Error Correction
in
Robust Storage Structures

by

Ian John Davis

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, 1988

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

....

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

....

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Error Correction

in

Robust Storage Structures

The need to develop reliable computer systems is of paramount importance in many endeavours. To achieve this goal all aspects of software and hardware engineering must be made independently and collectively as reliable as possible by all means available to the system architect. Almost all computer systems make extensive use of data structures, and it is therefore appropriate to ask how data structures can be made as reliable as possible.

Previously, global and local constraints were imposed on the distribution of errors in storage structures, and results developed which indicated when such constrained errors were correctable. Algorithms were then developed which corrected these classes of errors.

This dissertation shows that, under a potentially larger class of errors, correctable errors can be distinguished from errors which are not correctable. Guidelines are then presented for identifying this larger class of errors. Using these guidelines, a number of new correction algorithms are developed which perform correction whenever possible and otherwise report failure.

This dissertation also significantly extends the theory of local error correction; presents four new locally correctable tree structures; and uses mathematical models to describe the expected behaviour of local correction procedures operating on arbitrary and specific storage structures.

Acknowledgements

First and foremost I would like to acknowledge the contribution that my wife, Anne R. Crowe M.D., made to this research. Without her financial support I would have been unable to conduct this research. Without her emotional support I would certainly not have completed it. Her unswerving faith in me has made the impossible possible.

I also owe a great debt of gratitude to all the friends who supported me through difficult times. In particular, I would like to thank Carl Seger and Michel Devine for their friendship in a shared ordeal, and Wilfrid Laurier University for their encouragement in this endeavour. I would also like to express my thanks to my counsellor, W. M. Muirhead M.D. At the point when I first approached him for counselling, I did not believe that I could complete this dissertation. The resulting inner conflicts had plunged me from a state of depression into despair. Dr. Muirhead's interest in me, and the medication that he prescribed, proved to be a critical turning point in the production of this thesis.

Clearly, this work would not have been possible without the assistance of my Ph.D. supervisor, Dr. D. J. Taylor. I am grateful to him for laying the foundations upon which this work is built; for his willingness to accept me into this program; and for the many years of supervision and backing that he has given me throughout this course of study. His diligence, perseverance, and attention to detail, are reflected in every page of this thesis.

Thanks are also extended to Dr. J. P. Black and Dr. G. V. Cormack for the interest that they have showed in this work, and for the assistance that they gave Dr. D. J. Taylor in supervising this work.

The experimental results presented in this thesis were obtained using ISSS, and the numerous graphs were produced using Maple and GTS. This software was developed by many people over a number of years, and these people were in turn supported by others. Collectively, their contribution is appreciated.

You smug-faced crowd with kindling eye
Who cheer when soldier lads march by
Sneak home and pray you'll never know
The hell where youth and laughter go.

SEIGFRIED SASSOON

Chapter I

Introduction

1. Preamble

Many engineering and scientific endeavours currently being undertaken depend critically on the correct behaviour of computer hardware and software [5[6, 12, 27, 36, 47, 80, 116]139]. However, modern computer hardware and software are inherently very complex and thus subject to error from numerous sources [26[57, 140]142]. Some of these errors can be avoided by careful design [17[29]93], development [16[1, 51, 95]110], testing and periodic maintenance, but others will still occur [55[56, 84, 101]102].

If the occurrence of an error is not detected promptly then this error may cause a series of further errors which collectively result in a catastrophic failure. It is therefore desirable that errors be detected. Having detected an error (or collection of errors) some intelligent response must be made, if the consequences of this error are to be minimized. In many systems the best strategy is to report failure and enter a ‘‘down’’ state cleanly but rapidly [113]. However, in other systems such an approach is not viable, since entering a down state is itself a catastrophic failure. In such systems, errors must be contained, diagnosed, and, whenever possible, removed [9[10, 99]107].

Much research has been conducted into methods of protecting computer systems against hardware errors [35[39, 63, 65, 64]104]. These methods tend to use massive amounts of redundant hardware [14[31, 90, 100]133] to detect faulty circuits, and to predict the behaviour of a correct system. In some systems, redundant components also provide standby spares for critical components. Such techniques have also been proposed by many as a means of improving the reliability of software [8[34, 72]106], but some recent studies suggest that these techniques may be less useful than anticipated [11[40, 73]74]. This is because independently developed versions of software often contain similar types of faults, which collectively mask the presence of errors. At a higher level of abstraction, information is often protected by using error-correcting codes. Many good codes are known that support detection and correction of a wide variety of errors [58[67, 81, 91]103]. Attempts have also been made to formally prove that hardware and software correspond to their intended design [68[86]137], but these attempts have met with limited success [46[118]119]. More recently, programming languages have been designed that assist in the development of reliable software [37[85, 89, 115, 117]138]. None the less, faults still occur. In desperation, some researchers have attempted to verify hardware and software by resorting to studies of its empirical behaviour, or by attempting to predict remaining faults by extrapolating from previously discovered faults.

The above techniques for coping with occasional errors achieve success by masking faults that might lead to failure, rather than by removing observed errors. If computer systems containing faults or developing errors must function correctly over an unbounded period of time, then periodically any errors that have been detected must be removed. Diagnosing erroneous hardware and software, while potentially tedious, is not generally difficult if one has tools that allow the behaviour of isolated sets of components to be monitored. This is because most hardware and software operates in a deterministic manner, producing definable outputs from given inputs. Having diagnosed the cause of an error, faulty hardware can be replaced and faulty software corrected.

Faults in hardware and software may introduce errors into data structures. These errors are very much harder to diagnose, simply because data structures change over time, and can continue to appear correct even when they contain errors. Historically, designers have therefore tended to use backups and some form of recovery log to provide protection against data structure errors [28[76, 131, 134]136].

Unfortunately, the costs associated with backing up data structures are very high [30[32, 88, 112]120]. Backups consume vast amounts of potentially valuable storage. Backing up entire file systems usually involves suspending other ongoing computer activities for a considerable period of time, while any attempt to perform concurrent backups introduces severe scheduling and administrative problems, particularly in distributed systems. Recovery logs are produced concurrently with other ongoing activities, and therefore present fewer administrative problems than backups. However, recovery logs can also consume vast amounts of online storage, particularly if these logs are themselves to be protected against occasional errors. The logic needed to produce recovery logs is normally quite complex and therefore subject to error. In addition, this logic may, as a result of dramatically increasing the amount of input/output being performed within a computer system, seriously degrade the performance of that computer system.

Difficulties also arise when attempting to use backups to restore data to a correct state. It normally takes considerable time to coordinate such activities, and recovery, once initiated, often uses critical computing resources for a considerable period of time. Following the completion of all recovery activities, the status of the restored data remains very much in doubt. This is because the restored data is very dependent on the correctness of backups, recovery logs, and, perhaps most importantly, the recovery procedures used. Unfortunately, since such recovery procedures circumvent the need for error diagnosis, it is generally quite hard to ensure that all errors have been removed by the recovery procedure and that no new errors have been introduced. Often no effort is made to verify the correctness of the restored data and this naturally invites disaster.

Many users therefore endeavour to repair damaged data structures by guessing at the appropriate set of corrections [13[15, 38]96] and use backups only as a last resort. In desperation, others attempt this after discovering that backups have failed [135]. After a while the techniques used in making such guesses become formalized and embedded in ‘‘scavenger’’ programs that perform error detection and, when requested, automatic error correction [49[53, 78, 87, 97]109].

Clearly, we cannot expect scavenger programs to correct or even detect all errors. A small number of well chosen errors can undermine the behaviour of almost any conceivable scavenger. Conversely, a large number of errors can make any intelligent response impossible, either by destroying all meaningful data, or by altering this data so that it becomes grossly misleading. However, these problems can be minimized by ensuring that system designers develop data structures and scavengers in parallel, and use all of the redundancy present in the design of the data structures when designing scavenger programs.

2. Robust storage structures

Designers of robust storage structures are very ambitious, and arguably foolhardy, individuals. They seek to develop structures which are competitive with existing storage structures and algorithms which can correct unspecified errors in these structures [19[45, 52]129]. In order to be competitive with existing structures, robust storage structures must be easy to implement, use and update, and must make efficient use of both space and time [59]. Correction algorithms should also be efficient in both space and time, and should have a proven ability to perform correction under a large class of errors. In addition, they should be capable of reporting failure, with high probability, when uncorrectable errors are encountered; otherwise, there is a considerable risk that unexpected errors will cause correction algorithms to introduce new errors into already erroneous structures.

A major problem besets the designer of a robust storage structure. Adding useful redundancy to an existing storage structure is difficult. This redundancy is only useful if it allows an, as yet unknown, algorithm to perform error detection, or preferably error correction, in an intelligent manner. Ideally, the designer would be given an exact specification for the desired behaviour of the correction algorithm, and then would be able to determine easily the optimal method of achieving this behaviour by carefully inserting redundancy into the structure being corrected.

As a first step towards meeting this goal, a number of related issues must be at least partially resolved, before we can even envision what such a specification might contain, how it might assist in the development of a robust storage structure, and how it might be used to verify that the resulting storage structure was in some sense optimal.

1) Given that a correction algorithm will necessarily be unable to correct all errors, what are the specifications that a correction algorithm should operate under? Specifications challenge designers to meet certain desirable goals, and are necessary if algorithms are to be formally shown to be correct. However, these specifications also blinker designers, and may (if poorly constructed) encourage them to reject excellent methods of protecting structural data against errors, simply because these methods violate some unimportant detail of the specification. Thus the specifications and goals of error correction must be constantly reviewed and improved upon whenever possible.

2) In order for a correction algorithm to be viable, it must be capable of deducing from damaged instances what the most appropriate response to this damage is. However, unless some assumptions are made about the nature of the errors introduced into the instance, this is impossible. One simplifying assumption is that errors are independent and uniformly distributed. This assumption is attractive since it allows many correction algorithms to be developed that have good theoretical behaviour. However, such simplifying assumptions are hardly realistic, and it is therefore essential that the behaviour of detection and correction algorithms be studied when operating on more realistic types and distributions of errors [5070].

3) The above is obviously important if one wishes to design good robust storage structures, but provides no indication of how such robust storage structures are to be developed. There is therefore a need to develop theoretical guidelines [108[125]126], indicating the properties that robust storage structures must have if they are to facilitate certain types of error correction. This would allow designers to easily reject inappropriate designs for robust storage structures, thus concentrating their efforts on those that appeared most promising.

4) Having developed a robust storage structure and an associated correction algorithm which has a provable behaviour under a certain class of errors, the nature of the robust storage structure should be carefully reviewed, and additional properties of this structure identified. This is an important activity, since it may reveal that the structure is more robust than first believed, or that the structure can be corrected more efficiently or accurately using techniques not originally considered when designing the structure to meet a particular specification.

5) There is also a need to develop a variety of different robust storage structures, so that these can be evaluated and compared with existing storage structures. It is hoped that in the process new ideas and better methods of introducing redundancy into storage structures will be discovered.

6) The evaluation of robust storage structures and their associated correction routines has historically involved empirical studies, in which the size of the instance being corrected, and the distribution of errors within this instance were carefully controlled. There is a need to develop statistical models which provide an approximate indication of the likelihood of correction when these carefully controlled parameters vary. Such models would, if reasonably accurate, provide a much better indication of the expected behaviour of correction algorithms, and would allow more reasonable comparisons to be made between different robust storage structures and associated correction algorithms.

Previous research conducted into robust storage structures is reviewed in Chapter 2. Chapter 3 and Chapter 4 propose goals for error correction which are more general than those previously considered, and contain a number of theoretical results which may be used to determine when these goals can be achieved. Using these theoretical results, new correction algorithms are presented in Chapter 5 and Chapter 6, for various regular linked list structures, which would appear to be significantly better than previous correction algorithms proposed for similar structures. Three new robust tree structures are presented in Chapter 7, and in Chapter 8 mathematical models are developed which allow robust storage structures to be compared and analyzed very much more easily than was previously possible. Finally, in Chapter 9 conclusions are drawn and further avenues for research suggested.

Chapter II

Background

3. Terminology

For our purposes a *data structure* [122] is an abstract organization of data that allows units of data to be stored, accessed, and manipulated in a meaningful and useful way. A *storage structure* is a data structure design, and describes the type of nodes used to support the data structure, and the relationships between these nodes. A *storage structure encoding* defines one implementation of a storage structure, and thus specifies the *components* that exist within each node type, their representation and interpretation. A particular *instance* of a storage structure encoding is defined by the storage structure encoding to which it belongs, and by the *header nodes* that allow access to this instance. Finally, an *instance state* consists of all components (and associated values) currently occurring within that instance.

Storage structures typically contain several types of components. For example, *identifier* components explicitly establish the type of a node in a storage structure encoding, and identify the instance to which this node belongs. *Pointer* components establish access paths to nodes, typically by containing the address of these nodes. *Tag* components explicitly define the interpretation of other components in the instance. *Key* components explicitly define relationships between nodes in the instance. *Count* components explicitly define the number of nodes in an instance, the number of keys in a node, etc. *Checksum* components contain code words that facilitate error detection and/or correction of other components. Finally, *data* components contain information which, with respect to the storage structure specification, may be arbitrary.

Obviously, we may protest that data components contain values which are not arbitrary, but rather themselves highly structured. However, should we choose to accept this position, then we must either append to our specification of the storage structure the rules which data components satisfy, or must abandon the concept of an all-inclusive structural specification. In the former case such components no longer contain arbitrary values and therefore cease to be data components, while in the latter case our specification ceases to be authoritative and therefore becomes of little theoretical use.

Because data components may contain arbitrary values with respect to the storage structure specification, this storage structure specification cannot, in isolation, be used to correct or even detect errors in data components. Without loss of generality, we will therefore assume that storage structures contain no data components. Using this assumption it should be clear that a storage structure instance contains no detectable errors if and only if it is consistent with its structural specification.

Having developed techniques for ensuring that structural components of a storage structure are correct, we may then cease to assume that this storage structure contains only structural components. However, if we wish this structure to remain robust, we must then devise means of protecting data components against error, by using internal or external information not contained in the structural specification.

4. Previous research

Historically, robust storage structures and their associated error detection and correction routines were developed in an *ad hoc* manner. The need to develop good specifications for the behaviour of correction algorithms was first addressed by Dr. Taylor in his Ph.D. dissertation [121] in which he established some preliminary guidelines for the requirements to be imposed on algorithms that attempted structural error correction. These guidelines had many similarities with goals long considered desirable in coding theory, and resulted in some similar theories. However, the idea of applying such guidelines in the development of robust storage structures was a monumental step forward, and provided the foundation on which all subsequent research into robust storage structures has been built.

The early specifications for the behaviour of correction algorithms required that these algorithms be able to correct some small maximum number of errors in any instance being examined, by making some essentially realistic assumptions about the nature of the data memory space containing the erroneous instances [23[123]124]. Algorithms that met these specifications were called ‘‘global correction algorithms’’, while those that performed correction by examining only components reachable from the headers of an instance were considered ‘‘reasonable’’.

As part of his Ph.D. dissertation [22] Dr. Black expanded upon Dr. Taylor’s work by developing theories pertaining to ‘‘macro’’ changes that modified entire nodes, and introduced new results pertaining to the robustness of composite storage structures. He also applied axiomatic descriptions to storage structures [48[54]92], and catalogued the properties of existing storage structures [1821]. Towards the end of his thesis he presented a very interesting alternative specification for the behaviour of correction algorithms, which required that correction algorithms be able to correct an unbounded number of errors in a storage structure, if these errors were in some sense sufficiently distant from each other. This led to the development of a formal specifications for ‘‘local detection’’ and ‘‘local correction’’ procedures [24].

This collective body of theory pertaining to error correction led to the creation of a number of new robust storage structures and associated correction procedures [79[111, 114]141], many of which [20[98]130] were incorporated into a complex control system called ‘‘ISSS’’ [129]. When incorporated into this control system, data structures could be deliberately seeded with errors using a number of different techniques, and the behaviour of various algorithms which operated on these data structures then studied empirically.

5. System model

The system model [2] used to study robust implementations of data structures assumes that all storage structure instances reside in a common *data memory space* that is distinct from the *control memory space* used to support the operating system, to store executing programs, and to contain their associated working storage, registers, etc. This data memory space may be resident in main memory, or may be represented on disc or other peripheral devices. It may even be distributed among different devices and/or machines.

The data memory space comprises words of memory of some arbitrary fixed size. The number of words within this data memory space is finite, but very large. Each word in this data memory space has a distinct address, and contains in any given *data memory state* exactly one value.

Within this data memory space, storage structure instances may exist that each have a varying number of *data nodes* that are accessed via paths leading from some fixed set of *header nodes*. Each node is assumed to occupy some contiguous set of words within the data memory space, whose location is externally known if and only if the node is a header node. Internally, a uniquely identifying node name or number defines the location of each node. No word in a correct memory state occurs in more than one node, and typically all words in a correct data memory space occur in exactly one node.

Initially it is assumed that the data memory space contains some set of correct instance states. Instances may be changed by procedures which update these instances. Words in the data memory space may also be changed as a consequence of hardware, software, or other types of fault. If a word is assigned an incorrect value because of a fault, then it becomes erroneous. It remains erroneous until such time as it no longer contains an incorrect value. Although errors may be removed by faults, or by subsequent updates to the instances containing these errors, we obviously cannot rely on correction of errors by such means.

We will therefore periodically execute detection procedures which attempt to verify the correctness of the storage structure instances being examined. These procedures may be invoked as part of a preventative maintenance program, possibly occurring after each update, or may be specifically invoked when errors are detected by (or suspected in) concurrently executing software that manipulates these structures. Such procedures will also typically be executed following externally observable malfunctions, such as system failures.

When detection procedures report the presence of errors, it becomes the responsibility of a correction procedure both to correctly diagnose the nature of the errors observed, and, when appropriate, to remove these errors. Often the activities of detection and correction are closely associated and therefore integrated into a single procedure, which performs detection and potentially correction.

Generally, correction algorithms should only modify words in the data memory space when they have good reason to believe that this will reverse previously introduced erroneous changes. Otherwise the execution of such algorithms can be expected to introduce additional erroneous changes into the storage structure instance being examined, making subsequent correction very much harder, or impossible, to achieve.

While many processes are potentially concurrently accessing and updating instances of storage structures in the data memory space, it is assumed that a procedure attempting to detect or correct errors in the data memory space can use locking or other techniques to ensure that the instance currently being examined is not concurrently being updated. Thus the behaviour of these algorithms can be investigated in isolation, and the correctness of the structures being examined defined, either by reference to a detection procedure [3], or by presenting axiomatic specifications [2254].

Clearly, the system model described above is invalid in most computer systems, for a number of reasons. Typically, no data memory space exists for the exclusive purpose of representing all instances of storage structures, and such memory spaces as exist may vary in size. In virtual memory spaces, distinct words can cease to have distinct addresses, as a result of errors in the underlying mapping that supports these data memory spaces, and this can also occur as a result of hardware error. In most data memory spaces, a certain amount of duplication occurs, either in cache memory [66] or on peripheral devices, and therefore a single word in the data memory space may be capable of observably containing more than one value. Finally, in applications where reliable file structures might reasonably be used, we can expect to find hardware assuming some of the responsibility for detecting and correcting errors in the data memory space.

The assumption that detection and correction procedures observe instances in the data memory space which are not concurrently being updated, is also somewhat unrealistic in the presence of arbitrary errors, since such procedures may themselves inadvertently examine instances which they have not locked against concurrent access. Even when all instances are locked against concurrent access it is still possible for words to be occasionally changed as a result of faults. In cases where this might otherwise be of concern, we can assume that algorithms preserve their own sanity by rigorously examining each word in the data memory space at most once.

6. Crash recovery

It is often assumed that the robustness of a storage structure is merely a function of the amount of redundancy contained within that storage structure, and that the study of robust storage structures is therefore solely concerned with maximizing the robustness of useful storage structures, while minimizing the amount of data redundancy used.

This attitude is overly simplistic. Many factors determine the robustness of a storage structure. Obviously the type and frequency of errors that occur in a storage structure have a profound effect on the robustness of this storage structure. In particular, if most data structure errors occur as a result of instantaneous system crashes, then much can be accomplished by controlling the order in which updates are performed.

Although crash recovery is not addressed in this dissertation, by assigning a partial ordering to the sequence in which updates are applied to disc, some structures can be made crash resilient, without significantly degrading system performance. In particular, the logic that updates the Unix file system can ensure that this file system contains no structural errors following instantaneous system crashes [41].

This technique has also been used to show that a variety of linked-list structures can be made crash recoverable, and that these structures can be corrected by using existing local correction algorithms [127].

The possibility for performing crash recovery in binary trees has also been considered. In [132] it has been shown that existing global correction routines can perform crash recovery in some binary trees, if these binary trees are updated in a specific sequence using non-standard and somewhat inefficient update techniques. This paper assumes that correction algorithms have no knowledge of the method used to update such structures, Obviously, crash recovery of binary trees can be accomplished very much more simply and efficiently if such an assumption is not made.

Chapter III

Global correction

7. Introduction

This chapter explores how data memory spaces might be corrected when they contain at most some small bounded number of errors. After introducing some new notation and terminology, the previous body of theory that pertains to this problem is reviewed. This previous research attempts to identify the maximum number of errors which can necessarily be corrected in a data memory space containing some set of instances, and then, assuming that at most this number of errors occurs, suggests various methods of performing correction.

It is then suggested that correction algorithms might be able to tolerate more errors in the data memory space than allowed for by previous theories, if correction algorithms were designed so that they distinguished between correctable and uncorrectable sets of errors. After establishing bounds on the maximum number of errors that can be tolerated, if global correction is to be performed whenever possible, we present a new selective global correction algorithm which either corrects two errors in a mod(2) double-linked list or reports that these two errors have disconnected the structure.

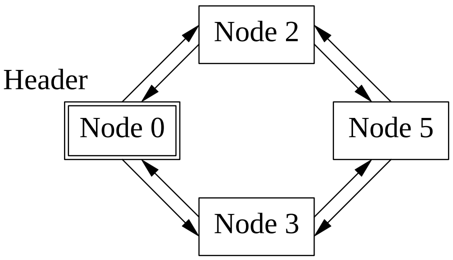
8. Notation and terminology

When errors are introduced into words of the data memory space, a new *corrupt* data memory state is produced. Any instance containing one or more such errors is also considered corrupt. Corrupt instances which contain detectable errors will be considered *incorrect*. Otherwise they continue to appear *correct*. In a corrupt instance the header nodes can by assumption be located, but other structural information becomes suspect. In particular, data nodes belonging to the original correct instance may become *disconnected* if all paths from the header nodes of this instance that correctly lead to these data nodes are damaged, while other arbitrary nodes may erroneously appear to be part of the corrupt instance. The state of an incorrect instance is therefore subjective. However, when the addresses of the header nodes of an incorrect instance are presented to an arbitrary deterministic algorithm, P, this algorithm examines, and possibly updates, a specific collection of words. These words constitute the state of the incorrect instance as observed by P, and when interpreted are referred to as components.

We will be primarily interested in the instance states x_1, x_2, etc. observed by a procedure, P, when the addresses, H_X, of the headers of an arbitrary instance, X, are presented to this procedure. It is stressed that although x_1 and x_2 represent different instance states, x_1 and x_2 have the same header nodes.

A correct instance state x_i will be denoted by x_i^c, and the set of all correct instance states of X by x^c. A specific data memory state containing an instance state x_i will be denoted by $[x_i]_m$. As observed in [121] the procedure P cannot distinguish between the memory state $[x_i]_m$ and the memory state $[x_i]_n$, since P examines only components in x_i. The collection of memory states that P cannot distinguish from $[x_i]_m$ form an equivalence class and will be denoted by $[x_i]$. A small example clarifying this notation is presented in Figure 3.1.

The *distance* $d([x_i]_m, [x_j]_n)$ between two arbitrary memory states $[x_i]_m$ and $[x_j]_n$ is simply the number of words in these two data memory states that differ. Generalizing, the distance $d([x_i], [x_j])$ between two non-empty sets of memory states $[x_i]$ and $[x_j]$ is the minimum distance between any $[x_i]_m$ and any $[x_j]_n$. If $d([x_i]_m, [x_j]) = d([x_i], [x_j])$ then $[x_i]_m$ is a *closest* member in $[x_i]$ to $[x_j]$. Finally, if $0 < d([x_i], [x_j]) \leq d([x_k], [x_j])$ for all $x_k \neq x_j$ then $[x_i]$ is closest to $[x_j]$. Note that there may be more than one set (or members of a set) of data memory states, which are closest to another set of data memory states, and that the distance between sets of data memory states is zero if and only if these sets contain some common member.



The instance state x_1^c

Node	Id	Ptr	Ptr
0:	-1	2	3
1:	0	6	-5
2:	-1	5	0
3:	-1	0	5
4:	-1	7	12
5:	-1	3	2
6:	0	0	0

The memory state $[x_1^c]_1$

Node	Id	Ptr	Ptr
0:	-1	2	3
1:	?	?	?
2:	-1	5	0
3:	-1	0	5
4:	?	?	?
5:	-1	3	2
6:	?	?	?

The memory states $[x_1^c]$

Figure 0.1. An instance state and the memory states containing it

9. Connection

We will consider an arbitrary word occurring in x_i^c to be *erroneous* in $[x_j]_k$ if the value of this word is different in x_i^c and $[x_j]_k$. A correct instance state, x_i^c, whose header node addresses are known, remains *connected* in $[x_j]_k$, if the address of each word in x_i^c can be determined without examining any erroneous words in $[x_j]_k$. Similarly, x_i^c remains connected in x_j if $x_i^c \subseteq x_j$ and all words in x_i^c can be located in x_j without examining any erroneous words in x_j. Finally, if all members of x^c remain connected in the data memory space, when at most n errors are introduced into the data memory space, then X is n-connected.

It should be noted that if x_i^c remains connected in x_j and an arbitrary error (with respect to x_i^c) is removed from x_j, producing x_k, then x_i^c remains connected in x_k.

Example 0.1

Consider a circular linked list having a single header node. Every node in this list contains n forward pointers, which correctly address the next node in the list. For simplicity, assume that nodes contain no other components. If the structure contains at most $n-1$ errors then every node in this structure contains at least one correct pointer. Thus, this structure can be traversed by an algorithm which examines no erroneous components. Therefore the structure is $(n-1)$-connected. However, the structure is not n-connected, since n errors can disconnect any non-null instance of this structure. Now consider placing $m \geq n$ errors into m distinct nodes of some instance state, x_i^c, of this storage structure, producing the corrupt instance state x_j. If $n \geq 2$ then x_i^c necessarily remains connected in x_j, as justified above. Otherwise, x_i^c remains connected in x_j if and only if x_j contains a single error and this error occurs in the pointer which correctly contains the header address, H_X.

10. Valid State Hypothesis

In any data memory state containing no errors it will be assumed that the Valid State Hypothesis holds. This asserts [22121] that for any word w belonging to a correct data memory state:

a) If w can be interpreted as a node identifier of an arbitrary instance state x_i^c, then w contains none of the set of node identifier values for X unless w is a node identifier component of x_i^c.

b) If w can be interpreted as a pointer component of an arbitrary instance state x_i^c, then w contains the address of no node in x_i^c unless w is a pointer component of x_i^c.

Any data memory state satisfying the Valid State Hypothesis is *valid*. The subset of the set of data memory states $[x_i^c]$ which are valid when assumed to contain only x_i^c will be denoted by $[x_i^c]^v$. When the data memory state contains more than one instance state we will say that $[x_i^c]^v$ is valid (or satisfies the Valid State Hypothesis) with respect to x_i^c. Similarly, a single memory state $[x_i^c]_m \in [x_i^c]^v$ will be denoted by $[x_i^c]_m^v$. It is stressed that $[x_i^c]_m^v$ is not necessarily valid, since a data memory state is valid if and only if it is valid with respect to all instance states contained within it. Indeed, if x_i^c contains a header node of some other instance, Y, then no member of $[x_i^c]^v$ is valid, since Y is necessarily invalid in $[x_i^c]$. Additional examples of invalidity are presented later in Example 3.2.

If $r < \min(d([x_i^c]^v, [x_j^c]))$ for all x_i^c and x_j^c satisfying $x_i^c \neq x_j^c$, then the instance X is *r-detectable*. Similarly, if $r < \min(d([x_i^c]^v, [x_j^c]^v))$ for all x_i^c and x_j^c satisfying $x_i^c \neq x_j^c$, then the instance X is *r-absolute-detectable*. These and other properties become *exact*, if they are maximal. When all instances of a storage structure have a particular property, as is typically the case, the storage structure also has this property [121].

11. Reasonable procedures

Two types of procedure will be of particular importance to us. A detection procedure, P_1, when presented with the addresses, H_X, of the headers of an arbitrary instance, X, on which it has been designed to operate, and a data memory state $[x_i]_m$, determines if $x_i \in x^c$. A correction procedure, P_2, when presented with the same information, either attempts to transform $[x_i]_m$ into some suitable $[x_j^c]_n^v$, or reports that it is unable to perform this transformation.

Unfortunately, determining that $[x_j^c]_n \in [x_j^c]^v$, involves examining every component in the data memory state. This is obviously not very desirable when attempting to correct isolated storage structure instances, since it implies that the correction time will be dependent on the size of the data memory state, rather than the size of the instance being corrected, which may possibly be very small. In addition, algorithms that examine the entire data memory state will almost certainly require that this data memory space be in a quiescent state, even though most instances will be correct, and thus potentially modifiable.

It therefore seems appropriate to develop *reasonable* algorithms [121] which, when presented with the header node addresses, H_X, of x_i, locate all other nodes of x_i by using pointers occurring in x_i, which form paths from these header nodes. Although a reasonable algorithm operating on a corrupt data memory state may continue to observe instances, x_i, whose size is bounded only by the size of the data memory space, we will endeavour to design reasonable algorithms which display this behaviour only when certain assumptions about the nature of errors encountered are violated. In this chapter, it will be assumed that reasonable algorithms operate on instances that contain some bounded number of errors. When nodes within these instances contain node identifiers, this implies [121] that reasonable algorithms need traverse at most some bounded number of nodes not present in the original instance being corrected.

12. Seeming validity

Although we wish to develop correction procedures which exploit violations of the Valid State Hypothesis, reasonable algorithms cannot by their nature determine if $[x_j^c]_n$ is valid. However, a reasonable correction algorithm operating on $[x_i]_m$ may be able to determine that some other data memory state $[x_j^c]_n$ contains violations of the Valid State Hypothesis with respect to x_j^c, even though reasonable algorithms that operated on $[x_j^c]_n$ can not. This is because x_i may contain nodes not in x_j^c, whose component values violate the Valid State Hypothesis with respect to x_j^c when occurring in $[x_j^c]_n$.

We will consider $[x_j^c]_n$ to be *seemingly valid* with respect to an instance state x_i, if violations of the Valid State Hypothesis with respect to x_j^c do not occur, or can be detected in $[x_j^c]_n$ only by examining components neither in x_i nor x_j^c. When the context is clear we will denote a seemingly valid data memory state $[x_j^c]_n$ by $[x_j^{cv}]_n$, the class of seemingly valid data memory states containing x_j^c by $[x_j^{cv}]$, and the class of all data memory states which are seemingly valid with respect to x_i by $[x^{cv}]$.

If components in x_j^c violate the validity of x_j^c, then serious problems arise. This is because $d([x_i], [x_j^{cv}])$ and $d([x_i], [x_j^c]^v)$ are undefined for all x_i, since $[x_j^{cv}] = [x_j^c]^v = \varnothing$. Fortunately, if any correct instance state, x_j^c, was not seemingly valid with respect to itself, then this would contradict our assumption that a data memory state containing no errors is valid, since a correct data memory state containing x_j^c is not valid. Therefore, we may assert that any x_j^c is seemingly valid with respect to itself.

Example 0.2

Consider the data memory state, $[x_1^c]_1$, presented previously in Figure 3.1. This data memory state contains three violations of the Valid State Hypothesis. The first word in node 4 contains the value -1 which, in this example, is the correct node identifier value for x_1^c and the second and third words in node 6 contain the value 0 which, when interpreted as a pointer, is the address of the header node of x_1^c. However, a reasonable algorithm examining x_1^c will observe none of these violations. Thus $[x_1^c]_1$ is seemingly valid with respect to x_1^c.

Now change one pointer in x_1^c so that it addresses node 4, node 6, or node 1, producing the data memory state $[x_2]_1$. Then, provided that x_2 contains at least one of the words (described above) that violates the validity of $[x_1^c]_1$, any algorithm which transforms $[x_2]_1$ back into $[x_1^c]_1$, as a result of examining x_2, can observe that $[x_1^c]_1$ violates the Valid State Hypothesis. Thus $[x_1^c]_1$ is not seemingly valid with respect to x_2.

13. Correction procedures

Ideally, a reasonable correction procedure, P, attempting to correct an instance state x_1 residing in a data memory state $[x_1]_1$, would determine the class of data memory states $[x_2^{cv}]$ that was observably closest to $[x_1]_1$. If two or more such classes existed the correction procedure should report failure, since the appropriate correction would be ambiguous. Similarly, if x_1 was disconnected with respect to some unknown instance state x_3^c and $d([x_1],[x_3^c]^v) \le d([x_1],[x_2^{cv}])$ then P should report failure, since the original correct instance might have become disconnected. Otherwise, the erroneous data memory state $[x_1]_1$ should optionally be transformed into the closest correct and seemingly valid data memory state $[x_2^{cv}]_1$.

Unfortunately, such a strategy seems very difficult to implement efficiently, or even effectively. Allowing arbitrary errors to occur greatly increases the complexity involved in determining the nature of those errors. The search for the nearest correct instance suggests that we must perform some form of exhaustive search. In addition, we must use some form of deductive reasoning to deduce when the nearest correct instance may have become disconnected, within the observed instance state x_1. Since the size of x_1 is objective only with respect to a given algorithm, algorithms which perceive the size of x_1 to be large may be more successful at detecting violations of the Valid State Hypothesis with respect to x_2^c than those that perceive x_1 to be small. Thus even the notion of an ideal reasonable correction algorithm becomes cloudy.

As an alternative strategy, it seems appropriate to assume that some maximum number of errors, n, occurs, and to consider an instance state, x_1, *correctable* if there exists exactly one $[x_2^{cv}]$ for which $d([x_1],[x_2^{cv}]) \le n$. The instance state x_1 is correctable by a reasonable algorithm if x_2^c remains connected in x_1. If all instance states of X containing at most n errors are correctable by a reasonable algorithm, then the instance X is *n-correctable* [121].

Consider introducing errors into some $[x_1^c]_1^v$ producing $[x_2]_1$, and then presenting H_X to a reasonable correction algorithm, P. If P introduces no new errors into the data memory space, then P *tolerates* the set of errors in x_2. Otherwise, P is *misled* by the set of errors in x_2. If P can determine the location of an error in x_2, then P *identifies* this error. Finally, if P can identify all erroneous components in x_2, which occur in x_1^c, and determine their correct value in x_1^c, then P can *correct* the errors introduced into x_1^c.

Lemma 0.1

$$d([x_i]_m,[x_j^c]^v) \ge d([x_i],[x_j^c]^v) \ge d([x_i],[x_j^{cv}]) \ge d([x_i],[x_j^c]).$$

Proof

Since $[x_i]_m \in [x_i]$ and $[x^c_j]^v \subseteq [x^{cv}_j] \subseteq [x^c_j]$ the inequalities follow immediately.

Lemma 0.2

If $[x^{cv}_j]_1$ is seemingly valid with respect to x_i, then $d([x_i], [x^{cv}_j]) = d([x_i], [x^c_j]^v)$.

Proof

The proof corresponds closely to the proof of Theorem 4.2.1. presented in [121]. However, the claim made in this earlier theorem is incorrect.

If some member in $[x^{cv}_j]$ which is closest to $[x_i]$ is also a member of $[x^c_j]^v$ then the result follows immediately. So assume otherwise. Then every member in $[x^{cv}_j]$ which is closest to $[x_i]$ violates the validity of x^c_j. These violations must occur in words belonging to neither x_i nor x^c_j since all data memory states in $[x^{cv}_j]$ are seemingly valid with respect to both x_i and x^c_j. But this implies that these violations can be removed from some data memory state $[x^{cv}_j]_1$ which is closest to $[x_i]$, producing some $[x^c_j]^v_2$ which is as close to $[x_i]$ as $[x^{cv}_j]$. This contradicts the assertion that every member in $[x^{cv}_j]$ which is closest to $[x_i]$ violates the validity of x^c_j.

As observed in [121], $d([x_i], [x^c_j])$ will be less than $d([x_i], [x^{cv}_j])$ whenever components occurring in x_i but not in x^c_j violate the validity of x^c_j. Thus, the minimum number of changes needed to transform a damaged instance of a storage structure into a correct instance may be less than the number needed to cause the observed damage in a correct and valid instance.

Correction algorithms that ignore observable violations of the Valid State Hypothesis and merely convert an incorrect instance into the nearest correct instance lying within some number of changes of the incorrect instance, cannot always perform correction when the number of errors exceeds half the exact detectability of the storage structure, since under at least one choice of errors the damaged instance is, by definition of detectability, correctable in more than one way.

However, as shown in [121], correction algorithms which convert examined instances of a storage structure into a correct instance by considering how the damaged instance may have arisen in a data memory state assumed to satisfy the Valid State Hypothesis, may be able to guarantee correction even when the number of errors anticipated exceeds half the detectability of the storage structure being corrected.

Lemma 0.3

If x_1^c and x_3^c are contained in x_2 then $d([x_1^c]^v, [x_3^c]^v) \leq d([x_1^c]^v, [x_2]) + d([x_2], [x_3^c]^v)$.

Proof

Suppose that there exists no $[x_2]_1$ which is closest to both $[x_1^c]^v$ and $[x_3^c]^v$. Then since any two members of $[x_2]$ differ only in the components not contained in x_2, and x_3^c is contained in x_2, any member $[x_2]_2$ which is closest to $[x_1^c]^v$ must contain words outside x_2 which violate the validity of x_3^c. But these words do not occur in x_1^c, and therefore can be changed so that they violate the validity of neither x_1^c nor x_3^c, contradiction. Thus there exists some $[x_2]_1$ which is closest to both $[x_1^c]^v$ and $[x_3^c]^v$.

Let $[x_1^c]_1^v$ be such that $d([x_1^c]_1^v, [x_2]_1) = d([x_1^c]^v, [x_2]_1)$ and let $[x_3^c]_1^v$ be such that $d([x_3^c]_1^v, [x_2]_1) = d([x_3^c]^v, [x_2]_1)$. Then, since the distance function is a metric when applied to individual data memory states [121], $d([x_1^c]^v, [x_3^c]^v) \leq d([x_1^c]_1^v, [x_3^c]_1^v) \leq d([x_1^c]_1^v, [x_2]_1) + d([x_2]_1, [x_3^c]_1^v) = d([x_1^c]^v, [x_2]) + d([x_2], [x_3^c]^v)$.

Counterexample 0.1

It is not necessarily the case that $d([x_1^c]^v, [x_3^c]^v) \leq d([x_1^c]^v, [x_2]) + d([x_2], [x_3^c]^v)$. For example, let $x_2 = x_1^c$. Then $d([x_1^c]^v, [x_2]) + d([x_2], [x_3^c]^v) = d([x_1^c], [x_3^c]^v)$ which, as already observed, may be less than $d([x_1^c]^v, [x_3^c]^v)$.

Theorem 0.1

If a storage structure is c-connected, r-absolute-detectable and $n = \min(c, \lfloor r/2 \rfloor)$ then the storage structure is n-correctable.

Proof

Consider introducing at most n errors into some correct and valid instance state $[x_1^c]^v$ producing the instance state $[x_2]$. Then, since the storage structure is c-connected, for $c \geq n$, there exists a reasonable procedure which observes an instance state x_2 containing all of the components in x_1^c.

Suppose that x_2 is not n-correctable. Then there exists some $x_3^c \neq x_1^c$ satisfying $d([x_2], [x_3^{cv}]) \leq n \leq c$ which therefore remains connected in some suitably chosen x_2 satisfying the above. By Lemma 3.2, $d([x_2], [x_1^{cv}]) = d([x_2], [x_1^c]^v)$ and $d([x_2], [x_3^{cv}]) = d([x_2], [x_3^c]^v)$. Therefore, by Lemma 3.3, $d([x_1^c]^v, [x_3^c]^v) \leq d([x_1^c]^v, [x_2]) + d([x_2], [x_3^c]^v) = d([x_1^{cv}], [x_2]) + d([x_2], [x_3^{cv}]) \leq 2 * n \leq 2 * \lfloor r/2 \rfloor \leq r$. But the storage structure is at least r-absolute-detectable, contradiction. Therefore, x_2 is correctable.

This theorem is similar to Theorem 4.3.3 presented in [121] and reproduced as Theorem 4.3 in [122]. However, the earlier theorem had a procedural proof and required that all nodes contained node identifiers. Because this theorem has a proof which is not procedural, this theorem does not require that all nodes contain node identifiers.

Example 0.3

Consider a standard circular double-linked list, having an identifier component in each node, but no count. Within a non-empty instance of such a structure an arbitrary number of consecutive nodes can be deleted, by changing a forward and backward pointer appropriately. The resulting structure contains no detectable errors, implying that the structure is at most 1-detectable. However, a single error, in either an identifier component or a pointer component, can always be corrected. In the former case, all pointers appear correct, implying that the identifier component is in error. In the latter case, the erroneous pointer either addresses a node having an incorrect identifier component or, when assumed to be correct, causes at least one node having a correct identifier to become disconnected, in violation of the Valid State Hypothesis. The described structure is therefore exactly 1-detectable and 1-correctable.

Alternatively observe that, when transforming some instance state $[x_1^c]^v$ into some distinct instance state $[x_2^c]^v$, at least five changes are required to add or delete nodes from the instance, at least six changes are required to reorder nodes within the instance, and at least eight changes are required to replace nodes within the instance with nodes from outside the instance. The storage structure is therefore 4-absolute-detectable [124], and thus, by Theorem 3.1, 1-correctable.

14. Selective correction

Historically, correction algorithms have been required to correct all errors introduced into a data structure under some given assumption about the nature of the errors introduced into that structure. Correction algorithms have therefore been developed that have essentially undefined behaviour when the number of errors encountered exceeds the number for which correction is certain. This is unfortunate, since it avoids the need to address a number of important issues.

What types of error should a global correction algorithm be expected to tolerate? How might a correction algorithm identify the location of these errors and in particular determine when these errors cause disconnection? How might a correction algorithm correct these types of error whenever possible? How might a correction algorithm determine that the assumptions under which it is operating have been violated, and what ought it to do in all of the above cases?

We begin addressing these issues by considering the types of error which a global correction algorithm might be expected to tolerate. Firstly, it seems appropriate to continue to demand that global correction algorithms have no *a priori* knowledge about the cause of errors, and thus can make no assumption about the distribution of the errors observed. Secondly, it seems appropriate to continue to anticipate that at most some number of errors occur globally within the structure being corrected. It only remains to determine the number of errors that such a correction algorithm should be able to tolerate, while either performing correction or alternatively reporting that correction is impossible. In this latter case, the correction algorithm may optionally correct some of the errors in the instance, but is not permitted to introduce new errors into the instance being examined.

Consider allowing an increasing number of errors to be introduced into correct instances of an arbitrary storage structure. When at most some some small number of errors is allowed to occur, which may possibly be zero, all damaged instances remain both connected and closer to the original correct instance than to any other correct instance. When more errors are allowed to occur, all damaged instances may continue to be correctable, because they remain connected and are closer to one correct and valid instance than any other. When further errors are allowed to occur, some new damaged instance states may be created which are uncorrectable, either because they have become disconnected, or because they may be derived from two or more distinct but correct and seemingly valid instances. Finally, if we allow sufficient errors to occur, it becomes possible for damaged instances that were previously correctable to now be derived from more than one correct and seemingly valid instance, and thus to cease to be correctable.

Given that we wish to correct errors whenever possible, it therefore seems appropriate to assume that the maximum number of errors that can occur is one less than the number of errors needed to invalidate the assertion that some erroneous instance is correctable. A storage structure will therefore be considered *n-selective-correctable*, if all instances which are correctable by a reasonable procedure when fewer than *n* errors occur in the instance continue to be correctable when at most *n* errors are assumed to occur.

Thus, formally, a storage structure X is n-selective-correctable if, for all x_i and all x_j^c remaining connected in x_i which satisfy $d([x_i], [x_j^{cv}]) < n$, either there exists some $x_k^c \neq x_j^c$ such that $d([x_i], [x_k^{cv}]) \leq d([x_i], [x_j^{cv}])$ or for all $x_k^c \neq x_j^c$ $d([x_i], [x_k^{cv}]) > n$.

An *n-selective global correction routine*, when operating on an instance containing at most *n* errors, must correct all correctable instances, and tolerate all other incorrect instances. Typically we will require that these routines be reasonable, and operate on storage structures which are at least *n*-selectively correctable. We will expect these algorithms to explicitly report when correction is not possible, and to further indicate the cause of failure. Failure may occur because the instance being corrected appears disconnected, capable of being corrected in more than one way, or to contain more errors than have been assumed to have occurred. The decision to report failure in each of these cases can be justified on the grounds that it allows a more appropriate correction technique (such as resorting to backups, or more global correction routines) to then be applied.

We now establish various bounds on the selective correctability of an arbitrary robust storage structure, and then discuss how exact values can be determined for specific storage structures. Finally, in Section 3.10, we discuss how these bounds can assist in the development of n-selective global correction routines.

Lemma 0.4

Let a storage structure instance X be at least c-correctable, at most r-detectable, and exactly n-selective-correctable. Then $c \leq n \leq r$.

Proof

Since the storage structure is at least c-correctable, all instance states containing at most c errors are correctable. Thus the structure is at least c-selective-correctable. Consider some x_i^c residing in a valid data memory state which can be converted into some distinct x_j^c by applying $r + 1$ changes. Such an instance state must exist since the storage structure instance is at most r-detectable. All correct instances are 0-correctable, since distinct correct instances must differ, but x_i^c is not $(r + 1)$-correctable since it lies $r + 1$ changes from x_j^c. Thus the storage structure instance X is at most r-selective-correctable.

Example 0.4

It seems clear that many storage structures will have exactly the same correctability and selective correctability. To show that there also exist storage structures which are exactly n-correctable, n-detectable, and thus, by Lemma 3.4, n-selective correctable, consider a circular linked list X having n pointers per node, with fixed pointer distances $(d_1, d_2, \cdots, d_n)$, satisfying $|d_i| = 1$ for $1 \le i \le n$.

Since at least 3n changes are required to reorder nodes within this structure, at least 2n changes are required to replace nodes in this structure with nodes from outside the structure, and at least n changes are required to delete or add nodes to the structure the structure is $(n - 1)$-detectable, by Theorem 8 of [124].

Now consider the minimum number of changes needed to transform an arbitrary correct and valid instance state $[x_1^c]^v$, into a distinct correct and valid instance state $[x_2^c]^v$. At least 2n changes are needed to reorder or replace nodes in $[x_1^c]^v$ as justified above. However, at least 2n changes are now also needed to add nodes to or delete nodes from $[x_1^c]^v$. The structure is therefore $(2n - 1)$-absolute-detectable. Since n pointers address each node, the structure is $(n - 1)$-connected. The structure is therefore, by Theorem 3.1, $(n - 1)$-correctable.

Theorem 0.2

If a storage structure is r-detectable, and $n = \lceil r/2 \rceil$, then the structure is at least n-selective-correctable.

Proof

Consider a correct and valid data memory state, $[x_1^c]_1^v$, which has been transformed into a corrupt data memory state, $[x_2]_1$, by applying at most $n-1$ changes, and assume that x_2 is correctable.

Then, since the storage structure is r-detectable, $d([x_1^c]^v, [x_2]_1) + d([x_2]_1, [x_3^c]) \geq d([x_1^c]^v, [x_3^c]) \geq r+1$, for all $x_3^c \neq x_1^c$. Thus, $d([x_2]_1, [x_3^c]) \geq d([x_1^c]^v, [x_3^c]) - d([x_1^c]^v, [x_2]_1) \geq (r+1) - (n-1) = r + 2 - \lceil r/2 \rceil = \lfloor r/2 \rfloor + 2 > n$ for all $[x_3^c] \neq [x_1^c]$. Therefore, x_2 remains correctable when at most n errors are assumed to occur.

Counterexample 0.2

It should not be assumed in the above that $[x_2]_1$ is necessarily correctable when $d([x_1^c]^v, [x_2]_1) < n$. Consider for example a mod(4) double-linked list [44] that has been altered to have two identifiers per node. This structure has four consecutive header nodes, a forward pointer in each node which addresses the next node, a back pointer in each node that points back four nodes, and a count in one of the header nodes within the instance. The storage structure is 5-detectable, and therefore, by Theorem 3.2, 3-selective-correctable. It is however only 1-correctable, since two changes can disconnect a node.

Lemma 0.5

If a storage structure is exactly r-detectable, as demonstrated by a sequence of $r+1$ changes to a correct and valid instance state, x_0^c, which produce instance states x_1, x_2, ..., x_r, x_{r+1}^c, and x_0^c remains connected in x_i, for $0 \leq i \leq r+1$, then the storage structure is exactly n-selective-correctable for $n = \lceil r/2 \rceil$.

Proof

Let $m = \lfloor r/2 \rfloor$. Then $d([x_m], [x_0^{cv}]) = m$ and $d([x_m], [x_{r+1}^{cv}]) = r + 1 - m = r + 1 - \lfloor r/2 \rfloor = \lceil r/2 \rceil + 1 = n + 1 > m$. Since x_0^c remains connected in x_m, x_m is therefore m-correctable but not $n+1$-correctable. Thus the storage structure is at most n-selective-correctable. However, by Theorem 3.2, the storage structure is at least n-selective-correctable. Thus the storage structure is exactly n-selective-correctable.

Corollary 0.1

If a global examination of the data memory space is allowed, and the absolute detectability of a storage structure is r, then the absolute selective correctability of that storage structure is exactly $\lceil r/2 \rceil$.

Theorem 0.3

If $d([x_1^c]^v, [x_2]_1) \le n$, $d([x_2]_1, [x_3^{cv}]) \le n$ for some $x_1^c \ne x_3^c$, and x_1^c remains connected in x_2, then the structure is at most n-selective-correctable.

Proof

The proof proceeds by induction. In the base case, $n = 1$ so $d([x_1^c]^v, [x_2]_1) \le 1$ and $d([x_2]_1, [x_3^{cv}]) \le 1$, implying that the detectability of the storage structure is at most 1. Thus, by Lemma 3.4, the structure is at most 1-selective-correctable.

So assume that the selective correctability is at most $k - 1$ whenever $d([x_1^c]^v, [x_2]_1) \le k - 1$ and $d([x_2]_1, [x_3^{cv}]) \le k - 1$. Consider the case when $d([x_1^c]^v, [x_2]_1) \le k$.

Consider correcting one of the errors in x_2, producing the data memory state $[x_4]_1$, and let x_5^{cv} be the nearest corrupt but correct and seemingly valid instance to $[x_4]_1$. This is shown in Figure 3.2 below. Since x_1^c remains connected in x_2, x_1^c remains connected in x_4. Since one of the errors in x_2 (with respect to x_1^c) has been removed in x_4, $d([x_1^c]^v, [x_4]_1) = d([x_1^c]^v, [x_2]_1) - 1 \le k - 1$, and $d([x_4]_1, [x_3^{cv}]) \le d([x_2]_1, [x_3^{cv}]) + 1$.

If $d([x_2]_1, [x_3^{cv}]) \le k - 2$ then $d([x_4]_1, [x_3^{cv}]) \le k - 1$, implying by the inductive assumption that the structure is at most $(k - 1)$-selective-correctable.

If $d([x_2]_1, [x_3^{cv}]) = k - 1$ then $d([x_4]_1, [x_3^{cv}]) \le k$. If $d([x_4]_1, [x_5^{cv}]) \le k - 1$ then the structure is at most $(k - 1)$-selective-correctable by the inductive assumption. So assume that $d([x_4]_1, [x_5^{cv}]) > k - 1$. Then, $d([x_4]_1, [x_1^c]^v) \le k - 1 < d([x_4]_1, [x_5^{cv}]) \le d([x_4]_1, [x_3^{cv}]) \le k$, implying that $d([x_4]_1, [x_5^{cv}]) = k$. This implies that x_4 is correctable when at most $k - 1$ errors are assumed to occur, but not when k or more errors are assumed to have occurred. Thus the structure is at most $(k - 1)$-selective-correctable.

Finally, suppose that $d([x_2]_1, [x_3^{cv}]) = k$. If $d([x_4]_1, [x_5^{cv}]) \le k - 1$ then the instance is at most $(k - 1)$-selective-correctable by our inductive assumption. So assume that

$d([x_4]_1, [x_5^{cv}]) \geq k$. Then $d([x_4]_1, [x_1^c]^v) \leq k - 1 < d([x_4]_1, [x_5^{cv}]) \leq d([x_4]_1, [x_3^{cv}]) \leq k + 1$. This implies that x_4 is correctable when $k - 1$ errors are assumed to occur, but not when $k + 1$ errors are assumed to occur. Thus the structure is at most k-selective-correctable.

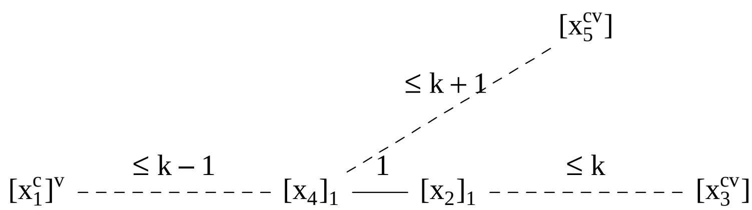


Figure 0.2. Effect of correcting an error in x_2 producing x_4

Corollary 0.2

If $d([x_1^c]^v, [x_2]_1) \leq n$, $d([x_2]_1, [x_3^{cv}]) < n$ for some $x_1^c \neq x_3^c$, and x_1^c remains connected in x_2, then the storage structure is at most $(n - 1)$-selective-correctable.

15. Calculating the maximum selective correctability

While the above results establish bounds on the value of the selective correctability of a storage structure, they do not necessarily identify the maximum value of the selective correctability of an arbitrary storage structure. Often the easiest way to identify the exact selective correctability of a storage structure is to assume that it has the minimum possible value subject to the above bounds, and then to present an example showing that this value is also maximal.

In general, however, the selective correctability can only be determined by carefully considering how successive changes introduced into instances of a storage structure affect the apparent minimum distance between correct and seemingly valid instances. Since this distance is reduced only by changes that disconnect nodes, it is appropriate to assume that changes that disconnect nodes are applied first.

Example 0.5

Consider again the helix(3) multi-linked list. Such a structure is 2-connected, 2-correctable, 6-detectable, and 10-absolute-detectable. Since it is 6-detectable, it is at least

3-selective-correctable. Consider introducing errors into the three pointer components in some node N_1 in the instance. Since the instance is 6-detectable and these errors do not cause disconnection, these errors can be corrected. However, if it is assumed that 4 errors may have occurred, then alternatively N_1 may not belong to the instance being corrected, contain a corrupt identifier, and be addressed by three incorrect pointers within the instance which should correctly address a common disconnected node. Thus the structure is exactly 3-selective-correctable.

Now consider the following modifications to this structure. Firstly, we will remove the count, thus producing a structure that is 5-detectable, since six changes can delete an arbitrary number of consecutive nodes. Secondly, we will require a correct structure to contain some number of nodes that is a multiple of three. This makes it harder for less than six changes to disconnect a node and yet produce a instance that is close to some corrupt but correct and seemingly valid instance. Finally, we will artificially increase the number of changes needed to exchange nodes in the instance with nodes outside the instance, by placing 5 independent node identifiers in each node.

The structure remains at least 3-selective-correctable, since it is 5-detectable. However 3 changes can no longer both disconnect a node and be part of those changes that constrain the detectability of the structure. Disconnecting three consecutive nodes, or some multiple thereof, requires six changes, no subset of which disconnects any node. Although one or two nodes can be disconnected using less than six changes, deleting or adding additional nodes so that the structure continues to contain some multiple of three nodes, requires more than five changes. This is because each node added or deleted contains five node identifiers. Finally, if nodes are reordered, or nodes within the instance replaced by the same number of nodes outside the instance then at least 11 changes are required. This rather bizarre structure is 5-selective-correctable, even though the structure is exactly 5-detectable, and only 2-correctable. Thus it is possible to reduce the detectability of a structure while increasing its selective correctability.

16. Using the selective correctability

Having identified some lower (and ideally maximal) bound on the selective correctability of a structure, which is greater than the correctability of the structure, this bound can be used to assist in the development of an *n-selective* global correction routine, which behaves at least as well as historical global correction routines operating on the same structures.

Suppose that an arbitrary storage structure is n-selective-correctable, and that nodes within this structure contain node identifiers. Consider a reasonable algorithm that is trying to correct some corrupt data memory state $[x_2]_1$. Then, as demonstrated in [121], a reasonable algorithm exists which can (by performing an exhaustive search) identify (whenever it exists) some x_1^c having the property that x_1^c remains connected in some suitably chosen x_2, and $d([x_2], [x_1^c]) = r < n$. Having found such an $[x_1^c]$ it is possible to identify within the subjective set of components forming x_2, the number of additional changes s needed to apparently preserve the Valid State Hypothesis when transforming some member of $[x_2]$ into a member of $[x_1^{cv}]$. Thus it is possible for a reasonable algorithm to determine if there exists some x_1^c which remains connected in x_2 such that $d([x_2], [x_1^{cv}]) = r + s < n$ [121].

Theorem 3.3 ensures that there exists at most one x_1^c having the above properties. Thus if such an x_1^c is discovered, and it is assumed that at most n errors occur within the instance being corrected, then the appropriate correction involves reversing the $r < n$ errors that have been introduced into x_1^c to produce x_2. It should be noted that the s components that violate the Valid State Hypothesis cannot be corrected by the algorithm that corrects x_2, since it cannot determine the correct values of components that lie outside of x_1^{cv}. However, having detected such violations of the Valid State Hypothesis, these violations may potentially be removed by correcting other instances within the data memory state.

So, suppose that the reasonable algorithm discovers no $[x_1^{cv}]_1$ for which $d([x_2], [x_1^{cv}]_1) < n$. Then the reasonable algorithm next searches for some $[x_1^{cv}]_2$ for which $d([x_2], [x_1^{cv}]_2) = n$.

If no $[x_1^c]_2$ is discovered which has the property that $d([x_2], [x_1^c]_2) = n$, then either $[x_2]_1$ has become disconnected with respect to all correct instances lying within n changes of x_2, or no such correct instance exist. Therefore, the appropriate action is to report failure.

Alternatively, if for all x_1^c satisfying $d([x_2], [x_1^c]) = n$, $d([x_2], [x_1^{cv}]) > n$, then an attempt must be made to remove violations of the Valid State Hypothesis, by correcting other instances, before possibly making further attempts to correct x_2.

Finally, if some x_1^c satisfies $d([x_2], [x_1^{cv}]) = n$ then the reasonable algorithm should continue to search for some $[x_3^{cv}]_3 \notin [x_1^{cv}]$ having the property that $d([x_2], [x_3^{cv}]) = n$. If such an $[x_3^{cv}]_3$ is discovered then the algorithm should report that the desired correction is ambiguous, before terminating correctly.

If none of these events occurs then the reasonable algorithm has discovered exactly one class of data memory states $[x_1^{cv}]$ that remain connected in x_2, having the property that $d([x_1^c]^v, [x_2]) = n$. By the corollary to Theorem 3.3, no other $[x_3^{cv}]$ exists which lies closer to $[x_2]$. However, it is possible that x_2 may be disconnected with respect to some x_3^c, and $d([x_2], [x_3^{cv}]) = n$. Thus it is necessary to determine if any such $[x_3^{cv}]$ exists. If such an x_3^{cv} may exist, then the reasonable algorithm should report that it cannot determine whether x_2 has become disconnected with respect to the initial correct and valid instance. Otherwise, the reasonable correction algorithm should perform correction by converting $[x_2]$ to $[x_1^{cv}]$.

Although one might envision reasonable global selective correction algorithms displaying the necessary intelligence to identify when it was possible that some invisible fragment of $[x_3^{cv}]$ has become disconnected in $[x_2]$, it is more efficient to provide these algorithms with some description of the set of $[x_i]$ which lie exactly n changes from two correct and seemingly valid instance states, exactly one of which has become disconnected in x_i. Typically, either no such $[x_i]$ exist, or there are few types of errors that produce such $[x_i]$ and these have been identified when determining the upper bound on the selective correctability of the storage structure X.

Similarly, it is typically far better to provide global selective correction algorithms with some efficient algorithm which allows them to identify when some $[x_1^{cv}]$ remains connected in x_2, and has the property that $d([x_1^c]^v, [x_2]) \leq n$ than to require that these algorithms consider the set of all possible sets of at most n errors that may have been introduced into the instance state x_2.

Example 0.6

Consider a mod(2) double-linked list. This structure [44] has two consecutive header nodes, a forward pointer in each node, a back pointer in each node that points back two nodes, a count in one of the header nodes, and a node identifier in each node. Since at least 7 changes are needed to reorder nodes within this instance, at least 5 changes are needed to replace nodes within the instance with nodes outside the instance, and at least 4 changes are needed to delete nodes from or add nodes to the instance, the instance is exactly 3-detectable. By Theorem 3.2, it is therefore at least 2-selective-correctable, and by closer examination exactly 2-selective-correctable.

We will assume that at most two errors occur in an instance state, x_2, and selectively correct these errors by using a number of different correction routines. Each routine will maintain a small table of nodes addressed by pointers in x_2 which contain node identifiers of x_2, but which currently are not known to belong to the instance being corrected. Having accomplished correction, each node remaining in this table will be considered to contain an error, since it contains an invalid node identifier. Each correction routine will undo any changes applied and report failure if more than two errors are observed by it.

We will first attempt correction using any of the mod(2) local correction algorithms described in [44]. If local correction succeeds then we must ensure that we have not applied two changes to the data memory state $[x_2]_1$ converting it into a member of $[x_1^{cv}]$ when $[x_2]_1$ also lies two changes from some member of $[x_3^{cv}]$, satisfying $x_1^c \neq x_3^c$. This only occurs if x_3^c is damaged so that a back pointer in some node N_{-1} addresses N_2 rather than N_1 and the forward pointer in N_2 addresses N_0 instead of N_1, as shown in Figure 3.3. Correction is therefore aborted if (as a result of performing local correction) we correct a back pointer which appears to point back one node, and reduce the count by 1. Otherwise, since local correction is successful, the correction algorithm terminates normally.

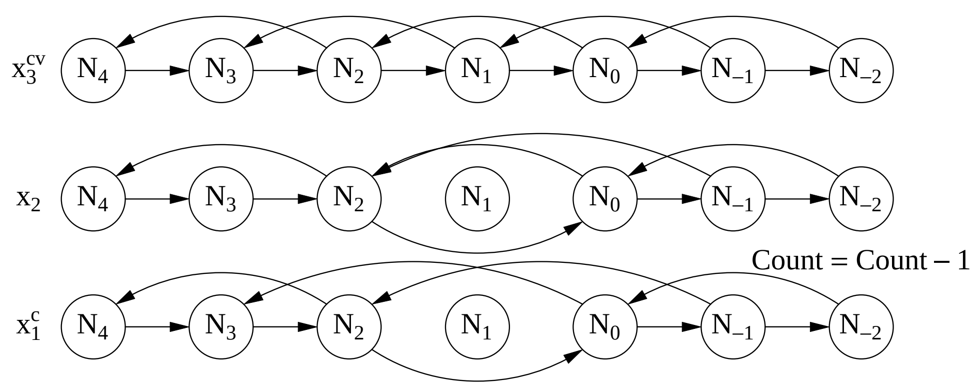


Figure 0.3. The instance x_2 lies two changes from both x_1^c and x_3^c

Conversely, if local correction fails then the locality constraint must have been violated, implying that the two errors occur in pointers within a single locality of the linked list which correctly would appear as shown in Figure 3.4.

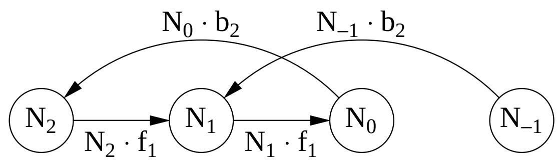


Figure 0.4. A mod(2) locality

We next attempt to perform determining-set correction [22] by assuming that the forward pointers are correct, and if this fails, attempt determining-set correction by assuming that the back pointers are correct, as shown in Figure 3.5. Since the count of the number of nodes in the instance must be correct if two pointers are in error, the number of pointers to be followed during these and subsequent correction attempts is known. If both of these correction attempts fail then we know that the two errors occur in a forward pointer and a back pointer. Thus, the errors occur in one of $\{N_{-1} \cdot b_2, N_1 \cdot f_1\}$, $\{N_0 \cdot b_2, N_2 \cdot f_1\}$, $\{N_0 \cdot b_2, N_1 \cdot f_1\}$, or $\{N_{-1} \cdot b_2, N_2 \cdot f_1\}$.

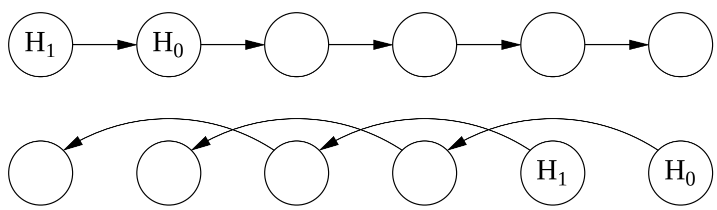


Figure 0.5. First method of constructing determining sets

We next assume that one of two other rather strange determining sets contains no errors and attempt correction using each of these determining sets. These two determining sets consist of the set of back pointers forming a linked list from one of the header nodes together with the forward pointers in the nodes that these back pointers address; and the same determining set constructed using the other header node as shown in Figure 3.6.

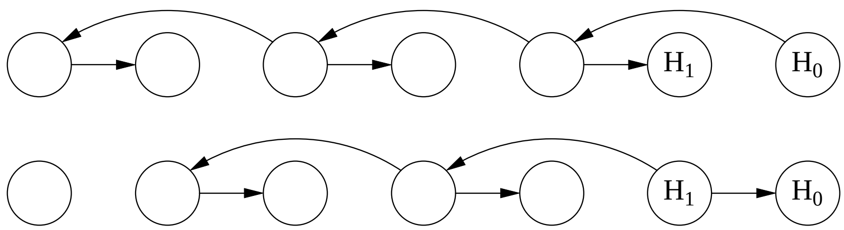


Figure 0.6. Second method of constructing determining sets

If this attempt to perform correction fails, then either $\{N_0 \cdot b_2, N_1 \cdot f_1\}$ is the pair of pointers in error or $\{N_{-1} \cdot b_2, N_2 \cdot f_1\}$ is the pair of pointers in error and the instance is disconnected. In the former case we can construct three determining sets, one of which contains no errors, and thus once again correct the two errors in the instance being corrected. The three determining sets are constructed by traversing the list from a header node by following a back pointer then a forward pointer and then a back pointer, and repeating these three steps. Each determining set is constructed by beginning at a different step in this sequence as show in Figure 3.7.

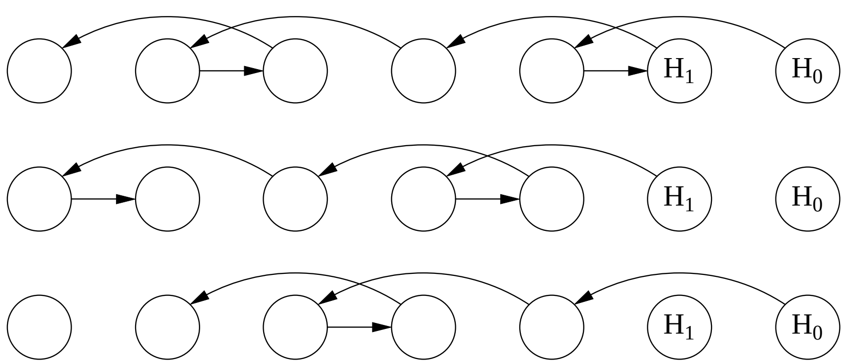


Figure 0.7. Third method of constructing determining sets

If any of the above determining-set correction routines detect at most two errors then these errors are corrected, prior to the selective correction algorithm terminating normally. Otherwise, the correction algorithm reports that the instance being corrected is either disconnected, or contains more than two errors.

The above correction procedure may appear rather cumbersome but can be implemented simply and efficiently. Since local correction and determining-set correction routines operate in linear time, the overall 2-selective correction algorithm also operates in linear time.

Chapter IV

Local correction

17. Introduction

In the previous chapter we have discussed how the introduction of a small bounded number of errors into a correct and valid instance might be corrected by a global correction routine, and have shown that a larger (but still bounded) number of errors can be tolerated in some storage structures if we relax the constraints under which a global correction algorithm operates. By requiring that global correction algorithms correctly diagnose the nature of the errors encountered, rather than necessarily correct all errors encountered, we can often construct algorithms that correct a very much larger class of errors than historical correction algorithms, and which, in addition, correctly report failure when some types of uncorrectable errors are encountered.

Unfortunately, the number of errors that can be tolerated by these correction algorithms typically remains small. This is because a small number of well chosen errors can mislead global correction algorithms, simply because the detectability of most robust storage structures is small [124]. However, if we assume that erroneous components are distributed fairly evenly throughout the instance being corrected, a large number of errors can potentially be corrected. It is this assumption which is exploited by a local correction procedure [44[43, 42, 130]127].

Informally, a local correction procedure visits all of the components of a storage structure instance state, x_1, in some deterministic order, by following pointers from the headers of the instance, and corrects errors when these are first encountered. A component becomes *trusted* once it has been ensured that the component is correct. Errors are identified and corrected by examining previously trusted components, and at most some constant number of potentially erroneous *untrusted* components. This bounded set of untrusted components forms a *locality* which is assumed to contain at most some constant number of errors.

After introducing some notation and terminology, this chapter reviews, revises and expands upon the theory of local correction. It is shown that votes cannot always be used to develop optimal local correction algorithms and that therefore the previous theories pertaining to local correction, which relied on votes, are necessarily incomplete.

The concept of local connection is therefore developed and new results presented which establish lower bounds on the local correctability of an arbitrary storage structure, given that the local connectivity and local detectability of this storage structure are known.

It is then suggested that local-correction algorithms might be able to tolerate more errors in a locality, than allowed for by previous theories, if these local-correction algorithms were designed so that they distinguished between correctable and uncorrectable sets of errors. After establishing bounds on the maximum number of errors in a locality that can be tolerated by an algorithm which wishes to perform local correction whenever possible, we present in Chapter 5 and Chapter 6 two algorithms which use these new bounds to perform selective local correction on various multi-linked list structures.

18. Local detection

A precise characterization of local detection [24] requires that a storage structure have associated with it a *local-linearisation* function, f, which, when presented with the data memory state, $[x_1]_1$, and the addresses, H_X, of the header nodes of a possibly ill-defined instance state, x_1, returns a sequence of possibly duplicated ordered pairs, (w_i, v_i). Each w_i describes the node address, offset, and thus word location, of a one-word component in $[x_1]_1$, and each v_i the value of the word representing that component in $[x_1]_1$. In addition, f returns a boolean flag indicating if the instance state, x_1, appears correct.

For brevity, since the original notation can easily be recovered, we will abbreviate $\mathrm{f}([\mathrm{x}_1]_1, \mathrm{H}_\mathrm{X})$ to $\mathrm{f}(\mathrm{x}_1)$. Except when relevant, we will also, for convenience, ignore the boolean result returned by f. This allows us to use the notation $\mathrm{f}(\mathrm{x}_1)$ to denote the sequence of ordered pairs, $(\mathrm{w}_\mathrm{i}, \mathrm{v}_\mathrm{i})$, produced by f.

Denote the number of tuples in $\mathrm{f}(\mathrm{x}_1)$ by $|\mathrm{f}(\mathrm{x}_1)|$, and the initial subsequence of $\mathrm{f}(\mathrm{x}_1)$ containing exactly k tuples by $\mathrm{f}(\mathrm{x}_1)_\mathrm{k}$. If $\mathrm{k} \leq 0$ then $\mathrm{f}(\mathrm{x}_1)_\mathrm{k} = \emptyset$, while if $\mathrm{k} > |\mathrm{f}(\mathrm{x}_1)|$ then $\mathrm{f}(\mathrm{x}_1)_\mathrm{k} = \mathrm{f}(\mathrm{x}_1)$.

By extension, we will use $[\mathrm{f}(\mathrm{x}_1)_\mathrm{k}]$ to denote the set of all data memory states in which the words described in $\mathrm{f}(\mathrm{x}_1)_\mathrm{k}$ contain the values indicated in $\mathrm{f}(\mathrm{x}_1)_\mathrm{k}$. Thus, for example, $\mathrm{d}([\mathrm{f}(\mathrm{x}_1)_\mathrm{k}], [\mathrm{f}(\mathrm{x}_1)]) = 0$, for all k.

An *r-local-linearisation function*, f, when presented with a data memory state, $[x_1]_1$, and the addresses, H_X, of the header nodes of x_1 returns a *linearisation*, $f(x_1)$, satisfying

1. Completeness: $\forall\ x_1^c,\ [f(x_1^c)] = [x_1^c]$. Thus, the linearisation of any correct instance state, x_1^c, contains exactly those tuples which represent components occurring in x_1^c.

2. Determinism: $\forall\ x_1\ \forall x_2$ satisfying $f(x_1)_{k-1} = f(x_2)_{k-1}$, either $|f(x_1)| = |f(x_2)| = k-1$, or the *k'th* ordered pair in both $f(x_1)$ and $f(x_2)$ describes the same component. However, the value of this component may differ in the two linearisations.

3. Locality constraint: $\exists\ k_f$, such that $\forall\ x_1\ \forall\ x_2^c$, either $d([f(x_1)_{|f(x_1)|-k_f}], [x_2^c]^v) = 0$, or $d([f(x_1)], [x_2^c]^v) > r$. Thus, in any linearisation containing at most r errors, all errors occur in the last k_f tuples of the linearisation.

4. Detection: If $f(x_1)$ contains no erroneous components, then the boolean result returned by f indicates that $f(x_1)$ appears to contain no errors. Conversely, if $f(x_1)$ contains between 1 and r erroneous components, then the boolean result returned by f indicates that $f(x_1)$ contains errors.

A storage structure is r-local-detectable if it has an r-local-linearisation function. We prove, in Lemma 4.2, that the definitions of completeness and locality constraint presented here are equivalent to the apparently stronger definitions presented in [24].

Lemma 0.1

If X has an r-local-linearisation function then X is at least r-detectable.

Proof

Assume, that X is not r-detectable. Then there exists $x_1^c,\ x_2^c \neq x_1^c$, and data memory states, satisfying $1 \leq d([x_1^c]_1, [x_2^c]_1) \leq r$. If $[x_1^c]_1$ is the original correct data memory state then f reports that $f(x_1^c)$ appears to contain no errors. Now suppose that $[x_2^c]_1$ is the original correct data memory state and that this is transformed into $[x_1^c]_1$ by the introduction of between 1 and r errors. Then, by detection, f reports that $f(x_1^c)$ contains errors, contradiction. Therefore X is r-detectable.

Although it was implied in earlier work that an r-local-linearisation function could detect up to r errors in the linearisations that it produced, this was not stated, and does not follow from completeness, determinism and the locality constraint. For example, consider a 1-local-detectable standard double-linked list [24]. Change the structure by allowing the count to arbitrarily contain one of two distinct values. If the count continues to be placed only at the end of a linearisation, then an undetectable error can be placed in the count, without violating the locality constraint. Thus the structure remains 1-local-detectable according to earlier definitions, but is now 0-detectable.

If f is an r-local-linearisation function, but not an $(r+1)$-local-linearisation function, and $f(x_1)$ contains more than r errors with respect to some original correct and valid data memory state, then we will say that the locality constraint has been violated. If we assume that the locality constraint is not violated, then all ordered pairs in $f(x_1)_{|f(x_1)|-k_f}$ are correct. All such ordered pairs will be considered *trusted*, while other ordered pairs appearing in $f(x_1)$ will be considered *untrusted*. The set of untrusted ordered pairs in $f(x_1)$ will be denoted $U_f(f(x_1))$.

Given the assumption that $f(x_1)$ contains between 1 and r errors, it does not follow that a component described in an untrusted tuple of $f(x_1)$ is necessarily incorrect, or even that such a component is potentially incorrect. Because the ordered pairs in $f(x_1)$ may be duplicated, trusted ordered pairs may also occur in the untrusted set of ordered pairs. Less obviously, if for some ordered pair, $(w_i, v_i) \in f(x_1)$, all $[x_2^c]_1^v$ satisfying $d([x_2^c]_1^v, [f(x_1)]) \leq r$, also contain the value v_i in w_i, then v_i is necessarily the correct value for w_1. This is because we are assuming that the original data memory state contains only correct instances and is valid, and that, regardless of the total number of errors introduced into this original data memory, $f(x_1)$ contains at most r erroneous components.

Equivalently, given the assumption that $f(x_1)$ contains between 1 and r errors, the last ordered pair in $f(x_1)_i$ may describe an erroneous component if and only if there exists some x_2^c satisfying $0 \leq d([f(x_1)_{i-1}], [x_2^c]^v) < d([f(x_1)_i], [x_2^c]^v) \leq d([f(x_1)], [x_2^c]^v) \leq r$.

Lemma 0.2

Given an r-local-linearisation function, f, an r-local-linearisation function, g, can be constructed which, for any x_1 and x_2^c, produces linearisations satisfying either $d([g(x_1)_{|g(x_1)|-k_f}], [x_2^c]^v) = 0$ or $d([g(x_1)_j], [x_2^c]^v) = 0$, $d([g(x_1)_{j+1}], [x_2^c]^v) = 1$ and $d([g(x_1)_{j+k_f}, [x_2^c]^v) > r$, for some j.

That is, informally, given an r-local-linearisation function, f, an r-local-linearisation function, g, can be constructed which produces linearisations either satisfying the locality constraint, or containing some subsequence of at most k_f components containing more than r errors, beginning with the earliest erroneous component.

Proof

Assume that the function f operates on $[x_1]_1$. Then the function g simulates f, but excludes from the linearisation $f(x_1)$ produced by f, any tuple previously emitted by f.

The linearisation $g(x_1)$ describes exactly the components in $f(x_1)$. Therefore, since f satisfies the completeness and detection properties so does g. The behaviour of g also satisfies the determinism property. If $f(x_1)$ contains more than r erroneous components, then so does $g(x_1)$. So assume that $f(x_1)$ contains at most r erroneous components. Then, by the locality constraint, these erroneous components first occur in the last k_f tuples of $f(x_1)$. Thus, by construction, these erroneous components first occur in the last k_f tuples of $g(x_1)$. Therefore, g is an r-local-linearisation function, satisfying the locality constraint for $k_g = k_f$.

Now assume that $g(x_1)$ violates the conclusions of this lemma. Then not all errors in $g(x_1)$ can occur in the last k_f tuples of $g(x_1)$, and therefore, since g is a linearisation function, $g(x_1)$ contains more than r errors. Let the earliest error in $g(x_1)$ occur in the last tuple of $g(x_1)_{i+1}$. Then, by assumption, $g(x_1)_{i+k_f}$ contains at most r errors and $|g(x_1)| > i + k_f$.

Remove all errors occurring in $[x_1]_1$ which do not occur in $g(x_1)_{i+k_f}$, giving $[x_3]_1$, and then produce the linearisation $g(x_3)$. By determinism, $g(x_3)_{i+k_f} = g(x_1)_{i+k_f}$ and $|g(x_3)| > i + k_f$. Therefore, since the earliest error in $g(x_3)$ occurs in $g(x_3)_{i+1}$, this error occurs in a trusted component of $g(x_3)$, even though $g(x_3)$ contains at most r errors. This implies that g does not satisfy the locality constraint, and is therefore not an r-local-linearisation function, contradiction.

Thus, the first error in any linearisation produced by g must lie in a sequence of at most k_f components, either occurring at the end of the linearisation, or containing more than r errors.

19. Votes

When attempting to develop a locally detectable storage structure, three inter-related issues must be addressed. Firstly, the rules that define the storage structure must be established.

Secondly, an appropriate local-linearisation function must be selected. Finally, this function must be shown to be an r-local-linearisation function, for some r which, ideally, is maximal.

Although the above activities are in practice inter-related, the task of selecting a promising local-linearisation function, for any given storage structure, is typically not difficult. However, the task of demonstrating that the function is an r-local-linearisation function, and that r is maximal for this storage structure, is certainly not trivial.

In [24] it was proposed that *votes* be used to demonstrate that a storage structure was r-local-detectable. Let V be a predicate of three arguments, $V(f(x_1), w_i, v)$, where $f(x_1)$ is a linearisation produced by a local-linearisation function, f, $(w_i, v_i) \in f(x_1)$, and v is a possible value for w_i. Then $V(f(x_1), w_i, v)$ is a *vote* on w_i if, assuming that there are no erroneous trusted components,

1. V does not examine w_i.

2. $V(f(x_1), w_i, v_i)$ *true* implies zero or multiple errors in w_i and untrusted components of $f(x_1)$ examined by V.

3. $V(f(x_1), w_i, v_i)$ *false* implies one or more errors in w_i and untrusted components of $f(x_1)$ examined by V.

The applications of two votes, $V_1(f(x_1), w_i, v)$ and $V_2(f(x_1), w_i, v)$, were considered distinct, if either the untrusted components used by the two votes were disjoint, or at least one of the two votes evaluated *false*. Two votes, V_1 and V_2, were considered distinct if, for all linearisations, $f(x_1)$, all components, w_i, and all test values, v, the applications of the votes were distinct.

An *r-detectable substructure instance* with principal component w_i was defined to consist of the components evaluated by r distinct votes, when these votes were applied to the principal component w_i. The *target* components associated with w_i were defined to be those untrusted components whose correct value could be determined using only the correct value of w_i, and possibly other trusted components in $f(x_1)$.

It was then shown [24] that a storage structure was r-local-detectable, if corresponding to every correct instance of the storage structure there was a sequence of r-detectable substructure instances satisfying:

1. The targets of the substructure instances partition the instance, the size of all such targets being bounded by a constant,

2. The trusted components in each substructure instance appear in targets of preceding substructure instances, and

3. All other components of each substructure instance appear in targets no later than the j'th succeeding instance in the sequence, for some constant j.

In [128] it was observed that the above holds even if it is only required that votes be distinct in correct substructure instances. This is because we can mechanically test each substructure instance to ensure that the application of all votes within this substructure instance is distinct. If it is determined that this is the case then we are satisfying the earlier constraint on votes, and the above statements therefore holds. Otherwise, since votes within the substructure instance are not distinct, we can conclude that this substructure instance contains error(s), and therefore terminate the linearisation in the vicinity of the first error encountered.

Both of the above results identified sufficient conditions to ensure that a storage structure was at least r-local-detectable, but neither provided any evidence to indicate whether these conditions were necessary, as well as sufficient. This issue is resolved below.

Counterexample 0.1

It is not always possible to use r distinct votes to detect between 1 and r errors in any instance state of an r-local-detectable storage structure.

Proof

Consider a standard linked binary tree, X, in which each node contains a key, a left pointer, and a right pointer. In our robust implementation of this tree we will ensure that all components are of the same size, and add two additional checksum components, producing nodes which contain exactly five components. For simplicity, assume that components are represented by 2 * b bits.

Partition the five components in each node of x_1 into b disjoint code words of ten bits, by selecting two bits from each of the five components, and consider one such code word. Treat each pair of bits from a common component as a binary value ranging between zero and three, and use an arbitrary perfect Hamming code [6791] over a Galois field of four elements to checksum the three (two-bit) data values, by using the remaining two (two-bit) checksum values. One such code is presented in Appendix A2.

Then, errors in one or two of these five (two-bit) values are always detectable, while carefully selected errors in any three of these (two-bit) values transform a correct (ten-bit) code word into a different correct (ten-bit) code word [91].

Since errors in one or two components within a node are always detectable, but errors in three components within a node are not necessarily detectable, the structure is exactly 2-detectable. Since we can arrange that our local-linearisation function, f, visits nodes of this instance in some deterministic order, and emits all components within a node when that node is first visited, it follows that the structure is exactly 2-local-detectable.

Now assume that x_1 contains errors which affect only one code word in some node, N_0, and further assume that in this erroneous code word, containing five (two-bit) values, all sets of errors in at most three (two-bit) values are undetectable, if suitable errors are also placed in the other two (two-bit) values. Such an x_1 can be constructed merely by ensuring that keys are sufficiently variable, and that the data memory state is capable of containing some modest collection of nodes.

Suppose that some vote $V_1(f(x_1), w_i, v)$ uses less than three components in N_0 to verify the assertion that v is the correct value for the first principal component, w_i, in N_0. Then, V_1 can examine at most two of the five (two-bit) values in the erroneous code word. Any of the remaining three (two-bit) values can independently be assigned an arbitrary value while being contained in a correct code word, provided that we also change, if necessary, the other (two-bit) values not examined by V_1. In particular, the (two-bit) value contained in w_i is not examined by V_i, and therefore has a correct value which cannot be determined, even if the values examined by V_1 are correct. This implies that V_1 cannot detect some single errors in w_i, contradiction.

Thus any vote $V_1(f(x_1), w_i, v)$ must examine at least three components. Therefore any pair of distinct votes V_1 and V_2 on w_i must examine at least six components. But, by construction, components not occurring in N_0 cannot assist any vote on w_i, and, by definition, neither can w_i. Thus, there are four components that can usefully be examined by votes on w_i, and therefore there is only one distinct vote on the principal component w_i.

It is perhaps unfortunate that r-local-linearisation functions cannot always be developed that employ the existing theory (or theories) of voting. As indicated in [24] the use of votes provides a simple constructive method of establishing lower bounds on the local detectability and local correctability of an arbitrary storage structure. In addition the use of votes assists in the development of clear, concise, local detection and correction procedures.

However, it is important to recognize that not all storage structures can use votes to arrive at a maximal r-local-linearisation function, and that therefore historical methods of attempting to establish the local detectability and local correctability of a storage structure may not always be appropriate. We therefore present some new definitions, and then proceed to establish stronger relationships between the local detectability, local connectedness, and local correctability of an arbitrary storage structure than presented elsewhere.

20. Local connection

Let the function, Q_f, when presented with a linearisation, $f(x_1)$, and the addresses, H_X, of the header nodes of x_1, return a set of ordered pairs, (w_i, v_i), where w_i describes the location of a one-word component in $[x_1]_1$, and v_i a possible value for this component. We can implicitly assume that Q_f is presented with the appropriate header addresses, and will therefore use $Q_f(f(x_1))$ to denote the set of ordered pairs produced when Q_f is presented with $f(x_1)$.

An r-local-linearisation function, f, is c-local-connected if there exists a *c-connection function*, Q_f, and associated constant, z_f, such that $\forall x_1 \forall x_2^c$ satisfying $1 \leq d([f(x_1)], [x_2^c]^v) \leq c \leq r$:

A. Bound constraint: $1 \leq |Q_f(f(x_1))| \leq z_f$.

B. Connection constraint: $\exists (w_i, v_i) \in f(x_1)$ and $(w_i, v_i') \in Q_f(f(x_1)) \cap f(x_2^c)$ such that $v_i' \neq v_i$.

Thus, informally, an r-local-linearisation function is c-local-connected if there exists a function Q_f, which, when presented with any linearisation $f(x_1)$ containing between 1 and c errors, identifies a bounded set of possible values for specific components in $[x_1]_1$, at least one of which is the correct value for an erroneous component in $f(x_1)$. The function, Q_f, may return a number of values for a single component in $[x_1]_1$, not all of which are necessarily distinct, or necessarily differ from the current value of this component.

The (possibly unknown) erroneous components in any linearisation, $f(x_1)$, containing between 1 and c errors, whose location and correct value are recorded in a tuple produced by a connection function, Q_f, are called *principal components* of $Q_f(f(x_1))$, and will be denoted p_i. Clearly, $p_i \in U_f(f(x_1))$. It is stressed that this definition of principal component supersedes the definition of principal component presented in [24], and used earlier in this chapter.

A storage structure is c-local-connected if it has a c-local-connected linearisation function.

Lemma 0.3

If an instance, X, has a c-local-connected linearisation function, f, then X has a c-local-connected linearisation function, g, having a connection function, Q_g, which when presented with a linearisation containing between 1 and c erroneous components, includes among its principal components the earliest erroneous component in the linearisation.

Proof

Without loss of generality, assume that f produces linearisations containing no duplicated components. We will construct a sequence of components, $g(x_1)$, using a function, g, that is derived from f, and then show that g satisfies the conditions of this lemma.

The linearisation $g(x_1)$ contains as its initial subsequence the linearisation $f(x_1)$. If f reports that $f(x_1)$ appears to contain no errors, then g reports that $g(x_1)$ appears to contain no errors and terminates. Otherwise, g reports the detection of errors. This ensures that g satisfies the completeness and detection properties of a local-linearisation function.

So, suppose that f reports that $f(x_1)$ contains errors. Then for each $(w_{i_2}, v_{i_2}) \in Q_f(f(x_1))$ for which there exists a $(w_{i_2}, v_{i_2}') \in U_f(f(x_1))$ satisfying $v_{i_2} \neq v_{i_2}'$, g independently changes the value of w_{i_2} to v_{i_2}, producing a new instance x_{1,i_2} and then uses f to produce a new linearisation $f(x_{1,i_2})$. For each $f(x_{1,i_2})$ thus produced, g identifies the set of tuples in $f(x_{1,i_2})$ describing components not already in $g(x_1)$ which occur within k_f tuples of (w_{i_2}, v_{i_2}), and appends these tuples to $g(x_1)$, in the sequence that they appear in $f(x_{1,i_2})$. The above process is repeated recursively, producing all $f(x_{1,i_2 \cdots i_{j+1}})$ from those $f(x_{1,i_2 \cdots i_j})$ for which $0 < |Q_f(f(x_{1,i_2 \cdots i_j}))| \leq z_f$, while $j < c$.

Upon completion of this process, $g(x_1)$ contains a very large but bounded set of untrusted tuples, $U_g(g(x_1))$, beginning with those untrusted tuples in $f(x_1)$. In $g(x_1)$ no tuples contain modified values, since any component modified by g has already been added to $g(x_1)$ and $g(x_1)$ contains no duplicated components. If $g(x_1)$ contains at most r errors, then so does $f(x_1)$, implying that all errors in $g(x_1)$ occur in untrusted components. Thus g satisfies the locality constraint for some very large k_g. Finally, g uses only the values of components already in $g(x_1)$ when appending new components to $g(x_1)$, and is deterministic. Thus g is an r-local-linearisation function.

Associate with the local-linearisation function g the connection function $Q_g(g(x_1)) = \bigcup_{|Q_f(f(x_{1,i_2 \cdots i_j}))| \leq z_f} Q_f(f(x_{1,i_2 \cdots i_j}))$. Then $Q_g(g(x_1))$ contains a large but bounded set of elements. Now assume that $g(x_1)$ contains between 1 and $c \leq r$ errors. Then we wish to prove that $(p_0, v_0) \in Q_g(g(x_1))$, where v_0 is the correct value for the earliest erroneous component, p_0, in $f(x_1)$ and thus $g(x_1)$.

Since $g(x_1)$ contains $c \leq r$ errors, $f(x_1)$ also contains at most c errors, implying that there exists some $(p_1, v_1) \in Q_f(f(x_1)) \subset Q_g(g(x_1))$ such that p_1 is the earliest principal component of $Q_f(f(x_1))$, and v_1 is its correct value. If $p_1 = p_0$ then the proof is complete. So assume that $p_1 \neq p_0$.

Then g produces some $f(x_{1,i_2})$ in which the error in p_1 has been corrected, but the earlier error in p_0 has not. If $f(x_{1,i_2})$ contained more than $c - 1$ errors, then, by the locality constraint and Lemma 4.2, at least c of these errors would occur within k_f components of p_0. But, by construction, each such error therefore occurs in $g(x_1)$, as does the error in p_1, implying that $g(x_1)$ contains more than c errors, contradiction. Thus $f(x_{1,i_2})$ contains between 1 and $c - 1$ erroneous components.

By iteratively repeating the reasoning applied to $f(x_{1,i_2 \cdots i_j})$ to $f(x_{1,i_2 \cdots i_{j+1}})$ we deduce that g produces some linearisation $f(x_{1,i_2 \cdots i_{j+1}})$ containing at most $c - j$ errors. If any such linearisation containing $c - j$ errors included p_0 as a principal component then so would $Q_g(g(x_1))$. So assume otherwise. Then g produces some linearisation containing only the error in p_0, since g produces linearisations while $j < c$. But p_0 is therefore a principal component of this linearisation, contradiction. Thus p_0 is a principal component of $Q_g(g(x_1))$.

21. Local correction

A local-linearisation function, f, is *c-local-correctable* [24] if:

Correction constraint: There exists a c-connection function, P_f, which, when presented with a linearisation produced by f, emits at most one tuple. Such a c-connection function will also be called a c-*correction function*.

A storage structure is c-local-correctable if it has a c-local-correctable linearisation function.

Since there are a finite number of data memory states, P_f is computable, and therefore our definition of local correctability is equivalent to the apparently stronger definition of local correctability given in [24], which considered a storage structure to be c-local-correctable if there existed an r-local-linearisation function, f, and an improvement procedure, P, which could correct at least one error in any linearisation, $f(x_1)$, containing between 1 and $c \leq r$ errors.

Lemma 0.4

If a storage structure has an r-local-linearisation function, f, and associated c-correction function, P_f, then the storage structure has an r-local-linearisation function, g, and associated c-correction function, P_g, such that whenever $g(x_1)$ contains between 1 and c errors, $P_g(g(x_1))$ is the earliest erroneous component in $g(x_1)$.

Proof

By definition, a c-correction function, P_f, is also a c-connection function. Therefore, using the construction described in Lemma 4.3, we can produce a c-connection function, Q_g, whose principal components include the earliest erroneous component in $g(x_1)$. If $g(x_1)$ contains between 1 and c errors, then at each step of this construction, P_f, being a correction function, identifies exactly one principal component of Q_g, and therefore all tuples in $Q_g(g(x_1))$ describe principal components of $Q_g(g(x_1))$. So let $P_g(g(x_1)) = (p_0, v_0)$ where p_0 is the earliest component in $g(x_1)$ for which $(p_0, v_0) \in Q_g(g(x_1))$. Then P_g satisfies the conditions of this corollary.

Theorem 0.1

If a storage structure has a 2r-local-linearisation function, f, that is also r-local-connected, then the storage structure is at least r-local-correctable.

Proof

Let the function h behave exactly like the function g, described in Lemma 4.3, but internally generate all appropriate linearisations $f(x_{1,i_2..i_j})$ while $j \le r + 1$. Then we wish to show that h has an associated r-local-correction function, P_h. So assume that h returns a linearisation, $h(x_1)$, which contains between 1 and r errors.

Using the arguments given in Lemma 4.3, h internally generates some linearisation, $f(x_{1,i_2 \cdots i_j})$, in which all errors occurring in the k_f components beginning with the earliest erroneous component, p_0 in $f(x_1)$, have been removed. In the linearisation $f(x_{1,i_2 \cdots i_j})$ therefore, either p_0 must be trusted, or f must have signalled that $f(x_{1,i_2 \cdots i_j})$ apparently contains no errors.

Conversely, consider the linearisation, $f(x_{1,i_2 \cdots i_k})$, produced by h when h fails to correct the earliest erroneous component, p_0. By determinism, the linearisation, $f(x_{1,i_2 \cdots i_k})$, contains p_0, and, since p_0 remains in error, therefore contains at least one error. Suppose that $f(x_{1,i_2 \cdots i_k})$ contains more than 2r errors. Then, since f emits no duplicated components, by the proof of Lemma 4.2, more than 2r errors occur in components lying not more than k_f components ahead of p_0 within $f(x_{1,i_2 \cdots i_k})$. Since the function, h, introduces at most r errors into any linearisation that it produces, by construction, more than r errors in $f(x_{1,i_2 \cdots i_k})$ also exist in $h(x_1)$. But $h(x_1)$ contains at most r errors, contradiction. Thus, $f(x_{1,i_2 \cdots i_k})$ contains between 1 and 2r errors. By the locality constraint and the detection properties, f therefore reports that $f(x_{1,i_2 \cdots i_k})$ contains errors, and places p_0 and all subsequent components in $f(x_{1,i_2 \cdots i_k})$ in $U_f(f(x_{1,i_2 \cdots i_k}))$.

Thus, if the function h produces an erroneous linearisation $h(x_1)$ containing at most r errors, then the function $P_h(h(x_1)) = (p_0, v_0)$ mimics the functions f and h to determine the earliest erroneous component p_0 in $h(x_1)$ and its correct value v_0. Specifically, p_0 is the first component that becomes trusted, or is reported to be correct, in some $f(x_{1,i_2 \cdots i_j})$ once h has modified it, and v_0 is its modified and now correct value. Since the function $P_h(h(x_1))$ exists, the structures on which f operate are r-local-correctable.

Example 0.1

Consider a circular regular multi-linked list having $f > 0$ forward pointers in each node which address the next node, $0 < b < f$ back pointers in each node which address the previous node, and a single header. Any number of consecutive data nodes in such a structure can be deleted by applying $f + b$ changes. Thus the local detectability of the structure is at most $f + b - 1$.

Assume that a local correction procedure is traversing this multi-linked list forwards and has arrived at node N_0, and wishes to identify the node, N_1, which follows N_0. Let the current set of untrusted components consist of all forward pointers in N_0, and the back pointers in the nodes addressed by these forward pointers. Then, provided that the Valid State Hypothesis holds, any set of at most $f + b - 1$ errors in this locality is detectable, implying that the structure is exactly $(f + b - 1)$-local-detectable.

Since N_1 is addressed by some forward pointer within the locality, unless all forward pointers in N_0 are damaged, the linearisation is $f - 1 \geq (f + b - 1)/2$ local-connected. Therefore, by Theorem 4.1, the linearisation is at least $\lfloor (f + b - 1)/2 \rfloor = \lceil (f + b)/2 \rceil - 1$ locally correctable.

However, it does not follow that this linearisation is at most $(\lceil (f + b)/2 \rceil - 1)$-local-correctable. Suppose that an arbitrary locality contains at most $n < f$ errors. Then, since f is $(f - 1)$-local-connected, the node, N_1, that follows N_0 must be addressed by some correct pointer in the locality being corrected.

Let the local correction algorithm consider the possibility that each node addressed by a forward pointer in N_0 is N_1. When the local correction algorithm correctly guesses that N_1 follows N_0, it will observe at most n errors in the locality being examined, since the locality contains at most n errors. Now suppose that the local correction algorithm erroneously guesses that some node N_x follows N_0. Then all forward pointers in N_0, all back pointers in N_1, and all back pointers in N_x occur in the locality being examined, and by the Valid State Hypothesis, now appear incorrect unless they contain errors. This is shown in Figure 4.1. Thus the local correction algorithm will conclude that the locality contains at least $2 * b + f - n$ errors.

Now eliminate any possible ambiguity about the node that correctly follows N_0, in a linearisation containing at most n errors, by requiring that n be the largest value satisfying $n < 2 * b + f - n$, or equivalently $n < b + \lceil f/2 \rceil$. Then N_1 can always be uniquely identified. The storage structure is therefore exactly $\min(f, b + \lceil f/2 \rceil) - 1$ locally correctable. Thus when $f \geq 2 * b$ the local correctability of this storage structure will exceed half the local detectability of this storage structure by at least $\lfloor b/2 \rfloor$.

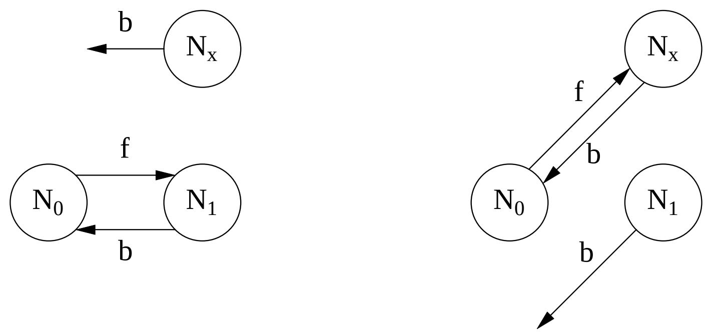


Figure 0.1. Pointer changes needed to replace N_1 with N_x

The above example demonstrates that we can design storage structures whose local correctability exceeds half the detectability of the same storage structure by an arbitrary amount. In Chapter 7 it is shown that a robust AVL tree exists, whose detectability, local detectability, correctability, and local correctability are all equal. Collectively, these examples demonstrate that the Valid State Hypothesis can have a profound effect on the relationship between the local detectability and local correctability of an arbitrary storage structure. Much which was stressed in Chapter 3 should therefore be reiterated here.

Theorem 0.2

If an r-local-linearisation function, f, has a c-local-correction function, P_f, and produces linearisations, $f(x_1)$, which occur as the initial subsequence of linearisations, $g(x_1)$, produced by an r-local-linearisation function, g, then g has the c-local-correction function $P_g(g(x_1)) = P_f(f(x_1))$. Therefore, g is a c-local-correctable linearisation function.

Proof

Suppose that $g(x_1)$ contains between 1 and c errors. Then, since $f(x_1)$ is contained in $g(x_1)$, $f(x_1)$ contains at most c errors. By determinism and completeness, if $f(x_1)$ contains no errors then $x_1 \in x^c$, implying that $g(x_1)$ contains no errors, contradiction. Thus $f(x_1)$ contains between 1 and c errors. Therefore, since f is c-local-correctable, let $P_f(f(x_1)) = (w_1, v_1)$.

Define $A \subseteq [x^c]^v$ so that $[x_2^c]^v \in A$ if and only if $d([f(x_1)], [x_2^c]^v) \leq c$. Then the component w_1 has the value v_1 in all $[x_2^c]^v \in A$, but has value $v_1' \neq v_1$ in both $f(x_1)$ and $g(x_1)$.

Since $g(x_1)$ contains at most c errors, there exists at least one $[x_j^c]^v$ satisfying $d([g(x_1)], [x_j^c]^v) \le c$. Since $d([f(x_1)], [x_j^c]^v) \le d([g(x_1), [x_j^c]^v)$ all such $[x_j^c]^v \in A$. Therefore all such $[x_j^c]^v$ contain w_1 and require that it have the value v_1. Thus $P_g(g(x_1)) = P_f(f(x_1))$ is a c-local-correction function for g. Therefore, g is c-local-correctable.

Earlier in this chapter we have created new linearisation functions by deterministically appending a bounded number of components to erroneous linearisations. Similarly, when designing or enhancing local-correction algorithms, linearisation functions are often modified so that they add a bounded number of additional components to the end of erroneous linearisations. The above theorem is important, since it provides a lower bound on the local correctability of these resulting linearisation functions.

22. Local-correctable linearisations

A correct linearisation, $f(x_1^c)$, is r-local-correctable if $\forall\ x_2^c \ne x_1^c$, $d([f(x_1^c)], [x_2^c]^v) > r$. An erroneous linearisation, $f(x_3)$, satisfying $0 < d([f(x_3), [x^c]^v) \le r$, is r-local-correctable if there exists at least one tuple $(w_i, v_i') \in f(x_3)$ and value $v_i \ne v_i'$, such that $\forall\ x_4^c$ satisfying $d([f(x_3)], [x_4^{cv}]) \le r$, $(w_i, v_i) \in f(x_4^c)$. Linearisations having neither of the above two properties are not r-local-correctable.

Lemma 0.5

A local-linearisation function, f, is r-local-correctable if and only if all $f(x_1)$ produced by f, satisfying $d([f(x_1)], [x^c]^v) \le r$, are r-local-correctable.

Proof

Suppose that f is r-local-correctable. Then, by Lemma 4.1, the instance, X, is r-detectable and all $f(x_1^c)$ therefore r-local-correctable. Further, since f is r-local-correctable, $\exists$ a correction function, P_f, such that for any $f(x_3)$ and $\forall\ x_4^c$ satisfying $1 \le d([f(x_3)], [x_4^c]^v) \le r$, $P_f(f(x_3)) = (w_i, v_i) \in f(x_4^c)$, $(w_i, v_i') \in f(x_3)$ and $v_i \ne v_i'$. Thus, if $d([f(x_3)], [x^c]^v) \le r$, $f(x_3)$ is r-local-correctable. The converse result follows similarly.

23. Local-correctable instance states

An instance state, x_1, residing in the data memory state $[x_1]_1 = [x_{1,0}]_1$, is c-local-correctable, with respect to a correction function, $P_f(f(x_{1,i})) = (p_{1,i}, v_{1,i})$, if there exists a sequence of c-local-correctable linearisations $f(x_{1,0}) \cdots f(x^c_{1,k})$ satisfying $d([f(x_{1,i})], [x^c_{1,k}]^v) \le c$, for $0 \le i \le k$, where each $f(x_{1,i+1})$ is produced from the data memory state used to produce $f(x_{1,i})$, after first changing the value of $p_{1,i}$ to $v_{1,i}$.

Suppose that a storage structure, X, has a local-linearisation function, f, which is c-local-correctable, and a correction function, P_f, which can be implemented. Then we can attempt to correct an erroneous instance state, x_1, occurring in a data memory state, $[x_{1,0}]_1$, by using a c-local-correction procedure, Ψ, which assumes that x_1 is c-local-correctable. Beginning with $[x_{1,0}]_1$, Ψ iteratively produces linearisations, $f(x_{1,i})$, and while $P_f(f(x_{1,i}))$ identifies a possible correction, changes $p_{1,i}$ to $v_{1,i}$, in the data memory state used to derive $f(x_{1,i})$, before producing $f(x_{1,i+1})$.

The procedure Ψ terminates successfully once a linearisation containing no detectable errors is produced, and reports failure if it is determined that some linearisation produced by Ψ contains more than c errors. The instance state, x_1, observed by Ψ comprises exactly those components occurring in some $f(x_{1,i})$.

Lemma 0.6

Given a local-correction procedure, Ψ, it is possible to construct a local-correction procedure, Φ, which detects, in those instance states correctable by Ψ, any violation of the locality constraint, and which further identifies the subset of instance states corrected by Ψ which are seemingly valid with respect to the original incorrect instance.

Proof

Let Φ simulate Ψ. If Ψ reports failure then so does Φ. Otherwise, since Ψ halts, Φ can identify both the x_1 observed by Ψ, and the $x^c_{1,k}$ that was produced by Ψ. Φ can therefore identify all components in x_1 which are either erroneous in $x^c_{1,k}$, or which violate the validity of $x^c_{1,k}$. Therefore Φ can verify that all $f(x_{1,i})$ produced by Ψ satisfy the locality constraint with respect to $x^c_{1,k}$, and that $x^c_{1,k}$ is seemingly valid with respect to x_1.

Counterexample 0.2

Even if a local-correction procedure Ψ, operating on a locally correctable instance state, x_1, produces a correct and seemingly valid instance state, x_2^c, there may exist some alternative correct and seemingly valid instance state, x_3^c, satisfying $d([x_1], [x_3^{cv}]) < d([x_1], [x_2^{cv}])$.

Proof

Consider a regular linked list, x_3^c, containing two forward pointers per node that address the next node, and one back pointer per node that addresses the previous node in the list. Each node also contains a node identifier. Then, as justified earlier, this storage structure is exactly 1-local-correctable.

Assume that another instance of this storage structure, y_4^c, occurs in the set of data memory states, $[x_3^c, y_4^c]^v$. Link into x_3^c any m consecutive nodes in y_4^c producing the set of damaged data memory states $[x_1]$. This involves changing two forward pointers and a back pointer in x_3^c, and doing likewise in y_4^c. Thus $d([x_1], [x_3^c, y_4^c]^v) = 6$.

Now consider the behaviour of Ψ when operating on x_1. It will observe no pointer errors, and will therefore conclude when examining nodes which correctly belong to y_4^c that these nodes contain erroneous node identifiers. Since each such identifier will be changed by Ψ, Ψ will produce an instance state x_2^c satisfying $d([x_1], [x_2^{cv}]) = m$. To produce the desired counterexample, set $m > 6$.

24. Selective local correction

In Chapter 3 we discussed how the theory of selective correction could be applied to global correction, and used as a fundamental concept the notion of a correctable instance state. Having now established the concept of a c-local-correctable linearisation, it seems appropriate to explore how the theory of selective correction might be applied to a locally correctable storage structure.

A storage structure, X, which is exactly c-local-correctable, is exactly s-selective-local-correctable if it has a c-local-linearisation function, f, which produces linearisations which are either not locally correctable, or are at least s-local-correctable, for some s which is maximal. Such linearisation functions will be termed s-selective-local-correctable linearisation functions.

Thus, informally, if X is c-local-correctable and s-selective-local-correctable, then X has a local correction procedure which can both correct one error in all linearisations containing between 1 and c errors, and can correct one error in all locally correctable linearisations containing less than s errors, even when it is assumed that these linearisations may contain up to s errors.

If we did not require that f be a c-local-correctable linearisation function, then it might sometimes be possible to increase the selective-local-correctability of some storage structures by using linearisation functions having higher local-detectability but not being c-local-correctable. While such linearisation functions may be more tolerant of errors, or possibly more successful at correcting errors because they employ smaller localities than alternative c-local-correction functions, it seems intuitively appealing that s-selective-local-correctable linearisation functions, also be c-local-correctable linearisation functions.

Typically, selective-local-correction algorithms will use linearisation functions which have larger localities than linearisation functions used by historical local-correction algorithms, but these selective-local-correction algorithms will be able to correct more errors in these larger localities than historical correction algorithms. Because of these differences, it should be clear that not all errors correctable by a c-local-correction procedure are necessarily correctable by an s-selective-local-correction procedure. Indeed, since different algorithms may employ different local-linearisation functions to assist in performing correction, different s-selective-local-correction algorithms may have very different characteristics when operating on the same erroneous instance state. Obviously, however, our goal is to produce good selective-local-correction procedures which correct a larger percentage of errors than corresponding local-correction procedures.

Lemma 0.7

If an instance, X, is at least c-local-correctable, at most r-detectable, and exactly s-selective-local-correctable then $c \leq s \leq r$.

Proof

Since X is at least c-local-correctable, there exists a local-linearisation function, f, whose linearisations, when containing less than c errors, are c-local-correctable. Therefore, $s \geq c$.

Since X is at most r-detectable, there exists x_1^c and $x_2^c \neq x_1^c$ satisfying $d([x_1^c]^v, [x_2^c]) \leq r + 1$. The linearisation $f(x_1^c)$ is trivially 0-local-correctable, but, by the above, not $(r + 1)$-local-correctable. Therefore $s \leq r$.

Regrettably, we have been unable to prove that the local detectability of a storage structure is at least equal to the selective-local-correctability. Consider, for example, a storage structure which is 0-local-correctable, r-local-detectable and $(r + 1)$-detectable. Then, by the above definitions, correct linearisations are $(r + 1)$-local-correctable. If no incorrect linearisation is locally correctable, the storage structure is therefore $(r + 1)$-selective-correctable. Even if some incorrect linearisations are locally correctable, it may be possible to ensure, by using the Valid State Hypothesis or otherwise, that all such linearisations are $(r + 1)$-local-correctable. However, given the typically large number of such local-correctable linearisations, and the requirement that all be $(r + 1)$-local-correctable, it seems unlikely that such a perverse storage structure will ever be found.

Theorem 0.3

If a storage structure, X, has an r-local-linearisation function, f, and associated c-local-correction function, P_f, then the storage structure has an r-local-linearisation function, h, and associated c-local correction function, P_h, such that all c-local-correctable linearisations produced by h are also $(r - c)$-local-correctable.

Proof

Select some r-local-linearisation function, f, which is c-local-correctable, and without loss of generality assume that f emits no duplicate components. Construct the c-local-correctable r-linearisation function, $h(x_1)$, from the r-local-linearisation function f, by using the correction function, P_f, and the sequence of linearisations $f(x_{1,0}), f(x_{1,1}) \cdots f(x_{1,m})$, where $m \leq c$, as described in the proof of Theorem 4.1. Thus, while $m \leq c$, the construction continues until some $f(x_{1,m})$ is produced in which the earliest modified ordered pair $P_h(h(x_1)) = (p_0, v_0)$ is either trusted, or occurs in the last k_f tuples of a linearisation containing no detectable errors. Denote the initial subsequence of $f(x_{1,m})$ which contains (p_0, v_0) and the k_f or fewer tuples following (p_0, v_0) by $f(x_{1,m})_t$.

Now assume that $h(x_1)$ is c-local-correctable. If $h(x_1^c)$ contains no detectable errors, then $h(x_1^c)$ is trivially $(r-c)$-local-correctable. So assume that $h(x_1)$ contains between 1 and c errors. Then, since $h(x_1)$ contains at most c errors and f is a c-local-correctable linearisation function, the construction of each $f(x_{1,i})$ is well defined, and therefore $f(x_{1,m})_t$ is well defined. Thus all components in $f(x_{1,m})_t$ are contained in $h(x_1)$.

So suppose that $h(x_1)$ is c-correctable but not $(r-c)$-local-correctable. Then there exists some x_2^c satisfying $c < d([h(x_1)], [x_2^{cv}]) \leq r - c$, which either does not contain p_0, or which requires that p_0 have the value $v_0' \neq v_0$. In $f(x_{1,m})$ some ordered pair at or before the ordered pair (p_0, v_0) is therefore erroneous with respect to x_2^c. Since at most $m \leq c$ components have different values in $f(x_{1,m})_t$ and $h(x_1)$ and all components in $f(x_{1,m})_t$ are contained in $h(x_1)$,
$1 \leq d([f(x_{1,m})_t], [x_2^{cv}]) \leq d([f(x_{1,m})_t], [h(x_1)]) + d([h(x_1)], [x_2^{cv}]) \leq m + (r-c) \leq c + (r-c) = r.$

But this implies that, with respect to x_2^c, $f(x_{1,m})_t$ contains at most r errors, at least one of which either occurs in the trusted tuple (p_0, v_0) or occurs in some earlier trusted tuple. Since this violates the locality constraint, f is not an r-local-linearisation function, contradiction. Thus the assumption that $h(x_1)$ is not $(r-c)$-local-correctable is false, and $h(x_1)$ is therefore $(r-c)$-local-correctable.

Corollary 0.2

If a storage structure, X, has an r-local-linearisation function, f, which is exactly c-local-correctable, and $r \geq 2 * c + 1$, then X is at least $(c+1)$-selective-local-correctable.

25. Applications

In Chapter 3 we showed that all instances which were r-global-detectable, were $\lceil r/2 \rceil$-selective-correctable, and that therefore we could safely assume that any instance which we wished to correct contained up to $\lceil r/2 \rceil$ errors, provided that we took some care when examining instances that contained exactly $\lceil r/2 \rceil$ errors.

We have now produced a corresponding but weaker result, showing that if a storage structure has a c-local-correctable r-local-linearisation function, then it has an r-local-linearisation function for which all c-local-correctable instance states remain locally correctable even when linearisations are assumed to contain up to $r - c \geq \lceil r/2 \rceil$ errors.

Superficially, the use of such an r-local-linearisation function in a correction procedure may seem rather dubious. Although c-correctable instance states will continued to be corrected appropriately, provided that no linearisation contains more that $r - c$ errors, such local-linearisation functions are likely to use larger constants to define the size of their localities, and these localities are therefore likely to contain a larger number of errors. Typically, large numbers of errors do not conspire to mislead a correction algorithm, and we would therefore expect such correction procedures to be rather conservative.

However, as with selective global correction, having established that linearisations may contain more than c errors, while continuing to support c-local-correctability, we are free to consider how such linearisations may typically be corrected when assumed to contain at most $r - c$ errors, provided that we take care to detect uncorrectable linearisations. If we are able to develop strategies for correcting the majority of such errors, then we are likely to develop correction procedures which perform at least as well as historical local correction procedures.

Unfortunately, the majority of existing robust storage structures are not locally correctable, and of those which are locally correctable many have $r = 2 * c$ and thus $r - c = c$. Among the storage structures which are not locally detectable are the single-linked list [123], the chained and threaded binary tree [125], the mod(2) chained and threaded binary tree [114], the chained and threaded B-tree [20], the double binary tree [98], and the robust UNIX file structure [111].

One of the earliest robust structures to be presented was the double-linked list [122], and this is 1-selective-local-correctable, since it is 1-local-detectable, even though it is 0-local-correctable. However, erroneous pointers are often not correctable, and therefore any selective local correction algorithm operating on an erroneous instance of this structure will, at best, tend to correct only identifiers and the count.

Out of the small collection of existing locally correctable storage structures, many have the same local-correctability and selective-local-correctability. These structures include the mod(2) linked list [123], the locally correctable B-tree [130], the checksummed binary tree presented in Counterexample 4.1, and the locally correctable AVL Tree presented in Chapter 7.

Fortunately, there are families of robust storage structures which are exactly c-local-correctability, and which have s-selective-local-correction algorithms which can correct almost all linearisations containing at most $s = r - c = c + 1$ errors. Indeed, some of the structures presented in the next two chapters are always $(c + 1)$-local-correctable, unless the instance state being corrected has become disconnected.

Chapter V

Correcting mod(k) linked lists

26. Regular linked lists

Regular linked lists form a very important class of robust storage structures for many reasons. They are easily described, implemented, and analyzed, and for this reason the properties of robust linked lists are, for the most part, well understood. Because regular linked lists are well understood, it is typically easy to find linked lists which satisfy predefined properties, and this makes them ideal candidates for both example, and counterexample.

The organization of pointers within a regular linked list can be described by a vector of pointer distances, since any pointer in any node of a regular linked list correctly points forwards or backwards the same distance as any pointer occurring at the same offset in any other node within the structure. For conciseness and clarity, in this and subsequent chapters, we will therefore describe the organization of the pointers of a regular linked list by means of a vector. A single-linked list therefore has a $(+1)$ pointer structure, while the mod(2) structure presented at the end of chapter 3 has a $(+1, -2)$ pointer structure.

Since we assume that errors affect independent components, the physical ordering of pointers within nodes is irrelevant. For definiteness, we may imagine that pointers occur in nodes in the order that they are defined within the vector describing the pointer structure. Similarly, since we are uninterested in the data contained in an arbitrary regular linked list, reversing the sign of all values in a vector describing the pointer structure produces a regular linked list structure which is essentially unchanged. However, once again for the sake of definiteness, we will consider positive pointer distances to point forward the specified number of nodes, and negative pointer distances to point back the specified number of nodes.

Most of the material presented in this chapter has already been published [44].

27. The mod(k) linked list

A modified(k), or mod(k), linked list storage structure [18[21]114] is a circular double-linked list of nodes, in which each node contains a forward pointer that links it to the next node, and a back pointer that links it to the *k'th* previous node. A particular *instance* of a mod(*k*) structure consists of *k* consecutive *header* nodes, whose addresses are known, and all nodes reachable by following pointers from these header nodes. These header nodes are contained within the double-linked list of nodes, and are the only nodes in the instance when the instance is empty. Each node within an instance contains an *identifier* whose value uniquely identifies the instance to which the node belongs. A *count* of the number of non-header nodes within an instance is stored in one of the header nodes of the instance. An *error* is an incorrect value in a single pointer, identifier, or count component [123].

The mod(1) double-linked list is 2-global detectable, 1-global correctable [123124], but not 1-local-correctable [24]. Since the structure is 1-local-detectable, it is 1-selective-local-correctable. However, this observation is of little practical value, since selective-local-correction algorithms will tend to correct only errors in node identifiers and the count.

The mod(2) regular linked list is 3-global-detectable, 1-global correctable, 2-selective-global correctable, 2-local-detectable, 1-local-correctable, and 1-selective-local-correctable. It can be corrected by using the selective-global-correction algorithm presented in Chapter 3, or by using at least three seemingly different 1-local-correction algorithms [24[44]125]. In [127] it has been shown that local correction algorithms operating on mod($k \geq 2$) structures can also perform crash recovery.

A mod($k \geq 3$) linked list, while remaining exactly 1-local-correctable, is 3-local-detectable, and thus 2-selective-local-correctable.

We now develop, by stages, an algorithm which is proven to perform 2-selective-local-correction on mod($k \geq 3$) linked lists. When operating on mod($k \geq 4$) linked lists, this algorithm either corrects up to two errors in a single correction locality, or reports that the two errors in the locality have disconnected the instance being corrected. In a mod(3) linked list one other pair of errors in a single correction locality cannot be corrected.

This algorithm, like its predecessors, proceeds backwards from the header nodes of the mod($k \geq 3$) instance state, iteratively attempting to identify the correct address of the previous node. This previous node is called the *target*. Because the algorithm performs 2-selective-local-correction, we will subsequently assume that this algorithm encounters at most two errors in any locality examined by it. We will also assume that the Valid State Hypothesis holds. Pseudocode for this algorithm is presented in Appendix B1.

28. Terminology

Nodes will be labelled N and subscripted by the correct forward distance from them to the last trusted node. The last trusted node is therefore N_0, while earlier trusted nodes have negative subscripts. The target node is always N_1.

Back pointers will be labelled b and forward pointers f with subscripts indicating the correct distance spanned by these pointers. Pointers will be prefixed by the node in which they reside, or, by extension, a path that addresses them. When appropriate, superscripts will indicate the number of consecutive occurrences of a pointer type within a path. $N_x \cdot b_k / N_{x+k} \cdot f_1$ represents exactly one of $N_x \cdot b_k$ and $N_{x+k} \cdot f_1$. Figure 5.1 illustrates this notation, by showing a locality in a mod(k) list.

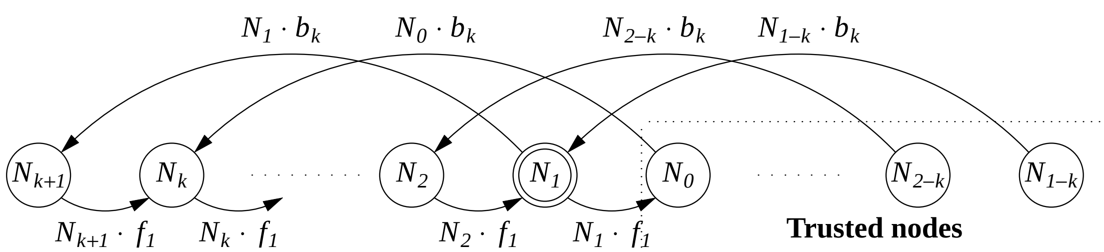


Figure 0.1. A correct mod(k) locality

When explicitly discussing the k header nodes these will be labelled H. In a correct instance $H_i \cdot f_1 = H_{i-1}$ for $0 < i < k$. If the instance is not empty then $H_0 \cdot f_1$ addresses the first non-header node, and $H_0 \cdot b_k$ the last non-header node. Otherwise, in an empty instance, $H_0 \cdot f_1 = H_{k-1}$ and $H_i \cdot b_k = H_i$ for $0 \le i < k$. This is illustrated in Figure 5.2.

29. Votes

One method of attempting to identify the target is to use *votes* [24]. In this structure, each *constructive* vote is a function which follows a path from a trusted node and returns a *candidate* node, N_n, for consideration as the target. Constructive votes are labelled C and distinguished by subscripts. Each *diagnostic* vote is a predicate which when presented with a candidate node, N_n, assumes that this candidate is the target node, N_1, examines a path proceeding from this candidate, and returns true if this path appears correct. Diagnostic votes are labelled D, and are also distinguished by subscripts.

A candidate receives the *support* of each constructive vote that returns it, and each diagnostic vote which returns true when presented with it. A *weighted vote* is a vote which has associated with it a non-negative constant called its *weight*. The weight assigned to a vote X will be labelled $\bar{X}$. Each candidate *receives a vote* equal to the sum of the weights of all votes which support it. If the candidate is not the target then it is an *incorrect* candidate. Votes are *distinct* if they cannot support the same candidate as a result of using a common component. The following votes are used in performing 2-selective-local correction on a mod($k \geq 3$) linked list.

Vote	Pointers followed	Compared with
C_1	$N_{1-k} \cdot b_k$	
$C_i, 2 \leq i \leq k$	$N_{i-k} \cdot b_k \cdot f_1^{i-1}$	
D_1	$N_n \cdot f_1$	N_0
$D_i, 2 \leq i \leq k$	$N_n \cdot b_k \cdot f_1^{k-i+1}$	$N_{i-k} \cdot b_k$

These votes will be assigned weights later. For notational convenience the set of votes $\{C_i : 2 \leq i \leq k\}$, will be referred to as C_0. Similarly, the set of votes $\{D_i : 2 \leq i \leq k\}$, will be referred to as D_0.

30. Proof of correctness

Lemma 0.1

If an instance of a mod($k \geq 3$) structure contains at most two errors, it can be determined if this instance is empty. Having determined that an instance is empty, any errors in the instance can be trivially corrected.

Proof

In a mod($k \geq 3$) instance $k + 2 \geq 5$ components indicate when the instance is empty. Specifically, the back pointer in each of the k header nodes points back zero nodes, the forward pointer in the header node H_0 addresses the last header node H_{k-1}, and the count is zero. For the mod(3) structure, this is shown in Figure 5.2. Given at most two errors, the instance is therefore empty if and only if at least three of these components indicate that the instance is empty.

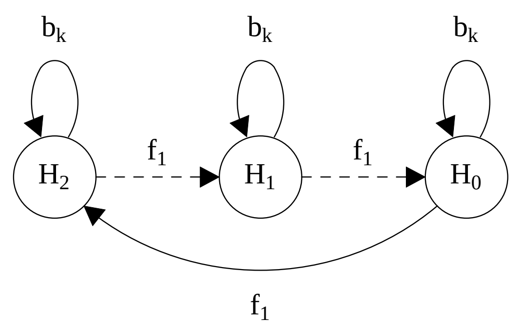


Figure 0.2. An empty instance of a mod(k = 3) structure

Lemma 0.2

If a connected target is always to receive a vote of at least one-half, and any incorrect candidate is always to receive a vote of at most one-half, whenever at most two errors occur in any locality within a mod($k \geq 3$) structure, it is necessary that the voting weights satisfy the following inequalities:

$$1)\ \bar{C}_1 = \bar{D}_1 = \bar{C}_0 = \bar{D}_0 = 1/4$$

$$2)\ \bar{C}_i + \bar{D}_i \leq 1/4, \text{ for } 2 \leq i \leq k$$

$$3)\ \sum_{j=i}^{k} \bar{C}_j + \sum_{j=2}^{i-1} \bar{D}_j \leq 1/4, \text{ for } 3 \leq i \leq k$$

Proof

Damaging any two of $\{N_{1-k} \cdot b_k,\ N_1 \cdot f_1,\ N_2 \cdot f_1,\ N_1 \cdot b_k\}$ causes the corresponding two votes in the set $\{C_1, D_1, C_0, D_0\}$ to fail to support the target. This leaves only the other two votes supporting the target. Damaging two of $\{N_{1-k} \cdot b_k,\ N_n \cdot f_1,\ N_2 \cdot f_1,\ N_n \cdot b_k\}$ appropriately causes the corresponding two votes in the set $\{C_1, D_1, C_0, D_0\}$ to support an incorrect candidate N_n. Since the target is required to receive a vote of at least one-half, and incorrect candidates are required to receive a vote of at most one-half, it follows that any pair of the above votes must necessarily have weights that sum to one-half. Solving gives $\bar{C}_1 = \bar{C}_0 = \bar{D}_1 = \bar{D}_0 = ¼$.

Suppose that $N_{i-k} \cdot b_k$ is damaged, for some $2 \le i \le k$. Then the target loses the support of votes C_i and D_i. If C_i and D_i had weights that summed to more than one-quarter, the target would be left receiving a vote of less than one-half when $N_1 \cdot f_1$ was also damaged. Since it is required that the target receive a vote of at least one-half, it is therefore necessary that $\bar{C}_i + \bar{D}_i \le ¼$, for $2 \le i \le k$.

Now suppose that $N_i \cdot f_1$ is damaged, for some $3 \le i \le k$. Then the target loses the support of all votes $C_{i \le j \le k}$ and $D_{2 \le j \le i-1}$. If these votes had weights that summed to more than one-quarter, the target would again receive a vote of less than one-half when $N_1 \cdot f_1$ was also damaged. Thus it is necessary that $\sum_{j=i}^{k} \bar{C}_j + \sum_{j=2}^{i-1} \bar{D}_j \le ¼$, for $3 \le i \le k$.

Lemma 0.3

If no more than two errors occur in any locality within a mod($k \ge 3$) structure; the instance being corrected is not empty; forward pointers are corrected when this first becomes possible; and votes are modified so that they do not support any of the last k trusted nodes, then the constraints imposed on voting weights in Lemma 5.2 ensure that (1) the target receives a vote of at least one-half, and (2) incorrect candidates receive a vote of at most one-half.

Proof of (1)

Since the instance is not empty, the target is distinct from the last k trusted nodes. Thus, modifying votes so that they cannot support any of the last k trusted nodes leaves the vote for the target unchanged. Since $\bar{C}_1 = \bar{D}_1 = \bar{C}_0 = \bar{D}_0 = ¼$, damaging any of $\{N_{1-k} \cdot b_k,\ N_1 \cdot f_1,\ N_2 \cdot f_1,\ N_1 \cdot b_k/N_{k+1} \cdot f_1\}$ removes a vote of one-quarter from the target. Since $\bar{C}_i + \bar{D}_i \le ¼$ for

$2 \le i \le k$, damaging any other back pointer in the locality removes a vote of at most one-quarter from the target. Since $\sum_{j=i}^{k} \bar{C}_j + \sum_{j=2}^{i-1} \bar{D}_j \le ¼$ for $3 \le i \le k$, damaging any other forward pointer in the locality removes a vote of at most one-quarter from the target. When multiple errors occur in the locality the target loses the support of at most those votes containing errors. Thus if two errors occur in the locality the target loses the support of at most two sets of votes each having weights that sum to at most one-quarter. Since all weights sum to one, the target therefore receives a vote of at least one-half.

Proof of (2)

Suppose that C_1 supports an incorrect candidate, N_n, which is therefore distinct from the last k trusted nodes. Then $N_{1-k} \cdot b_k$ contains an error. $N_{1-k} \cdot b_k$ is distinct from $N_n \cdot b_k$ since N_n is not a trusted node, and inductively $N_{1-k} \cdot b_k$ is distinct from $N_i \cdot b_k$ for $0 \ge i \ge 2 - k$. Thus an error in $N_{1-k} \cdot b_k$ causes only C_1 to support N_n. Thus C_1 is distinct from all other votes.

Suppose that D_1 and some $C_{2 \le i \le k}$ support N_n, as a result of both using $N_n \cdot f_1$. Then $N_n \cdot f_1$ addresses the last trusted node. If forward pointers have been repaired as early as possible, at least the last $k - 1$ forward pointers in the trusted set are correct, since $k - 1$ forward pointers can be corrected in the headers during initialisation. All pointers followed by C_i, after C_i uses $N_n \cdot f_1$, are therefore correct. This implies that C_i supports one of the last k trusted nodes, contradiction. Thus D_1 is distinct from C_0.

Now suppose that D_1 and some $D_{2 \le i \le k}$ support N_n, as a result of both using $N_n \cdot f_1$. Since the instance being examined is not empty, some other distinct error must exist in components used by D_i in supporting N_n, for D_i to use $N_n \cdot f_1$. After using $N_n \cdot f_1$, D_i can follow at most $k - i$ forward pointers. Thus D_i addresses one of the trusted nodes N_0 through N_{i-k}. Since D_i supports N_n, $N_{i-k} \cdot b_k$ must also address this node. No error can exist in $N_{i-k} \cdot b_k$ since two distinct errors exist in pointers followed by D_i, and $N_{i-k} \cdot b_k$ is distinct from both of these pointers. Since the instance is not empty $N_{i-k} \cdot b_k$ therefore points back between 1 and $k - 2$ nodes. But $N_{i-k} \cdot b_k$ correctly points back k nodes, contradiction. Thus D_1 is distinct from D_0.

The above demonstrates that C_1 and D_1 are distinct from all other votes. If C_1 and D_1 support N_n, they contain two distinct errors, and these errors cause no other vote to support N_n. In this case N_n receives a vote of one-half, since $\bar{C}_1 = \bar{D}_1 = ¼$. If neither C_1 nor D_1 support N_n, then N_n receives a vote of at most one-half, since $\bar{C}_0 = \bar{D}_0 = ¼$. Thus if N_n is to receive a vote of more than one-half, it must receive the support of one of C_1 or D_1, and a single independent error must cause N_n to receive the support of votes that sum to more than one-quarter.

If a single error occurs in a back pointer $N_{i-k} \cdot b_k$, for some $2 \le i \le k$, then C_i and D_i may support N_n, but no other vote can, since back pointers within the locality are distinct. Such an error cannot cause N_n to receive a vote of more than one-quarter, since we require that $\bar{C}_i + \bar{D}_i \le ¼$, for $2 \le i \le k$.

So suppose that a single error in a forward pointer $N_x \cdot f_1$ causes votes supporting N_n to sum to more than one-quarter. Then it must cause some $C_{2 \le i \le k}$, and some $D_{2 \le j \le k}$ to support N_n, since $\bar{C}_0 = \bar{D}_0 = ¼$. Since N_x is correctly addressed by the path used by C_i, N_x lies within the instance. If N_n lies outside the instance, and the Valid State Hypothesis holds, then inductively no correct path from N_n addresses a node within the instance. But the path used by D_j in supporting N_n correctly passes through N_x which lies within the instance. Thus N_n lies within the instance.

Since an error occurs in $N_x \cdot f_1$, N_x is not one of the last $k-1$ trusted nodes. Since D_j correctly passes through $N_x \cdot f_1$ in supporting N_n, and N_n is not one of the last k trusted nodes, N_n lies strictly between N_x and N_0. Since C_i supports N_n but follows only forward pointers after using the erroneous $N_x \cdot f_1$ pointer, N_n lies between N_0 and N_x, contradiction. Thus no single error can cause N_n to receive a vote of more than one-quarter.

Lemma 0.4

If weights satisfying the requirements of Lemma 5.2 are used, then in a mod($k \ge 3$) structure damaging two of $\{N_{1-k} \cdot b_k, N_1 \cdot f_1, N_2 \cdot f_1, N_1 \cdot b_k/N_{k+1} \cdot f_1\}$ causes the target to receive a vote of one-half. In a mod(3) structure damaging two of $\{N_{-2} \cdot b_3, N_{-1} \cdot b_3, N_0 \cdot b_3, N_1 \cdot f_1\}$, also causes the target to receive a vote of one-half. The weights $\bar{C}_1 = \bar{D}_1 = ¼$; $\bar{C}_2 = \bar{D}_k = 3/16$; and $\bar{C}_3 = \bar{D}_{k-1} = 1/16$, satisfy the requirements of Lemma 5.2, and ensure that the target receives a vote of more than one-half in all other cases.

Proof

For an error to remove a vote of one-quarter from the target, it must damage all votes with non-zero weights in one of the expressions in Lemma 5.2 that sum to one-quarter. The target receives a vote of exactly one-half when two errors are introduced into the locality, and each independently removes a vote of one-quarter from the target. Because $\bar{C}_1 = \bar{D}_1 = \bar{C}_0 = \bar{D}_0 = ¼$, damaging any two of $\{N_{1-k} \cdot b_k, N_1 \cdot f_1, N_2 \cdot f_1, N_1 \cdot b_k/N_{k+1} \cdot f_1\}$ therefore removes a vote of one-half from the target.

In a mod(3) structure, we require that $\bar{C}_2 + \bar{D}_2 \leq ¼$; $\bar{C}_3 + \bar{D}_3 \leq ¼$; $\bar{C}_2 + \bar{C}_3 \leq ¼$; and $\bar{D}_2 + \bar{D}_3 \leq ¼$. Collectively these inequalities imply that $\bar{C}_2 + \bar{D}_2 = ¼$, and $\bar{C}_3 + \bar{D}_3 = ¼$. Thus in a mod(3) structure damaging any two of $\{N_{-2} \cdot b_3, N_{-1} \cdot b_3, N_0 \cdot b_3, N_1 \cdot f_1\}$ also removes a vote of one-half from the target.

Assume that the weights proposed are used. Then the only equations that sum to one-quarter in Lemma 5.2 are those identified above as necessarily summing to one-quarter. Since $\bar{C}_2$, $\bar{C}_3$, $\bar{D}_{k-1}$, and $\bar{D}_k$ are each non-zero, the single errors that cause the target to lose a vote of one-quarter in a mod($k \geq 4$) structure occur only in $\{N_{1-k} \cdot b_k, N_1 \cdot f_1, N_2 \cdot f_1, N_1 \cdot b_k/N_{k+1} \cdot f_1\}$.

In a mod(3) structure the single errors that cause the target to lose a vote of one-quarter occur in $\{N_{-2} \cdot b_3, N_{-1} \cdot b_3, N_0 \cdot b_3, N_1 \cdot f_1, N_2 \cdot f_1, N_1 \cdot b_3/N_4 \cdot f_1\}$. The target receives a vote of more than one-half when one of $\{N_2 \cdot f_1, N_1 \cdot b_3/N_4 \cdot f_1\}$ and one of $\{N_{-1} \cdot b_3, N_0 \cdot b_3\}$ are damaged. Thus if the proposed weights are used, then the target receives a vote of one-half only under the types of damage suggested.

Lemma 0.5

In a mod(3) structure, damage that causes $N_{-1} \cdot b_3$ to address N_1, and $N_0 \cdot b_3$ to address N_2, is indistinguishable from damage that causes $N_{-2} \cdot b_3$ to address N_2, and $N_2 \cdot f_1$ to address N_0. Thus it cannot always be determined if the target is connected.

However, if the weights proposed in Lemma 5.4 are used, nodes contain identifier components, and at most two errors occur in any locality, then in all other cases it can be determined if the target is connected.

Proof

If all candidates receive a vote of less than one-half then the target must be disconnected, since Lemma 5.3 ensures that the target receives a vote of at least one-half. Conversely, if any candidate receives a vote of more than one-half this must be the target, since Lemma 5.3 ensures that no incorrect candidate receives such a vote. So assume that no candidate receives a vote of more than one-half, but some candidate receives a vote of exactly one-half. Then either this is the only candidate or multiple candidates exist. These cases are addressed separately.

Single candidate: If all constructive votes agree on a common candidate $\mathbf{N_n}$, and $\mathbf{N_n}$ receives a vote of one-half, then $\mathbf{N_n}$ receives no diagnostic votes. Thus either $\mathbf{N_n}$ is the target and both $\mathbf{N_1 \cdot f_1}$ and $\mathbf{N_1 \cdot b_k/N_{k+1} \cdot f_1}$ have been damaged, or $\mathbf{N_{1-k} \cdot b_k}$ and $\mathbf{N_2 \cdot f_1}$ address an incorrect candidate. In either case the identifier field in the candidate addressed must be unchanged, since at most two errors exist in the locality. Thus if the node addressed lies outside the instance this can be immediately detected, and disconnection reported.

Suppose instead that N_n lies within the instance. Consider following $N_n \cdot b_k \cdot f_1^k$. If N_n is the target, then since $N_1 \cdot f_1$ and $N_1 \cdot b_k/N_{k+1} \cdot f_1$ are damaged and represent the only damage in the locality, this path must either arrive at some node other than N_n, or arrive back at N_n prematurely. Conversely, if N_n is an incorrect candidate, but clearly not a trusted node since it receives a vote of one-half, then all pointers used in the above path are correct. Since N_n lies within the instance, this path must address N_n without passing through N_n. These tests can therefore be used to detect disconnection when all constructive votes agree on a common candidate.

Multiple candidates: If the target is disconnected and constructive votes do not all agree on a common candidate, then $\mathbf{N_{1-k} \cdot b_k}$ and $\mathbf{N_2 \cdot f_1}$ must address distinct incorrect candidates or address no node. Since it is assumed that some candidate $\mathbf{N_n}$ receives a vote of one-half, $\mathbf{N_n}$ must receive a vote of one-quarter from diagnostic votes. For $\mathbf{N_n}$ to receive a vote of one-quarter from $\mathbf{D_0}$, either $\mathbf{N_n \cdot b_k/N_{n+k} \cdot f_1}$ or both $\mathbf{N_0 \cdot b_k}$ and $\mathbf{N_{-1} \cdot b_k/N_{n+k-1} \cdot f_1}$ must be damaged. But these pointers are distinct from $\mathbf{N_{1-k} \cdot b_k}$ and $\mathbf{N_2 \cdot f_1}$, since $\mathbf{N_n}$ is not a trusted node. This implies that three errors exist in the locality contradicting the assumption that at most two errors occur in any locality. Thus the diagnostic vote must come from $\mathbf{D_1}$.

For D_1 to support an incorrect candidate N_n, $N_n \cdot f_1$ must contain an error that causes it to address N_0. Since $N_2 \cdot f_1$ is the only erroneous forward pointer in the locality, N_n must be N_2.

Since $N_2 \cdot f_1$ addresses N_0, C_0 does not support N_2. Thus C_1 does. The statement of the lemma has acknowledged that if this occurs in a mod(3) structure, then it cannot be determined if the target is connected. However, for a mod($k \geq 4$) structure in this case $N_{4-k} \cdot b_k$ is consistent with pointers $N_{2-k} \cdot b_k$ and $N_{3-k} \cdot b_k$ if and only if disconnection occurs.

Theorem 0.1

If the conditions of Lemma 5.5 are satisfied, and it has been determined that the target is connected as described in Lemma 5.5, then the target can always be identified.

Proof

If the target is the only candidate, or receives a vote greater than any other candidate, then the target is trivially identifiable. For an incorrect candidate N_n to receive the same vote as the target, both must receive a vote of one-half. Lemma 5.4 has established that the target receives a vote of one-half only if two of $\{N_{1-k} \cdot b_k, N_1 \cdot f_1, N_2 \cdot f_1, N_1 \cdot b_k/N_{k+1} \cdot f_1\}$ are damaged, or in a mod(3) structure if two of $\{N_{-2} \cdot b_3, N_{-1} \cdot b_3, N_0 \cdot b_3, N_1 \cdot f_1\}$ are damaged.

Suppose that constructive votes not supporting the target disagree. Then two distinct pointers used by correct constructive votes must be damaged. Thus either $N_{1-k} \cdot b_k$ and $N_2 \cdot f_1$ are damaged, or in a mod(3) structure two of $\{N_{-2} \cdot b_3, N_{-1} \cdot b_3, N_0 \cdot b_3\}$ are damaged. In the first case the target is disconnected, while in the second each invalid candidate receives a vote of less than one-half. Thus an incorrect candidate N_n receives a vote of one-half only if all constructive votes not supporting the target support this candidate.

Since N_n is an incorrect candidate it must be supported by at least one constructive vote. Thus one of $\{N_{1-k} \cdot b_k, N_{-1} \cdot b_k, N_0 \cdot b_k, N_2 \cdot f_1\}$ must be damaged. If no other error exists in the locality then N_n receives a vote of one-quarter. Thus a second error in the locality must cause additional votes to support N_n whose weights sum to one-quarter.

Suppose that a second error occurs in $N_1 \cdot f_1$. Then N_n receives a vote of at most one-quarter from constructive votes, since $N_1 \cdot f_1$ is not used by correct constructive votes. D_1 cannot support any candidate, since neither $N_1 \cdot f_1$ nor $N_2 \cdot f_1$ address N_0. Since N_n receives a vote of one-half, all non-zero votes in D_0 must therefore support N_n. For this to occur either $N_n \cdot b_k/N_{n+k} \cdot f_1$, or both $N_0 \cdot b_k$ and $N_{-1} \cdot b_k/N_{n+k-1} \cdot f_1$ must be damaged. $N_n \cdot b_k$ is correct since N_n is not one of the last k trusted nodes, and only two errors occur in the locality. $N_0 \cdot b_k$ and $N_{-1} \cdot b_k$ cannot both be damaged since it is assumed that an error occurs in $N_1 \cdot f_1$. One of $\{N_{n+k} \cdot f_1, N_{n+k-1} \cdot f_1\}$ therefore contains an error and is thus one of $\{N_1 \cdot f_1, N_2 \cdot f_1\}$. However, in this case $N_n \cdot b_k$ correctly addresses one of $\{N_1, N_2, N_3\}$. This implies that N_n is one of the last k trusted nodes, which it is not. Thus if any incorrect candidate receives the same vote as the target, $N_1 \cdot f_1$ must be correct.

If $N_n \cdot f_1$ does not address N_0, then since $N_1 \cdot f_1$ must, the target can be immediately identified. So suppose that both $N_1 \cdot f_1$ and $N_n \cdot f_1$ address N_0. Since $N_n \cdot f_1$ is distinct from $N_1 \cdot f_1$ it contains an error. Since only two errors exist in the locality, $N_n \cdot f_1$ must therefore be either $N_2 \cdot f_1$ or $N_{k+1} \cdot f_1$. $N_n \cdot f_1$ cannot be $N_2 \cdot f_1$ since an erroneous $N_n \cdot f_1$ addresses N_0 while an erroneous $N_2 \cdot f_1$ address N_n, which is distinct from N_0. Thus $N_n \cdot f_1$ is $N_{k+1} \cdot f_1$, implying that N_n is N_{k+1}. The two errors in the locality thus occur in $N_{k+1} \cdot f_1$ and one of $\{N_{1-k} \cdot b_k, N_{-1} \cdot b_k, N_0 \cdot b_k, N_2 \cdot f_1\}$. $N_1 \cdot b_k$ and $N_{k+1} \cdot b_k$ are therefore correct, since N_{k+1} is not a trusted node. $N_1 \cdot b_k$ therefore addresses the incorrect candidate N_{k+1}. $N_{k+1} \cdot b_k$ however does not address the target, since N_n is not the trusted node N_{1-k}. Thus if $N_n \cdot f_1$ and $N_1 \cdot f_1$ address N_0, the candidate whose back pointer addresses the other candidate must be the target.

31. Comparisons

The method presented here for improving the robustness of a standard double-linked list requires the presence of one additional identifier component per node, the presence of $k - 1$ additional header nodes, and a count component. This storage overhead is typically smaller than that required if error correcting codes are used, since at least two checksum components are needed to protect two data components against single errors [25].

The modification to the distance spanned by back pointers will increase the cost of performing updates in the proposed structure, and an alternative structure having two header nodes, an identifier, a forward pointer, and a virtual back pointer has therefore been proposed [4]. The virtual backpointer in node N_i contains the exclusive *OR* of the addresses of N_{i+1} and N_{i-1}. The true back pointer can therefore be determined by performing an exclusive *OR* of the virtual backpointer with $N_i \cdot f_1$. Similarly, the forward pointer $N_i \cdot f_1$ can be verified by performing an exclusive *OR* of the virtual backpointer with the address of the previous node. This clever modification to the backpointer produces a locally correctable structure which is as strongly connected as a mod(3) structure, and a correction algorithm which is competitive with historical methods of correcting mod(k) structures.

Empirical results presented in Appendix C1 suggest that the 2-selective-local-correction algorithm presented here is superior to previous mod(k) local correction algorithms, when applied to mod($k \geq 3$) structures. The results of using mathematical Markov models, justified in Chapter 8 and presented in Appendix F, reinforce the results presented in Appendix C1. However, since this selective-local-correction algorithm cannot correct mod(2) structures, these other algorithms are still valuable.

Chapter VI

Correcting helix(k) linked lists

32. Motivation

In Chapter 5 we presented an algorithm which performed 2-selective-local-correction on mod($k \geq 3$) linked lists. It is naturally of interest to ask if we can develop selective-local-correction algorithms for more complex linked-list structures having k pointers per node, particularly since the spiral($k \geq 3$) structure [24] which has k pointers per node, has been shown to be exactly $(k-1)$-local-correctable.

A spiral($k \geq 3$) regular linked list is similar to a mod(k) linked list but has the pointer structure (+1,+2 ... +k–1,–k), while the helix($k \geq 3$) linked list has the pointer structure (+1,–2 ... –k+1,–k). It has been shown in [128] that by traversing these structures backwards, each structure has 2k distinct votes on the location of the target node, k of which are constructive. Therefore, as justified in [24], these storage structures are at least $(2k-1)$-local-detectable. Since 2k locally undetectable changes can replace a node in such a list with one occurring at some distant point in the linked list (which therefore already has the correct node identifier) these structures are exactly $(2k-1)$-local-detectable. Thus they are k-selective-local-correctable.

The spiral(3) linked list has one unfortunate property. Having elected to traverse a spiral(3) list either forwards or backwards it is possible to insert 3 errors into the list which makes the chosen traversal method fail, even though all nodes can be reached by traversing the linked list in the opposite direction. The helix(3) structure was developed specifically to avoid this problem, and when generalized became the helix($k \geq 3$) structure.

The rest of this chapter concentrates on the helix($k \geq 3$) linked list, and on developing a k-selective-local-correction algorithm for this structure. This algorithm, like the spiral(k) local-correction algorithm, proceeds backwards from the header nodes of the instance state, iteratively attempting to identify the correct address of the previous node.

Because the algorithm performs k-selective-local-correction, we will subsequently assume that this algorithm encounters at most k errors in any locality examined by it. We will also assume that the Valid State Hypothesis holds.

Pseudocode for this algorithm is presented in Appendix B2. Most of the material presented in this chapter has already been published [43].

33. Votes

The helix(k) local-correction algorithm uses the following unweighted votes:

Vote	Path followed	Compared with node or path
C_i, $1 \le i < k$	$N_{-i} \cdot b_{i+1}$	
C_k	$N_{2-k} \cdot b_k \cdot f_1$	
D_1	$N_n \cdot f_1$	N_0
D_i, $2 \le i < k$	$N_n \cdot b_i$	$N_0 \cdot b_{i+1}$
D_k	$N_n \cdot b_k$	$N_0 \cdot b_2 \cdot b_{k-1}$

The node addressed by $N_0 \cdot b_2 \cdot f_1$ is also considered to be a candidate, even if this node is addressed by no constructive vote. Given that at most k errors occur in any locality, this ensures that some pointer correctly addressing the target lies within the locality being considered, unless the target is disconnected. Note that in a helix(3) structure, $N_0 \cdot b_2$ is the only backpointer addressing N_2 that is not used by any constructive vote. This is shown in Figure 6.1. For helix($k \ge 4$) structures other backpointers have this property and can be used instead of $N_0 \cdot b_2$ if so desired.

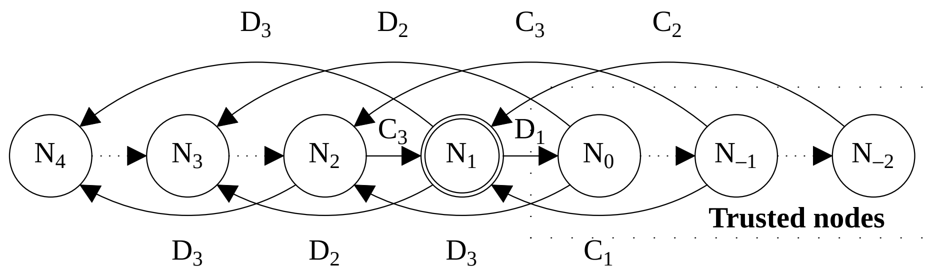


Figure 0.1. Pointers used in a correct helix(*3*) locality

34. Proof of correctness

Lemma 0.1

If an instance of a helix($k \geq 3$) structure contains at most k errors, it can be determined if this instance is empty. Having determined that the instance is empty, any errors in the instance can be trivially corrected.

Proof

Consider a correct empty instance of a helix($k \geq 3$) structure, shown in Figure 6.2. Since the pointers in a helix(k) structure form a circular multiply-linked list, and an empty instance contains only the k header nodes that define this instance, the b_k pointers in each of the k header nodes point back zero nodes, while the b_{k-1} pointers in each of these k header nodes point forward one node. In addition the f_1 pointer in the earliest header node addresses the last header node, and the count is zero.

Now consider a correct non-empty instance of a helix($k \geq 3$) structure. The only component described above that remains unchanged is the b_{k-1} pointer in the earliest header node, which always correctly addresses the last header node. At least $2k + 1$ components therefore contain values which can independently be used to determine if the instance is empty. Since at most k of these components contain errors, the majority of these $2k + 1$ components remain correct. A helix($k \geq 3$) instance containing at most k errors is therefore empty if and only if at least $k + 1$ of the above components confirm this.

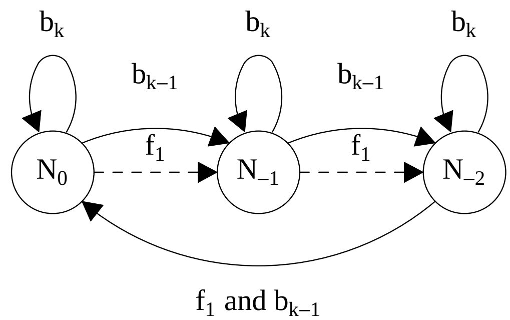


Figure 0.2. An empty instance of a helix(k = 3) structure

Lemma 0.2

If $r \leq k$ errors occur in any locality within a helix($k \geq 3$) structure, the instance being corrected is not empty, and votes are modified so that they do not support any of the last k trusted nodes, then **(a)** the target receives at least $\mathbf{2k - r \geq k}$ votes, and **(b)** incorrect candidates receive at most $\mathbf{r \leq k}$ votes.

Proof of (a)

Since the instance is not empty, the target is distinct from the last k trusted nodes. Thus, modifying votes so that they cannot support any of the last k trusted nodes leaves the vote for the target unchanged. In a correct non-empty instance each vote supporting the target uses distinct pointers. Since r pointers are assumed to be damaged, at most r votes can fail to support the target. The other $2k - r$ votes must therefore continue to support the target.

Proof of (b)

Each vote supporting an incorrect candidate N_n contains at least one error. If N_n is to receive more than r votes as a result of r errors, then at least one of the votes supporting N_n must contain only errors present in other votes that also support N_n.

If a shared error occurs in a forward pointer then it must be shared by D_1 and C_k, since no other vote uses a forward pointer. Since D_1 supports N_n, the pointer $N_n \cdot f_1$ addresses N_0. Since C_k shares the pointer $N_n \cdot f_1$ with D_1 it supports the node that this pointer addresses. Therefore C_k supports N_0. But N_0 is trusted and thus receives no votes, contradiction.

The only error in a back pointer that could be shared by votes supporting an incorrect candidate N_n, must occur in the b_{k-1} pointer used by D_k, since all other back pointers used either occur at different offsets, or originate in nodes that are known to be distinct. This error can be shared with at most one of C_{k-2}, D_{k-1} and (when $k \geq 4$) D_{k-2}, since no other vote uses a b_{k-1} pointer. These possibilities are shown in Figure 6.3, Figure 6.4, and Figure 6.5. For this shared error to cause N_n to receive more than r votes as a result of r errors, no vote supporting N_n may contain more than one error.

If C_{k-2} and D_k both use the erroneous pointer $N_{2-k} \cdot b_{k-1}$, and the instance being corrected is not empty, then D_k contains at least two errors since $N_0 \cdot b_2$ incorrectly addresses N_{2-k}. If D_{k-2} and D_k both use the erroneous pointer $N_0 \cdot b_{k-1}$, then D_k contains at least two errors since $N_0 \cdot b_2$ incorrectly addresses itself. Finally, if D_{k-1} and D_k both use the erroneous pointer $N_n \cdot b_{k-1}$, then at least one of $N_0 \cdot b_k$ and $N_n \cdot b_k$ must be in error since they originate in distinct nodes, but address a common node.

Since at most one error can be shared by two votes supporting an incorrect candidate N_n, and then only if some vote supporting N_n contains at least two errors, N_n receives at most r votes when r errors are introduced into any locality.

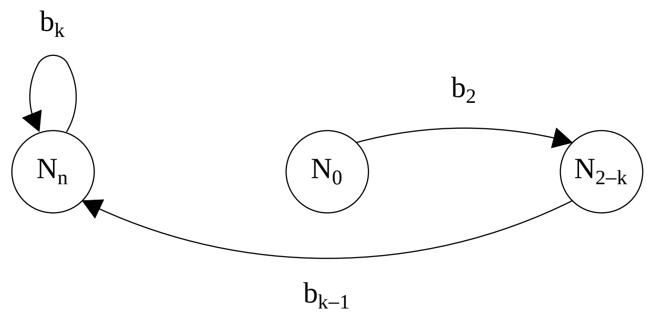


Figure 0.3. Configuration if C_{k-2} and D_k share b_{k-1}

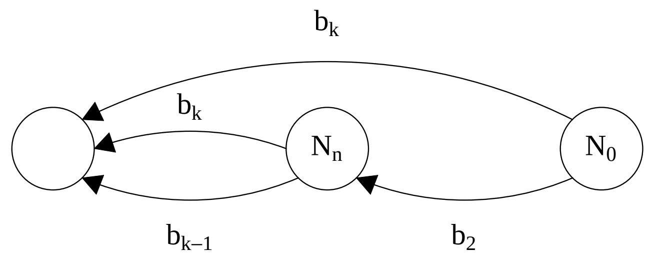


Figure 0.4. Configuration if D_{k-1} and D_k share b_{k-1}

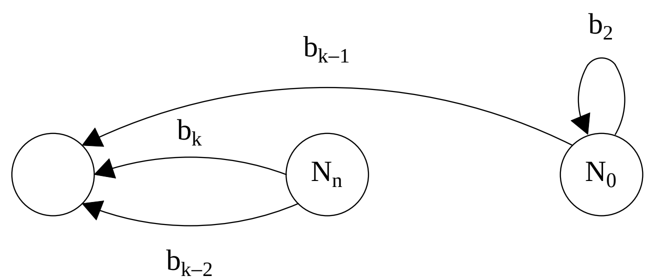


Figure 0.5. Configuration if D_{k-2} and D_k share b_{k-1}

Lemma 0.3

In a helix(3) structure, changing $N_2 \cdot f_1$ to address N_0, and the other two pointers correctly addressing N_1 so that they address N_2, is indistinguishable from damage that causes $N_{-1} \cdot b_3$ and $N_0 \cdot b_2$ to address N_1, and $N_0 \cdot b_3$ to address N_2. Thus it cannot always be determined if the target is connected. However, if nodes contain identifier components, and at most k errors occur in any locality, then in all other cases it can be determined if the target is connected.

Proof

If all k pointers correctly addressing the target have become damaged then the target is disconnected. Otherwise, since at most k errors occur in any locality, the target is connected, and either supported by one of the constructive votes, or addressed by the path $N_0 \cdot b_2 \cdot f_1$.

If no candidate receives k or more votes then the target must be disconnected, since Lemma 6.2 ensures that the target receives at least k votes. Conversely, if any candidate receives more than k votes this must be the target. So assume that some candidate receives k votes and no candidate receives more than this. Then either this is the only candidate or multiple candidates exist. These cases are addressed separately.

Single candidate: If only one candidate $\mathbf{N_n}$ exists, and this candidate is the target node $\mathbf{N_1}$, then only diagnostic votes contain errors, implying that $\mathbf{N_{2-k}} \cdot \mathbf{b_k}$ is correct. Conversely, if $\mathbf{N_n}$ is not the target, the path $\mathbf{N_0} \cdot \mathbf{b_2} \cdot \mathbf{f_1}$ and all paths used by constructive votes incorrectly address $\mathbf{N_n}$ and thus contain errors. Since only **k** errors occur in the locality, each path contains one error and the error in the path $\mathbf{N_0} \cdot \mathbf{b_2} \cdot \mathbf{f_1}$ also occurs in the path $\mathbf{N_{2-k}} \cdot \mathbf{b_k} \cdot \mathbf{f_1}$ used by $\mathbf{C_k}$. Thus $\mathbf{N_2} \cdot \mathbf{f_1}$ contains an error but once again $\mathbf{N_{2-k}} \cdot \mathbf{b_k}$ does not.

Since $N_{2-k} \cdot b_k$ is correct and addresses N_2, it can easily be determined if $N_n = N_2$. Similarly, since at most k errors occur in any locality, N_n must have an undamaged identifier field, allowing it to be easily determined if N_n lies outside the instance being corrected. Finally, it can easily be determined if N_n is one of the last k trusted nodes. In any of the above cases N_n is clearly not the target node N_1.

So suppose that N_n lies within the instance, but has an address that differs from N_2, N_1, and each of the last k trusted nodes. If $N_n \cdot f_1$ contains an error, then this pointer must be used by C_k since each incorrect pointer in the locality is used by some constructive vote, but no other constructive vote uses f_1. Since C_k contains only this one error and supports N_n, $N_n \cdot f_1$ must both occur in and address N_2, as shown in Figure 6.6. This implies that $N_n = N_2$, contradiction. Thus $N_n \cdot f_1$ is correct. Conversely, if N_n is the target node N_1, then since all of the diagnostic votes associated with $N_{n=1}$ are damaged $N_n \cdot f_1$ contains an error. Thus N_n is the target if and only if $N_n \cdot f_1$ contains an error.

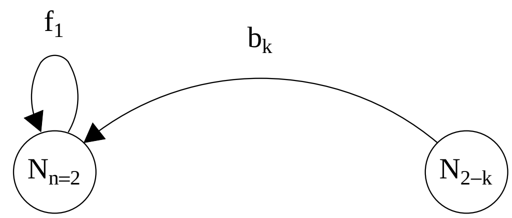


Figure 0.6. $N_n \cdot f_1$ being used by C_k to support N_n

The pointer $N_n \cdot f_1$ cannot address N_0 since it is known that N_n receives no diagnostic votes. If this pointer addresses any other trusted node then it contains an error since N_n is distinct from the last k trusted nodes. This pointer also clearly contains an error if it addresses itself. In any of the above cases, since $N_n \cdot f_1$ is known to be in error, N_n is the target. So assume that $N_n \cdot f_1$ addresses N_x which is distinct from N_n and the last k trusted nodes. Then $N_x \cdot b_k$ is correct since it is distinct from all of the b_k pointers containing errors.

Consider following the path $N_n \cdot f_1 \cdot b_k$, and then $k-1$ forward pointers, as shown in Figure 6.7. If $N_n \cdot f_1$ is correct then none of these $k-1$ forward pointers can be the erroneous $N_2 \cdot f_1$ pointer, since N_n is not one of the last k trusted nodes. Thus all $k-1$ forward pointers are also correct and form a path that arrives back at N_n. Conversely, if $N_n \cdot f_1$ is incorrect then $N_2 \cdot f_1$ is correct and thus the path followed must either fail to arrive back at N_n, or, in using $N_n \cdot f_1$ more than once, arrive back at N_n prematurely. Thus N_n is the target if and and only if the above path appears incorrect.

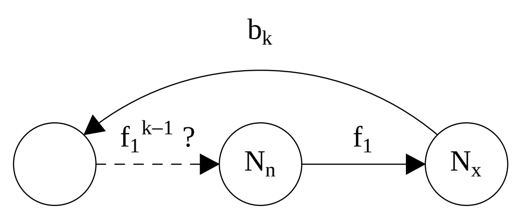


Figure 0.7. The path $N_n \cdot f_1 \cdot b_k \cdot f_1^{k-1}$

Multiple candidates: If constructive votes agree on a common candidate, but support a different candidate from that addressed by $\mathbf{N_0 \cdot b_2 \cdot f_1}$ then the target is connected. Otherwise, since constructive votes disagree, any candidate $\mathbf{N_n}$ receiving **k** votes must receive at least one diagnostic vote. If the target is disconnected, then all errors occur in pointers correctly addressing

$\mathbf{N_1}$. Only the diagnostic vote $\mathbf{D_1}$ can use one of these erroneous pointers to support $\mathbf{N_n}$. However, this implies that $\mathbf{N_n}$ is $\mathbf{N_2}$, and that $\mathbf{N_2} \cdot \mathbf{f_1}$ addresses $\mathbf{N_0}$.

The statement of the lemma has acknowledged that if this damage occurs in a helix(3) structure, then it cannot be determined if the target is connected. However, for a helix($k \geq 4$) structure the pointer $N_{-1} \cdot b_3$ is unused and thus correct since k other pointers within the locality are known to be in error. Since this pointer correctly addresses N_2 it can be used to determine if the candidate receiving k votes is indeed N_2. If it is, then the target is disconnected. Otherwise, this candidate is the target.

Theorem 0.1

If the conditions of Lemma 6.3 are satisfied, and it has been determined that the target is connected as described in Lemma 6.3, then the target can always be identified.

Proof

If the target is the only candidate, or receives a vote greater than any other candidate, then the target is trivially identifiable. For an incorrect candidate N_n to receive the same vote as the target N_1, both must receive k votes.

Suppose that $N_1 \cdot f_1$ contains an error. Then this error must be used by some vote supporting the incorrect candidate N_n, since otherwise $k - 1$ errors could cause k votes to support an incorrect candidate, contradicting Lemma 6.2. The only vote that can utilize such an error in $N_1 \cdot f_1$ is C_k, and then only if $N_{2-k} \cdot b_k$ erroneously addresses N_1. But in this case C_k contains two errors that are used by no other vote that supports N_n. This implies that $k - 2$ errors cause the remaining $k - 1$ votes to support N_n. Once again this contradicts Lemma 6.2. Thus $N_1 \cdot f_1$ must be correct.

Since $N_1 \cdot f_1$ is correct we can trivially identify the target if $N_n \cdot f_1$ does not address N_0. So suppose that $N_n \cdot f_1$ contains an error that causes it to also address N_0. Since it is known that each error in the locality damages a vote correctly supporting the target, the incorrect candidate, N_n, must be N_2. But in this case the damage to $N_2 \cdot f_1$ implies that $N_{2-k} \cdot b_k$ is correct and therefore addresses the incorrect candidate N_n. Thus if both $N_1 \cdot f_1$ and $N_n \cdot f_1$ address N_0 then the target is that node not addressed by $N_{2-k} \cdot b_k$.

35. Conclusions

The above results are the natural progression of ideas first developed in Chapter 4 and Chapter 5. The helix(k) selective-local-correction algorithm, presented in Appendix B2, is slightly longer than the mod(k) selective-local-correction algorithm, presented in Appendix B1, but somewhat easier to prove correct.

Empirical results, presented in Appendix C2, suggest that the helix(k) selective-local-correction algorithm is significantly better than the spiral(k) local-correction algorithm [24], when operating on comparable structures. This is hardly surprising, since the spiral(k) local-correction algorithm assumes that at most $k - 1$ errors occurred in any locality, and therefore makes no attempt to either detect disconnection or to behave intelligently when k errors occur in a correction locality.

Chapter VII

Locally correctable trees

36. Introduction

A binary tree is a storage structure which allows rapid retrieval of data. The structure consists of a collection of *nodes* that each contain two *link* pointers, and a *key*. Each node with the exception of the *header* node is addressed by exactly one link residing in its *parent* node, and is considered to be a *child* of this parent node. Obviously, since the structure is finite, some links are unused. These links typically contain some special value indicating that they are *null*. A *full* node has two children, an *incomplete* node has one child, and a *leaf* node has no children. In a binary search tree the keys within the structure are arranged in such a way that all keys reached by following a ‘‘left’’ link out of any node are lexicographically smaller than the key recorded in this node, while all keys reached by following a ‘‘right’’ link are larger than this key.

Classical trees are not robust. Errors in keys are undetectable, unless these errors affect the key ordering, while errors in non-null links disconnect the structure [123]. A number of binary trees have been proposed that allow a limited number of errors to be detected and corrected, by performing a global examination of the erroneous storage structure instance [21[98, 111, 114, 126]141].

In this chapter we will consider what are perhaps the three most widely used tree structures having no previously known corresponding locally-correctable robust tree structure, and will present 1-local-correctable versions of each of these three structures. These structures are in order, the binary search tree, the AVL tree, and the m-ary trie. The 1-local-correctable AVL tree was first presented in [42]. A 1-local-correctable checksummed binary tree has already been described in Chapter 4, and two 1-local-correctable B-trees are described in [83130].

All of the trees presented in this chapter have nodes which contain a node identifier, two link components, a key component, and one additional *arc* pointer component. The arc pointer performs two functions. Firstly, it ensures that the structures containing it are 1-connected. Secondly, it assists in developing structures which are 1-locally-correctable. When key ordering is actively used to assist in performing local-correction, we will ensure that keys are themselves locally-correctable, by placing one additional *checksum* component in each node.

Nodes will be labelled N and distinguished by subscripts. Each tree will have one header node, denoted N_H. Left links will be labelled l, right links r, and arcs a. Arbitrary links will be labelled c, and keys k. Identifiers will be labelled id, and checksums, when present, s. Components will be prefixed by the node in which they reside, or by extension the path that addresses them. The symbol ∅ will be used to denote null pointers.

37. A sibling-linked search tree

37.1. Description

Let the left link, right link, and arc pointer, in the header node N_H of a sibling-linked tree, all address the root node in the tree. If no root node exists then these three pointers are null.

Consider a *full* node in a standard binary tree. We can increase the number of paths to each child node, by arranging that the arc pointer in each child node addresses the sibling node.

Now consider an incomplete node, N_x, in a standard binary tree. One link in this node addresses the child of this node, while the other is null. We can increase the number of paths to this single child by arranging that both links in N_x address this single child, provided that we either flag the location of the null link in the identifier component of N_x, or are willing to use the key in both N_x and the child node to determine if this is a left child node or a right child node. Arc pointers in solitary child nodes address the parent node.

Finally, include in each node a checksum component of the same size as the key component, which will be used to ensure that keys and checksums are themselves locally-correctable. Collectively, the above describes the organization of a sibling-linked tree.

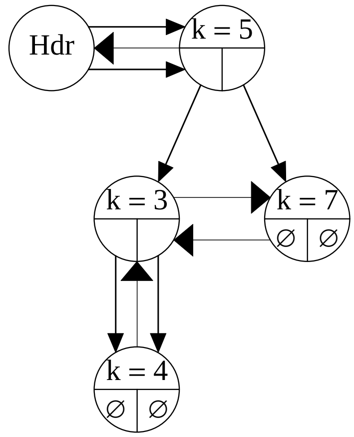


Figure 0.1. An example of the sibling-linked search tree

37.2. Local correction

Correction proceeds by selecting some node, N_x, which is addressed by a trusted pointer, but whose components are untrusted. By examining a small number of untrusted components, which are collectively assumed to contain at most one error, the correct values of both links in this node can be deduced. These can then be corrected if necessary, as can the arcs in the nodes that they address. This process continues until all pointers are trusted.

Correction is accomplished by guessing that N_x is a leaf node, an incomplete node, or a full node. When full, we further guess which of the two links is correct. Associated with each of these four guesses are two pseudo-votes, shown in Figure 7.2.

When guessing that the node N_x is a leaf, the votes are $N_x \cdot l = \varnothing$ and $N_x \cdot r = \varnothing$. When guessing that a node has only one child, the votes are $N_x \cdot l = N_x \cdot r$ and either $N_x \cdot l \cdot a = N_x$ or $N_x \cdot r \cdot a = N_x$. When guessing that $N_x \cdot l$ correctly addresses one of two children below N_x, the votes are $N_x \cdot l \cdot a = N_x \cdot r$ and $N_x \cdot l \cdot a \cdot a = N_x \cdot l$. Similarly, when guessing that $N_x \cdot r$ correctly addresses one of two children below N_x the votes are $N_x \cdot r \cdot a = N_x \cdot l$ and $N_x \cdot r \cdot a \cdot a = N_x \cdot r$.

If no errors exist in the locality examined, then a cursory examination of the votes employed reveals that correct guesses will receive two votes, and other guesses will receive no vote. If one error exists in the locality examined, then no guess will receive two votes, and guesses which receive no votes must be incorrect. Therefore, if only one guess receives a vote,

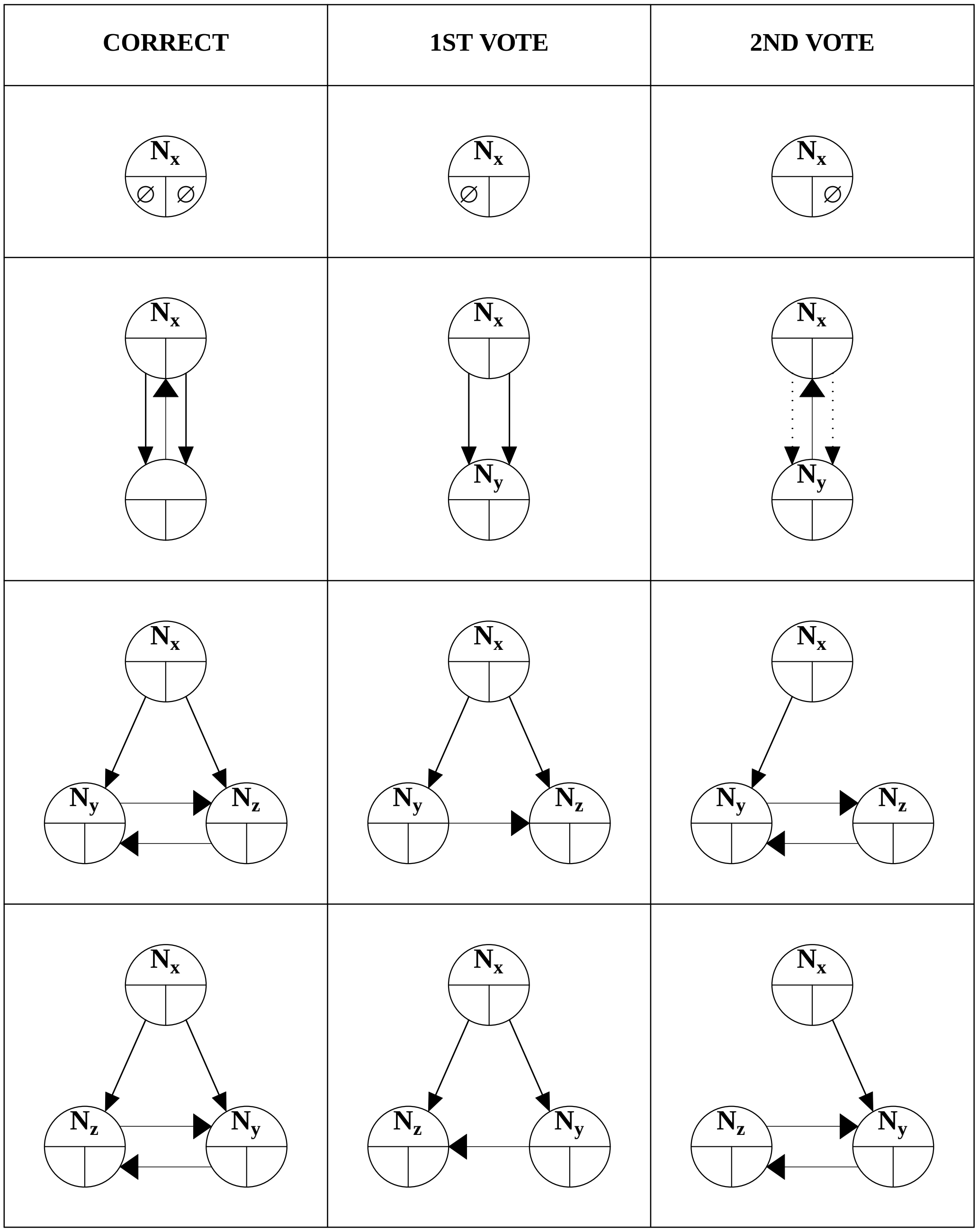


Figure 0.2. Pseudo votes used in the sibling-linked tree

then this guess must be correct, allowing the single error in the locality to be corrected.

Otherwise, additional effort is needed in order to identify the correct guess.

If $N_x \cdot l$ or $N_x \cdot r$ fails to address a node in the memory space, or addresses the previously identified sibling of N_x, then this link in error. Given the assumption that only one error occurs in any locality, such errors can be trivially corrected. It will therefore subsequently be assumed that such errors have not occurred.

Suppose that we add a bounded number of additional components to the locality being examined, when this locality is discovered to contain an error. If the locality constraint continues to be satisfied, then these additional components contain no errors. Thus if any pointer in the locality addresses a node with an invalid node identifier, then by adding these identifiers to the locality, the local-correction procedure can conclude that the pointer addressing this invalid node identifier is in error, and can therefore be corrected. So assume otherwise.

If any of the guesses receiving one vote suggests that N_x has two child nodes, N_y and N_z, where $N_z = N_y \cdot a$, then add $N_x \cdot k$, $N_y \cdot k$, and $N_z \cdot k$ to the locality. Since we assume that the locality constraint continues to be satisfied, none of these three keys is in error. Because keys in a binary search tree are ordered, we can conclude that this guess is incorrect if these three keys have an illegal ordering. Such a guess can also obviously be rejected if N_x is the header node of the instance, or if either N_y or N_z is a direct ancestor of N_x. Otherwise, N_y and N_z must be correctly ordered siblings, and this guess therefore correct.

Once it has been established that N_x is not a full node, eliminate any guess that suggests that it might be. If only one guess now receives a vote then this guess must be correct. Otherwise, the two remaining guesses are that N_x is a leaf node, and that N_x has one child. This occurs if and only if one of the links in N_x is null, and the other addresses a node N_y, satisfying $N_y \cdot a = N_x$. We have previously ensured that N_y is not the sibling node of N_x. Thus, since $N_y \cdot a = N_x$, N_y is the only child of N_x.

37.3. Correcting keys

Having corrected the pointers in the instance, the keys and identifiers can be corrected by using their associated checksums. It may seem unreasonable to correct keys and identifiers after pointers have been corrected, and yet base the correction of pointers in part on these potentially erroneous keys and identifiers. However, if anything, it is our assumption that no locality contains more than one error that is unreasonable. If this assumption holds, then the only keys and identifiers that are examined during the correction of pointers are necessarily correct.

Initially, when designing the storage structure, some cyclic ordering is associated with the nodes in the tree. One satisfactory choice is to use a pre-order traversal. Such an ordering places a parent node before its left child and the left child before the right child, assuming that both such nodes exist. This particular ordering allows the node following any node to be located simply and efficiently. The successor of the last node in the ordering is defined to be the header.

Let the node N_{x+1} follow the node N_x within this ordering, and let the checksum, $N_x \cdot s$, satisfy $N_x \cdot s = N_{x-1} \cdot k + N_x \cdot k$, for all N_x. Thus if any $N_y \cdot k$ is inserted, or updated, $N_y \cdot s$ and $N_{y+1} \cdot s$ must also be updated. N_y is updated anyway, and the above ordering ensures that $N_{y+1} \cdot s$ can be updated by performing one additional probe. N_{y-1} need never be retrieved since, prior to any change, $N_{y-1} \cdot k = N_y \cdot s - N_y \cdot k$. Thus keys and checksums can be inserted and updated efficiently.

Initially, since the key in the header is known, this component is corrected if in error and then added to the set of trusted components. The correct value of each successive key is determined iteratively by using $N_x \cdot k$, $N_x \cdot s - N_{x-1} \cdot k$ and $N_{x+1} \cdot s - N_{x+1} \cdot k$ as votes [24] that agree if and only if they evaluate the correct value for $N_x \cdot k$. Having corrected $N_x \cdot k$ if necessary, $N_x \cdot k$ becomes trusted. Since $N_{x-1} \cdot k$ is also trusted, $N_x \cdot s$ can now be corrected if incorrect before also becoming trusted. Correction is complete when all links, arcs, identifiers, keys, and checksums have been thus corrected.

38. A robust AVL tree

38.1. Description

In a height-balanced (AVL) binary tree, the heights of the left and right subtrees below any node differ by at most one [75]. An identifier exists in each node which indicates the current direction of any such imbalance in the two subtrees below this node. Because the tree is height-balanced, expected retrieval times are reduced, and worst case insert and delete operation times are logarithmic.

The AVL tree structure being considered will be made more robust by adding additional redundancy to the nodes of the structure. In addition to the height balancing information present in each node identifier, each node identifier will also contain two flags explicitly identifying the location of null links within the node. Each node will also contain an arc pointer. If desired, keys may be protected by associating checksums with them, as described above.

Within a correct structure all pointers in the header node, N_H, address the root node if this exists. Links address child nodes as expected, and arc pointers form a cyclic single linked list which links nodes in the order defined by the following node traversal:

```
Visit(N_H)
If N_H · r ≠ ∅  {      /* Not null */
        Visit(N_H · r)
        Traverse(N_H · r)
}
```

```
Traverse(N) {
        If N · l ≠ ∅ Visit(N · l)
If N · r ≠ ∅ Visit(N · r)
        If N · l ≠ ∅ Traverse(N · l)
        If N · r ≠ ∅ Traverse(N · r)
}
```

Figure 0.3. Arc traversal order in the AVL tree

38.2. Global characteristics

The structure described above has the following global characteristics. It can be traversed using either the links or the arcs and is thus 1-connected. It can be reconstructed by either using correct links, or by using correct arcs and identifiers, even if identifiers do not contain height-balance information. The structure is therefore 2-determined [125]. We will show that the structure can be corrected when at most one error occurs in every bounded correction locality even if height-balance flags are absent. The structure is therefore 1-locally-correctable, and thus trivially both 1-local-detectable and 1-correctable. However, without these flags certain pairs of changes within subtrees are undetectable, as shown in Figure 7.5, since they leave the structure appearing internally to be correct. Thus, if height balance flags are absent, this structure is unusual, since it has exactly the same detectability, local detectability, correctability, and local correctability.

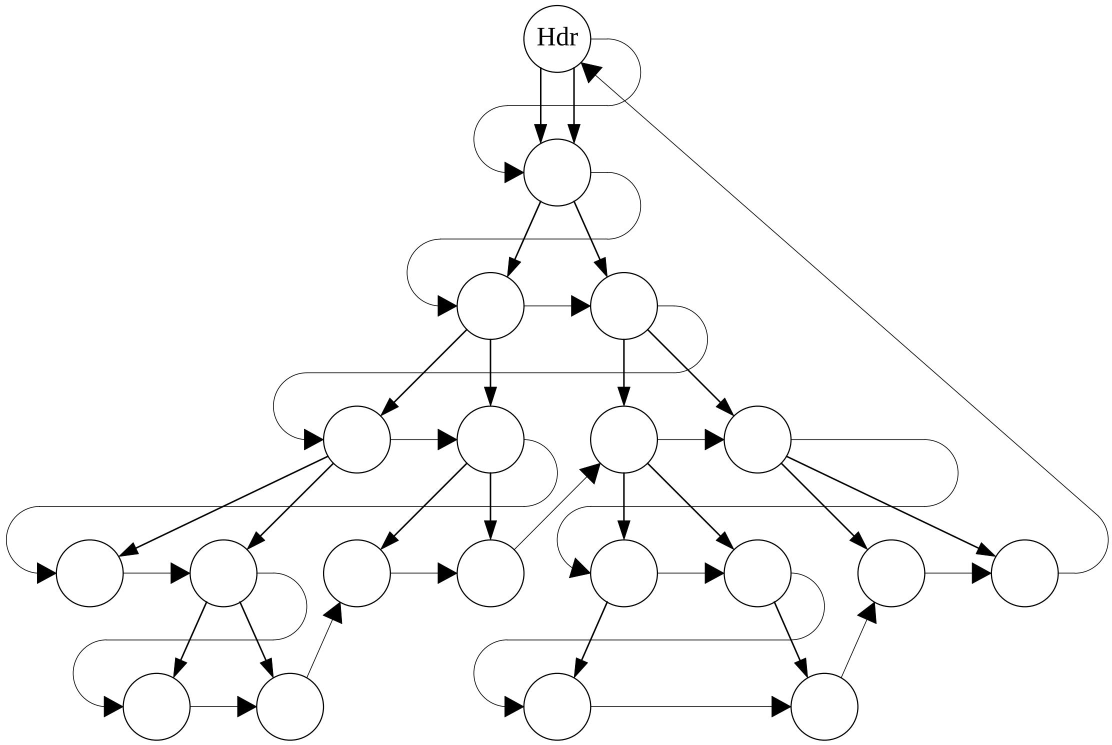


Figure 0.4. An example of the proposed AVL tree

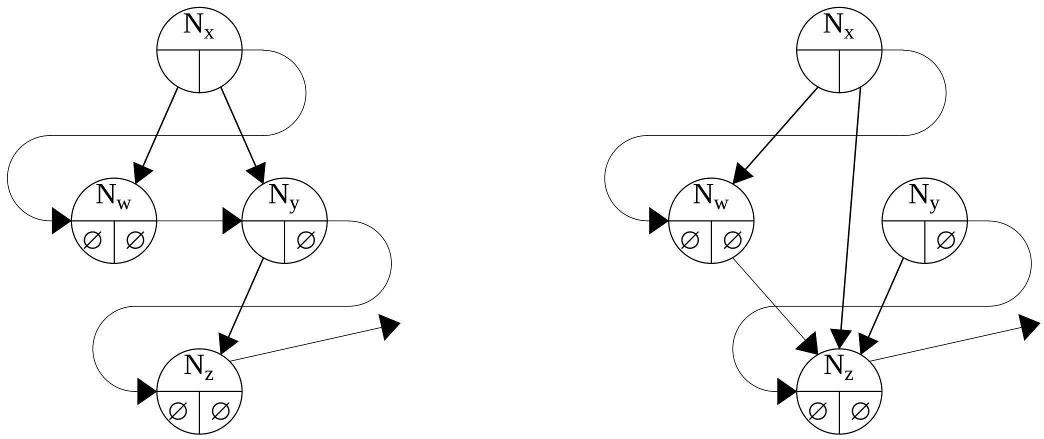


Figure 0.5. A pair of undetectable changes in $N_w \cdot a$ and $N_x \cdot r$

Given that node identifiers do contain height balance flags, the structure is 2-detectable since any undetectable transformation of the instance requires at least three changes, and certain sets of three changes are indeed undetectable as shown in Figure 7.6.

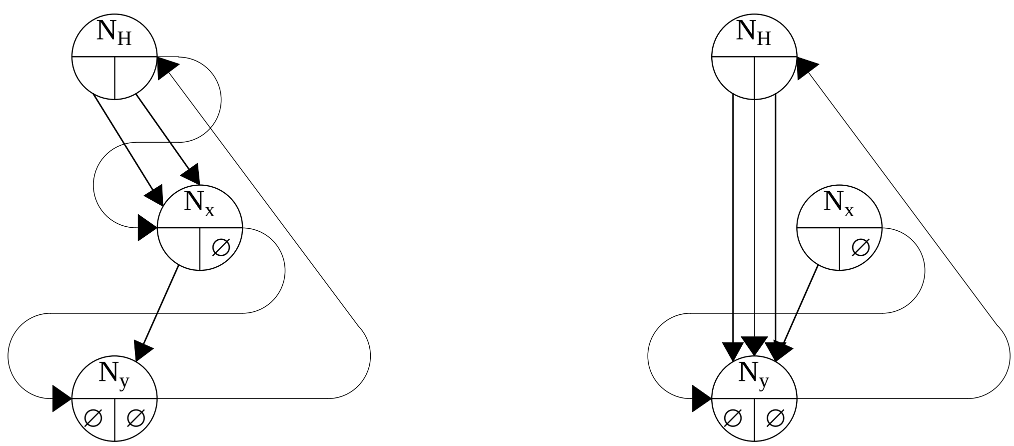


Figure \n(H0.6. Three undetectable changes in $N_H \cdot l$, $N_H \cdot r$ and $N_H \cdot a$

38.3. Local correction

When correct, the AVL tree can be traversed in the same sequence by either following the single-linked list formed by the arcs, or by traversing the links that form the tree. Each step of the arc traversal involves examining one new arc. Following the same traversal using links is considerably more complex, but still involves examining at most a bounded number of new components at each step, as justified below.

Suppose that we have arrived at some non-null link $N_x \cdot c$ and wish to identify the non-null link $N_m \cdot c = N_x \cdot c \cdot a$ so that we can proceed to the next step of the traversal. Then, as shown in Figure 7.7, at most four new null links will be examined before it is determined that N_x has no children or grandchildren that might contain this link. The search for this link then continues by proceeding up the tree from N_x, until we arrive at a node N_y having N_x in its left subtree. Since the tree is balanced, the node addressed by $N_y \cdot r$ exists and therefore contains the next two links in the ordering. If this node is a leaf node then two further null links will be encountered, before repeating the ascent of the tree from N_y until we encounter some N_z having N_y in its left subtree. Because the tree is balanced, the node addressed by $N_z \cdot r$ exists and has at least one child. Thus in the worst case we will encounter a seventh null link $N_z \cdot r \cdot l$, before encountering the non-null link $N_z \cdot r \cdot r$. Conversely, if no further non-null link exists, then during one of the two ascents up the tree the header node will be encountered, signalling that the traversal is complete.

Having detected a discrepancy between arcs and links as a result of performing the above parallel traversal, any single error causing this discrepancy must occur in the last arc examined, or in the links examined during the last two steps of the parallel traversal, since either a null link encountered in the previous step of the traversal erroneously contained the same value as the desired link, or some error was encountered during the current step of the traversal.

Since we know that a single error exists in the above components we can assume that no error occurs in a bounded number of other new components that the correction procedure wishes to examine, following the detection of an error. Identifying null links that contain erroneous values and non-null links that have erroneously become null is therefore trivial since node identifiers contain flags indicating the location of null links within these nodes. Correcting such links is also trivial since null links correctly contain a known value, and non-null links correctly

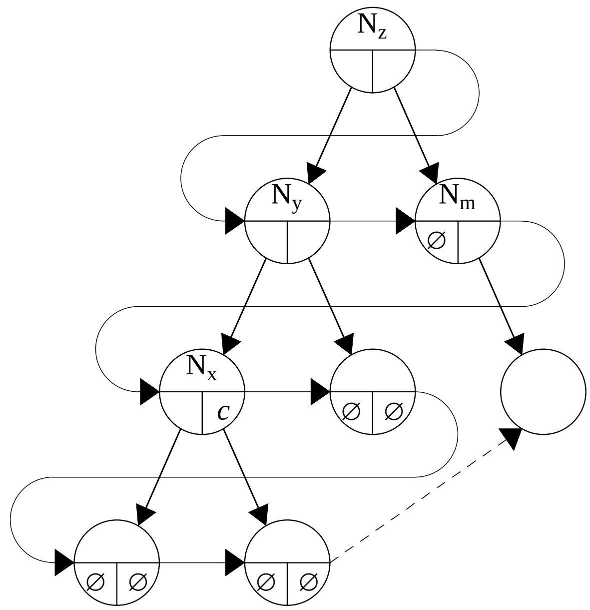


Figure 0.7. Maximum null links ∅ between $N_x \cdot c$ and $N_m \cdot c$

contain the same value as the last arc examined.

So, assume that null links within the locality being corrected contain no errors, and that non-null links appear non-null even if erroneous. Then the error within this locality must occur either in the last arc examined, or in the non-null link that was expected to contain the same value as this last arc. If the location of the error can be determined, the error can be trivially corrected since these two pointers agree when correct.

If the erroneous pointer addresses a non-existent node this can be detected when we attempt to access this node. Similarly if the erroneous pointer addresses a node outside of the instance being corrected, this can be detected by examining the identifier in this node.

So suppose that $N_x \cdot l$ and $N_w \cdot a$ address different nodes within the instance being corrected, as shown in Figure 7.8, and let N_z be the node that both should address. Then one further traversal step using only correct links can be performed, arriving at the node addressed by $N_m \cdot c = N_z \cdot a$. This is because N_x either has a right child addressed by $N_z \cdot a$, or the node correctly addressed by $N_x \cdot l$ can be assumed to be a leaf since the tree is balanced. Having identified the correct value of $N_z \cdot a$, using only correct pointers, we can identify N_z and thus the erroneous pointer, since N_z is not N_w, and no other correct node within the structure contains an arc with the same value as N_z.

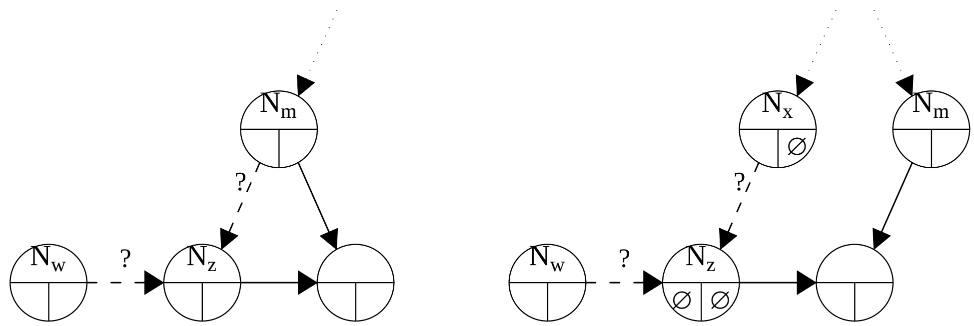


Figure 0.8. Possible configurations if left link or arc is in error

Now suppose that $N_x \cdot r$ and $N_w \cdot a$ address different nodes within the instance being corrected. If $N_x \cdot l$ is null or addresses an internal node, then the next non-null link $N_m \cdot c$ within the link traversal does not depend on $N_x \cdot r$, and correction can be performed as described above.

A problem arises however if $N_x \cdot r \neq N_w \cdot a$ and $N_x \cdot l$ addresses a leaf node, since in this case determining the correct value of $N_m \cdot c$ involves determining the correct value of $N_x \cdot r$. This occurs for example if one of the changes depicted in Figure 7.5 occurs. Fortunately, in this case $N_x \cdot r$ correctly addresses a subtree containing at most three nodes, since the tree is balanced. Thus if either $N_w \cdot a$ or $N_x \cdot r$ address a subtree containing more than three nodes, then the other pointer must be correct. Otherwise, the algorithm locates the next non-null link $N_n \cdot c$ not under N_x, by temporarily assuming that $N_x \cdot r$ is null. Then it locates the node N_z visited last within each subtree, and rejects the possibility that $N_x \cdot r$ correctly addresses this subtree if $N_z \cdot a \neq N_n \cdot c$. If neither subtree is rejected during this process, then both contain N_z. Since the two subtrees are distinct but each contains at most three nodes, one subtree must contain the single node N_z, while the other has as its root the parent of N_z. Thus, both $N_w \cdot a$ and $N_x \cdot r$ correctly address this larger subtree.

Having corrected all pointers, identifiers can be corrected. Correction of the height balance flags in each node can be accomplished efficiently by using a post-order traversal, if the height of each left subtree visited is stacked, until the height of the corresponding right subtree has been established. Correcting the other information in each node identifier is trivial.

39. A robust trie

39.1. Description

A trie is an m-ary tree structure which allows rapid retrieval of data [75]. The structure consists of a collection of nodes that each address, in some manner, an ordered set of at most m children. The keys present within this structure are represented in an alphabet of m characters, and each possible path from the header node to a leaf node spells out consecutive characters of one key within the structure. To avoid some keys being initial subsequences of other keys, keys are usually terminated by a special termination character.

In general, the cost of storing a vector of m pointers in each node of a trie is prohibitive, and we therefore will use lists to represent tries [7]. The first child below any node is addressed by the child link in its parent node, while all the siblings of this child node are linked to this first child node by using a single sibling link in each node. Since absent siblings are not included in this linked list, each node also contains a single character key identifying the relative position of this node within the sibling list. The last sibling pointer in the sibling list addresses the parent node.

The above structure degenerates into a (+1,-1) linked list when m is one, and into an unconstrained, but non-standard, binary tree when m is two. However, it is also possible to use the above structure to represent a constrained binary tree, in which links have their more normal interpretation, by redefining child links as left links, and sibling links as right links. For this reason we will denote child links by l and sibling links by r. In such a constrained binary tree at most m right links can be followed successively before arriving at a null right link.

Such a constraint is easily enforced if rotations are allowed. Simply detect violations of this constraint when attempting to insert any node, and re-establish this constraint by rotating the parent of this node to the left of this node so that it becomes a left child of this node. The rotation is particularly straightforward and can be accomplished easily, even if the structure contains the redundancy described below. However, it should be stressed that this rotation increases the amount of imbalance below any rotation point, and will therefore tend to encourage the construction of unbalanced trees.

39.2. Additional redundancy

The tree structure being considered will be made more robust by adding additional redundancy to the nodes of the structure. Each node will contain a node identifier which, when correct, identifies the instance to which this node belongs. The identifier will also contain two flags, explicitly identifying the location of null links within this node. Each node will also contain an arc pointer. If desired, keys may also be protected by using checksums as described above.

Null right links address the parent node of the node that they reside in. As a special case (since the header node has no parent) the null right link in N_H addresses N_H. If the tree is empty, then all pointers in N_H are null. Otherwise, $N_H \cdot l$ correctly addresses the root node, and $N_H \cdot r$ is null. Arc pointers form a cyclic single linked list, by generalizing the cyclic linked list formed by arc pointers in the AVL tree. Specifically, this linked list is ordered using the following node traversal:

Visit(N_H)
Traverse(N_H)
	Traverse(N) {
	    For each child N_i of N Visit(N_i)
	For each child N_i of N Traverse(N_i)
	}

Figure 0.9. Arc traversal order in the m-ary Trie

As shown in Figure 7.10, this structure is similar to the robust AVL tree, and, like the AVL tree without height balance flags, is 1-detectable, 1-local-correctable, and thus also 1-local-detectable and 1-correctable. As before, single pointer errors will be detected by performing a parallel traversal of links and arcs.

39.3. Local correction

The only possible traversal using arcs is to follow the single-linked list formed by these arcs. Each step of this traversal involves examining one new arc. Following the same traversal using links is considerably more complex, but errors can still be detected and corrected using a bounded locality, as justified below.

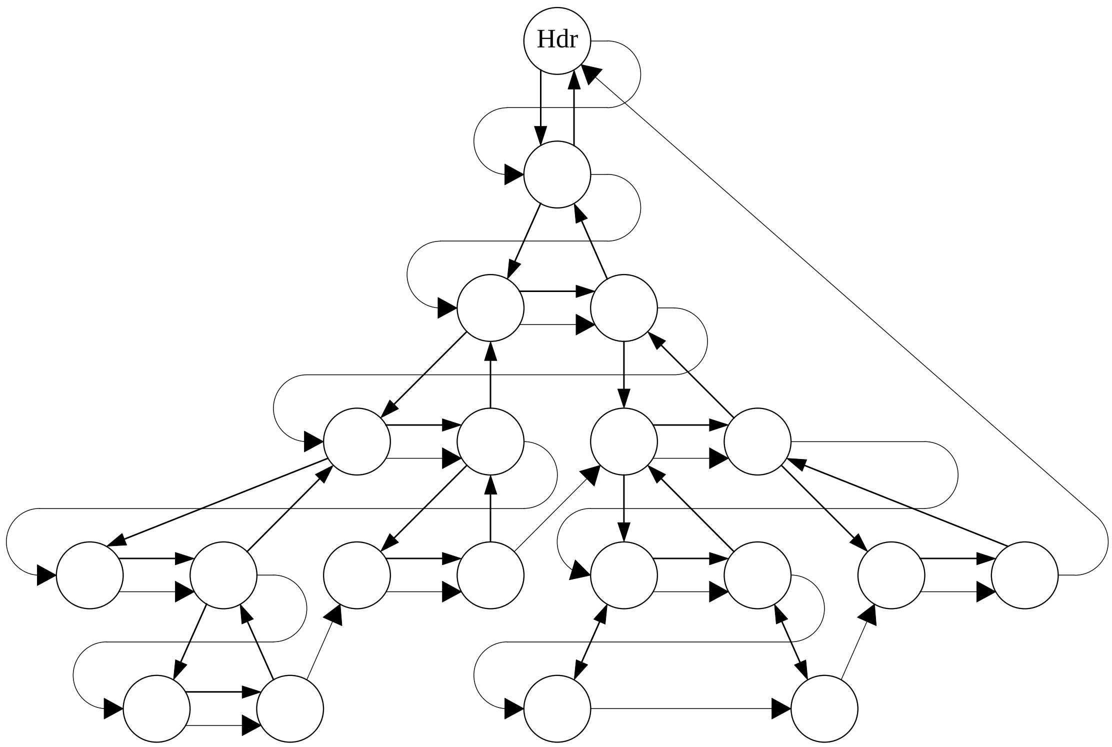


Figure 0.10. An example of a robust binary trie

Suppose that in performing the above parallel traversal we have arrived at some link $N_x \cdot c$ and some arc $N_w \cdot a$ and wish to determine if $N_x \cdot c$ is null. The value of this link and the identifier $N_x \cdot id$ provide two distinct votes to assist in resolving this issue. In cases where these two votes agree we can conclude that both votes are correct since at most one error is anticipated in either vote. Otherwise one vote contains an error implying that we can assume that a bounded number of other components are correct.

In particular, $N_w \cdot a$ must be correct. If $N_w \cdot a$ addresses N_H, the suspect link must therefore be null. So suppose that $N_w \cdot a$ does not address N_H, and thus does not address a previously trusted node. Then $N_x \cdot l$ is null if and only if $N_w \cdot a$ does not address a descendant (the first child) of N_x, while $N_x \cdot r$ is null if and only if $N_w \cdot a$ does not address a node (the following sibling) having the same parent as N_x. This is shown in Figure 7.11.

Figure 0.11. Examples for which $N_x \cdot l \neq \emptyset$ and $N_x \cdot r \neq \emptyset$

Consider the nodes visited when following the path $N_w \cdot a \cdot r^m$, given that this path terminates prematurely if N_H is encountered or if N_x is encountered when attempting to determine if $N_x \cdot r$ is null. Since the correction locality contains some error not occurring in $N_w \cdot a$, we can assume that all pointers in this path are correct. The link $N_x \cdot l$ is therefore null if and only if N_x is not encountered on this path, while the link $N_x \cdot r$ is null if and only if the parent of N_x is not encountered. Having determined if the suspect link is null, the offending link or identifier can be trivially corrected.

Since null links can be distinguished from non-null links whenever at most one error occurs in a locality, the links and arcs can be successfully traversed in parallel, given that no locality contains more than one error. Any remaining discrepancy encountered in such a parallel traversal implies that the last link and arc pointer examined should agree, but don't. All that is required of the correction algorithm upon detecting such a discrepancy, is that it identify the pointer in error, and assign it the value of the other pointer, before continuing.

Suppose that the discrepancy occurs between $N_x \cdot l$ and $N_w \cdot a$. Then we wish to determine which pointer addresses the first child of N_x. If either pointer addresses N_x then this pointer is clearly in error. So assume otherwise. Then as before, follow the two paths $N_x \cdot l \cdot r^m$ and $N_w \cdot a \cdot r^m$. If N_x is not the first trusted node on one such path then this path must be incorrect. So assume otherwise. Then both $N_x \cdot l$ and $N_w \cdot a$ address descendents of N_x, and the paths $N_x \cdot l \cdot r^m$ and $N_w \cdot a \cdot r^m$ merge at some node, N_y, prior to arriving back at N_x, since N_x is addressed by only one right link occurring in an untrusted node.

Since $N_x \cdot l \neq N_w \cdot a$, at least one path must visit a node, N_z, prior to arriving at N_y. This path is correct if N_z is the previous sibling of N_y, and incorrect if N_z is the last child of N_y. This is shown in Figure 7.12. But we can easily determine if N_z is a child of N_y by following the pointers $N_y \cdot l \cdot r^m$. Thus the first child below any node can always be identified.

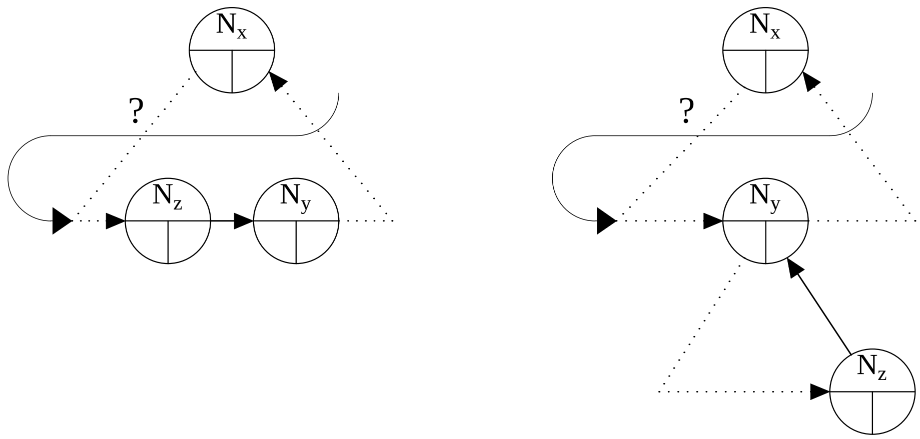


Figure 0.12. Both $N_x \cdot l$ and $N_w \cdot a$ address descendants of N_x

Now suppose that the discrepancy occurs between $N_x \cdot r$ and $N_x \cdot a$ and let N_p be the trusted parent of N_x. Follow the paths $N_x \cdot r^m$ and $N_x \cdot a \cdot r^m$ and reject any path that fails to visit N_p or visits some trusted node prior to arriving at N_p. So assume that both paths first visit the trusted node N_p. Then as before both paths must merge at some node N_y. Perform the operation described above to determine if either path visits a descendent of N_y, and reject any such path. If neither path is rejected then the correct path contains the earlier sibling of N_x, and therefore the longer path is correct. Having identified the erroneous path, we can determine if $N_x \cdot r$ or $N_x \cdot a$ is in error, and thus can correct this pointer. Thus, all pointers and identifiers in the above trie structure are locally correctable.

40. Conclusions

The above structures are of considerable interest for two reasons. Firstly, it is hoped that these structures have some practical application. Secondly, the existence of these structures significantly enlarges the set of known locally-correctable robust storage structures, and this is of benefit to those working in this field.

Of the various tree structures presented in this dissertation, the locally correctable check-summed tree structure is most easily implemented and used, and involves the least overhead when being accessed. The locally correctable binary search tree, presented in this chapter, is also not difficult either to implement or to use, and typically involves only a small overhead.

The locally correctable AVL tree has two weaknesses. Firstly, even though the AVL tree is typically a constrained binary search tree, we have not used keys to enhance the robustness of this structure. Secondly, we have produced a binary tree which has more constraints imposed on it than we might have wished, and this seems unfortunate.

The m-ary trie structure is the first unconstrained locally-correctable m-ary tree structure to be presented. The structure is specifically designed to be efficient in its use of space, and can, if desired, be used to represent a constrained binary tree in which links have their standard interpretation.

Results pertaining to the behaviour of the sibling-linked tree are presented in Appendix E, and F. Correction algorithms have not yet been implemented for the other trees presented in this chapter, and these structures appear too complex to be modelled accurately using the techniques described in the next chapter.

Chapter VIII

Mathematical models

41. Introduction

Having designed a robust storage structure, the intrinsic properties of this storage structure are naturally of some interest. Similarly, having designed a correction procedure for such a robust storage structure, the behaviour of this procedure is of interest.

Historically, the robustness of a storage structure was typically considered a function of its global detectability and global correctability. Although this function was never defined explicitly, it was not difficult to claim informally that a storage structure which was $(n + 1)$-correctable was more robust than a storage structure which was only n-correctable. This was obviously rather simplistic, and ignored many important issues discussed in Chapter 2 and Chapter 3.

In Chapter 4 this simple model for describing the robustness of a storage structure breaks down. We could perhaps provide upper and lower bounds on the number of errors that can be corrected by a local correction procedure when operating on a locally correctable storage structure, but neither bound realistically describes the robustness of such a structure. At the very least, given an arbitrary instance state of some specific size, which is known to contain some particular error distribution, we would like to know the probability that these errors are locally correctable, and in cases when they are not, we would like to know the probability that these errors disconnect the instance state being examined.

More generally, given an arbitrary storage structure and a known error distribution, we would like to know the expected number of nodes that can be traversed prior to encountering a disconnected node, and the expected number of nodes that can be traversed by a correction algorithm before failing. We may also wish to know the variance in these measures. Such measures can be used to directly compare different storage structures, and assist in identifying appropriate sizes for instance states.

Previously, such issues could only be explored by resorting to empirical studies of the robustness of instance states, such as presented in Appendix C. This approach was unsatisfactory for a number of reasons. Firstly, any results obtained pertained to specific, typically small, instance states of robust storage structures, and provided little information about the behaviour of algorithms operating on different instance states, or related robust storage structures. Secondly, these results were subject to statistical error, and this made the results difficult to compare or interpret accurately. Thirdly, prior to being able to perform such studies, a considerable amount of software needed to be developed, tested and debugged, and despite every effort to produce correct code, it was difficult to claim that empirical results were not biased by undetected errors. Finally, considerable computing resources were required in order to produce statistically significant results.

In this chapter, combinatorial and Markov models will be considered, which may be used to study the robustness of locally correctable storage structures. The combinatorial model provides general insights into the behaviour of a 1-local-correction procedure, when operating on an arbitrary storage structure known to contain a specific number of errors. The Markov models are very much more detailed and may be used to investigate the behaviour of many specific robust storage structure algorithms, when errors are assumed to occur with some fixed set of probabilities.

42. Constant error model

Historically, it was assumed that some maximum number of errors occurred in a storage structure instance, and therefore, when studying the behaviour of global correction routines, some constant number of errors was randomly placed in the instance being corrected. Later, when studying the behaviour of local-correction procedures, a constant number of errors continued to be introduced during any one experiment.

This error distribution can be trivially modelled by translating correct instance states into some suitably chosen sequence of components, some constant number of which contain errors. Unfortunately, this model needs further clarification when considering the behaviour of a correction procedure, since correction procedures can examine components not in the original correct instance, and can examine single components in many different contexts.

It is not difficult to ignore the fact that correction algorithms examine components not described within our model, since all errors are assumed to occur within our model, and it is these errors that are of interest. There are at least two alternative simplifying assumptions that can be made about the way in which an r-local-correction procedure, Ψ, visits components.

We can assume that Ψ produces linearisations in which the order of components is invariant, and that therefore any linearisation produced by Ψ is merely an initial subsequence of the linearisation emitted when operating on the original correct instance state. This will be called the *unperturbed* constant error model, and can be modelled mathematically by assuming that Ψ operates on an sequence of *n* components, some *e* of which contain errors. If we make the further simplifying assumption that there are always exactly *m* untrusted components in any correction locality, then we can assert that the locality constraint is violated, if and only if some subsequence of *m* components, within this sequence of *n* components, contains more than *r* uncorrected errors.

Alternatively, we can assume that having made a change to a component in a linearisation, subsequent components are visited in an order which is independent of the order in which these components were previously visited. One method of approximating this assumption mathematically, is to assume that after each error is corrected, remaining errors are redistributed throughout the possibly erroneous components. This will be called the *perturbed* constant error model.

Neither of the above models reflect accurately the effect that errors will have on a local correction procedure, since they represent rather extreme assumptions. However, the unperturbed constant error model seems somewhat more realistic than the perturbed constant error model, and fortuitously this model is a little easier to work with. In this section we present some results that pertain to this error model. No results are presented for the perturbed constant error model.

Lemma 0.1

There are exactly $f(n, m, e) = \binom{n - (m-1) * (e-1)}{e}$ ways in which $e > 0$ errors can be inserted into a sequence of n components, so that no subsequence of m components contains more than 1 error.

Proof

The proof proceeds by induction. In the base case when $e = 1$ this single error can occur in any of the n components. Thus $f(n, m, 1) = n = \binom{n}{1}$ as desired. So assume that e errors occur in the sequence of n components, and that the proposed formula is correct for any smaller number of errors in the sequence.

Let the earliest error in the sequence occur at c_i, where the location, i, of c_i ranges between 0 and $n - 1$. Then since none of the other errors occur in the subsequence of m components beginning at c_i, we have the recurrence relation $f(n, m, e) = \sum_{i=0}^{n-1} f(n - m - i, m, e - 1)$.

Replacing f by the proposed formula we wish to show inductively that $\binom{n - (m - 1) * (e - 1)}{e} = \sum_{i=0}^{n-1} \binom{n - m - i - (m - 1) * (e - 2)}{e - 1}$. But $\sum_{i=0}^{n-1} \binom{n - m - i - (m - 1) * (e - 2)}{e - 1}$ $= \sum_{i=0}^{n-m-(m-1)*(e-2)} \binom{i}{e - 1} = \binom{n - m - (m - 1) * (e - 2) + 1}{e}$ by a basic combinatorial identity [62]. This is simply $\binom{n - (m - 1) * (e - 1)}{e}$ as desired.

Corollary 0.1

The probability $P(X \geq e)$ that at least e errors can be inserted into a sequence of n components so that no subsequence of length m contains more than one error is $\binom{n - (m - 1) * (e - 1)}{e} \Big/ \binom{n}{e}$.

Graphs of the above function for representative values of n, m, and e are presented in Appendix D. The above result is a generalization of the result for $f(n, 2, e)$ which appears to have first been presented in [71]. A corresponding result for the case when no error occurs exactly two components after a previous error is presented in [77], and this is generalized for the case when no error occurs exactly n components after a previous error in [69105].

We have been unable either to find, or to deduce, the more general formula for the case when at most r errors are allowed in any subsequence of m components. However, when both m and r are small, the patterns which are forbidden to occur in any subsequence of m components can be enumerated, and a recurrence relationship providing specific solutions of this problem then derived [6160]. Unfortunately, this derivation requires that we invert a square matrix having as many rows and columns as there are illegal patterns. Since the elements of this square matrix are themselves polynomials in two variables, this inversion rapidly becomes infeasible for even moderate values of m and r.

Lemma 0.2

The expected number of errors, E(e), that can be inserted into a sequence of n components, so that no subsequence of m components contains more than one error is $\sum_{e=1}^{n} \binom{n-(m-1)*(e-1)}{e} \Big/ \binom{n}{e}$. Graphs of this function for representative values of n and m are presented in Appendix D.

Proof

By the corollary to Lemma 8.1 the probability $P(X \geq e)$ that at least e errors can be inserted into n sequential components so that no subsequence of length m contains more than one error is $\binom{n-(m-1)*(e-1)}{e} \Big/ \binom{n}{e}$. The probability $P(X = e)$ that exactly e errors can be inserted is $P(X \geq e) - P(X \geq e+1)$. By definition, the expected number of errors E(e) that can be introduced is $\sum_{e=1}^{\infty} e * P(X = e)$. This is $\sum_{e=1}^{n} e * \binom{n-(m-1)*(e-1)}{e} \Big/ \binom{n}{e} - \sum_{e=1}^{n-1} e * \binom{n-(m-1)*e}{e+1} \Big/ \binom{n}{e+1}$. Cancelling common terms in this expression gives $\sum_{e=1}^{n} \binom{n-(m-1)*(e-1)}{e} \Big/ \binom{n}{e}$.

Theorem 0.1

If E(e) is the expected number of errors that can be introduced into a sequence of n components, such that no more than one error occurs in any subsequence of size m, then $E(e) \to \infty$ as $n \to \infty$.

Proof

It has been established in Lemma 8.2 that $E(e) = \sum_{e=1}^{n} \binom{n-(m-1)*(e-1)}{e} \Big/ \binom{n}{e} = \sum_{e=1}^{\left\lfloor \frac{n+m-1}{m} \right\rfloor} \prod_{i=0}^{e-1} \frac{n-(m-1)(e-1)-i}{n-i}$. So $E(e) \geq \sum_{e=1}^{c} \prod_{i=0}^{e-1} \left(1 - \frac{(m-1)(e-1)}{(n-i)}\right)$, for any arbitrary constant $c \leq n/m$. But $\lim_{n\to\infty} \prod_{i=0}^{(e-1)<c} \left(1 - \frac{(m-1)(e-1)}{(n-i)}\right) = 1$, since $\lim_{n\to\infty} \left(1 - \frac{(m-1)(e-1)}{(n-i)}\right) = 1$ when i and e are bounded above by the constant c and m is a given constant. Thus $\lim_{n\to\infty} E(e) \geq \sum_{e=1}^{c} 1 = c$. But c can be made arbitrarily large as $n \to \infty$. Thus $\lim_{n\to\infty} E(e) = \infty$.

Corollary 0.2

The expected number of errors that can be corrected by a local correction procedure, operating on a sufficiently large instance state, exceeds any given bound.

Proof

All local-correction procedures perform 1-local-correction. Lemma 8.2 provides a lower bound on the expected number of errors that can be corrected by a 1-local-correction procedure, when operating on an instance state containing n components, given that this procedure uses localities containing at most m untrusted components. Theorem 8.1 shows that, by increasing n, this lower bound can be made arbitrarily large.

Theorem 0.2

There are $N(n, m, e) = \binom{m}{e} + (n-m) * \binom{m-1}{e-1}$ ways in which e errors may be inserted into a linearisation containing n components so that all e errors occur in some locality of size m.

Proof

If an error occurs in the first component of the linearisation then e errors occur in a locality of size m if and only if all errors occur in the first m components. There are $\binom{m-1}{e-1}$ ways in which the other $e-1$ errors can be placed in the next $m-1$ components. Conversely, if the first component in the linearisation is correct, then the number of ways that the e errors can be placed in the linearisation so that they all occur in some locality of size m is $N(n-1, m, e)$. So $N(n, m, e) = N(n-1, m, e) + \binom{m-1}{e-1}$. Clearly, $N(m, m, e) = \binom{m}{e}$. Thus $N(n, m, e) = \binom{m}{e} + (n-m) * \binom{m-1}{e-1}$.

Theorem 0.3

The number of ways that e errors can be distributed in a linearisation containing n components so that no more than r errors occur in the first erroneous locality of size m is

$$\sum_{j=0}^{r-1} \binom{m-1}{j}\binom{n-m+1}{e-j}.$$

Proof

Let there be exactly j errors in the first erroneous locality, and let the first error occur at component i, where i ranges between 0 and $n-m-e+j$. Then the $j-1$ errors following the first error must occur in the next $m-1$ components of the linearisation, while the remaining $e-j$ errors must occur in the last $n-m-i$ components of the linearisation. The total number of ways of placing exactly j errors in the first erroneous locality is therefore

$$\sum_{i=0}^{n-m-e+j} \binom{m-1}{j-1}\binom{n-m-i}{e-j} = \binom{m-1}{j-1}\sum_{i=e-j}^{n-m} \binom{i}{e-j} = \binom{m-1}{j-1}\binom{n-m+1}{e-j+1}.$$

Now allowing j to range over its possible values we deduce that the total number of ways that at most r errors can occur in the first erroneous locality is exactly $\sum_{j=1}^{r} \binom{m-1}{j-1}\binom{n-m+1}{e-j+1} =$

$$\sum_{j=0}^{r-1} \binom{m-1}{j}\binom{n-m+1}{e-j}.$$

43. Markov models

Instead of assuming that some set number of errors occurs in an instance state, we may assume that each component examined has a constant probability of being in error. Then, we can construct discrete Markov models which describe various abstract properties of the robust storage structure being investigated, and in particular can predict with considerable accuracy the behaviour of various types of algorithms that operate on such structures.

A Markov model [82] is described by a collection of states, and transitions between these states. Given any two states, S_1 and S_2, which need not be distinct, there is some constant probability that a transition will occur from state S_1 to S_2, given that we are currently in state S_1. The probabilities of all transitions from any state, S_i, sum to unity. Having selected some start state, we wish to determine the probability that we arrive in some other state and may also wish to determine the expected number of transitions of a certain type that occur before arriving in this selected state. Such selected states are called final states, and allow no transition to any other state.

We will use Markov models to simulate the occurrence of specific events, such as the discovery of an error, the correction of an error, or the traversal of a node, and will wish to predict the number of such events that occur prior to some catastrophic event such as the discovery that the instance state being examined is disconnected or not correctable.

One clever way of counting the number of events of a particular type, which occur in a Markov model, involves transforming transition probabilities into generating functions, by multiplying transition probabilities by dummy variables. Specifically, suppose that during a given transition an event, X, occurs, which we wish to count. Then we multiply the probability of this transition by some dummy variable, x. Later, having deduced a formula for the probability of a move (potentially via many steps) from state S_1 to state S_2, we can extract the probability that X occurs exactly n times during this move, by setting all of the dummy variables except x to 1, expanding the derived formula as a polynomial in x, setting all but the coefficient of x^n to zero, and then setting x to 1. Various other types of result, such as the probability that an event occurs at least or at most n times, or that multiple events occur, can easily be deduced by using this technique.

Typically, we will wish to derive a generating function that encapsulates, for the events of interest, the expected number of such events that occur between beginning in the starting state and terminating in a final state. The rules for deriving such a generating function are straightforward. While we have more than one generating function describing a transition from state S_1 to S_2, replace these generating functions by their sum. This is equivalent to summing the probabilities of independent mutually exclusive but identical events, so that the overall probability of that event can be deduced. Having eliminated duplicate transitions, select some state S_1 which allows no direct transition to itself, and explode the transitions into S_1 with those out of S_1, by multiplying each such pair of transitions producing new transition rules for moves which essentially pass through S_1 but no longer explicitly reference S_1. This is equivalent to multiplying the probabilities of independent events when wishing to determine the probability that both occur. Eventually, we will either reduce the Markov model to a single transition from a start state to a final state, or will be obliged to remove a transition which moves from some state S_1 directly back to S_1. If the generating function associated with this loop is G, then multiply the probability of other transitions from S_1 by $1/(1 - G)$, and remove the loop transition. Thus, the sum of all output transition probabilities from S_1 remains 1. The above operations are then resumed until only one transition remains.

Example 0.1

Suppose that we wish to study a standard double-linked list, and are particularly interested in the connectivity of such a structure. Then, proceeding empirically, we might write an algorithm which, having been provided with an exact description of the errors introduced into the data memory state, attempted to traverse an instance state of such a structure, counted errors encountered and nodes traversed, and reported these counts when disconnection was detected or the traversal was complete.

Let us assume that the probability of an error being observed is p, and that we count the number of errors encountered in the dummy variable y. We will also count the number of nodes traversed using the dummy variable z. The Markov model which performs the desired simulation is presented in Figure 8.1. The Markov model starts in state S_0, and while no error is encountered in a forward pointer, proceeds forward through the instance state, counting the number of nodes traversed. When an error is encountered the Markov model moves to state S_1, in the process counting the error observed, and then proceeds to move backwards from the headers of the instance, continuing to accumulate the count of the number of nodes visited. When an error is encountered in a back pointer, the model enters its final state since the instance state being examined has been discovered to be disconnected.

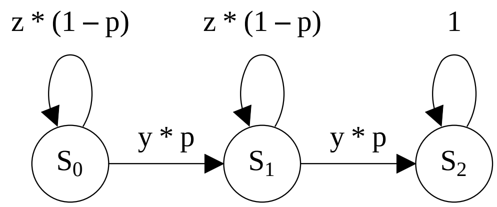


Figure 0.1. A Markov model for connectivity in a double linked list

The generating function, G, describing the behaviour of the Markov model in moving from S_0 to S_2, can easily be seen to be $G = \left(y * p/(1 - z * (1 - p))\right)^2$. Suppose that we wish to study the connectivity of a double-linked list containing 100 data nodes, given that components contain errors with probability p. Then, presumably using a symbolic manipulation package such as Maple [33], we set $y = 1$, expand G as a Taylor series around z, and truncate this series by eliminating terms of order greater than z^{99}. Setting $z = 1$, we derive a function of p which expresses the probability that disconnection will be discovered, prior to traversing all 100 data nodes. By allowing p to range between 0 and 1, we can then study the connectivity of a double-linked list without necessarily conducting any empirical studies.

Now suppose that we wish to know the expected number of nodes E(z) which can be traversed by such a procedure, before detecting that the instance state is disconnected. Then by setting $y = 1$, differentiating G with respect to z, and then setting $z = 1$, we replace terms of the form $a_b * z^b$ in G by $b * a_b$. Since a_b is the probability that our algorithm observes exactly b nodes before terminating, the resulting expression $E(z) = \sum_{i=0}^{\infty} b * a_b$ is the expected number of nodes visited. This easily derived formula is useful, since it describes the robustness of a double-linked list in terms of its size.

We may also easily determine the variance V(z) of the number of nodes visited by this traversal. This variance is $V(z) = E(z^2) - E^2(z)$. Consider setting $y = 1$ as before, differentiating G with respect to z twice, and then setting $z = 1$. Then we replace terms of the form $a_b * z^b$ in G by $b * (b - 1) * a_b$. Therefore, the resulting expression $\sum_{i=0}^{\infty} b * (b - 1) * a_b = E(z^2) - E(z)$. Adding E(z) to this expression produces an expression for $E(z^2)$, and subtracting the square of E(z) from this expression produces the desired formula for V(z).

44. Regular linked lists

A large number of regular linked lists have been analyzed both empirically and by using Markov models. To remain compatible with the Markov models, the empirical studies assumed that there was some fixed probability that an error occurred in any component. Thus the empirical studies differed from earlier studies, presented in Appendix C, which assumed that an instance state contained some specified number of errors. The results of these studies are presented in Appendix E and Appendix F.

Somewhat surprisingly, it was not possible to produce a Markov model which allowed the connectivity of a spiral(k) storage structure to be studied precisely. Suppose that the marked pointers in Figure 8.2 contain errors. Then N_1 is connected if there is a path from N_3 to N_2. However, as shown in Figure 8.3 there may be a path from N_3 to N_2 if and only if there is a path from N_6 to N_5. Thus, the task of finding a path to N_1 may involve recursion. Therefore, determining that N_1 is connected may involve examining an unbounded number of pointers, an unbounded number of which contain errors. This problem cannot therefore be described by a Markov model, since a Markov model has only a finite memory in which to record the errors which have so far been encountered.

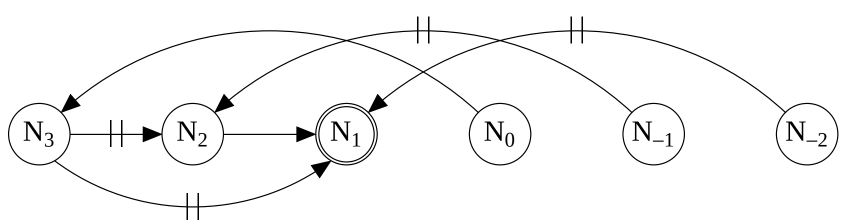


Figure 0.2. N_1 is connected if path from N_3 to N_2

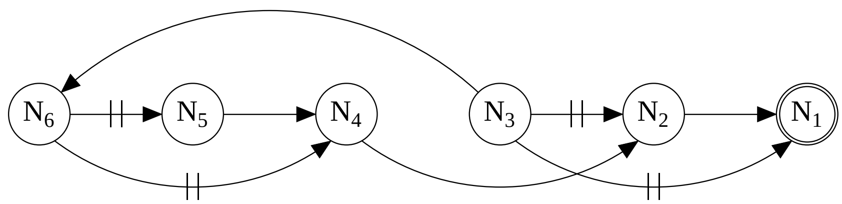


Figure 0.3. N_1 is connected if path from N_6 to N_5

The simplest locally correctable linked list analyzed was a double-linked list containing a forward pointer, a back pointer, and two additional checksum components. The error-correcting code presented in Appendix A1 was used to produce a 1-local-correctable structure, in which any one error in a node was correctable while any two errors were not. The Markov model describing the behaviour of a correction algorithm operating on this structure is presented in Figure 8.4. Errors occur with probability p. The dummy variable y indicates that the component being examined is in error. The dummy variable z indicates that local correction is advancing to the next node in the linked list.

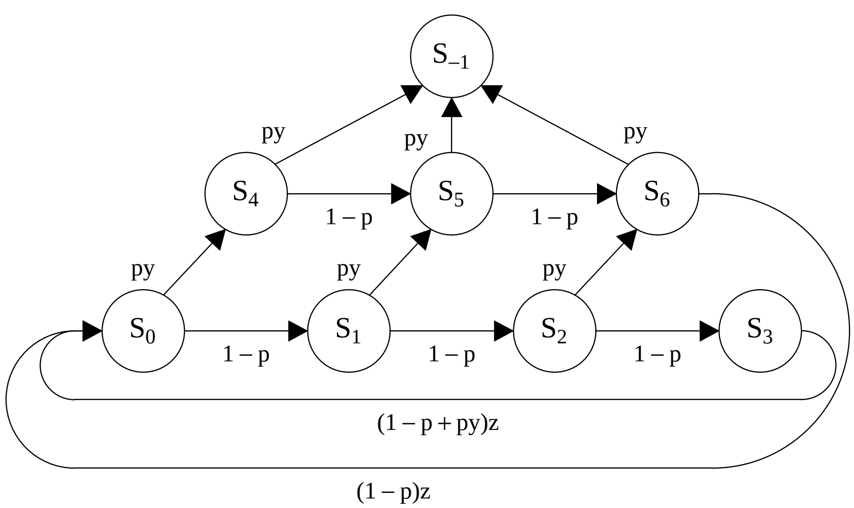


Figure 0.4. Correction of one error in any node containing four components

When considering more complex local-correction procedures the Markov models must typically use additional states to ‘‘remember’’ not only that an error has been encountered, as is the case in the example above, but also to record the exact set of components in the current locality known to contain errors, because errors occurring in one locality are often carried forward into a bounded number of subsequent localities, and thus can have subtle effects on the behaviour of the local-correction algorithm being modelled.

45. Tree structures

In order to construct Markov models which describe structures which are less regular than linked lists, we must provide not only the probability that an error occurs in any given state, but also the probability that we next visit a node of a particular type. For example, the behaviour of an algorithm traversing a tree structure is in part dependent on the distribution of full nodes, incomplete nodes, and leaf nodes.

When attempting to predict the behaviour of such an algorithm operating on a tree containing n nodes, we begin by determining the expected number of full nodes, incomplete nodes, and leaf nodes, in a tree containing n nodes. Then we can make the simplifying assumption that the probability of arriving at a node of a particular type is merely the expected number of such nodes in the tree divided by n.

We present below the expected number of leaf, incomplete, and full nodes in a binary search tree that is created by the insertion of keys in a random sequence. It seems clear given the importance of these results, and the ease with which they can be obtained, that these results are not new. However, these results are relevant and are easier to derive than to cite.

Lemma 0.3

If a binary search tree is created by the random insertion of $n \geq 2$ distinct keys, the expected number of leaf nodes is $(n+1)/3$, the expected number of incomplete nodes is the same, and the expected number of full nodes is $(n-2)/3$.

Proof

Assume that some set of n distinct keys is to be inserted into an initially empty binary search tree. There are n! possible permutations of n keys and therefore there are n! possible binary search trees which may be constructed, not all of which are distinct. Denote the total number of full nodes in all such trees of size n by f_n, the number of incomplete nodes by h_n, and the number of leaf nodes by l_n. Now consider how these variables change when the forest of all $(n+1)!$ trees containing $n+1$ nodes is constructed.

Since there are $n+1$ times as many trees, and full nodes in trees of size n remain full if an additional node is inserted in the tree, it is only necessary to identify the number of new full nodes which occur as a result of inserting the last node into each position in each tree in the forest. But this is merely h_n, since in exactly one tree of size $n+1$ will a particular incomplete node in a tree of size n become a full node. Thus $f_{n+1} = (n+1) * f_n + h_n$.

Applying the same reasoning to incomplete nodes, $h_{n+1} = (n+1) * h_n - h_n + 2 * l_n$, since overall h_n incomplete nodes will become full nodes while each leaf node has two ways of becoming an incomplete node. Finally, since a leaf node is added to each of the $(n+1)!$ trees in the forest of trees containing $n+1$ nodes, $l_{n+1} = (n+1) * l_n - 2 * l_n + (n+1)!$.

The equation $l_{n+1} = (n-1) * l_n + (n+1)!$ has a particular solution [94] $l_n = (n+1)!/3$ and this solution is exact for $n \geq 2$, since there are a total of two leaf nodes in the two possible trees containing two nodes. The equation $h_{n+1} = n * h_n + 2 * (n+1)!/3$ also has a particular solution $h_n = (n+1)!/3$ which is exact for $n \geq 2$. The expected number of leaf and incomplete nodes in one of these n! trees is therefore $2 * (n+1)/3$. Since each tree contains a total of n nodes, the expected number of full nodes is therefore $(n-2)/3$.

46. Conclusions

All of the correction procedures studied had finite Markov models that described exactly the set of errors which would cause these correction procedures to fail, given that multiple errors did not conspire to assist correction algorithms by masking the presence of errors.

In Appendix E and Appendix F many Markov models are presented, and the results of using these Markov models compared with empirical studies. Typically, the results produced by using Markov models very clearly correspond almost exactly to empirical results. Indeed, in the few cases where noticeable differences did occur, faults were subsequently discovered in the empirical studies. Had we not had such accurate tools for predicting the behaviour of the algorithms being studied, it seems likely that some of these faults would not have been found.

Although there are obvious benefits associated with using Markov models, some questions are not easily resolved by using Markov models. For example, it is relatively easy to construct models which count cases when correction algorithms operating on linked lists are misled, as a result of carefully selected values being placed in specific components. However, such models communicate nothing, unless it is also possible to determine the probability that errors do cause specific values to be placed in selected components.

Another problem with using Markov models to precisely describe the behaviour of more complex systems is the sheer size of the generating functions obtained. Although it was possible to produce a generating function describing the precise behaviour of the helix(3) correction algorithm described in Chapter 6, the resulting expression was more than 100,000 characters long, and even the problem of determining if this expression could be simplified proved to be computationally infeasible using available computing resources.

Chapter IX

Conclusions and further work

47. Conclusions

In Chapter 1 we presented a number of issues which we wished to explore in the hope that their resolution would lead to the discovery of good methods of implementing and correcting robust storage structures.

We began by suggesting that the specifications that correction routines operate under be reviewed, in the hope that better specifications might challenge rather than blinker the designers of robust storage structures.

In Chapter 3, we therefore stressed the role that the Valid State Hypothesis can play in the development of correction procedures, and then extended the desired behaviour of global correction algorithms, so that in cases where the number of errors encountered exceeded the number that could necessarily be corrected, these algorithms selectively performed correction, and otherwise reported failure. We then established upper and lower bounds on the selective correctability of arbitrary storage structures. Using these bounds, we showed how in a 1-global-correctable mod(2) linked list, 2 errors could always be corrected, provided that these two errors did not disconnect the instance state being examined.

In Chapter 4, we explored the issue of local correction, and showed that distinct votes could not always be used to detect errors in locally detectable structures. We therefore developed the notion of local connection functions, and showed that if a storage structure had a 2r-local-linearisation function which was r-local-connected, then the storage structure was at least r-locally-correctable. We then proposed the development of algorithms that performed selective-local-correction, and established upper and lower bounds on the selective local correctability of an arbitrary storage structure. In Chapter 5 and 6, we showed how this theory could be readily applied to the mod($k \geq 3$) linked list, and the helix($k \geq 3$) linked list.

Our next area of concern addressed the underlying assumptions about the nature of errors introduced into the data memory state. There is only a limited amount of information available about the frequency and types of faults that, in practice, lead to errors in robust storage structures, and we were therefore keen to address this issue. However, it soon became apparent that such questions were not likely to produce any definitive results, and this avenue for research was therefore abandoned. Rather more constructively, in Chapter 8, we presented two new theoretical models for studying the effects of introducing errors into robust storage structures, and derived a number of new results that pertained to these error models.

We then indicated that there was a need to develop additional guidelines, indicating the properties that robust storage structures must have if they are to facilitate certain types of error correction. The theoretical results presented in Chapter 3 and Chapter 4 address this issue.

We then indicated that existing robust storage structures should be carefully reviewed, in the hope that such structures might be corrected more efficiently or accurately using techniques not originally considered when designing the structure. This led to an algorithm, presented in Chapter 3, for performing 2-selective-global-correction on mod($k \geq 2$) linked lists, and an algorithm, presented in Chapter 5, for performing 2-selective-local-correction on mod($k \geq 3$) linked lists. The helix(k) linked list, presented in Chapter 6, also arose as the result of reviewing the existing spiral(k) linked list structure. As shown in Appendix C1 and Appendix C2, these algorithms are significantly better than previous algorithms when correcting similar linked list structures.

We also expressed a desire to develop a number of new robust storage structures so that these could be evaluated and compared with existing storage structures. It was hoped that, in the process, new ideas and better methods of introducing redundancy into storage structures would be discovered. In Chapter 4 we presented a checksummed binary tree, in Chapter 6 we presented the helix(k) linked list, and in Chapter 7 we presented three new tree structures, all of which were locally correctable.

Finally, we suggested that empirical studies provided only limited information about the behaviour of robust storage structures, and associated correction procedures, and that therefore there was a need to develop statistical models which, when presented with various parameters, accurately predicted the behaviour of robust storage structures, and their associated correction routines.

In Chapter 8 we therefore discussed the constant error model and showed, using this model, that the expected number of errors that could be corrected by a local correction procedure, when operating on a sufficiently large instance, exceeded any bound. We also developed results indicating the probability of performing 1-local-correction under a variety of different constraints, and presented these results graphically in Appendix D.

We then proposed using Markov models to simulate the introduction of errors into robust storage structures, so that we might predict the consequences of such errors. As shown in Appendix E and Appendix F, the results of using Markov models to predict the consequences of errors correlated almost exactly with the corresponding results obtained from empirical studies. This is obviously very exciting, and promises to make the evaluation of some robust storage structures very much easier, while also making such evaluations more accurate and complete.

Thus, this thesis has contributed to storage structure error correction in four main ways. Firstly, a number of new theoretical results have been presented which pertain to global and local error correction. Secondly, a theory has been developed which allows global and local correction algorithms to selectively perform correction and otherwise report failure. Thirdly, using these new theoretical results, better correction algorithms have been proposed for previously correctable storage structures, and a number of new robust storage structures designed. Finally, it has been shown that probability theory and Markov models are powerful tools for studying the properties of many robust storage structures.

48. Further work

Most of the unsolved problems pertaining to error detection and correction seem intrinsically hard, but are sufficiently important to merit further research.

The relationships between storage structures and local-linearisation functions are not well understood, and in particular it remains unclear how good local-linearisation functions can be derived from a given storage structure specification. A number of related issues should be addressed in the hope of resolving this question.

What are the constraints that limit the local detectability of a given local-linearisation function, and how does the local detectability of a linearisation function change as modifications are made to this linearisation function? How can the exact local detectability of a given linearisation function be determined? What is the relationship between linearisation functions capable of detecting different numbers of errors in different size localities, and how does one determine which is “better”? How does one translate a linearisation function having certain attributes into one that uses the smallest possible locality while continuing to satisfy these attributes?

In Chapter 4 we proved that an r-local-connected 2r-local-linearisation function, f, had an r-local-correction function, P_f. However, the existence of P_f does not necessarily imply the existence of an efficient r-local-correction procedure which can correct at least one error in any linearisation $f(x_1)$ containing between 1 and r errors, since P_f may not be computable by a reasonable procedure. Is it possible to prove that all local-correction functions are computable by a reasonable procedure, or alternatively to provide some theory which would at least allow us to identify those local-correction functions which are?

In Chapter 4 we also proved that an r-local-correctable $(2r+1)$-local-linearisation function was $(r+1)$-selective-local-correctable. It seems clear that some linearisations which are exactly r-local-correctable are more than $(r+1)$-selective-local-correctable. What are the identifying characteristics of such linearisation functions?

More generally, what is the relationship between local correction algorithms and other types of correction algorithm? Should we be developing correction algorithms that are designed to operate ‘‘correctly’’ under worst-case scenarios, rather than developing correction algorithms which meet other objectives? Perhaps no assumption should be made about the number of errors encountered by a correction procedure, in the hope that we might develop algorithms with good probabilistic behaviour, or algorithms capable of always transforming damaged instances into the nearest correct and seemingly valid instance. Since it is not known how to construct such correction algorithms, or even if the construction of such algorithms is feasible, this would be an interesting area for further research.

Historically, the Valid State Hypothesis has been used primarily when attempting to prove the correctness of algorithms that relied on it. However, many of the algorithms presented in this dissertation actively use the assumption that the Valid State Hypothesis holds to assist in performing correction. There is therefore a need for a theory which would identify the effect that the Valid State Hypothesis has on the correctability of an arbitrary structure.

In this dissertation we assumed that correction algorithms had no external knowledge about the cause of errors, or the possible propagation of errors, and that they therefore had to assume at best some probabilistic distribution of errors within the structure being corrected. However, often the cause of errors can be externally identified, or the propagation of errors carefully controlled. It would therefore be of interest to investigate the properties of robust storage structures when subjected to such restrictive classes of errors, and to attempt to design good algorithms for handling such errors. Most notably, power failures, deadlock, and user-initiated interruption can often lead to partially completed updates within robust storage structures. How should correction algorithms attempt to correct such errors within a single robust storage structure, and more generally how should correction algorithms rectify partially completed updates applied to composite storage structures?

Methods of simulating the types of error that arise as a result of incorrect concurrency control should also be investigated. It would be of interest to develop code that simulated this type of error, and to consider the possible consequences of allowing error correction to be performed concurrently with other forms of update.

Finally, the most ambitious researchers might attempt to unify the theoretical results which pertain to robust storage structures with the theory of error-correcting codes, and/or the theory of detection and correction of errors in hardware components and systems. There are many superficial similarities between these fields, and much can be learned from studying all three. In particular, if storage structure errors could be modelled using any of the existing models for hardware error detection and correction, then a huge body of accumulated knowledge would suddenly become available to the designers of robust storage structures.

APPENDIX A1

A tertiary perfect Hamming code

0	0	0	0
0	1	1	1
0	2	2	2
1	0	2	1
1	1	0	2
1	2	1	0
2	0	1	2
2	1	2	0
2	2	0	1

APPENDIX A2

A quaternary perfect Hamming code

0	0	0	0	0	2	0	0	2	1
0	0	1	1	1	2	0	1	3	0
0	0	2	2	2	2	0	2	0	3
0	0	3	3	3	2	0	3	1	2
0	1	0	1	2	2	1	0	3	3
0	1	1	0	3	2	1	1	2	2
0	1	2	3	0	2	1	2	1	1
0	1	3	2	1	2	1	3	0	0
0	2	0	2	3	2	2	0	0	2
0	2	1	3	2	2	2	1	1	3
0	2	2	0	1	2	2	2	2	0
0	2	3	1	0	2	2	3	3	1
0	3	0	3	1	2	3	0	1	0
0	3	1	2	0	2	3	1	0	1
0	3	2	1	3	2	3	2	3	2
0	3	3	0	2	2	3	3	2	3
1	0	0	1	3	3	0	0	3	2
1	0	1	0	2	3	0	1	2	3
1	0	2	3	1	3	0	2	1	0
1	0	3	2	0	3	0	3	0	1
1	1	0	0	1	3	1	0	2	0
1	1	1	1	0	3	1	1	3	1
1	1	2	2	3	3	1	2	0	2
1	1	3	3	2	3	1	3	1	3
1	2	0	3	0	3	2	0	1	1
1	2	1	2	1	3	2	1	0	0
1	2	2	1	2	3	2	2	3	3
1	2	3	0	3	3	2	3	2	2
1	3	0	2	2	3	3	0	0	3
1	3	1	3	3	3	3	1	1	2
1	3	2	0	0	3	3	2	2	1
1	3	3	1	1	3	3	3	3	0

APPENDIX B1

Pseudocode for mod(k≥3) correction algorithm

```
correct_headers();                          /* Terminate if null instance            */
for (count = 0; count < max_possible; count = count+1) {
    candidates = 0;
    for (i = 0; i < 3; i = i+1) {           /* Apply constructive votes              */
        N_x = N_{1−k+i} · b_k · f_1^{i−1};      /* For simplicity, assume N_x exists     */
        if (N_x ≠ any candidate[j]) {
            j = candidates++; candidate[j] = N_x; vote[j] = 0; }
        vote[j] = vote[j] + weight[i]; }    /* weight[i] = {1/4, 3/16, 1/16}        */
    for (i = 0; i < candidates; i = i+1) {  /* Apply diagnostic votes                */
        N_x = candidate[i];
        if (N_x · f_1 = N_0)                vote[i] = vote[i] + 1/4;
        if (N_x · b_k · f_1 = N_0 · b_k)        vote[i] = vote[i] + 3/16;
        if (N_x · b_k · f_1^2 = N_{−1} · b_k)   vote[i] = vote[i] + 1/16;
        if (N_x = any N_{0≥i>−k})    vote[i] = 0; }
    case 'Only N_a got vote > 1/2': break;
    case 'Only N_a got vote of 1/2':
        if (candidates = 1) {
            if (N_a · id bad or N_a · b_k · f_1^k ok)  abort(Target disconnected);
            break; }
        if (N_a · f_1 = N_0 and N_a = N_{2−k} · b_k and N_a = N_{3−k} · b_k · f_1) {
            if (k = 3)  abort(Target may be disconnected);
            if (N_{4−k} · b_k · f_1 = N_{3−k} · b_k)  abort(Target disconnected); }
    case 'Only N_a and N_b got vote of 1/2':
        if (N_a · f_1 ≠ N_b · f_1) {
            if (N_b · f_1 = N_0) N_a = N_b;
        } else if (N_b · b_k = N_a) N_a = N_b;
    case 'Otherwise':  abort(Target disconnected);
                                                /* N_a is target node N_1            */
    N_{1−k} · b_k = N_a; N_a · id = id; N_a · f_1 = N_0;   /* Assignments may be unnecessary  */
    if (N_a = last header) correct_count(); /* Terminate successfully                */
}
abort(Algorithm looping);
```

APPENDIX B2

Pseudocode for helix($k \geq 3$) correction algorithm

correct_headers(); /* Terminate if null instance */
for (count = 0; count < max_possible; count = count+1) {
    candidates = 0;
    for ($i = 0$; $i \leq k$; $i = i+1$) { /* Apply constructive votes */
        case 'i = 0': $N_x = N_0 \cdot b_2 \cdot f_1$; /* For simplicity, assume N_x exists */
        case 'i = k': $N_x = N_{2-k} \cdot b_k \cdot f_1$;
        case 'default': $N_x = N_{-i} \cdot b_{i+1}$;
        if ($N_x \neq$ any candidate[j]) {
            j=candidates++; candidate[j]=N_x; vote[j]=0; }
        if ($i \neq 0$) vote[j] = vote[j] + 1; } /* N_x = candidate[j] */
    for ($i = 0$; $i <$ candidates; $i = i+1$) { /* Apply diagnostic votes */
        N_x = candidate[i];
        for ($j = 1$; $j \leq k$; $j = j+1$) {
            case 'j = 1': if ($N_x \cdot f_1 = N_0$) vote[i] = vote[i]+1;
            case 'j = k': if ($N_x \cdot b_k = N_0 \cdot b_2 \cdot b_{k-1}$) vote[i] = vote[i]+1;
            case 'default': if ($N_x \cdot b_i = N_0 \cdot b_{i+1}$) vote[i] = vote[i]+1; }
        if ($N_x = N_j$, for any $0 \geq j \geq 1 - k$) vote[i] = 0; }
    case 'Only N_a has > k votes': break;
    case 'Only N_a has k votes':
        if (candidates = 1) {
            if ($N_a \cdot \mathrm{id}$ bad or $N_a = N_{2-k} \cdot b_k$ or $N_a \cdot f_1 = N_a$
            or $N_a \cdot f_1 \cdot b_k \cdot f_1^{k-1} = N_a$ without cycles) abort(Target disconnected); }
        else if ($N_a \cdot f_1 = N_0$ and $N_a = N_{1-k} \cdot b_k$ and $N_a = N_{2-k} \cdot b_{k-1}$) {
            if ($k = 3$) abort(Target may be disconnected);
            if ($N_a = N_{-1} \cdot b_3$) abort(Target disconnected); }
    case 'N_a and N_b have k votes':
        if ($N_a \cdot f_1 \neq N_b \cdot f_1$) {
            if ($N_b \cdot f_1 = N_0$) $N_a = N_b$; }
        else if ($N_a = N_{2-k} \cdot b_k$) $N_a = N_b$;
    case 'Otherwise': abort(Target disconnected);
    $N_a \cdot \mathrm{id} = \mathrm{id}$; $N_a \cdot f_1 = N_0$; /* N_a is the target node N_1 */
    for ($i = 2$; $i \leq k$; $i = i+1$) $N_{1-i} \cdot b_i = N_a$; /* Assignments may be unnecessary */
    if (N_a = last header) correct_count(); } /* Terminate successfully */
abort(Algorithm looping);

APPENDIX C1

Empirical results for double-linked list structures

C1.49. Explanation

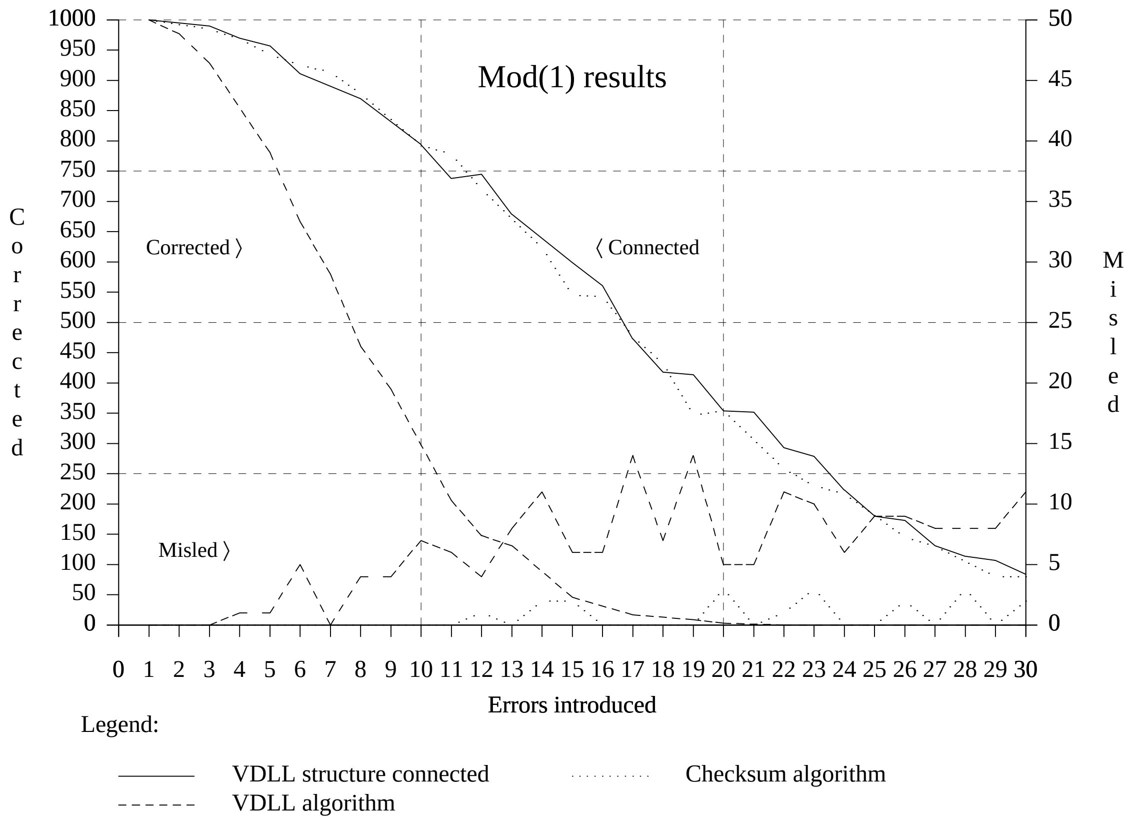
This appendix presents empirical results obtained when ‘‘random’’ errors were introduced into double-linked lists, a mod(2) structure, a mod(3) structure, and a mod(4) structure. Each instance contained 100 consecutively located nodes plus headers. Increasing numbers of pointers were randomly selected from within this instance, and modified by adding or subtracting a random number between 1 and 10. Because the instances being considered were small, the probability that errors caused disconnection was high. Because pointers were modified by a small amount, the probability that votes supported common incorrect candidates was high. This appendix is therefore somewhat pessimistic.

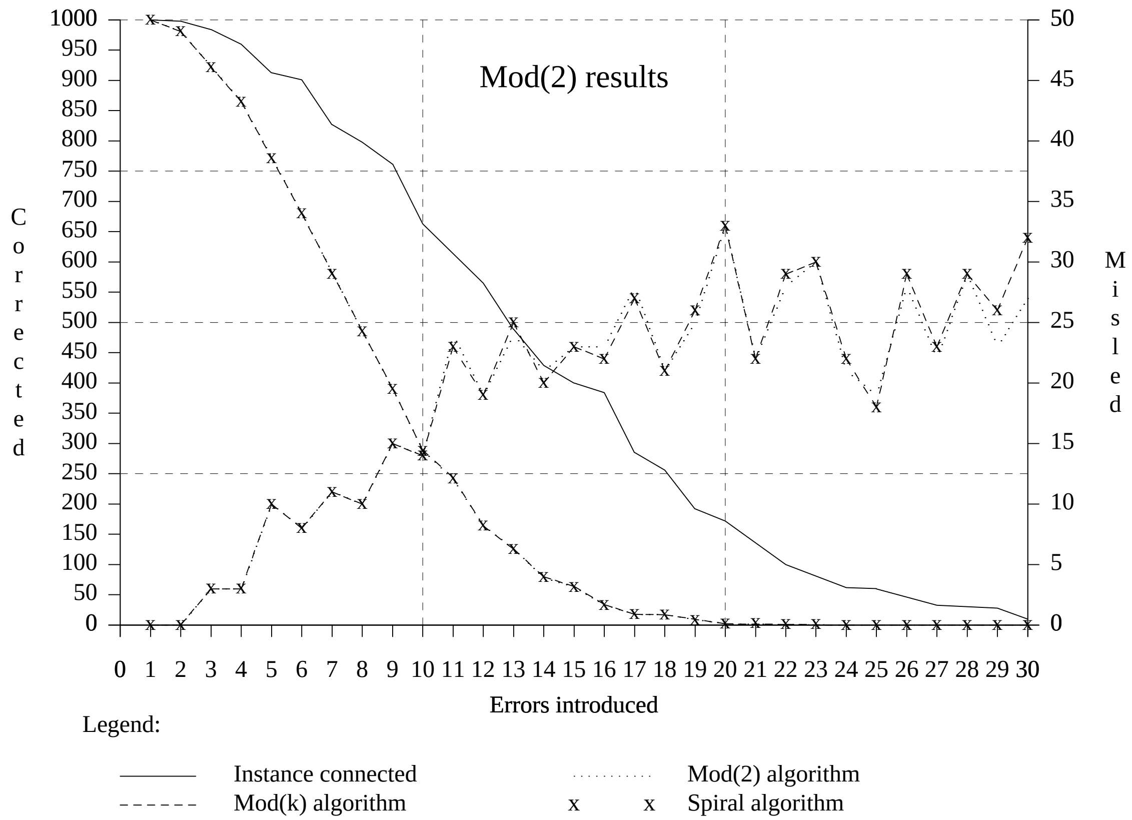
Since a standard (+1,−1) double-linked list is not locally correctable, two distinct methods were used to produce a locally correctable (+1,−1) linked list. Both methods employed a forward traversal. The first method added two checksum components, of the same size as the pointer components, to each node in the instance being corrected. A generalised perfect tertiary hamming code [25], presented in Appendix A1, then allowed single errors within nodes to be corrected. The algorithm reported failure if more than one component within a node required correction. The algorithm also reported failure if the (possibly corrected) back pointer in the current node failed to address the previous visited node.

The other method of producing such a list used two consecutive header nodes and stored $N_{x+1} \oplus N_{x-1}$, rather than N_{x-1}, in a *virtual* backpointer component, $N_x \cdot v$ [83]. Local correction in this VDLL structure was accomplished by using two constructive votes, $N_x \cdot f_1$ and $N_x \cdot v \oplus N_{x-1}$, together with a diagnostic vote, ‘$N_n \cdot id = id$ & $N_n \cdot f_1 \oplus N_n \cdot v = N_x$’, to identify the target node N_{x+1}.

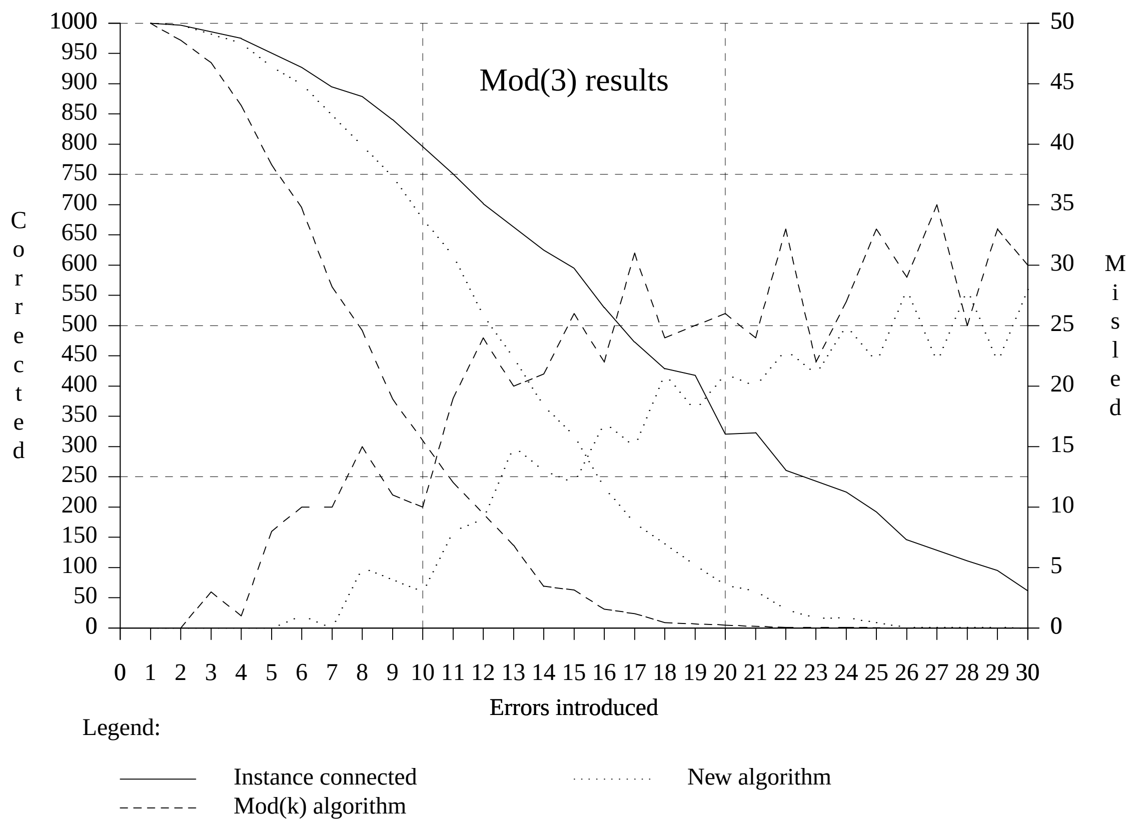
For the mod(2) instance, correction was attempted using a historical mod(k) 1-local-correction algorithm, which used the votes $N_1 = N_{1-k} \cdot b_k$, $N_1 = N_{2-k} \cdot b_k \cdot f_1$, and $N_1 \cdot f_1 = N_0$, the mod(2) local correction algorithm presented in [125], and the spiral local correction algorithm presented in [24]. For the mod(3) and mod(4) instances, correction was attempted using the mod(k) local correction algorithm, and the selective-local-correction algorithm described in Chapter 5 and Appendix B1.

Each mod(k) algorithm was executed on exactly the same ‘‘randomly’’ damaged instances. Each test was performed 1000 times before the number of pointers being damaged was increased. Statistics were collected on the number of times that the damaged instance remained connected, and was thus potentially correctable. Statistics were also collected on the number of times each algorithm was able to correct the structure, and the number of times that each algorithm was misled into attempting to apply an incorrect change.

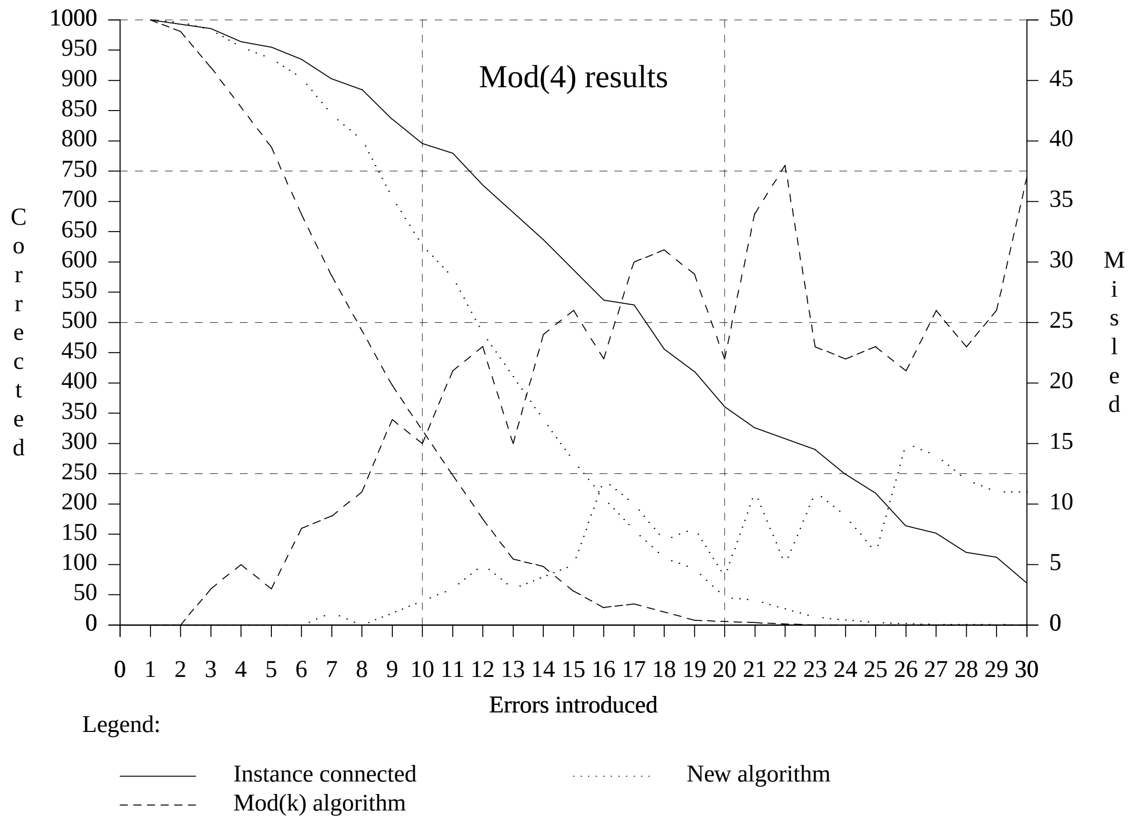




Mod(2) results
Corrected
Missed
Errors introduced
Legend:
Instance connected
Mod(k) algorithm
Mod(2) algorithm
x x Spiral algorithm



Mod(3) results
Corrected
Missed
Errors introduced
Legend:
Instance connected
Mod(k) algorithm
New algorithm



C1.50. Comments

When the double-linked list was protected by using an error-correcting code, nodes contained no identifier field, but still contained one additional component not present in the other structures considered. When compared to alternative robust linked lists requiring the same amount of storage space, the checksummed double-linked list performed rather poorly. Comparable results for the spiral(3) and helix(3) storage structures are presented in Appendix C2 and Appendix E2.

The VDLL structure performed very well. Empirical results suggest that it is as strongly connected as the mod(3) structure, and that its correction algorithm is competitive with the historical correction algorithms used to correct mod(k) structures.

While the behaviour of the mod(k) local-correction algorithm is similar to the spiral(2) local-correction algorithm, and to a lesser extent the selective-local-correction algorithm presented in Chapter 5, the mod(2) local-correction algorithm is quite different, since it uses two parallel traversals of the instance, and an elaborate fault dictionary to assist in correction. It is therefore surprising that the results of attempting to correct a mod(2) instance are almost identical, regardless of the algorithm used.

Under the various errors introduced, the mod(2) structure remained connected 44% of the time, the mod(3) structure 55% of the time, the VDLL structure 56% of the time, and the mod(4) structure 60% of the time. The VDLL correction algorithm corrected 26% of all errors, as did the mod(k) correction algorithm when operating on mod(2), mod(3), and mod(4) linked list structures.

Superficially it appears that the local correction algorithm outlined in Appendix B1 should correct more errors in a mod($k \geq 4$) structure than in a mod(3) structure. However the locality, in which it is assumed that at most two errors occur, is smaller in a mod(3) structure than in a mod($k \geq 4$) structure, and this becomes significant when many errors are introduced into the instance being corrected. It is therefore not surprising that this algorithm corrected 40% of errors in mod(3) instances, and 38% of errors in mod(4) instances.

The statistics presented above are very dependent on the number of errors introduced into the instance, the type of error introduced, and the size of the instance being damaged. However, these statistics provided some assurance that the selective-local-correction algorithm outlined in Appendix B1 is indeed superior to algorithms previously presented, when applied to a mod($k \geq 3$) structure.

APPENDIX C2

Empirical results for multi-linked list structures

C2.51. Explanation

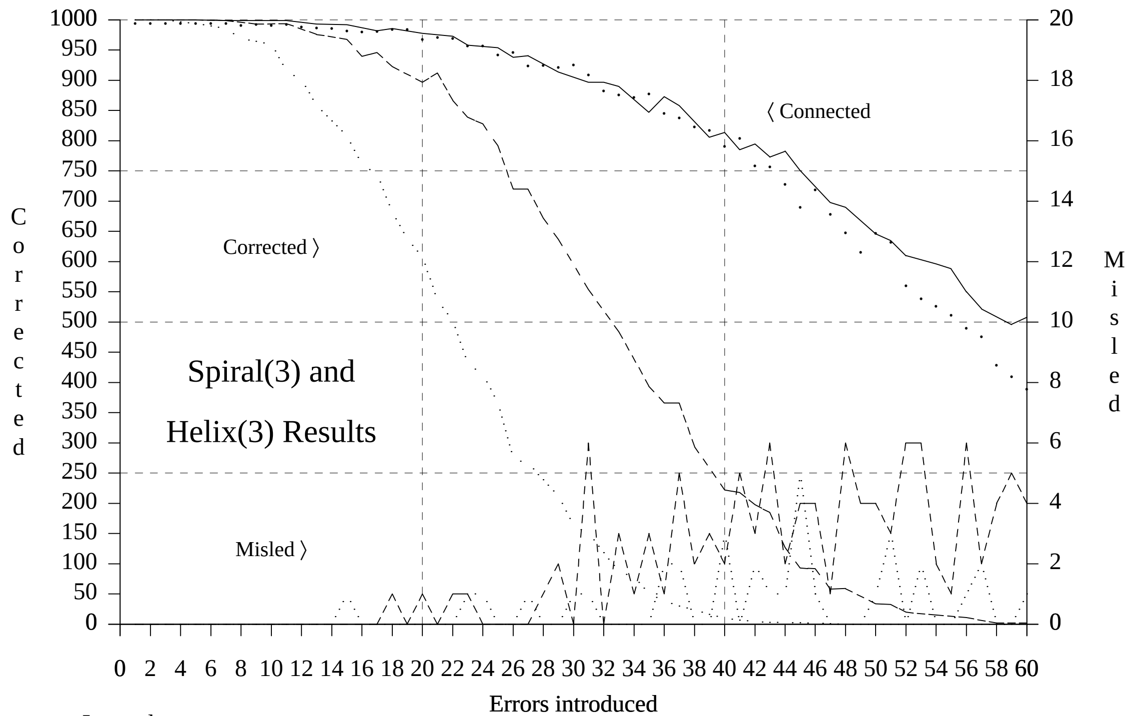
This appendix presents empirical results obtained when ‘‘random’’ errors were introduced into instances of a helix(3), helix(4), spiral(3), and spiral(4) structure.

Each instance contained 100 consecutively located nodes plus headers. Increasing numbers of pointers were randomly selected from within this instance, and modified by adding or subtracting a random number between 1 and 10.

The spiral(k) instances were corrected using the spiral correction algorithm described in [24]. This algorithm used the following votes to correct up to $k - 1$ errors in any locality. If a single candidate received $k + 1$ or more votes the algorithm concluded that this node was the target. Otherwise the algorithm reported failure.

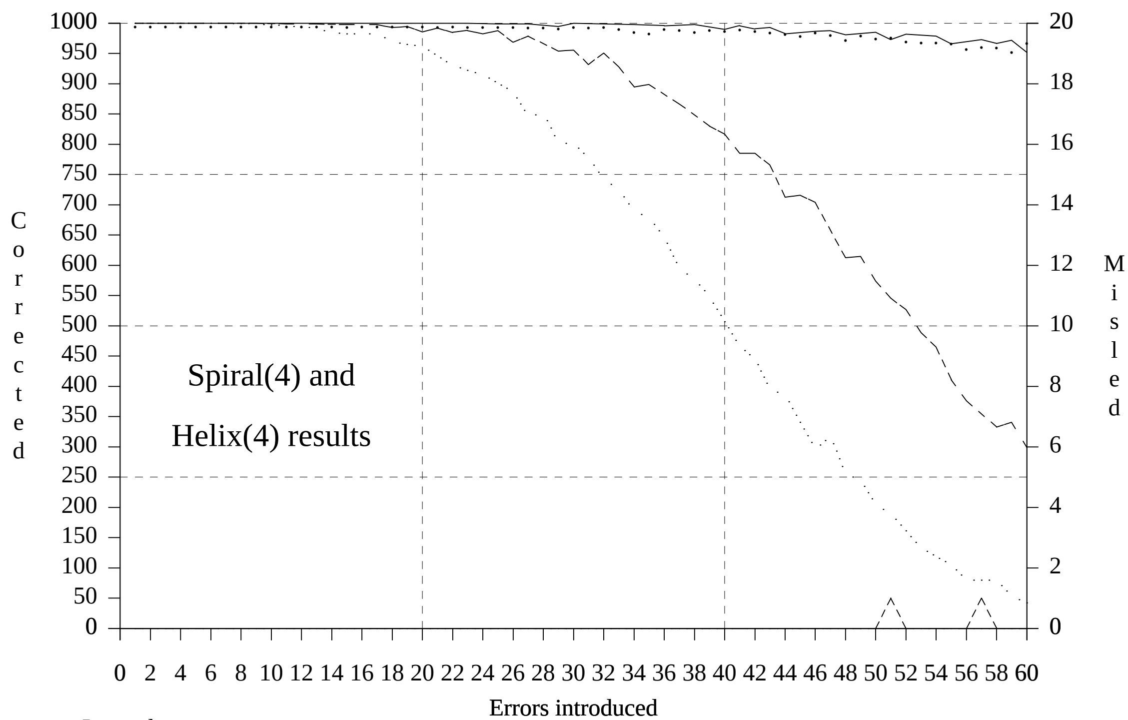
Vote	Path followed	Compared with
$C_i, 1 \le i < k$	$N_{1+i-k} \cdot b_k \cdot f_i$	
C_k	$N_{1-k} \cdot b_k$	
$D_i, 1 \le i < k$	$N_n \cdot f_i$	N_{1-i}

Unfortunately since the spiral and helix structures are different, it was impossible to execute the correction algorithms on the same ‘‘randomly’’ damaged instances. Thus the errors applied to each instance were related only by the above constraints. Each test was performed 1000 times on each instance before the number of pointers being damaged was increased. Statistics were collected on the number of times that each damaged instance remained connected, and was thus potentially correctable. Statistics were also collected on the number of times the appropriate algorithm was able to correct the instance presented to it, and the number of times that each algorithm attempted to apply an incorrect change.



Legend:

Helix(3) connected — Helix(k) algorithm
Spiral(3) connected — Spiral(k) algorithm



Legend:

Helix(4) connected — Helix(k) algorithm
Spiral(4) connected — Spiral(k) algorithm

C2.52. Comments

Under the various errors introduced, the helix(3) structure remained connected 85% of the time and the spiral(3) structure 84% of the time. The helix(4) and spiral(4) structures remained connected 99% of the time.

The helix(3) structure was corrected 54% of the time while the spiral(3) structure was corrected only 36% of the time. Similarly, the helix(4) structure was corrected 83% of the time, but the spiral(4) structure only 66% of the time. More informally, in the experiments conducted, the helix(k) algorithm generally behaved as well as the spiral(k) correction algorithm, even when the structures that it was correcting contained an additional 10 errors.

Somewhat surprisingly, the helix correction algorithm attempted more erroneous corrections than the spiral correction algorithm. In the helix(3) structure 111 erroneous corrections were attempted compared to 33 in the spiral(3) structure. Similarly, in the helix(4) structure 2 erroneous corrections were attempted compared to none in the spiral(4) structure. Various factors seem to have contributed to this discrepancy. Since the spiral correction algorithm failed more often, it encountered fewer errors, and thus had less opportunity to be misled. In addition, the helix correction algorithm could be misled when an incorrect candidate receives k votes, while the spiral correction algorithm could be misled only if an incorrect candidate receives at least $k + 1$ votes. This became particularly significant when constructive votes supported nodes outside of the instance being corrected. Given the nature of the diagnostic votes used, and the fact that only components within the instance were damaged, such nodes receive no diagnostic votes from the spiral correction algorithm, but could receive up to $k - 1$ diagnostic votes from the helix correction algorithm.

Although the spiral(3) and helix(3) structures are naturally much more robust than the mod(3) structure, since each node in a mod(k) structure contains only the two pointers f_1 and b_k, it is of some interest to compare the results presented above with those presented in Appendix C1. Therefore, instances of the helix(3) and spiral(3) structure containing more than 30 damaged pointers will be ignored. Under this scenario the spiral(3) and helix(3) structures remained connected 98% of the time. The helix(3) instances were corrected 90% of the time, and the spiral(3) instances 70% of the time.

APPENDIX D

The unperturbed constant error model

D.53. Explanation

This appendix graphs the behaviour of functions, derived in Chapter 8, which pertain to the unperturbed constant error model. This model assumes that n components in an instance state are traversed in some sequence, and that e randomly selected components initially contain errors. It further assumes that the order in which components are traversed is unperturbed by the correction of erroneous components.

We then assume that the local-linearisation function, being used to detect or correct errors in these components, produces linearisations which violate the locality constraint if and only if some subsequence of m components contains more than one error.

Superficially, it might appear that the above assumptions are reasonable when considering the behaviour of a 1-local-detection procedure, and that therefore the results presented here can be used to predict the behaviour of such algorithms. However, in the vast majority of cases, local detection will be accomplished successfully even when localities contain an arbitrary number of errors. This is because, typically, such erroneous localities will not be internally consistent. The results presented here therefore provide very pessimistic predictions of the behaviour of 1-local-detection procedures.

The unperturbed constant error model is more useful when attempting to understand the behaviour of a 1-local-correction procedure. Specifically, when 1-local-correction procedures use linearisations which always contain exactly m untrusted components, we will assume that an instance state is locally correctable, if and only if no subsequence (or window) of m consecutive components contains more than one error.

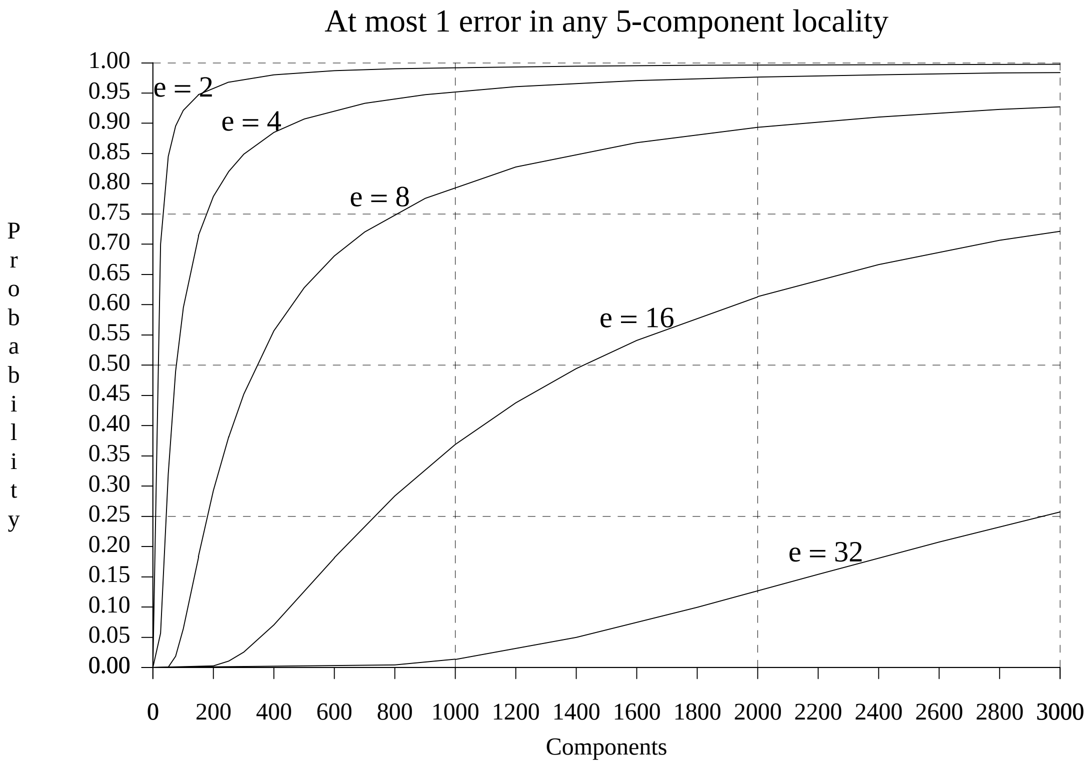
Graphs are presented in pairs, the first being a linear graph for small numbers of components, and the second using a logarithmic scale to depict results for large numbers of components.

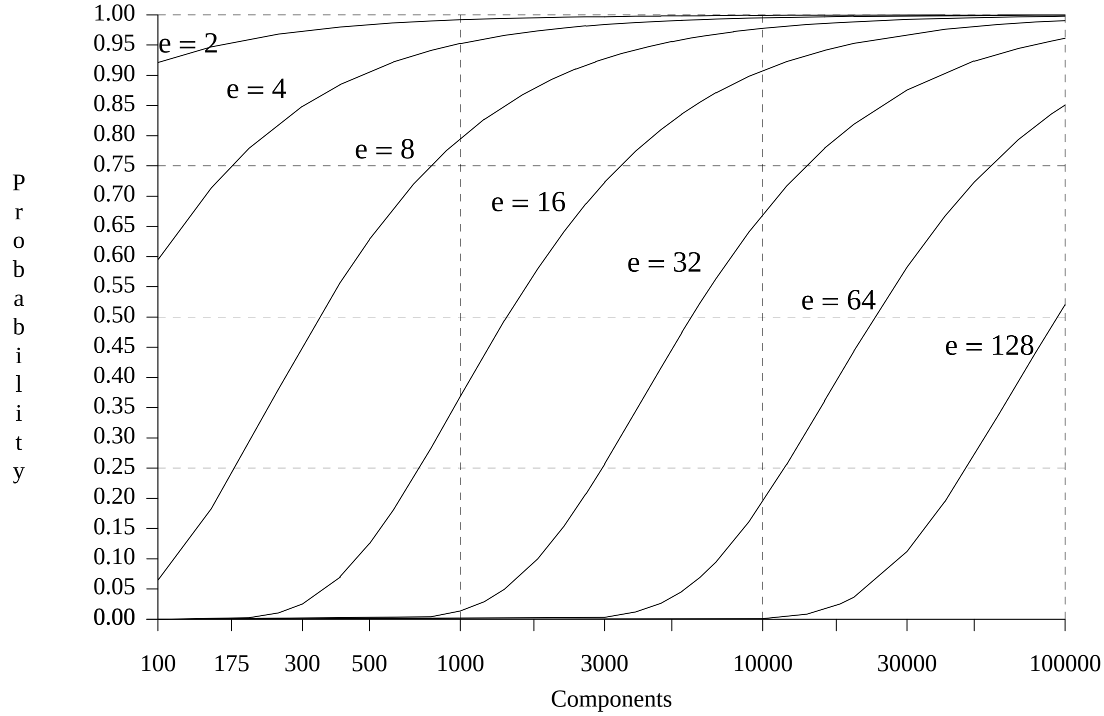
The first four sets of graphs plot the relationship between n, e and $\binom{n-(m-1)*(e-1)}{e} \Big/ \binom{n}{e}$ for m = 5, m = 10, m = 20, and m = 40. This formula expresses the probability that no subsequence of m components contains more than one error, given that exactly e errors occur in a sequence of n components.

These graphs suggest that when the size of a correction locality is quadrupled and the number of errors introduced into the structure reduced by half, similar results are obtained. Currently, this behaviour is not well understood. However, it is important since it implies that local-correction algorithms will perform reasonably well, even when they use large correction localities.

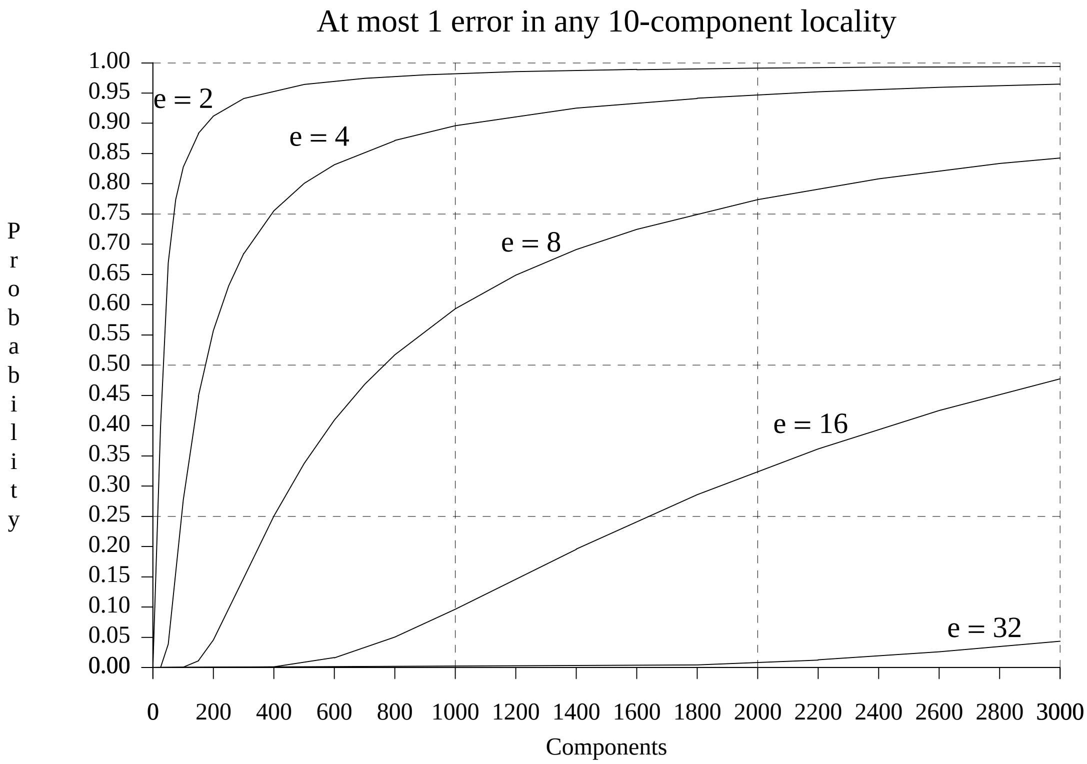
The last pair of graphs plots the relationship between m, n, and $\sum_{e=1}^{n} \binom{n-(m-1)*(e-1)}{e} \Big/ \binom{n}{e}$. This formula expresses the expected number of errors that can be placed in a sequence of n components, while placing no more than one error in any subsequence of m components.

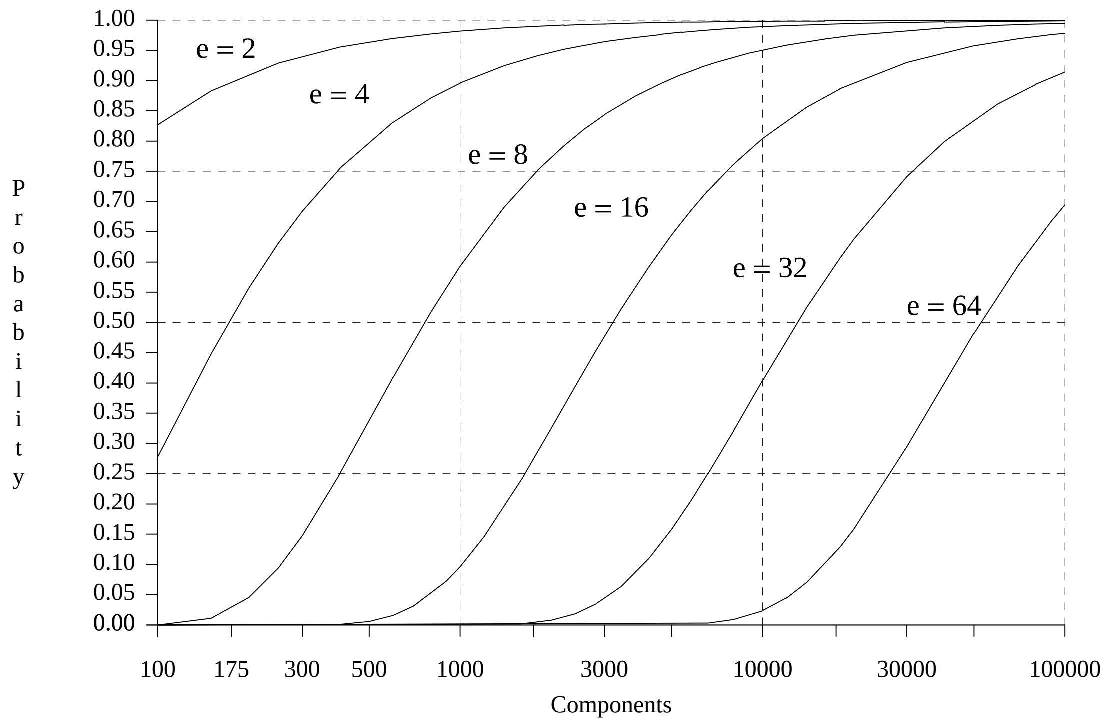
These graphs suggest that the expected number of correctable errors does not increase linearly as storage structures grow. This is not surprising and indicates that the ratio of correctable errors to total components will decrease as structures grow. However, the expected number of correctable errors increases almost linearly when n is large, and clearly increases more rapidly than log n. Thus, local-correction algorithms can be expected to perform very well when operating on large storage structures containing unrelated sets of errors.

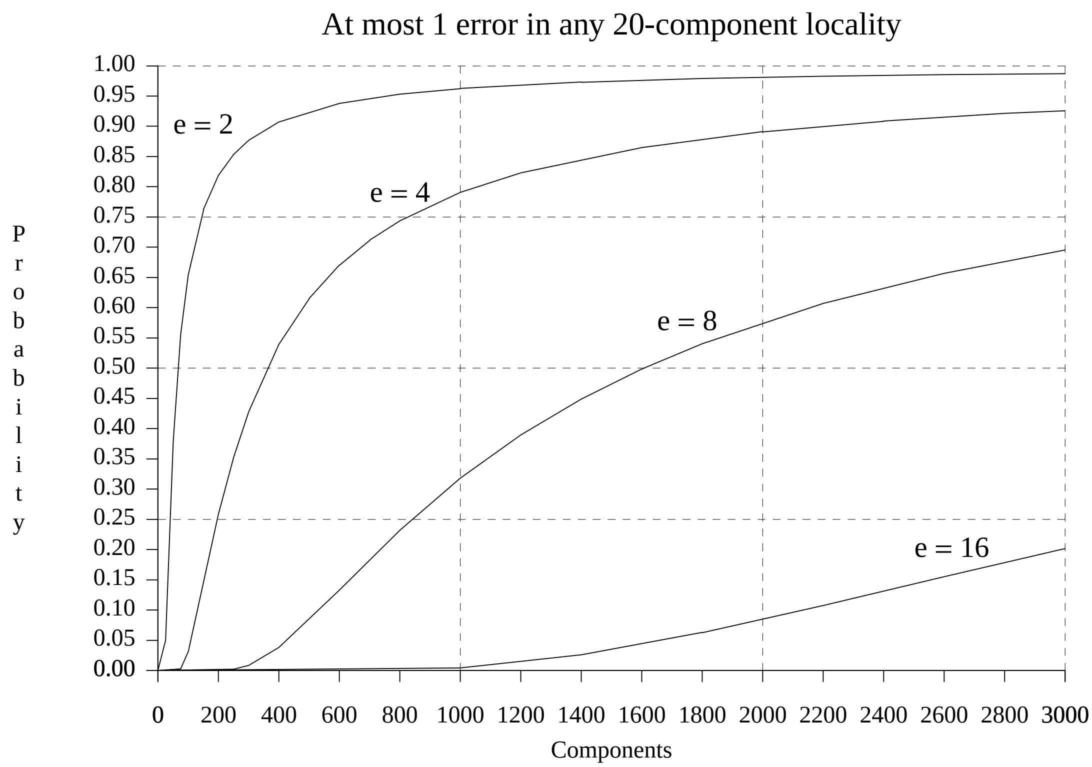

At most 1 error in any 5-component locality
Probability
Components
e = 2
e = 4
e = 8
e = 16
e = 32
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000

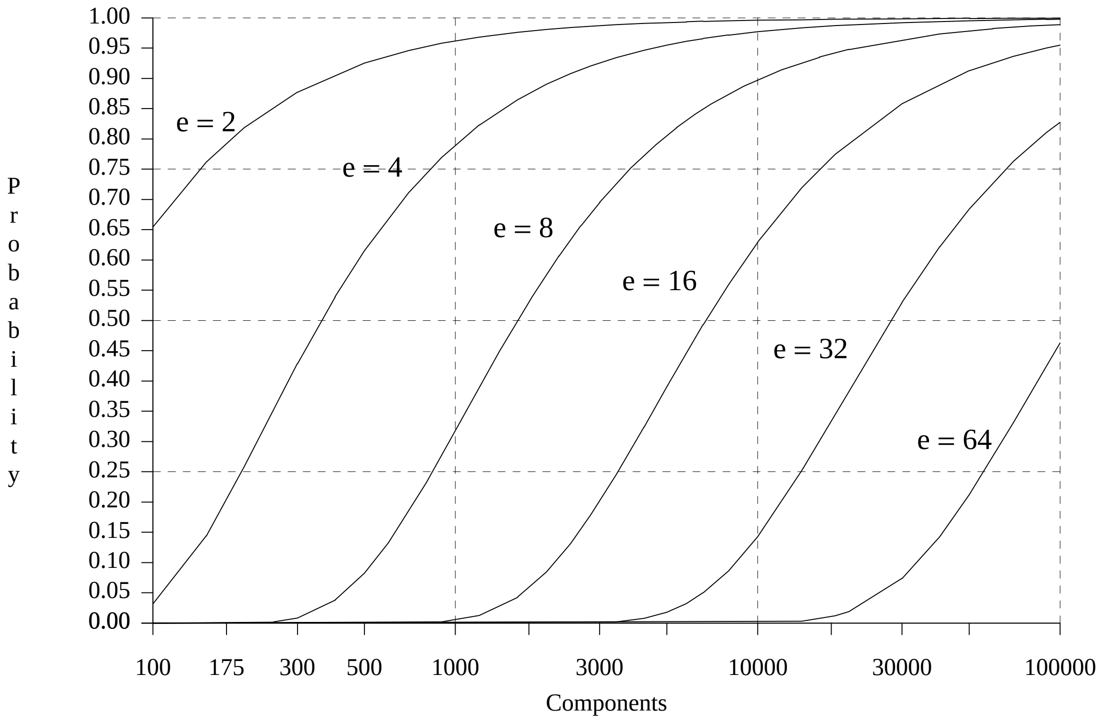


Probability
Components
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
100
175
300
500
1000
3000
10000
30000
100000
e = 2
e = 4
e = 8
e = 16
e = 32
e = 64
e = 128

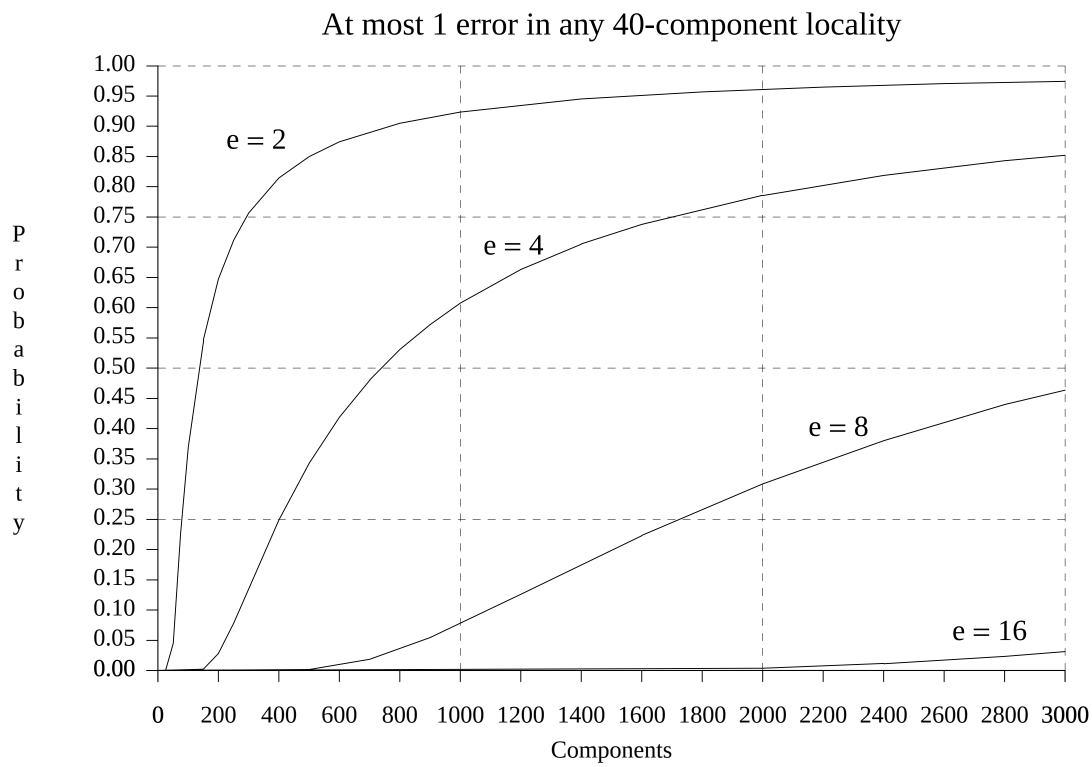

At most 1 error in any 10-component locality
Probability
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
e = 2
e = 4
e = 8
e = 16
e = 32
0 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000
Components


e = 2
e = 4
e = 8
e = 16
e = 32
e = 64
Probability
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
100
175
300
500
1000
3000
10000
30000
100000
Components

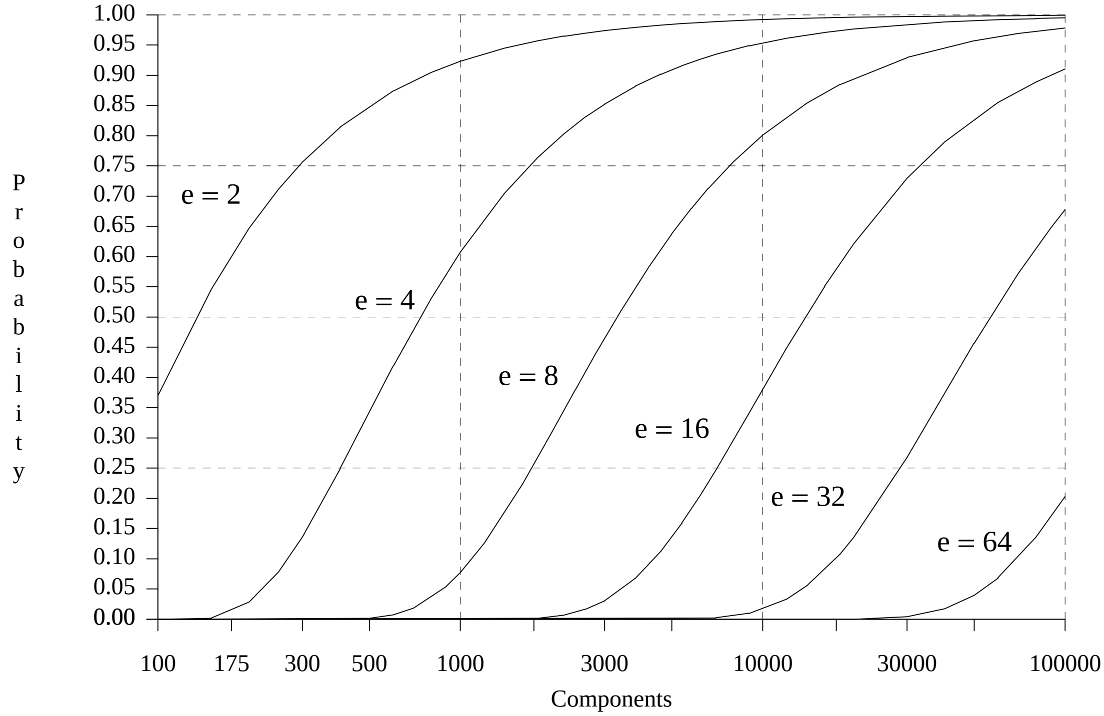

At most 1 error in any 20-component locality
Probability
Components
e = 2
e = 4
e = 8
e = 16
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
0.50
0.55
0.60
0.65
0.70
0.75
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
2400
2600
2800
3000



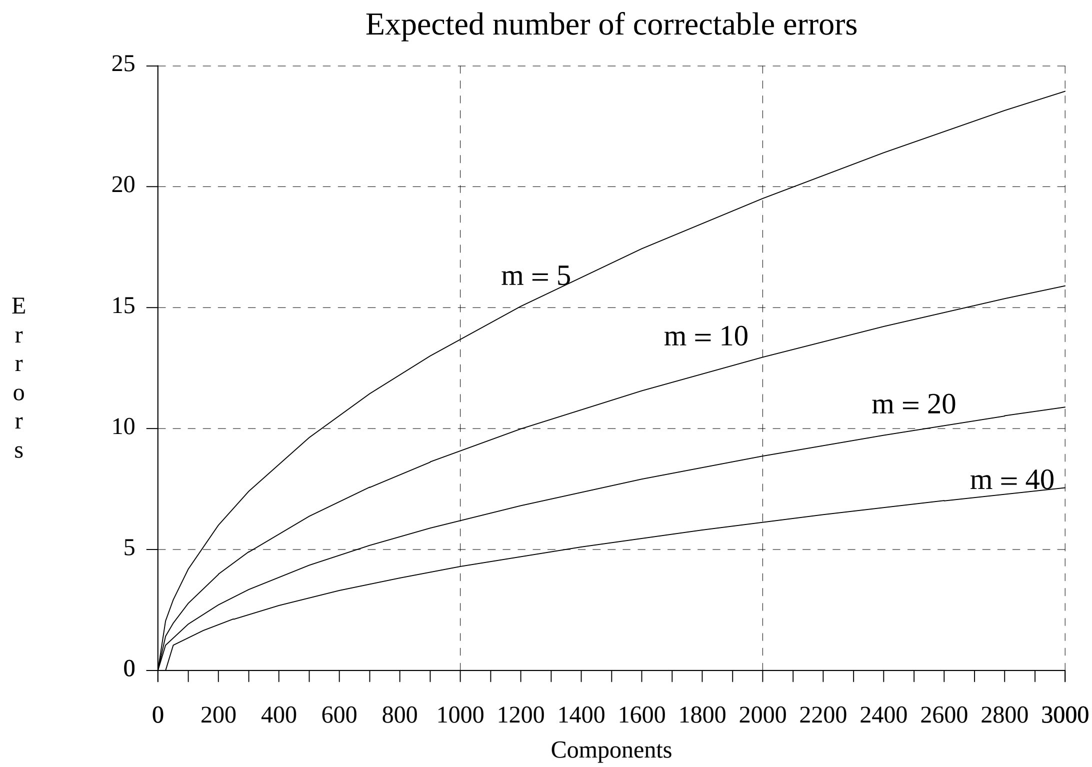
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Probability
e = 2
e = 4
e = 8
e = 16
e = 32
e = 64
100
175
300
500
1000
3000
10000
30000
100000
Components

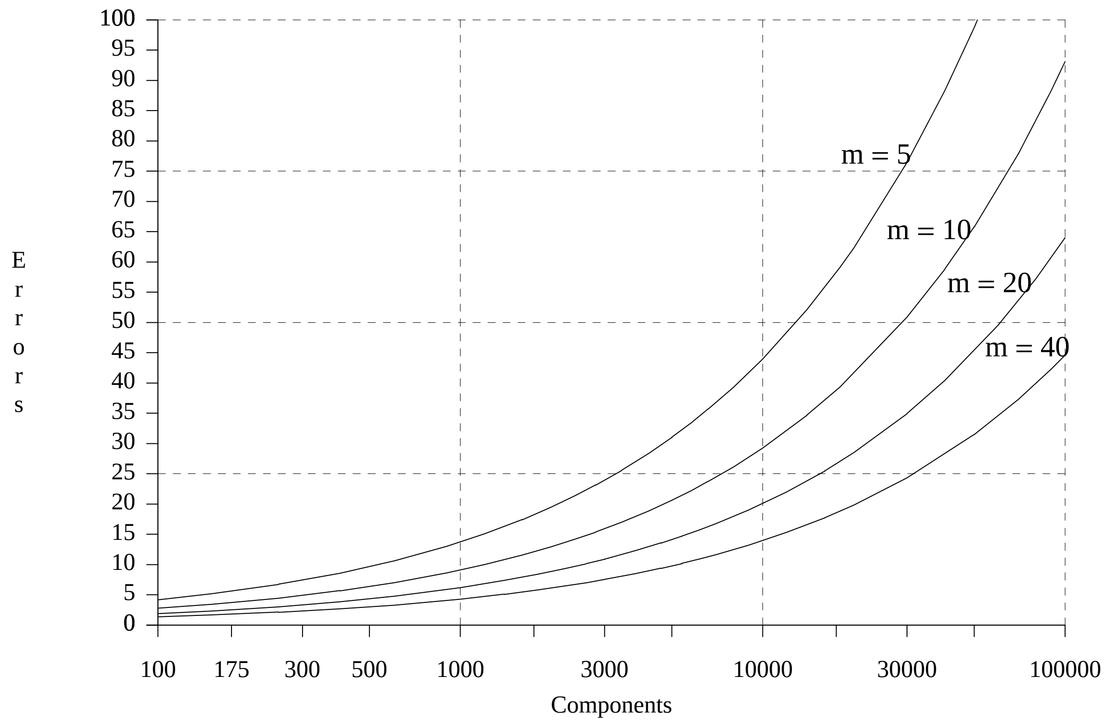


At most 1 error in any 40-component locality
Probability
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
e = 2
e = 4
e = 8
e = 16
0 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000
Components



Probability
Components
e = 2
e = 4
e = 8
e = 16
e = 32
e = 64
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
100
175
300
500
1000
3000
10000
30000
100000


Expected number of correctable errors
25
20
15
10
5
0
Errors
m = 5
m = 10
m = 20
m = 40
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
2400
2600
2800
3000
Components



E r r o r s
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
m = 5
m = 10
m = 20
m = 40
100
175
300
500
1000
3000
10000
30000
100000
Components

APPENDIX E
Analysis of connectivity using Markov models

E.54. Explanation

This appendix uses Markov models to present results pertaining to the connectivity of a number of storage structures, and compares these results with empirical data. Beginning in state 0, each model simulates an algorithm attempting to traverse a particular storage structure, in which errors occur with constant probability p. The Markov model uses the expression z^n, containing the dummy variable z, to indicate the successful traversal of n consecutive nodes.

Each Markov model is then transformed into a generating function, describing the overall transition from the start state to a final state. These generating functions are not presented, since they can easily be reconstructed from the Markov model. In all but the simplest structures, these generating functions are rather complex, and not easily depicted.

We will study two issues. Firstly, in order to remain compatible with the results presented in Appendix C, we will determine, for various structures and values of p, the expected number of instances that remain connected, given that we examine 1000 instances, each containing 100 data nodes. The results of this study will be compared with empirical data. Secondly, we will determine, for various structures and values of p, the expected number of nodes traversed prior to detecting disconnection, given that we are examining an instance containing at least this number of nodes. All of these results will be presented graphically.

Let G be a generating function whose coefficient of z^n indicates the probability that exactly n nodes can be reached from the headers of an instance. Then the number of instances out of 1000 which remain connected, and the expected number of nodes traversed at the point when disconnection is first detected, is calculated using the following ‘Maple’ code.

```
# Instances remaining connected out of 1000
O := proc(a) 0 end:        # The order term of the taylor series is 0
nodes := 100:       # There are 100 data nodes
for p from .005 by .005 to .15 do
        z := 'z':       # Let z be an arbitrary variable
        failed := taylor(G,z,nodes)*1000:      # Number disconnected
        z := 1:         # Eliminate the dummy variable 'z'
        print(p, evalf(1000-failed)):      # Print number connected
        od:

# Expected number of nodes traversed
result := diff(G,z): # Differentiate G with respect to z
z := 1:         # Eliminate the dummy variable z
for p from .005 by .005 to .15 do
        print(p, evalf(result)):     # Print expected number of errors
        od:
```

E.55. Markov models for linked lists

The Markov models used to study the connectivity of linked lists simulate a backwards traversal from the headers of the instance, iteratively selecting with probability q_i some transition generating function, $q_i * z^n$, indicating that the simulation has successfully retreated backwards from N_{x+n} to N_x, visiting in the process all intermediate nodes. If it is discovered that the model cannot proceed backwards because N_x is disconnected then the model enters a final state. Otherwise, when it becomes impossible to arrive at N_x by traversing the linked list backwards, the model simulates a forward traversal from the headers of the instance, iteratively selects with probability q_j some transition generating function $q_j z^m$ which indicates that we can advance forward from N_{x-m} to N_x once again visiting in the process all intermediate nodes. This forward traversal terminates when it is determined that some node is disconnected.

Each of the following tables contains four columns. The first two columns describe a possible transition between a state and an adjacent state. The third column provides the generating function associated with this transition, and the fourth column indicates when this transition occurs. State 0 represents the start state, and state -1 the final state.

Double-linked list			
This State	Next State	Generating Function	Comments
0	0	$z(1-p)$	$N_{x+1} \cdot b_1 = N_x$
0	1	p	$N_{x+1} \cdot b_1 \neq N_x$
1	1	$z(1-p)$	$N_{x-1} \cdot f_1 = N_x$
1	-1	p	$N_{x-1} \cdot f_1 \neq N_x$

Virtual double-linked list (VDLL)			
0	0	$z(1-p)$	$N_{x+1} \cdot v \oplus N_{x+2} = N_x$
0	1	p	$N_{x+1} \cdot v \oplus N_{x+2} \neq N_x$
1	1	$z(1-p)$	$N_{x-1} \cdot f_1 = N_x$
1	2	p	$N_{x-1} \cdot f_1 \neq N_x$
2	1	$z(1-p)$	$N_{x-1} \cdot v \oplus N_{x-2} = N_x$
2	-1	p	$N_{x-1} \cdot v \oplus N_{x-2} \neq N_x$

Mod(2) linked list			
0	0	$z(1-p)$	$N_{x+2} \cdot b_2 = N_x$
0	1	p	$N_{x+2} \cdot b_2 \neq N_x$
1	2	$(1-p)$	$N_{x+1} \cdot b_2 = N_{x-1}$
1	3	p	$N_{x+1} \cdot b_2 \neq N_{x-1}$
2	0	$z^2(1-p)$	$N_{x-1} \cdot f_1 = N_x$
2	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
3	3	$z(1-p)$	$N_{x-1} \cdot f_1 = N_x$
3	-1	p	$N_{x-1} \cdot f_1 \neq N_x$

Mod(3) linked list			
0	0	$z(1-p)$	$N_{x+3} \cdot b_3 = N_x$
0	1	p	$N_{x+3} \cdot b_3 \neq N_x$
1	2	$(1-p)$	$N_{x+2} \cdot b_3 = N_{x-1}$
1	3	p	$N_{x+2} \cdot b_3 \neq N_{x-1}$
2	0	$z^2(1-p)$	$N_{x-1} \cdot f_1 = N_x$
2	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
3	4	$(1-p)$	$N_{x+1} \cdot b_3 = N_{x-2}$
3	5	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$
4	6	$(1-p)$	$N_{x-2} \cdot f_1 = N_{x-1}$
4	-1	p	$N_{x-2} \cdot f_1 \neq N_{x-1}$
5	5	$z(1-p)$	$N_{x-1} \cdot f_1 = N_x$
5	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
6	0	$z^3(1-p)$	$N_{x-1} \cdot f_1 = N_x$
6	-1	p	$N_{x-1} \cdot f_1 \neq N_x$

Mod(4) linked list			
0	0	$z(1-p)$	$N_{x+4} \cdot b_4 = N_x$
0	1	p	$N_{x+4} \cdot b_4 \neq N_x$
1	2	$(1-p)$	$N_{x+3} \cdot b_4 = N_{x-1}$
1	3	p	$N_{x+3} \cdot b_4 \neq N_{x-1}$
2	0	$z^2(1-p)$	$N_{x-1} \cdot f_1 = N_x$
2	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
3	4	$(1-p)$	$N_{x+2} \cdot b_4 = N_{x-2}$
3	5	p	$N_{x+2} \cdot b_4 \neq N_{x-2}$
4	6	$(1-p)$	$N_{x-2} \cdot f_1 = N_{x-1}$
4	-1	p	$N_{x-2} \cdot f_1 \neq N_{x-1}$
5	7	$(1-p)$	$N_{x+1} \cdot b_4 = N_{x-3}$
5	8	p	$N_{x+1} \cdot b_4 \neq N_{x-3}$
6	0	$z^3(1-p)$	$N_{x-1} \cdot f_1 = N_x$
6	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
7	9	$(1-p)$	$N_{x-3} \cdot f_1 = N_{x-2}$
7	-1	p	$N_{x-3} \cdot f_1 \neq N_{x-2}$
8	8	$z(1-p)$	$N_{x-1} \cdot f_1 = N_x$
8	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
9	10	$(1-p)$	$N_{x-2} \cdot f_1 = N_{x-1}$
9	-1	p	$N_{x-2} \cdot f_1 \neq N_{x-1}$
10	0	$z^4(1-p)$	$N_{x-1} \cdot f_1 = N_x$
10	-1	p	$N_{x-1} \cdot f_1 \neq N_x$

Helix(3) linked list			
0	0	$z(1-p)$	$N_{x+3} \cdot b_3 = N_x$
0	1	p	$N_{x+3} \cdot b_3 \neq N_x$
1	0	$z(1-p)$	$N_{x+2} \cdot b_2 = N_x$
1	2	p	$N_{x+2} \cdot b_2 \neq N_x$
2	3	$(1-p)$	$N_{x+2} \cdot b_3 = N_{x-1}$
2	4	p	$N_{x+2} \cdot b_3 \neq N_{x-1}$
3	0	$z^2(1-p)$	$N_{x-1} \cdot f_1 = N_x$
3	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
4	5	$(1-p)$	$N_{x+1} \cdot b_2 = N_{x-1}$
4	6	p	$N_{x+1} \cdot b_2 \neq N_{x-1}$
5	0	$z^2(1-p)$	$N_{x-1} \cdot f_1 = N_x$
5	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
6	7	$(1-p)$	$N_{x+1} \cdot b_3 = N_{x-2}$
6	10	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$
7	8	$(1-p)$	$N_{x-2} \cdot f_1 = N_{x-1}$
7	-1	p	$N_{x-2} \cdot f_1 \neq N_{x-1}$
8	0	$z^3(1-p)$	$N_{x-1} \cdot f_1 = N_x$
8	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
10	10	$z(1-p)$	$N_{x-1} \cdot f_1 = N_x$
10	-1	p	$N_{x-1} \cdot f_1 \neq N_x$

Helix(4) linked list			
0	0	$z(1-p)$	$N_{x+4} \cdot b_4 = N_x$
0	1	p	$N_{x+4} \cdot b_4 \neq N_x$
1	0	$z(1-p)$	$N_{x+3} \cdot b_3 = N_x$
1	2	p	$N_{x+3} \cdot b_3 \neq N_x$
2	0	$z(1-p)$	$N_{x+2} \cdot b_2 = N_x$
2	3	p	$N_{x+2} \cdot b_2 \neq N_x$
3	4	$(1-p)$	$N_{x+3} \cdot b_4 = N_{x-1}$
3	5	p	$N_{x+3} \cdot b_4 \neq N_{x-1}$
4	0	$z^2(1-p)$	$N_{x-1} \cdot f_1 = N_x$
4	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
5	4	$(1-p)$	$N_{x+2} \cdot b_3 = N_{x-1}$
5	6	p	$N_{x+2} \cdot b_3 \neq N_{x-1}$
6	4	$(1-p)$	$N_{x+1} \cdot b_2 = N_{x-1}$
6	7	p	$N_{x+1} \cdot b_2 \neq N_{x-1}$
7	8	$(1-p)$	$N_{x+2} \cdot b_4 = N_{x-2}$
7	9	p	$N_{x+2} \cdot b_4 \neq N_{x-2}$
8	10	$(1-p)$	$N_{x-2} \cdot f_1 = N_{x-1}$
8	-1	p	$N_{x-2} \cdot f_1 \neq N_{x-1}$
9	8	$(1-p)$	$N_{x+1} \cdot b_3 = N_{x-2}$
9	11	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$
10	0	$z^3(1-p)$	$N_{x-1} \cdot f_1 = N_x$
10	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
11	12	$(1-p)$	$N_{x+1} \cdot b_4 = N_{x-3}$
11	13	p	$N_{x+1} \cdot b_4 \neq N_{x-3}$
12	14	$(1-p)$	$N_{x-3} \cdot f_1 = N_{x-2}$
12	-1	p	$N_{x-3} \cdot f_1 \neq N_{x-2}$
13	13	$z(1-p)$	$N_{x-1} \cdot f_1 = N_x$
13	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
14	15	$(1-p)$	$N_{x-2} \cdot f_1 = N_{x-1}$
14	-1	p	$N_{x-2} \cdot f_1 \neq N_{x-1}$
15	0	$z^4(1-p)$	$N_{x-1} \cdot f_1 = N_x$
15	-1	p	$N_{x-1} \cdot f_1 \neq N_x$

E.56. Markov model for binary trees

We also present the results of using Markov models to study the connectivity of the checksummed binary tree described in Chapter 4, and the sibling-linked binary tree described in Chapter 7. In the checksummed binary tree we consider a child node connected if either the link addressing it is correct or no other errors occur in the parent node, thus allowing this erroneous link to be corrected.

As justified in Lemma 8.3, in a binary tree created by random insertion, the probability of encountering a leaf node is $(n+1)/3n$, the probability of encountering an incomplete node is the same, and the probability of encountering a full node is $(n-2)/3n$. In the Markov models we will assume that n is large and therefore use 1/3 to approximate each of the above three probabilities.

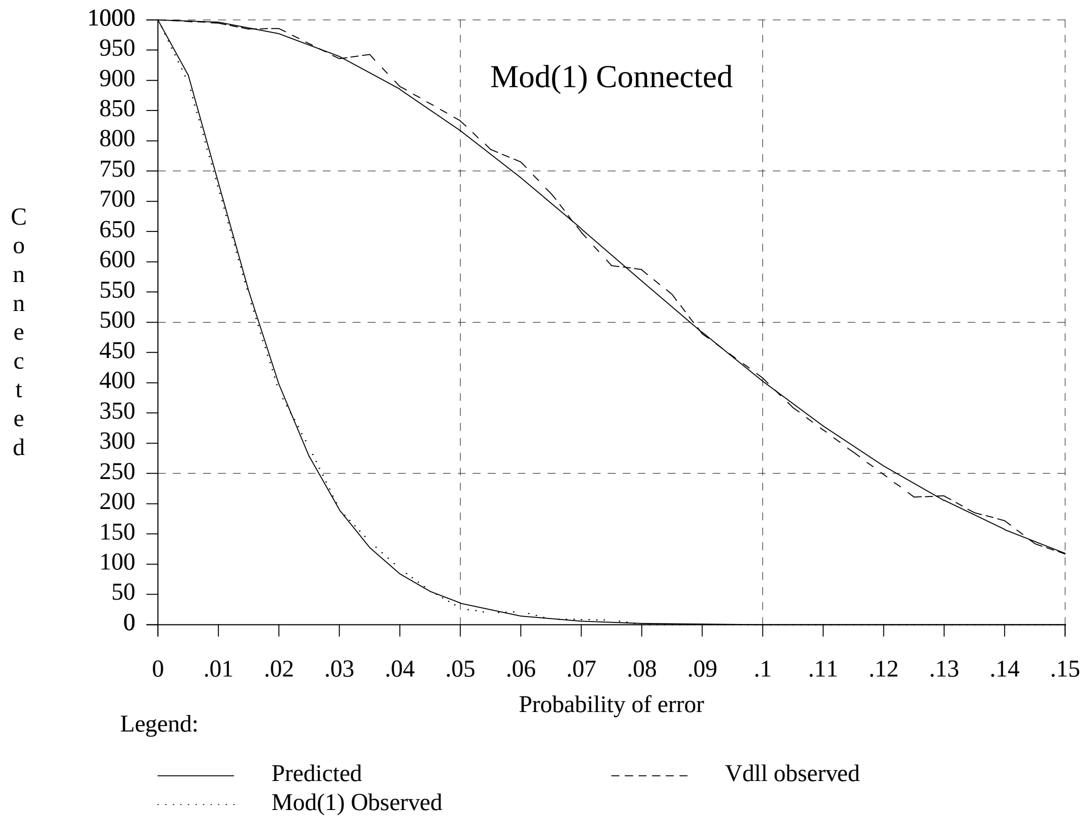
Checksummed binary tree			
This State	Next State	Generating Function	Comments
0	0	1/3	At leaf node - select another node
0	1	1/3	At an incomplete node
0	6	1/3	At a full node
1	0	$z(1-p)$	Link correct
1	2	p	Link contains an error
2	3	$(1-p)$	No error in 2nd component
2	-1	p	Two errors in node
3	4	$(1-p)$	No error in 3rd component
3	-1	p	Two errors in node
4	5	$(1-p)$	No error in 4th component
4	-1	p	Two errors in node
5	0	$z(1-p)$	Only one error in 5 components
5	-1	p	Two errors in node
6	7	$z(1-p)$	Left link correct
6	8	p	Left link in error
7	0	$z(1-p)$	Left link correct - Right link correct
7	3	p	Left link correct - Right link in error
8	3	$z(1-p)$	Left link in error - Right link correct
8	-1	p	Two errors in node

Sibling-linked binary tree			
0	0	1/3	At a leaf node - select another node
0	1	1/3	At an incomplete node
0	3	1/3	At a full node
1	0	$z(1-p)$	Single child addressed by left link
1	2	p	Single child not addressed by left link
2	0	$z(1-p)$	Single child addressed by right link
2	-1	p	Single child disconnected
3	4	$z(1-p)$	Left link correct
3	5	p	Left link in error
4	0	$z(1-p)$	Right link correct
4	6	p	Left link correct - Right link in error
5	6	$z(1-p)$	Left link in error - Right link correct
5	-1	p	Both links contain errors
6	0	$z(1-p)$	Other child addressed by arc pointer
6	-1	p	Other child disconnected

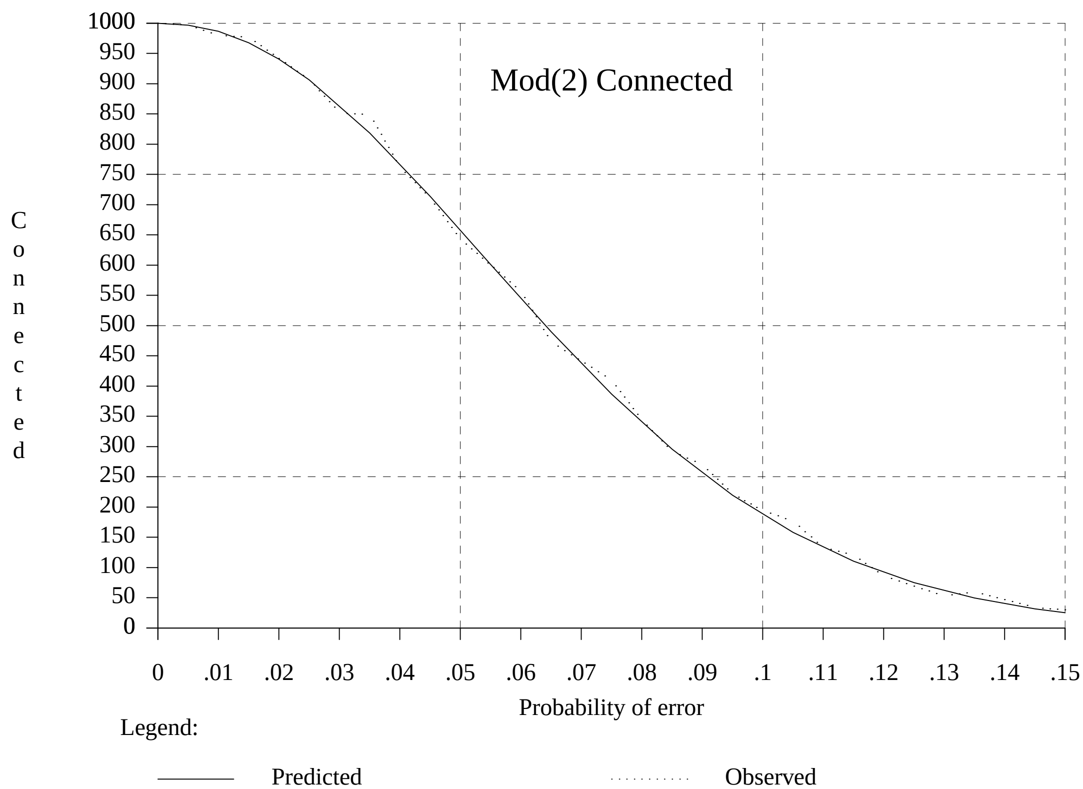
E.57. Graphs

In the following graphs, predicted results were obtained using Markov models and the Maple code presented at the beginning of this appendix. The observed results were obtained from corresponding empirical studies. In the empirical studies, storage structure instances contained 100 data nodes. Binary trees contained approximately the same number of full, incomplete, and leaf nodes, but no effort was made to ensure that these trees were balanced. Pointers, and when appropriate checksums, in each storage structure instance were assigned arbitrary incorrect values with the indicated constant probability and a test performed to determine if the instance remained connected. This test was repeated 1000 times and the total number of times that the instance remained connected recorded. This exercise was repeated under a variety of different error probabilities.

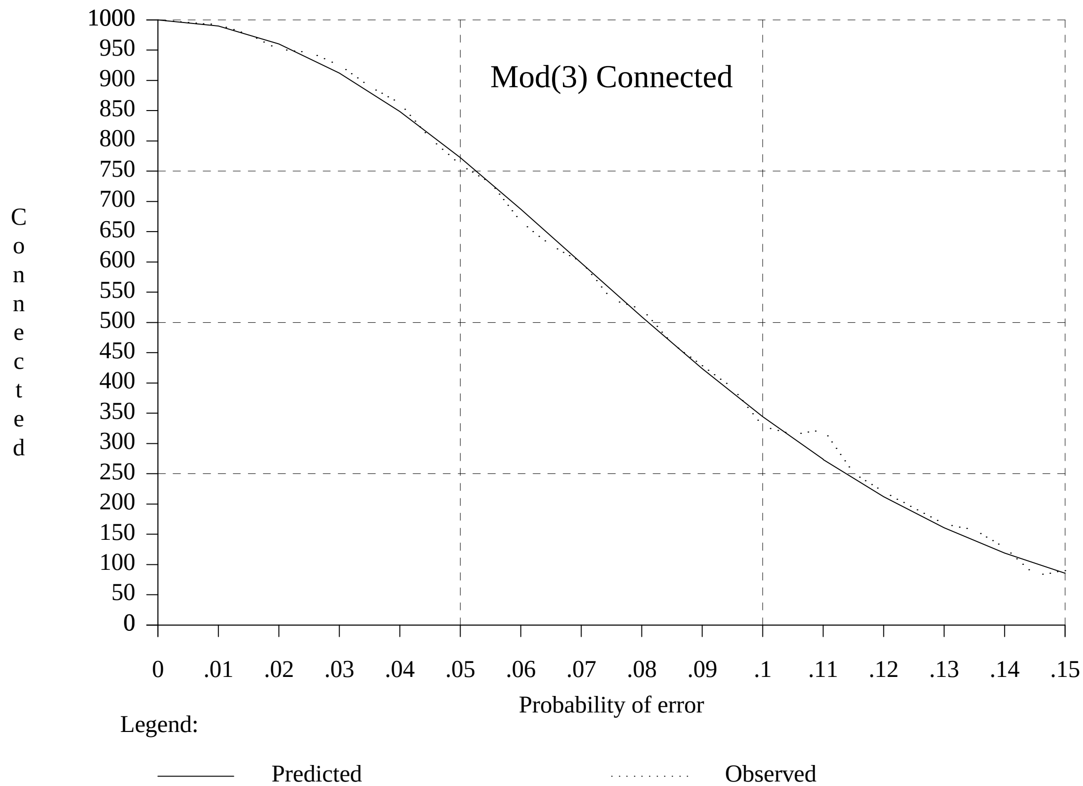
Empirical results were not obtained for the expected number of nodes capable of being traversed in large structures, given that components contained errors with constant probability. Such empirical studies would have required considerable computing resources, since large storage structures were involved, and probably would not have produced any significant new results.

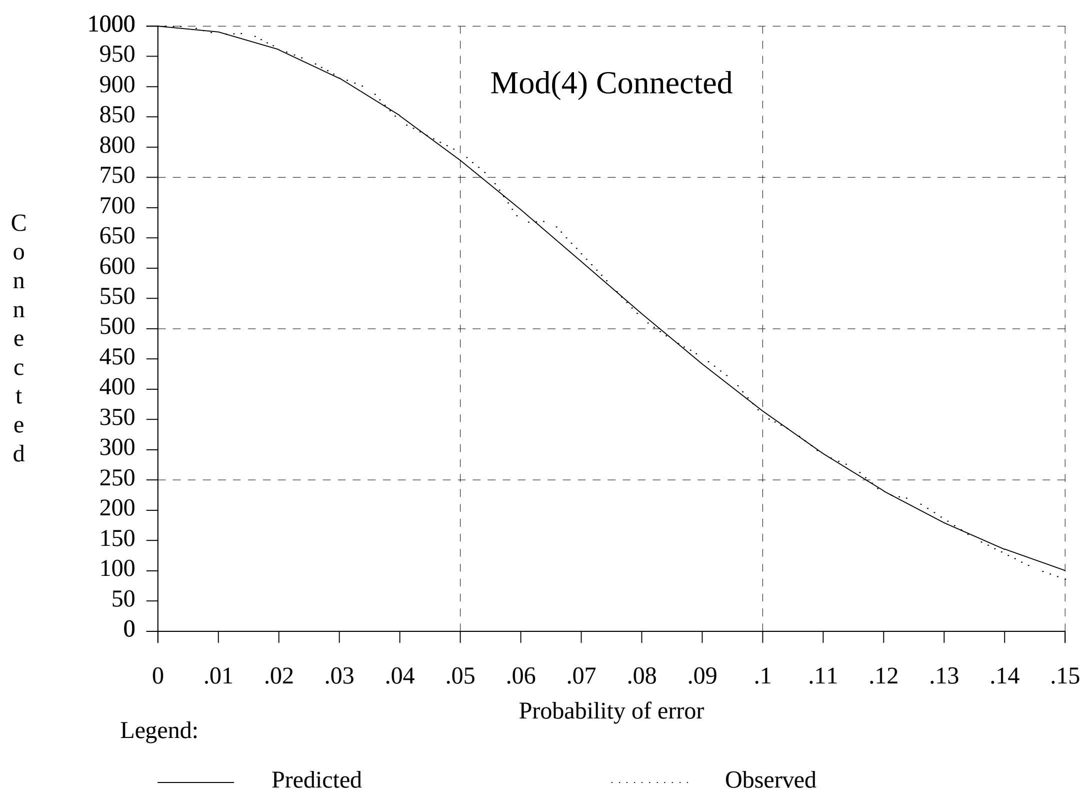


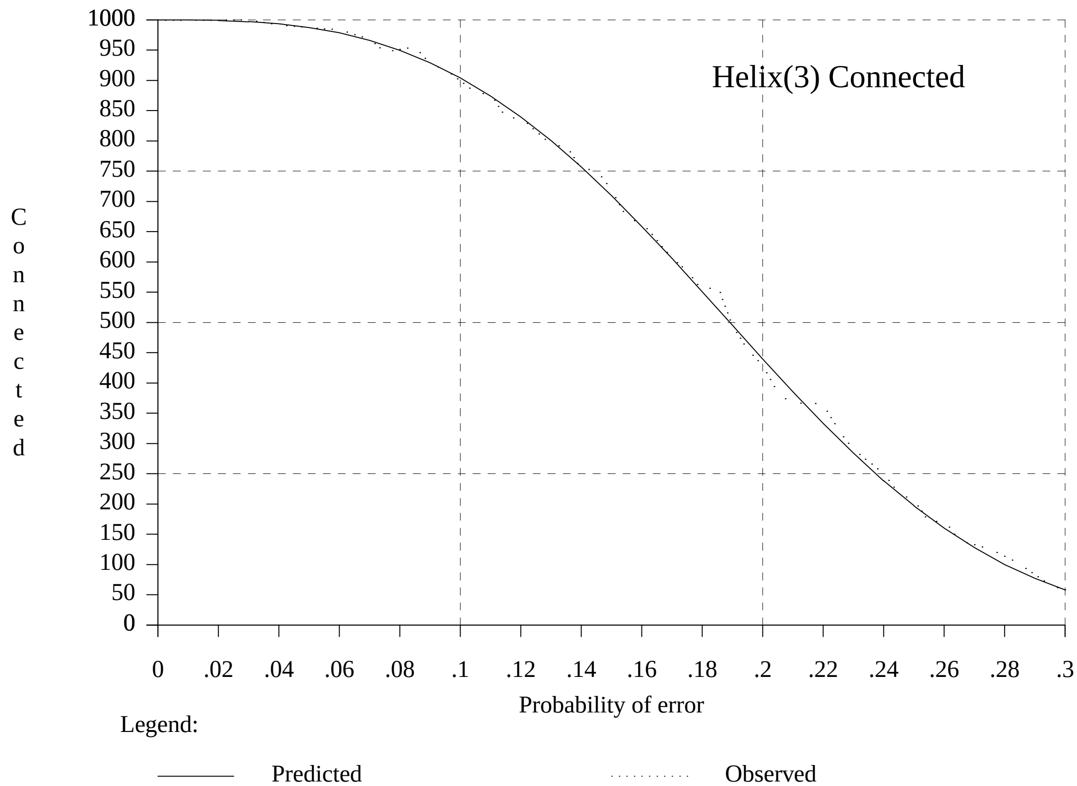
Mod(1) Connected
Connected
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0
.01
.02
.03
.04
.05
.06
.07
.08
.09
.1
.11
.12
.13
.14
.15
Probability of error
Legend:
Predicted
Vdll observed
Mod(1) Observed

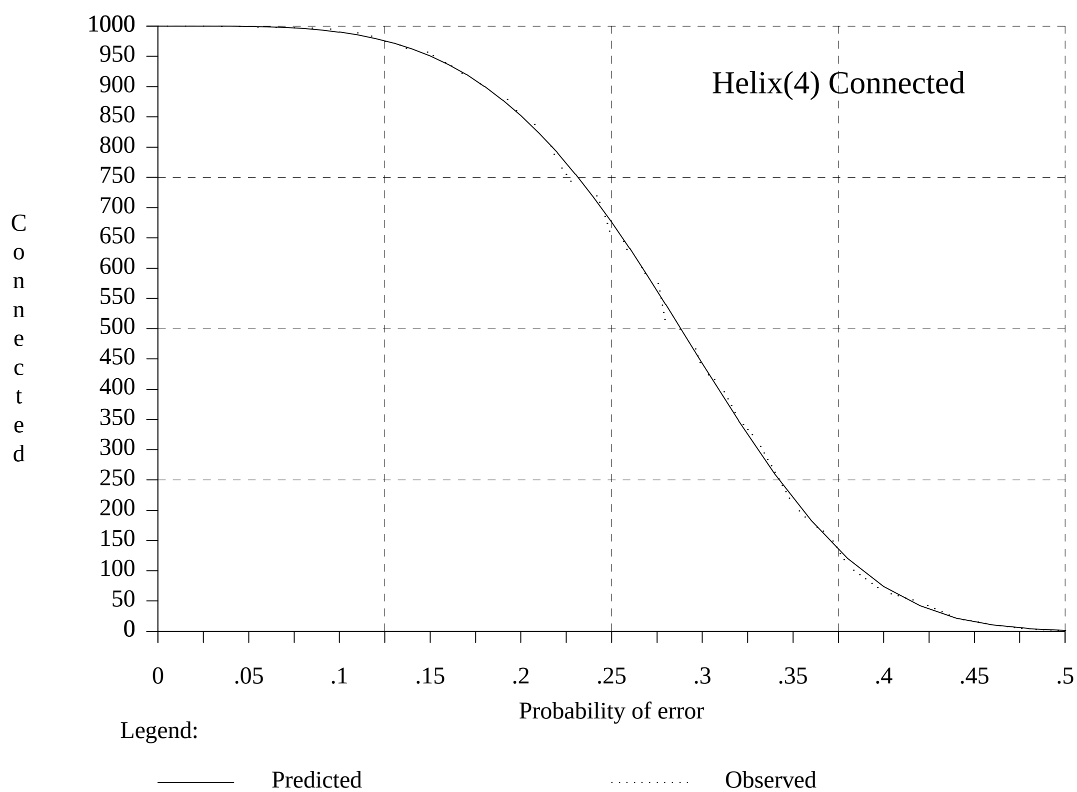


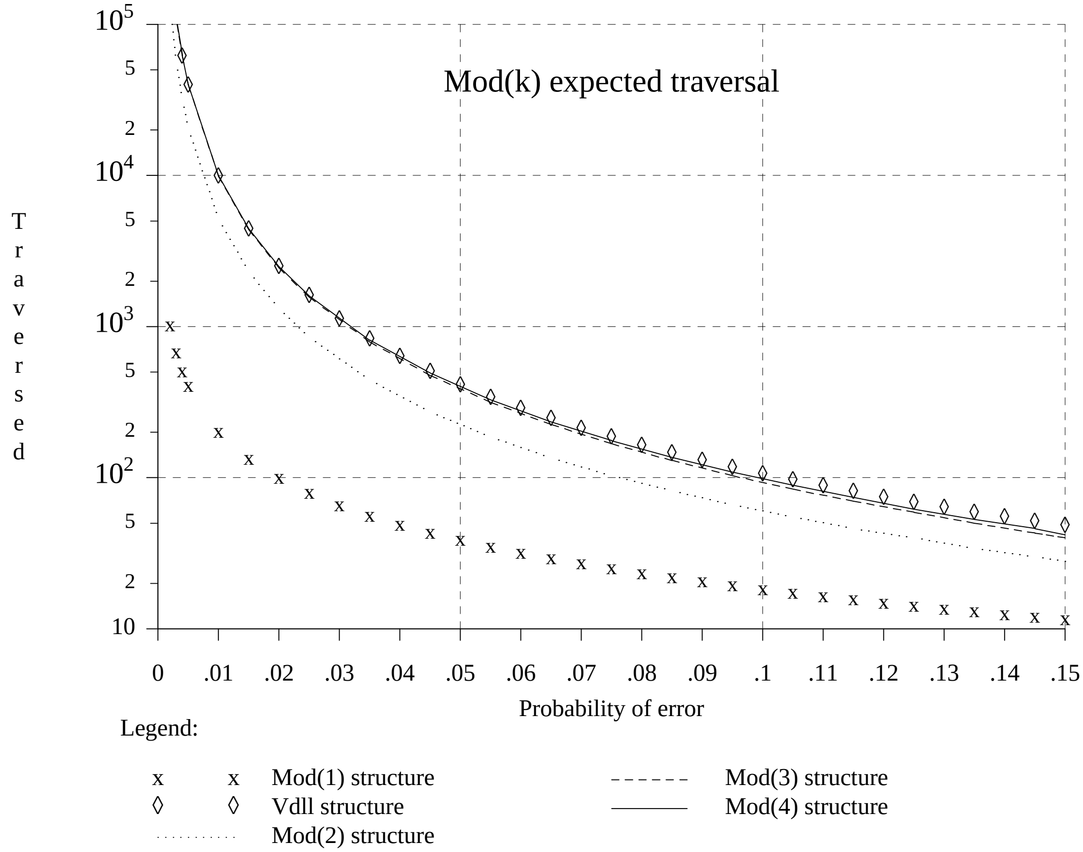
Mod(2) Connected
Connected
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0
.01
.02
.03
.04
.05
.06
.07
.08
.09
.1
.11
.12
.13
.14
.15
Probability of error
Legend:
Predicted
Observed

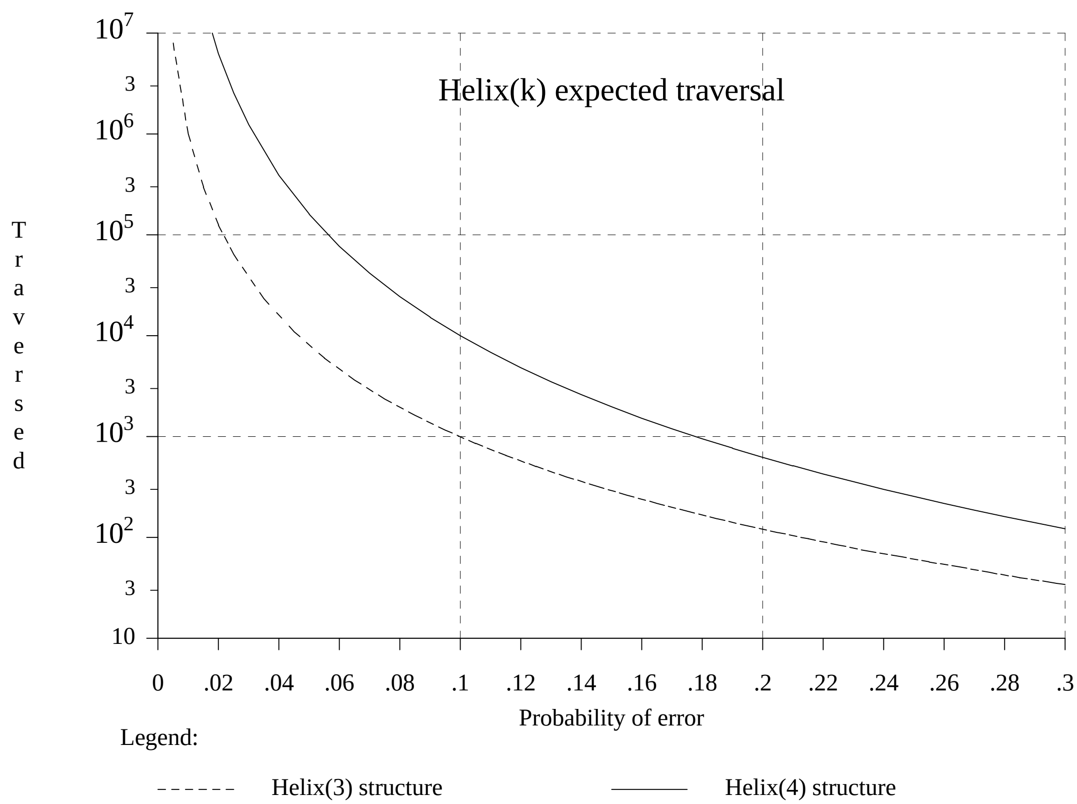

Mod(3) Connected
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
C o n n e c t e d
0
.01
.02
.03
.04
.05
.06
.07
.08
.09
.1
.11
.12
.13
.14
.15
Probability of error
Legend:
Predicted
Observed

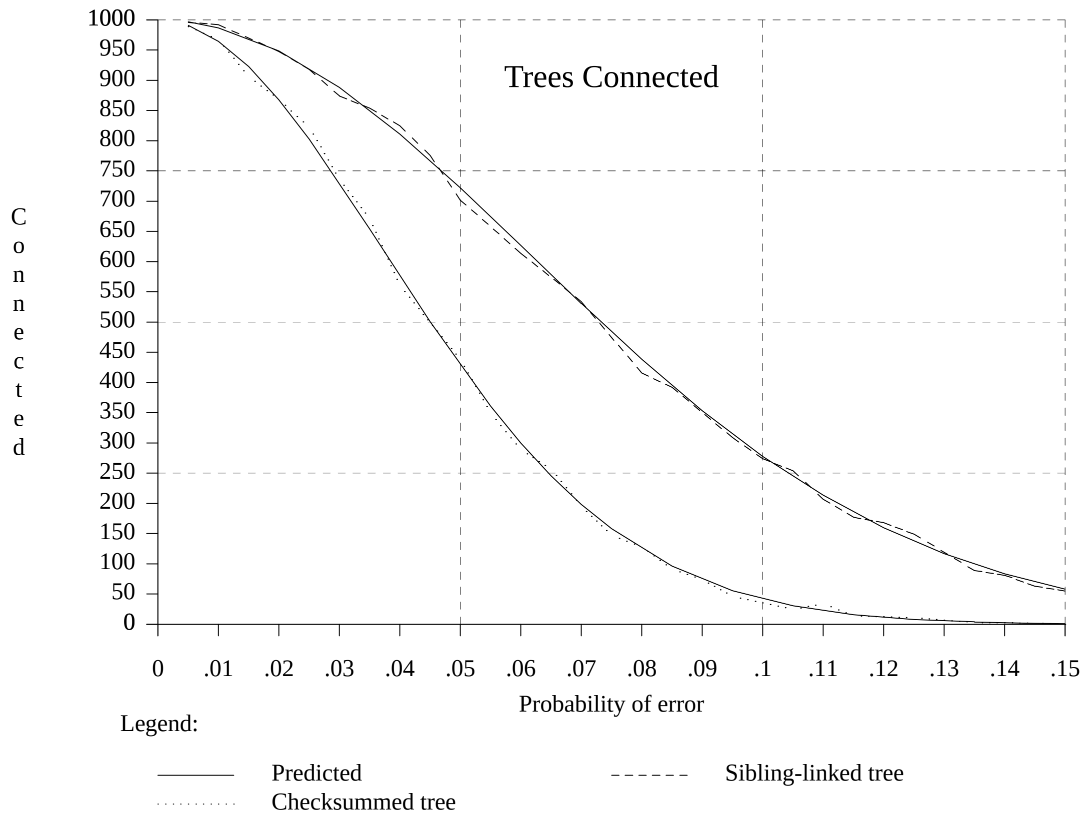

Mod(4) Connected
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
C o n n e c t e d
0
.01
.02
.03
.04
.05
.06
.07
.08
.09
.1
.11
.12
.13
.14
.15
Probability of error
Legend:
Predicted
Observed


Helix(3) Connected
C o n n e c t e d
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0
.02
.04
.06
.08
.1
.12
.14
.16
.18
.2
.22
.24
.26
.28
.3
Probability of error
Legend:
Predicted
Observed

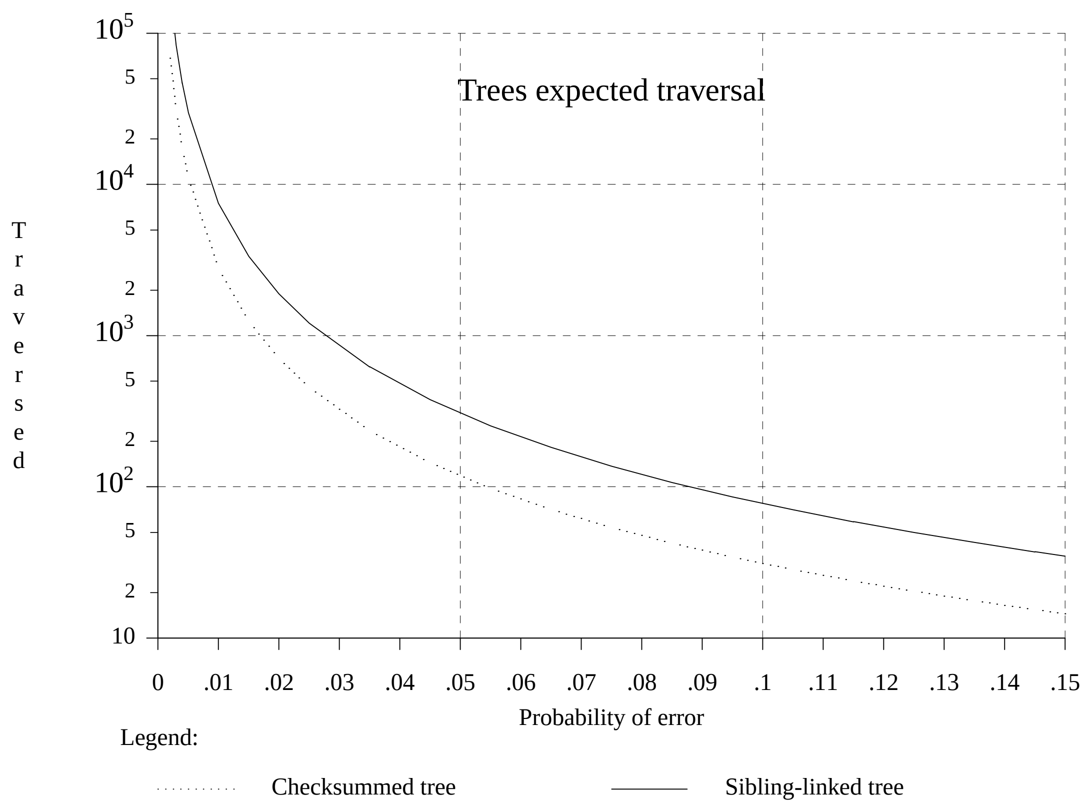

Helix(4) Connected
C o n n e c t e d
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0
.05
.1
.15
.2
.25
.3
.35
.4
.45
.5
Probability of error
Legend:
Predicted
Observed


Mod(k) expected traversal
Traversed
Probability of error
10^5
10^4
10^3
10^2
10
0
.01
.02
.03
.04
.05
.06
.07
.08
.09
.1
.11
.12
.13
.14
.15
Legend:
Mod(1) structure
Vdll structure
Mod(2) structure
Mod(3) structure
Mod(4) structure


Helix(k) expected traversal
Traversed
Probability of error
10^7
10^6
10^5
10^4
10^3
10^2
10
0
.02
.04
.06
.08
.1
.12
.14
.16
.18
.2
.22
.24
.26
.28
.3
Legend:
Helix(3) structure
Helix(4) structure



Trees Connected
Connected
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0 .01 .02 .03 .04 .05 .06 .07 .08 .09 .1 .11 .12 .13 .14 .15
Probability of error
Legend:
Predicted
Sibling-linked tree
Checksummed tree



Trees expected traversal
Traversed
10^5
5
2
10^4
5
2
10^3
5
2
10^2
5
2
10
0 .01 .02 .03 .04 .05 .06 .07 .08 .09 .1 .11 .12 .13 .14 .15
Probability of error
Legend:
Checksummed tree
Sibling-linked tree

APPENDIX F

Analysis of local correction using Markov models

F.58. Explanation

This appendix analyses the behaviour of some of the local correction algorithms described in Chapters 4, 5, and 7, using the techniques described in Chapter 8 and Appendix E. We will study two issues. Firstly, in order to remain compatible with the results presented in Appendix C, we will determine the expected number of instances that can be corrected, given that we attempt local correction on 1000 instances, each containing 100 data nodes, when selected components have a constant probability, p, of being in error. In our Markov models we assume that multiple errors do not conspire to assist or mislead correction algorithms. The results of this study will be compared with empirical data.

Secondly, we will determine, for various values of p, the expected number of nodes traversed by correction algorithms prior to failing, given that we are examining an instance containing at least this number of nodes. All of these results will be presented graphically.

F.59. Markov models for linked lists

We will compare local correction algorithms operating on a number of different unkeyed double-linked list structures. These structures are the checksummed double-linked list containing two additional checksum components, the virtual double-linked list (VDLL) [83], the mod(2) linked list, and the mod(3) linked list. Each node in the checksummed double-linked list was corrected by using the error correcting code presented in Appendix A1. The VDLL structure was corrected as described in [83], and the mod(2) linked list was corrected using the historical 1-local-correction algorithm described in Appendix C. The mod(3) linked list was corrected using the 2-selective-local-correction algorithm described in Chapter 5.

In each of the Markov models presented below we assume that local correction is attempting to determine the correct address of node N_x. If the correct address of this node cannot be determined, because of the errors in the correction locality, then the Markov model enters a final state. Otherwise the Markov model proceeds through all subsequent nodes whose addresses are necessarily identifiable given the errors in the locality, before trying to determine the address of some new N_x.

Somewhat surprisingly, the Markov model for performing local correction of the virtual double linked list was discovered to be identical to the Markov model for performing local correction in the mod(2) linked list. Results are not therefore presented for the local correction of virtual double linked lists.

Although we succeeded in producing a Markov model for the helix(3) local correction algorithm, the generating function produced from the helix(3) Markov model exceeded 100,000 characters, and was computationally intractable using available computing facilities.

1-local-correction of Checksummed linked list			
This State	Next State	Generating Function	Comments
0	1	$1 - p$	1st component correct
0	2	p	Error in 1st component
1	3	$1 - p$	Components 1 and 2 ok
1	4	p	Component 1 ok, 2 bad
2	5	$1 - p$	Component 1 bad, 2 ok
2	-1	p	Components 1 and 2 bad
3	6	$1 - p$	Components 1, 2 and 3 ok
3	7	p	Components 1 and 2 ok, 3 bad
4	8	$1 - p$	Components 1 and 3 ok, 2 bad
4	-1	p	Components 2 and 3 bad
5	9	$1 - p$	Components 2 and 3 ok, 1 bad
5	-1	p	Components 1 and 3 bad
6	0	$z(1 - p)$	All components correct
6	0	zp	Only component 4 bad
7	0	$z(1 - p)$	Only component 3 bad
7	-1	p	Components 3 and 4 bad
8	0	$z(1 - p)$	Only component 2 bad
8	-1	p	Components 2 and 4 bad
9	0	$z(1 - p)$	Only component 1 bad
9	-1	p	Components 1 and 4 bad

1-local-correction of Mod($k \geq 2$) linked list			
0	1	$1-p$	$N_{x+k} \cdot b_k = N_x$
0	2	p	$N_{x+k} \cdot b_k \neq N_x$
1	0	$z(1-p)$	$N_x \cdot f_1 = N_{x+1}$
1	3	p	$N_x \cdot f_1 \neq N_{x+1}$
2	3	$1-p$	$N_x \cdot f_1 = N_{x+1}$
2	-1	p	$N_x \cdot f_1 \neq N_{x+1}$
3	4	$1-p$	$N_{x+k-1} \cdot b_k = N_{x-1}$
3	-1	p	$N_{x+k-1} \cdot b_k \neq N_{x-1}$
4	0	$z^2(1-p)$	$N_{x-1} \cdot f_1 = N_x$
4	-1	p	$N_{x-1} \cdot f_1 \neq N_x$

2-selective-local-correction of Mod(3) linked list			
0	1	$1-p$	$N_{x+3} \cdot b_3 = N_x$
0	2	p	$N_{x+3} \cdot b_3 \neq N_x$
1	0	$z(1-p)$	$N_x \cdot f_1 = N_{x+1}$
1	3	p	$N_x \cdot f_1 \neq N_{x+1}$
2	4	$1-p$	$N_x \cdot f_1 = N_{x+1}$
2	5	p	$N_x \cdot f_1 \neq N_{x+1}$
3	6	$1-p$	$N_{x+2} \cdot b_3 = N_{x-1}$
3	7	p	$N_{x+2} \cdot b_3 \neq N_{x-1}$
4	8	$1-p$	$N_{x+2} \cdot b_3 = N_{x-1}$
4	7	p	$N_{x+2} \cdot b_3 \neq N_{x-1}$
5	7	$1-p$	$N_{x+2} \cdot b_3 = N_{x-1}$
5	-1	p	$N_{x+2} \cdot b_3 \neq N_{x-1}$
6	9	$1-p$	$N_{x-1} \cdot f_1 = N_x$
6	10	p	$N_{x-1} \cdot f_1 \neq N_x$
7	10	$1-p$	$N_{x-1} \cdot f_1 = N_x$
7	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
8	9	$1-p$	$N_{x-1} \cdot f_1 = N_x$
8	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
9	11	$1-p$	$N_{x+1} \cdot b_3 = N_{x-2}$
9	12	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$
10	13	$1-p$	$N_{x+1} \cdot b_3 = N_{x-2}$
10	-1	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$
11	0	$z^3(1-p)$	$N_{x-2} \cdot f_1 = N_{x-1}$
11	14	p	$N_{x-2} \cdot f_1 \neq N_{x-1}$
12	14	$1-p$	$N_{x-2} \cdot f_1 = N_{x-1}$
12	-1	p	$N_{x-2} \cdot f_1 \neq N_{x-1}$
13	15	$1-p$	$N_{x-2} \cdot f_1 = N_{x-1}$
13	-1	p	$N_{x-2} \cdot f_1 \neq N_{x-1}$
14	16	$1-p$	$N_x \cdot b_3 = N_{x-3}$
14	-1	p	$N_x \cdot b_3 \neq N_{x-3}$
15	17	$1-p$	$N_x \cdot b_3 = N_{x-3}$
15	-1	p	$N_x \cdot b_3 \neq N_{x-3}$
16	9	$z^2(1-p)$	$N_{x-3} \cdot f_1 = N_{x-2}$
16	-1	p	$N_{x-3} \cdot f_1 \neq N_{x-2}$
17	0	$z^4(1-p)$	$N_{x-3} \cdot f_1 = N_{x-2}$
17	-1	p	$N_{x-3} \cdot f_1 \neq N_{x-2}$

3-selective-local-correction of Helix(3) linked list $\{q := 1 - p\}$											
0	1	q	$N_{x+3} \cdot b_3 = N_x$	19	30	q	$N_{x+1} \cdot b_3 = N_{x-2}$	37	34	zq	$N_x \cdot f_1 = N_{x+1}$
0	2	p	$N_{x+3} \cdot b_3 \neq N_x$	19	31	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$	37	45	p	$N_x \cdot f_1 \neq N_{x+1}$
1	3	q	$N_{x+2} \cdot b_2 = N_x$	20	24	z^2q^5	Last 2 votes ok	38	21	zq^3	Last vote ok
1	4	p	$N_{x+2} \cdot b_2 \neq N_x$	20	-1	$1 - q^5$	Otherwise	38	-1	$1 - q^3$	Otherwise
2	4	q	$N_{x+2} \cdot b_2 = N_x$	21	24	zq	$N_x \cdot f_1 = N_{x+1}$	39	3	zq	$N_{x+2} \cdot b_2 = N_x$
2	5	p	$N_{x+2} \cdot b_2 \neq N_x$	21	32	p	$N_x \cdot f_1 \neq N_{x+1}$	39	33	p	$N_{x+2} \cdot b_2 \neq N_x$
3	0	zq	$N_x \cdot f_1 = N_{x+1}$	22	30	q	$N_{x+1} \cdot b_3 = N_{x-2}$	40	34	z^2q^3	Last vote ok
3	6	p	$N_x \cdot f_1 \neq N_{x+1}$	22	31	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$	40	-1	$1 - q^3$	Otherwise
4	6	q	$N_x \cdot f_1 = N_{x+1}$	23	24	z^2q^5	Last 2 votes ok	41	24	z^2q^2	Last vote ok
4	7	p	$N_x \cdot f_1 \neq N_{x+1}$	23	-1	$1 - q^5$	Otherwise	41	-1	$1 - q^2$	Otherwise
5	8	q	$N_{x+1} \cdot f_1 = N_x$	24	3	zq	$N_x \cdot f_1 = N_{x+1}$	42	46	q	$N_{x+1} \cdot b_3 = N_{x-2}$
5	-1	p	$N_{x+1} \cdot f_1 \neq N_x$	24	33	p	$N_x \cdot f_1 \neq N_{x+1}$	42	47	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$
6	9	q	$N_{x+2} \cdot b_3 = N_{x-1}$	25	34	zq	$N_x \cdot b_2 = N_{x-2}$	43	48	q	$N_{x-1} \cdot f_1 = N_x$
6	10	p	$N_{x+2} \cdot b_3 \neq N_{x-1}$	25	35	p	$N_x \cdot b_2 \neq N_{x-2}$	43	49	p	$N_{x-1} \cdot f_1 \neq N_x$
7	11	q	$N_{x+2} \cdot b_3 = N_{x-1}$	26	27	z^2q^3	Last vote ok	44	48	q	$N_x \cdot f_1 = N_{x+1}$
7	12	p	$N_{x+2} \cdot b_3 \neq N_{x-1}$	26	-1	$1 - q^3$	Otherwise	44	50	p	$N_x \cdot f_1 \neq N_{x+1}$
8	13	q	$N_x \cdot f_1 = N_{x+1}$	27	34	q	$N_{x+2} \cdot b_2 = N_x$	45	33	zq	$N_{x-1} \cdot f_1 = N_x$
8	14	p	$N_x \cdot f_1 \neq N_{x+1}$	27	36	p	$N_{x+2} \cdot b_2 \neq N_x$	45	-1	p	$N_{x-1} \cdot f_1 \neq N_x$
9	3	zq	$N_{x-1} \cdot f_1 = N_x$	28	37	zq^3	Last vote ok	46	33	zq	$N_x \cdot b_2 = N_{x-2}$
9	16	p	$N_{x-1} \cdot f_1 \neq N_x$	28	-1	$1 - q^3$	Otherwise	46	51	p	$N_x \cdot b_2 \neq N_{x-2}$
10	17	q	$N_{x+1} \cdot b_3 = N_{x-2}$	29	27	zq	$N_x \cdot f_1 = N_{x+1}$	47	27	z^2q^2	Last vote ok
10	18	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$	29	38	p	$N_x \cdot f_1 \neq N_{x+1}$	47	-1	$1 - q^2$	Otherwise
11	19	q	$N_{x-1} \cdot f_1 = N_x$	30	39	zq	$N_x \cdot b_2 = N_{x-2}$	48	52	q	$N_{x+1} \cdot b_3 = N_{x-2}$
11	20	p	$N_{x-1} \cdot f_1 \neq N_x$	30	40	p	$N_x \cdot b_2 \neq N_{x-2}$	48	53	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$
12	21	zq^5	Last 2 votes ok	31	27	z^2q^3	Last vote ok	49	24	z^2q^4	Last 2 votes ok
12	-1	$1 - q^5$	Otherwise	31	-1	$1 - q^3$	Otherwise	49	-1	$1 - q^4$	Otherwise
13	22	q	$N_{x+2} \cdot b_3 = N_{x-1}$	32	3	z^2q	$N_{x-1} \cdot f_1 = N_x$	50	24	z^2q^4	Last 2 votes ok
13	23	p	$N_{x+2} \cdot b_3 \neq N_{x-1}$	32	41	p	$N_{x-1} \cdot f_1 \neq N_x$	50	-1	$1 - q^4$	Otherwise
14	24	z^2q^6	Last votes ok	33	0	z^2q	$N_{x-1} \cdot f_1 = N_x$	51	34	z^2q^2	Last vote ok
14	-1	$1 - q^6$	Otherwise	33	42	p	$N_{x-1} \cdot f_1 \neq N_x$	51	-1	$1 - q^2$	Otherwise
16	25	q	$N_{x+1} \cdot b_3 = N_{x-2}$	34	33	q	$N_{x+2} \cdot b_2 = N_x$	52	3	z^2q	$N_x \cdot b_2 = N_{x-2}$
16	26	p	$N_{x+1} \cdot b_3 \neq N_{x-2}$	34	43	p	$N_{x+2} \cdot b_2 \neq N_x$	52	54	p	$N_x \cdot b_2 \neq N_{x-2}$
17	27	zq	$N_x \cdot b_2 = N_{x-2}$	35	34	z^2q^3	Last vote ok	53	27	z^2q^2	Last vote ok
17	28	p	$N_x \cdot b_2 \neq N_{x-2}$	35	-1	$1 - q^3$	Otherwise	53	-1	$1 - q^2$	Otherwise
18	29	zq^3	Last vote ok	36	44	q	$N_{x-1} \cdot f_1 = N_x$	54	34	z^2q^2	Last vote ok
18	-1	$1 - q^3$	Otherwise	36	-1	p	Disconnected	54	-1	$1 - q^2$	Otherwise

F.60. Markov model for binary trees

We also present the results of using Markov models to study local correction of the checksummed binary tree described in Chapter 4, and the sibling-linked binary tree described in Chapter 7.

In the analysis of the sibling-linked tree we use Markov models to separately predict the probability of performing correction when only pointers may contain errors, and when only keys and checksums may contain errors. The probability of correcting a sibling-linked tree when errors may occur in pointers, keys, and checksums, is approximated by multiplying these two probabilities.

If distinct dummy variables are used in distinct generating functions, and the Markov model for correcting pointers is extended by also allowing keys to contain errors with probability p, then the generating function describing correction of both keys and pointers in a sibling-linked tree is merely the product of the generating function describing the correction of each.

Unfortunately, since this composite generating function contains two distinct dummy variables, the techniques described in Chapter 8 cannot be used to identify the expected number of nodes traversed by an algorithm correcting both keys and pointers. Therefore, we present separately the expected number of nodes traversed in a sibling-linked tree when each of these two types of error occurs.

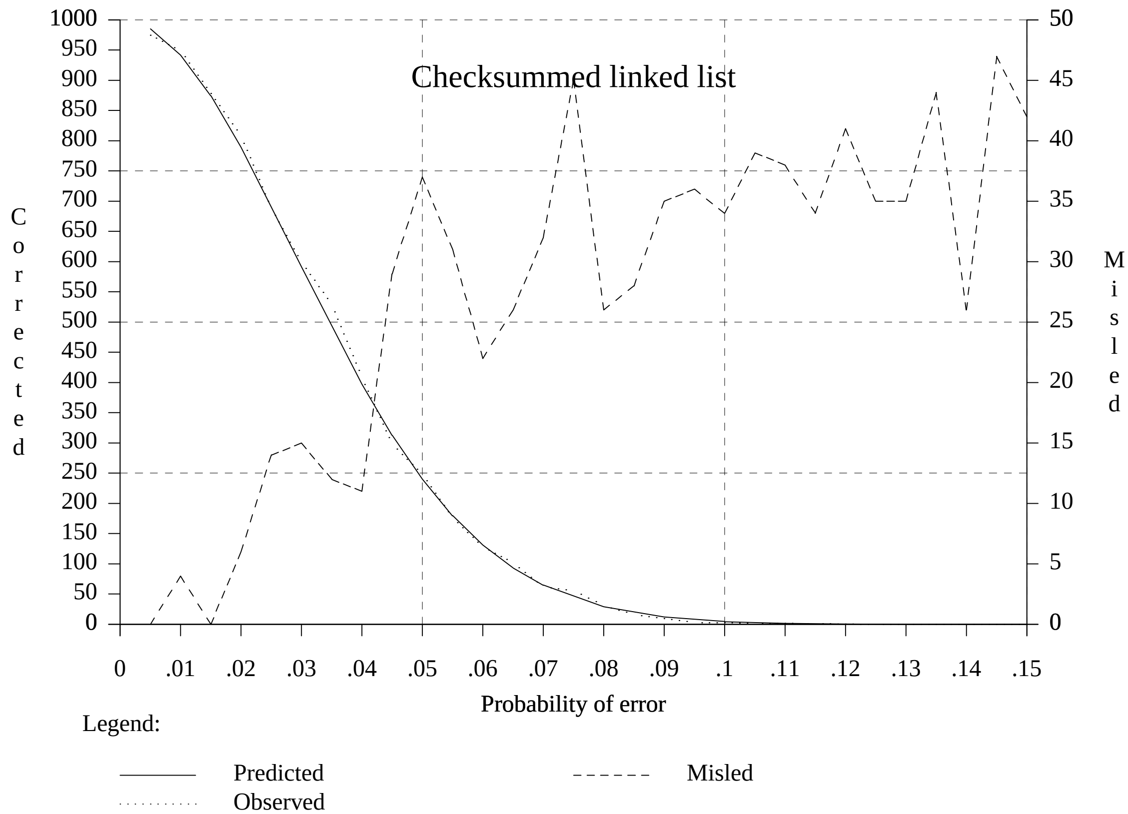
1-local-correction of Checksummed binary tree			
0	1	$1-p$	1st component ok
0	2	p	1st component bad
1	3	$1-p$	2nd component ok
1	4	p	2nd component bad
2	0	$z(1-p)\,4$	At least two errors
2	-1	$1-(1-p)\,4$	At least two errors
3	5	$1-p$	3rd component ok
3	6	p	3rd component bad
4	0	$z(1-p)\,3$	No other errors
4	-1	$1-(1-p)\,3$	At least two errors
5	0	$z(1-p)$	4th component ok
5	7	p	4th component bad
6	0	$z(1-p)\,2$	No other errors
6	-1	$1-(1-p)\,2$	At least two errors
7	0	$z(1-p)$	5th component ok
7	-1	p	At least two errors

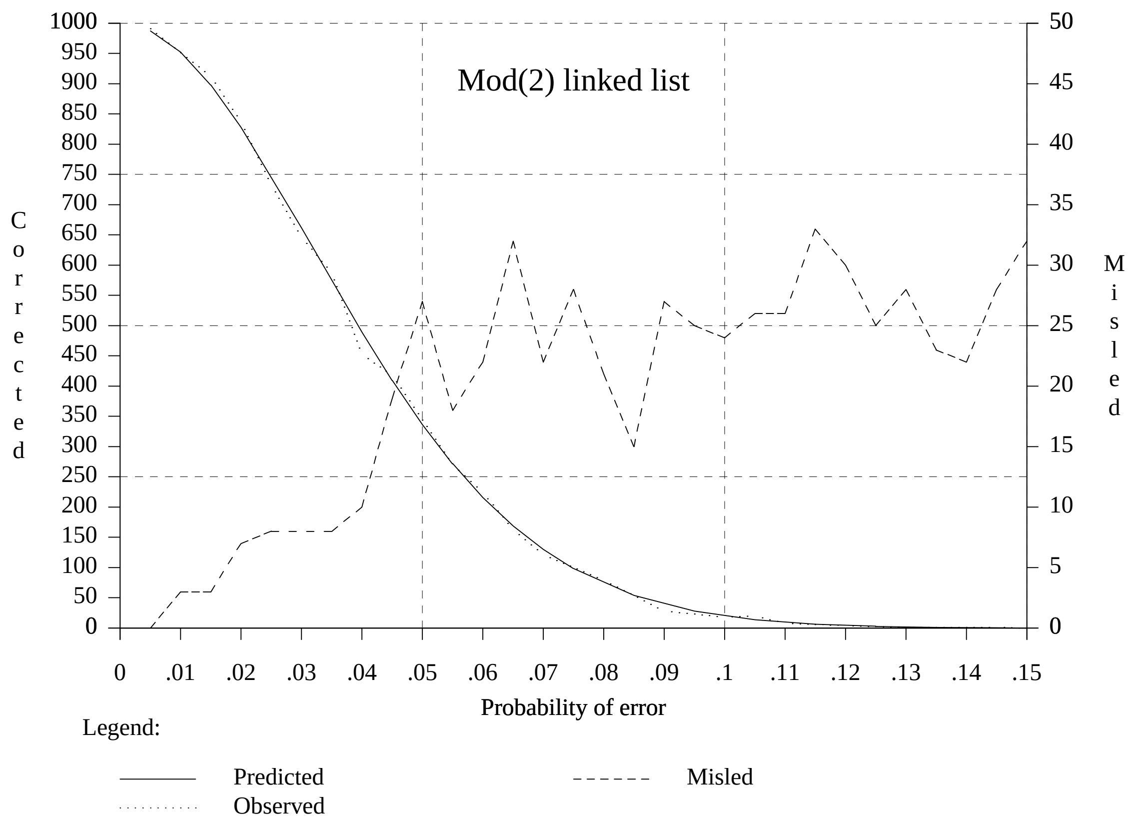
1-local-correction of keys in Sibling-linked tree			
0	1	$1-p$	$N_x \cdot$ s ok
0	2	p	$N_x \cdot$ s bad
1	0	$z(1-p)$	$N_x \cdot$ k ok
1	3	p	$N_x \cdot$ k bad
2	4	$1-p$	$N_x \cdot$ k ok
2	-1	p	$N_x \cdot$ k bad
3	5	$1-p$	$N_{x+1} \cdot$ k ok
3	-1	p	$N_{x+1} \cdot$ k bad
4	6	$1-p$	$N_{x+1} \cdot$ k ok
4	-1	p	$N_{x+1} \cdot$ k bad
5	0	$z^2(1-p)$	$N_{x+1} \cdot$ s ok
5	-1	p	$N_{x+1} \cdot$ s bad
6	0	$z^2(1-p)$	$N_{x+1} \cdot$ s ok
6	-1	p	$N_{x+1} \cdot$ s bad

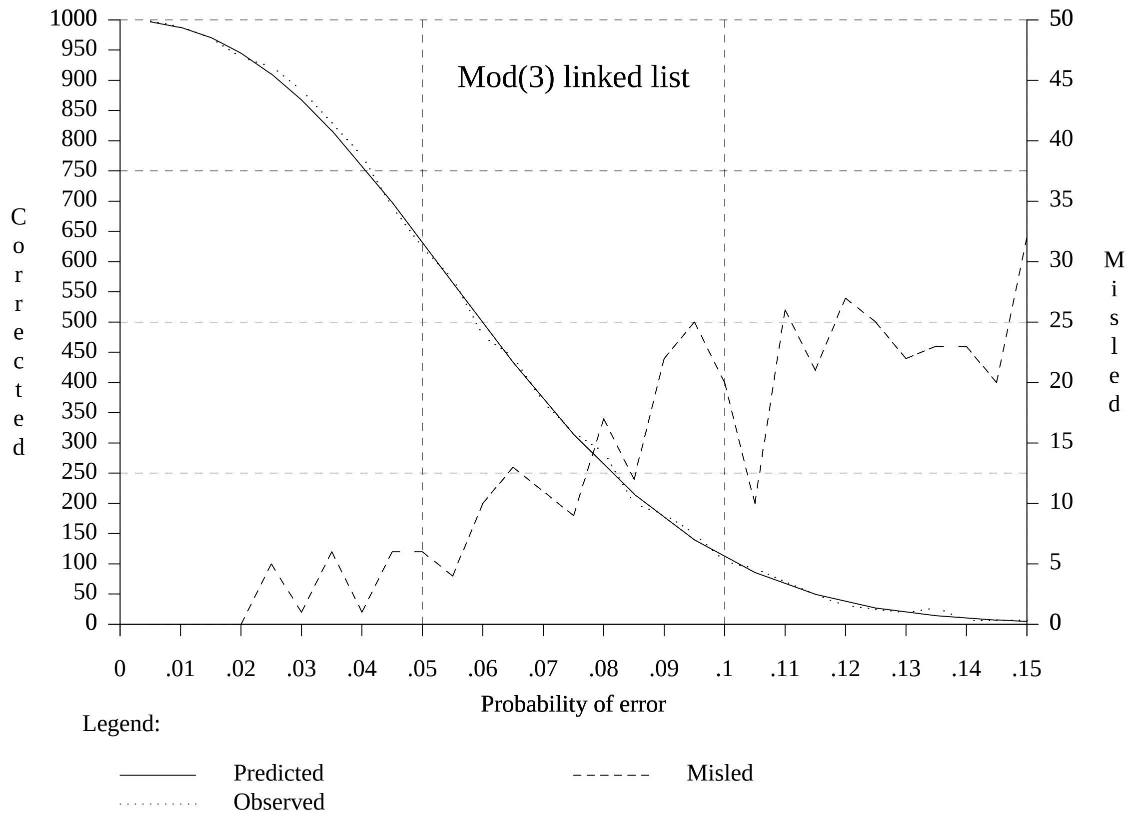
1-local-correction of pointers in Sibling-linked tree			
0	1	1/3	At a leaf
0	3	1/3	At incomplete node
0	7	1/3	At full node
1	0	$z(1-p)$	Left link null
1	2	p	Left link bad
2	0	$z(1-p)$	Right link null
2	-1	p	Both links bad
3	4	$1-p$	Left link ok
3	5	p	Left link bad
4	0	$z(1-p)$	Right link ok
4	6	p	Right link bad
5	6	$1-p$	Right link ok
5	-1	p	Disconnected
6	0	$z(1-p)$	Child arc ok
6	-1	p	Child arc bad
7	8	$1-p$	Left link ok
7	9	p	Left link bad
8	10	$1-p$	Right link ok
8	11	p	Right link bad
9	11	$1-p$	Right link ok
9	-1	p	Disconnected
10	0	$z(1-p)$	First arc ok
10	12	p	First arc bad
11	0	$z(1-p)^2$	Neither arc bad
11	-1	$1-(1-p)^2$	At least one arc bad
12	0	$z(1-p)$	Other arc ok
12	-1	p	At least one arc bad

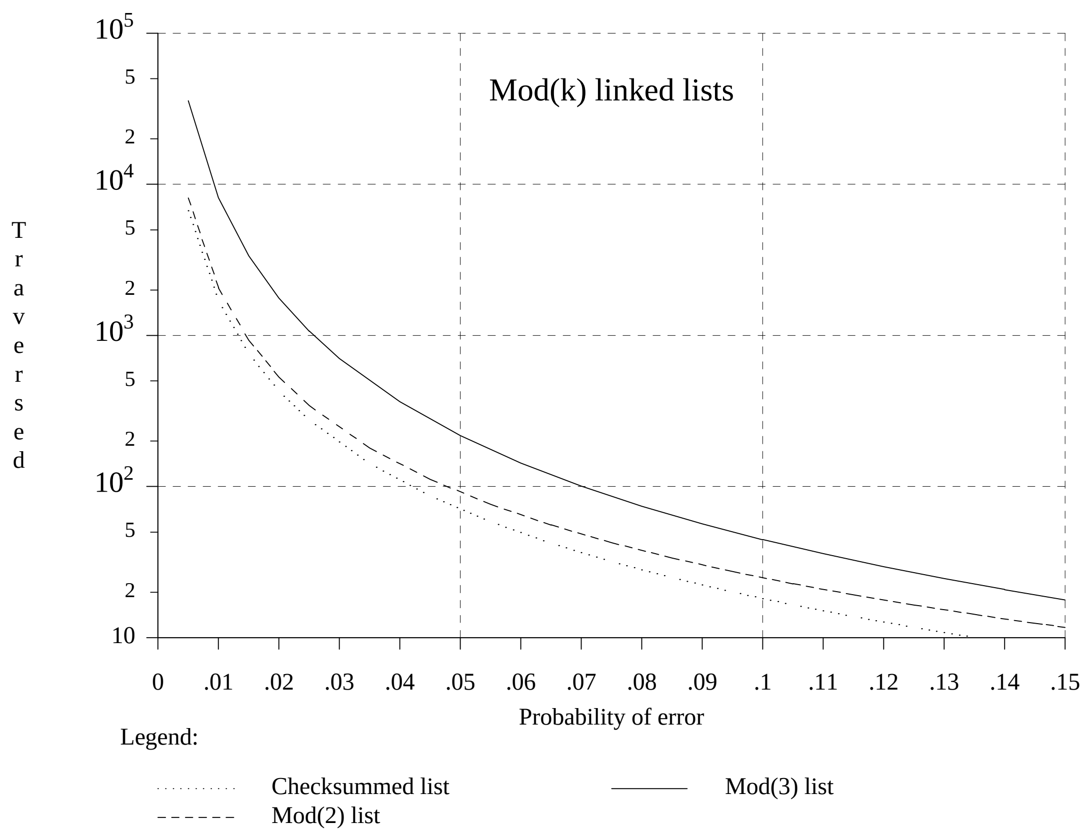
F.61. Graphs

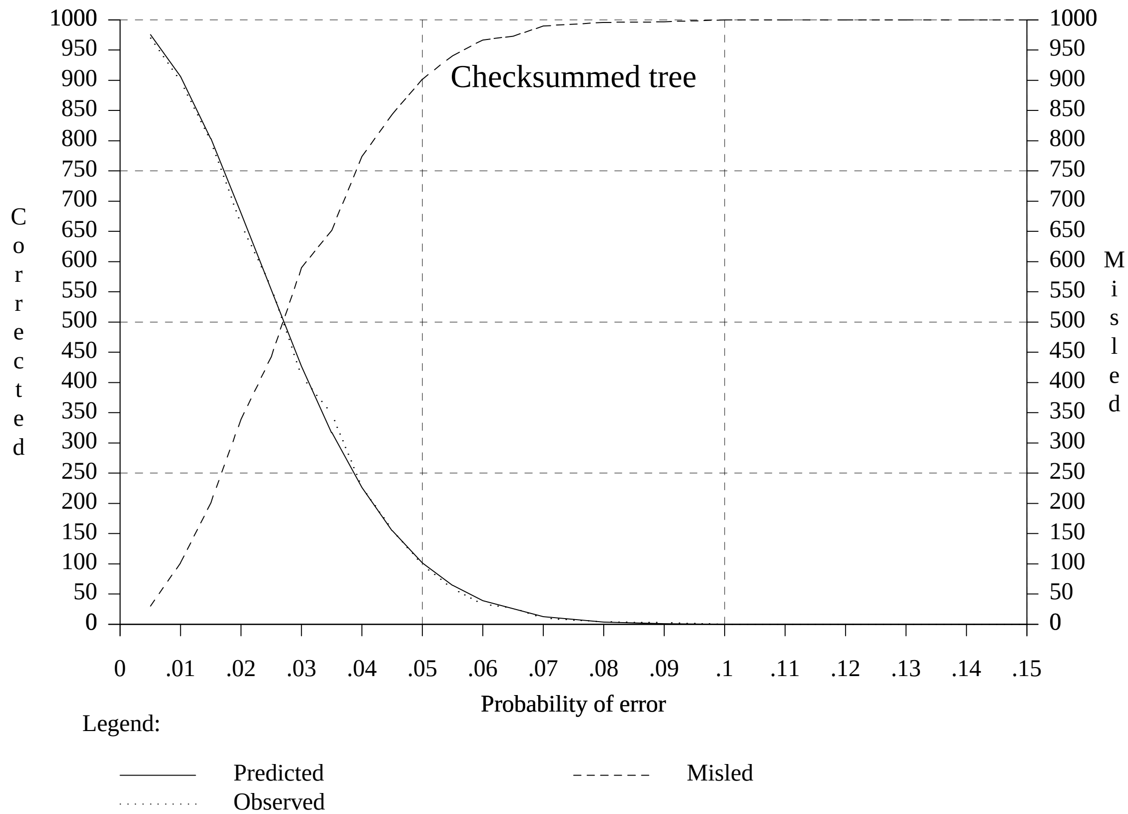
In the following graphs, predicted results were obtained using Markov models presented earlier in this appendix and the Maple code presented at the beginning of appendix E. The observed results were obtained from corresponding empirical studies. In the empirical studies storage structure instances contained 100 data nodes. Binary trees contained approximately the same number of full, incomplete, and leaf nodes, but no effort was made to ensure that these trees were balanced. Pointers, and when appropriate keys and checksums, in each storage structure instance were assigned arbitrary incorrect values with the indicated constant probability and a test performed to determine if the instance was correctable. This test was repeated 1000 times and the total number of times that the instance was corrected recorded. This exercise was repeated under a variety of different error probabilities.

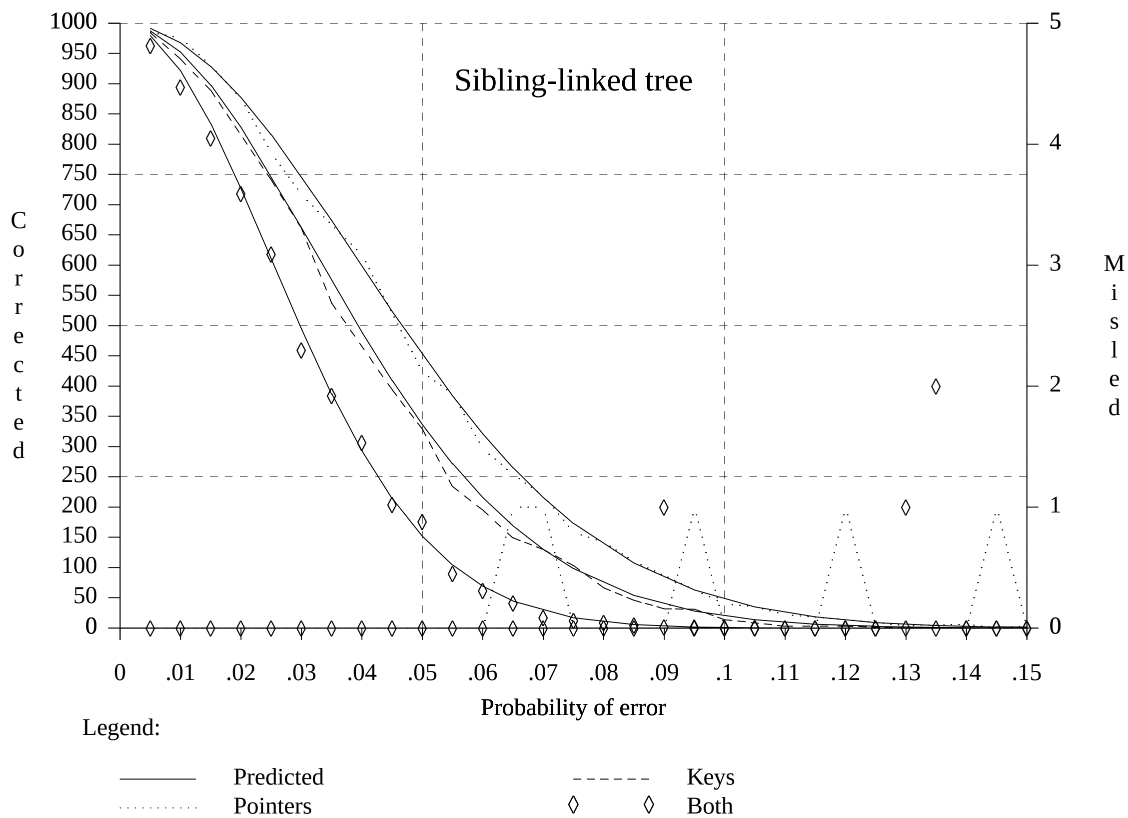

Checksummed linked list
Corrected
Misled
0 50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950 1000
0 5 10 15 20 25 30 35 40 45 50
0 .01 .02 .03 .04 .05 .06 .07 .08 .09 .1 .11 .12 .13 .14 .15
Probability of error
Legend:
Predicted
Observed
Misled

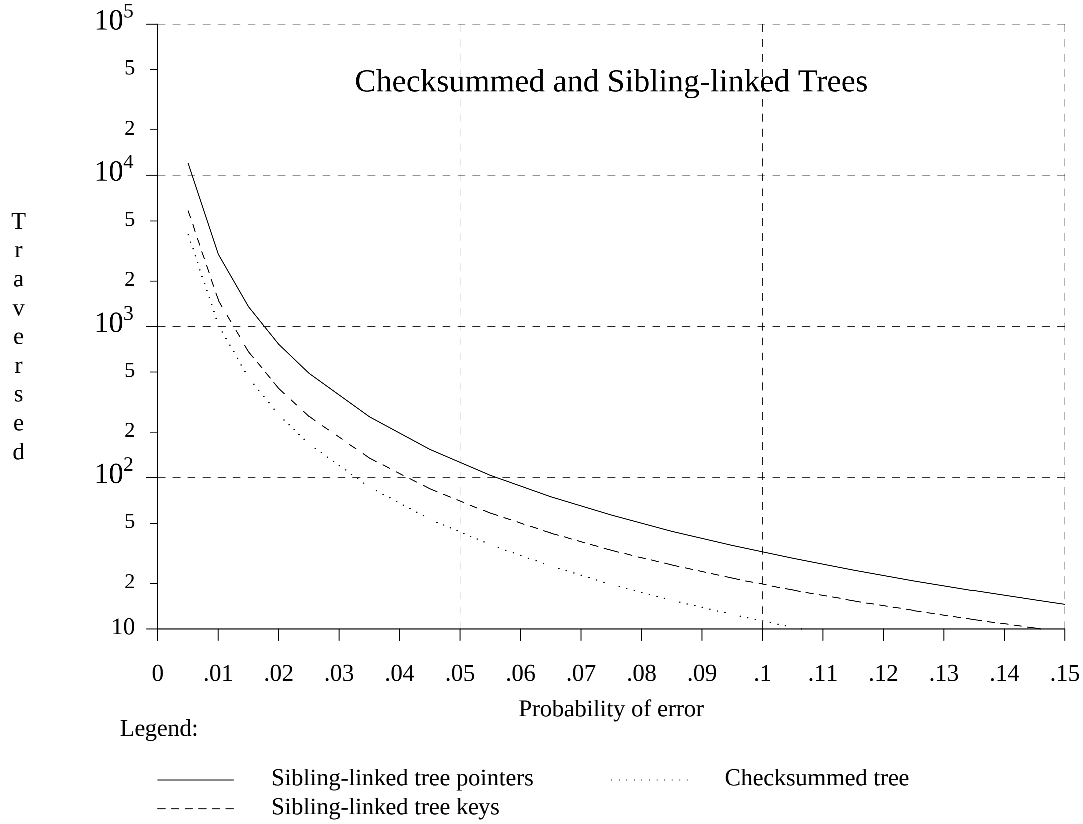

Mod(2) linked list
Corrected
Misled
0 50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950 1000
0 5 10 15 20 25 30 35 40 45 50
0 .01 .02 .03 .04 .05 .06 .07 .08 .09 .1 .11 .12 .13 .14 .15
Probability of error
Legend:
Predicted
Observed
Misled


Mod(3) linked list
Corrected
Misled
1000
950
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
50
45
40
35
30
25
20
15
10
5
0
0
.01
.02
.03
.04
.05
.06
.07
.08
.09
.1
.11
.12
.13
.14
.15
Probability of error
Legend:
Predicted
Observed
Misled


Mod(k) linked lists
Traversed
10^5
5
2
10^4
5
2
10^3
5
2
10^2
5
2
10
0
.01
.02
.03
.04
.05
.06
.07
.08
.09
.1
.11
.12
.13
.14
.15
Probability of error
Legend:
Checksummed list
Mod(3) list
Mod(2) list


Checksummed tree
Corrected
Misled
0
.01
.02
.03
.04
.05
.06
.07
.08
.09
.1
.11
.12
.13
.14
.15
Probability of error
Legend:
Predicted
Observed
Misled


Sibling-linked tree
Corrected
Misled
0
.01
.02
.03
.04
.05
.06
.07
.08
.09
.1
.11
.12
.13
.14
.15
Probability of error
Legend:
Predicted
Pointers
Keys
Both



References

"The DoD STARS Program," IEEE Computer 16(11) (November 1983). Special issue. J. A. Adams *et al*, "SDI: The grand experiment," IEEE Spectrum 22(9), pp. 34-64 (September 1985). A. V. Aho, J. E. Hopcroft, and J. D. Ullman, Data Structures and Algorithms, p. 167, Addison-Wesley, Reading, Mass. (1983). T. Anderson and R. Kerr, "Recovery blocks in action: A system supporting high reliability," Proc., 2nd Int. Conf. on Software Engineering, pp. 447-457, San Francisco, California (October 13-15, 1976). T. Anderson and B. Randell, Computing Systems Reliability, Cambridge University press (1979). T. Anderson and P. A. Lee, Fault Tolerance: Principles and Practice, Prentice-Hall, Englewood Cliffs, N. J. (1981). T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding, "An evaluation of software fault tolerance in a practical system," 15th Annual Symposium on Fault Tolerant Computing, pp. 140-145 (June 1985). M. M. Astrahan *et al*, "System R: Relational approach to database management," ACM Transactions on Database Systems 1(2), pp. 97-137 (June 1976). A. Avizienis, "Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing.," Proc. Int. Conf. on Reliable Software, pp. 458-464, Los Angeles, California (April 21-23, 1975). (Published as SIGPLAN Notices, vol. 10, no. 6, June 1975.). A. Avizienis, "Fault tolerant systems," IEEE Transactions on Computers C-25(12), pp. 1304-1312 (December 1976). A. Avizienis, "Fault tolerance: The survival attribute of digital systems," Proc. IEEE 66, pp. 1109-1125 (October 1978). F. T. Baker, "Chief programmer team management of production programming," IBM Systems Journal 11(1), pp. 56-73 (1972). F. T. Baker, "Structured programming in a production programming environment," Proc., Int. Conf. on Reliable Software, pp. 172-185, Los Angeles (April 21-23, 1975). J. P. Black, D. J. Taylor, and D. E. Morgan, "An introduction to robust data structures," Digest of Papers: 10th Annual Int. Symp. on Fault-Tolerant Computing, pp. 110-112, Kyoto, Japan (1-3 October 1980). J. P. Black, D. J. Taylor, and D. E. Morgan, "A case study in fault tolerant software," Software−Practice and Experience 11(2), pp. 145-157 (February 1981). J. P. Black, D. J. Taylor, and D. E. Morgan, "A robust B-tree implementation," Proc. 5th Int. Conf. on Software Engineering, pp. 63-70, San Diego, California (9-12 March 1981). J. P. Black, D. J. Taylor, and D. E. Morgan, "A compendium of robust data structures," Digest of Papers: 11th Annual Int. Symp. on Fault-Tolerant Computing, pp. 129-131, Portland, Maine (24-26 June 1981). J. P. Black, Analysis and design of systems of robust storage structures, Ph.D. Thesis, University of Waterloo, Ontario, Canada (July 1982). J. P. Black and D. J. Taylor, "A model for storage structures, encoding, and robustness," CS-84-45, Dept. of Computer Science, University of Waterloo (December 1984). J. P. Black and D. J. Taylor, "Local correctability in robust storage structures," CS-84-44, Dept. of Computer Science, University of Waterloo (December 1984). R. E. Blahut, Theory and practice of error control codes, pp. 54-55, Addison-Wesley (1983). G. S. Blair, J. R. Malone, and J. A. Mariani, "A critique of UNIX," Software−Practice and Experience 15(2), pp. 1125-1139 (December 1985). P. W. Bowman *et al*, "1A processor: Maintenance software," Bell System Technical Journal 56(2), pp. 255-287 (February 1977). J. B. Brenner, "A general model for integrity control," ICL Technical Journal(1), pp. 71-89 (November 1978). J. B. Brenner, C. P. Burton, A. D. Kitto, and E. C. P. Portman, "Project Little - an experimental utra reliable system," ICL Technical Journal, pp. 47-58 (May 1980). A. Brock, "An analysis of checkpointing," ICL Technical Journal 1(3), pp. 211-228 (November 1979). W. C. Carter and W. G. Bouricius, "A survey of fault-tolerant computer architecture and its evaluation," IEEE Computer 4(1), pp. 9-16 (January-February 1971). K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig, "Analytic models for rollback and recovery strategies in database systems," IEEE Transactions on Software Engineering SE-1(1), pp. 100-110 (March 1975). B. W. Char, K. O. Geddes, G. H. Gonnet, and S. M. Watt, Maple Users Guide, WATCOM Publications Limited, Waterloo (1985). L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," Digest of Papers: Eighth Annual Int. Symp. on Fault-Tolerant Computing, pp. 3-9, Toulouse, France (June 1978). K. Y. Chwa and S. L. Hakimi, "Schemes for fault tolerant computing: A comparison of modular redundancy and t-diagnosable systems," Information and Control 49(3), pp. 212-233 (June 1981). J. R. Connet, E. J. Pasternak, and B. D. Wagner, "Software defenses in real-time control systems," Proc. 2nd Fault Tolerant Computing Symp., pp. 94-99 (June 1972). J. R. Cordy and R. C. Holt, "Specification of Concurrent Euclid.," CSRG-115, Computer Systems Research Group, University of Toronto (July 1980). F. Cristian, "Exception handling and software fault tolerance," IEEE Transactions on Computers C-31(6), pp. 531-540 (June 1982). A. T. Dahbura and G. M. Masson, "A practical variation of the O(n**2.5) fault identification algorithm," Proc. Symposium on Fault Tolerant Computing, pp. 428-433 (June 1984). A. Damm, "The effectiveness of software error-detection mechanisms in real-time operating systems," 16th Annual Symposium on Fault Tolerant Computer Systems, pp. 171-176 (July 1986). I. J. Davis, Towards reliable file systems, M.Sc. Thesis, University of Toronto, Ontario, Canada (1982). I. J. Davis, "A locally correctable AVL tree," Digest of Papers: 17th Int. Symp. on Fault-Tolerant Computing, pp. 85-88, Pittsburgh, (6-8 July 1987). I. J. Davis, "Local correction of helix(k) lists," IEEE Transactions on Computers 38(5), pp. 718-724 (May 1989). I. J. Davis and D. J. Taylor, "Local correction of mod(k) lists," The Journal of Systems and Software 11(3), pp. 205-214. (March 1990). P. A. Dearnley, "An investigation into database resilience," The Computer Journal 19(2), pp. 117-121 (May 1976). R. A. DeMillo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," Communications ACM 22(5), pp. 271-280 (May 1979). P. J. Denning, "Fault tolerant operating systems," ACM Computing Surveys 8(4), pp. 359-389 (December 1976). M. E. D'Imperio, "Data structures and their representation in storage" in Annual Review in Automatic Programming 5, pp. 1-75, Pergamon Press, Oxford (1969). R. W. Drake and J. L. Smith, "Some technics for file recovery," Australian Computer Journal 3(4), pp. 162-170 (November 1971). J. R. Dunham, "Software errors in experimental systems having ultra-reliability requirements," 16th Annual Symposium on Fault Tolerant Computing Systems, pp. 158-164 (July 1986). D. Dunn, "Two observations on large software projects," ACM SIGSOFT Software Enginnering Notes 9(6), pp. 8-10 (October 1984). J. Earley, "Towards an understanding of data structures," Communications of the ACM 14(10), pp. 617-627 (October 1971). M. Edelberg, "Database contamination and recovery," ACM SIGFIDET Workshop on Data Description and Access Control (May 1974). A. L. Furtado, "Characterizing sets of data structures by the connectivity relation," Int. Journal of Computer and Information Sciences 5(2), pp. 89-110 (June 1976). J. R. Garman, "The ''Bug'' heard 'round the world," ACM SIGSOFT Software Engineering Notes 6(5), pp. 3-10 (October 1981). S. L. Gerhart and L. Yelowitz, "Observations of fallibility in applications of modern programming methologies," IEEE Transactions on Software Engineering SE-2(3), pp. 195-207 (September 1976). J. B. Goodenough, "Exception handling: Issues and a proposed notation," Communications of the ACM 18(12), pp. 683-696 (December 1975). E. Gorog, "Some new classes of cyclic codes used for blast-error correction," IBM Systems Journal 7(2), pp. 102-111 (April 73). C. C. Gotlieb and F. W. Tompa, "Choosing a storage schema," Acta Informatica 3, pp. 297-319 (1974). I. P. Goulden and D. M. Jackson, "An inversion theorem for cluster decompositions of sequences with distinguished subsequences," Journal of London Mathematics Society 20(3), pp. 567-576 (December 1979). I. P. Goulden and D. M. Jackson, Combinatorial Enumeration, John Wiley and Sons, New York (1983). D. H. Greene and D. E. Knuth, Mathematics for the Analysis of Algorithms, Birkh, Boston (1982). S. L. Hakimi and A. T. Amin, "Characterization of connection assignment of diagnosable systems," IEEE Transactions on Computers C-33(1), pp. 86-88 (January 1974). S. L. Hakimi and K. Nakajima, "Adaptive diagnosis: A new theory of t-fault diagnosable systems," Proc. 20th Annual Allerton Conf. on Comm., Control and Comp., pp. 231-240 (1982). S. L. Hakimi and K. Nakajna, "On adaptive systems diagnosis," IEEE Transactions on Computers C-33(3), pp. 234-240 (March 1984). V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, Computer Organization, McGraw-Hill (1984). R. W. Hamming, "Error detecting and error correcting codes," Bell System Technical Journal 26, pp. 147-160 (April 1950). C. A. R. Hoare, "Proof of correctness of data representations," Acta Informatica 1(4), pp. 271-281 (1972). F. K. Hwang, J. Korner, and V. K. W. Wei, "Selecting non-consecutive balls arranged in many lines," Journal of Combinatorial Theory, Series A 37, pp. 327-336 (1984). R. D. Joshi, "Reliable software systems," The Journal of Systems and Software 3(2), pp. 107-121 (June 1983). I. Kaplansky, "Solution of the ''Probleme des menages''," Bulletin American Mathematics Society 49, pp. 784-785 (October 1943). J. P. Kelly and A. Avizienis, "A Specification-oriented multi-version software experiment," 13th Annual International Symposium on Fault Tolerant Computing, pp. 120-126 (June 1983). J. C. Knight, N. G. Leveson, and L. D. St-Jean, "A large scale experiment in n-version programming," 15th Annual Symposium on Fault Tolerant Computing, pp. 135-139 (June 1985). J. C. Knight and N. G. Leveson, "An empirical study of failure probabilities in multi-version software," 16th Annual Symposium on Fault Tolerant Computer Systems, pp. 165-170 (July 1986). D. E. Knuth, The Art of Computer Programming, vol. 3: Sorting and Searching, p. Section 6.2, Addison-Wesley, Reading, Mass. (1973). W. H. Kohler, "A survey of techniques for synchronization and recovery in decentralized computer systems," Computing Surveys 13(2), pp. 149-183 (June 1981). J. Konvalina, "On the number of combinations without unit separation," Journal of combinatorial theory, Series A 31, pp. 101-107 (1981). T. J. Kowalski, "FSCK - The UNIX file system check program," Research report, Bell Laboratories, Murray Hill (May 1979). K. Kuspert, "Efficient error detection techniques for hash tables in database systems," 14th International Conference on Fault Tolerant Computing, pp. 198-203 (June 1984). J. H. Lala, "A Byzantine resilient fault tolerant computer for nuclear power plant applications," 16th Annual Symposium on Fault Tolerant Computer Systems, pp. 338-343 (July 1986). B. W. Lampson, "Redundancy and robustness in memory protection," Proc. IFIP, pp. 128-132, North Holland, Amsterdam (1974). T. G. Lewis and B. J. Smith, Computer principles of modeling and simulation, p. Section 8.3, Houghton Mifflin, Boston. (1979). C.-C. J. Li, P. P. Chen, and W. K. Fuchs, "Local concurrent error detection and correction in data structures using virtual backpointers," IEEE Transactions on Computers 38(11), pp. 1481-1492 (November 1989). H. Lin, "The software for star wars: An achilles heel?," MIT Technology Review (July 1985). B. H. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," ACM Transactions on Programming Languages and Systems 5(3), pp. 381-404 (July 1983). B. H. Liskov and S. N. Zilles, "Specification techniques for data abstraction," IEEE Transactions on Software Engineering SE-1(1), pp. 7-19 (March 1985). P. C. Lockemann and W. D. Knutsen, "Recovery of disk contents after system failure," Communications of the ACM 11(8), p. 542 (August 1968). G. H. Lohman and J. A. Muckstadt, "Optimal policy for batch operations: Backup, checkpointing, reorganisation and updating," ACM Transactions on Database Systems 2(3), pp. 209-222 (September 1977). D. B. Lomet, "Process structuring, synchronisation and recovery using atomic actions," SIGPLAN Notices 12(3), pp. 128-137 (March 1977). R. E. Lyons and Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," IBM Systems Journal 6(2), pp. 200-209 (April 1962). F. J. MacWilliams and N. J. A. Sloane, "The Theory of Error-correcting Codes (Part 1)" in North Holland Mathematical Library, North-Holland (1977). M. Marcotty, H. F. Ledgard, and G. V. Bochmann, "A sampler of formal definitions," ACM Computing Surveys 8(2), pp. 191-276 (June 1976). D. R. McGregor and J. R. Malone, "Design for a robust, simple and highly reliable filestore," Software−Practice and Experience 11(9), pp. 943-947 (September 1981). R. E. Mickens, Difference equations, Van Nostrand Reinhold (1987). H. D. Mills, "On the development of large reliable programs," Record, IEEE Symp. on Computer Software Reliability, pp. 155-159, New York City (April 30-May 2, 1973). D. E. Morgan and D. J. Taylor, "A survey of methods of achieving reliable software," Computer 10(2), pp. 44-53 (February 1977). J. K. Mullin, "Change area B-trees: A technique to aid error recovery," Computer Journal 24(4), pp. 367-373 (November 1981). J. I. Munro and P. V. Poblete, "Fault tolerance and storage reduction in binary search trees," Information and Control 62(2/3), pp. 210-218 (August/September 1984). G. J. Myers, Software reliability, John Wiley and Sons, New York (1976). D. L. Nelson and P. J. Leach, "The Architecture and applications of the Apollo Domain," IEEE Computer Graphics and Applications 4(4), pp. 58-66 (April 1984). P. G. Neumann, "Some computer-related disasters and other egregious horrors," ACM SIGSOFT Software Engineering Notes 10(1), pp. 6-7 (January 1985). D. L. Parnas, "Software aspects of strategic defense systems," American Scientist 73, pp. 432-440 (September/October 1985). (Reprinted in ACM SIGSOFT Software Engineering Notes 10-5 Oct 1985 pp15-23). W. W. Peterson and E. J. Weldon Jr., Error-Correcting Codes, MIT Press, Cambridge, Mass. (1972). F. P. Preparata, G. Metze, and R. T. Chien, "On the connection assignment problem of diagnosable systems," IEEE Transactions on Electronic Computers EC-16(6), pp. 848-854 (December 1967). H. Prodinger, "On the number of combinations without a fixed distance," Journal on combination theory, Series A 35, pp. 362-365 (1983). B. Randell, "System structure for software fault tolerance," IEEE Transactions on Software Engineering SE-1(2), pp. 220-232 (June 1975). B. Randell, "Reliable computing systems" in Operating Systems, An advanced course: Lecture notes in computer science, ed. G. Goos and J. Hartmanis 60, pp. 282-391, EDS (1978). B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability issues in computing system design," ACM Computing Surveys 10(2), pp. 123-165 (June 1978). D. D. Redell *et al*, "PILOT: An operating system for a personal computer," Communications of the ACM 23(2), pp. 81-91 (February 1980). N. W. Rickert, "The parable of the two programmers," ACM

SIGSOFT Software Engineering Notes 10(1), pp. 16-18 (January 1985). M. C. Sampaio and J. P. Sauve, “Robust trees,” Proc. 15th Int. Symp. on Fault Tolerant Computing, pp. 23-28, Ann Arbor, Michagan (June 19-21, 1985). H. H. Sayani, “Restart and recovery in transaction oriented information processing systems,” Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control, pp. 351-366 (May 1974). R. D. Schlichting and F. B. Schneider, “Fail stop processors: An approach to designing fault tolerant computing systems,” ACM Transactions on Computing Systems 1(3), pp. 222-238 (August 1983). S. C. Seth and R. Muralidhar, “Analysis and design of robust data structures,” Proc. 15th Int. Symp. on Fault Tolerant Computing, pp. 14-19, Ann Arbor, Michigan (June 19-21, 1985). S. K. Shrivastava and F. Panzieri, “The Design of a Reliable Remote Procedure Call Mechanism,” IEEE Transactions on Computers C-31(7), pp. 692-697 (July 1982). J. R. Sklaroff, “Redundancy Management Technique for Space Shuttle Computers,” IBM Journal of Research and Development 20(1), pp. 20-28 (January 1976). R. E. Strom and S. Yemini, “Typestate: A Programming Language Concept for Enhancing Software Reliability,” IEEE Transactions on Software Engineering SE-12(1), pp. 157-171 (January 1986). L. G. Stucki, “A case for software testing:...,” IEEE Transactions on Software Engineering SE-2(3), p. 194 (September 1976). A. G. Tanenbaum, “In defense of program testing; or, Correctness proofs considered harmful.,” SIGPLAN Notices 11(5), pp. 64-68 (May 1976). A. N. Tantawi and M. Ruschitzka, “Performance analysis of check pointing strategies,” ACM Transactions on Computing Systems 2(2), pp. 123-144 (May 1984). D. J. Taylor, Robust data structure implementations for software reliability, Ph.D. Thesis, University of Waterloo, Ontario, Canada (August 1977). D. J. Taylor, “Robust storage structures for data structures,” Research Report CS-80-36, Waterloo (October 1980). D. J. Taylor, D. E. Morgan, and J. P. Black, “Redundancy in data structures: Improving software fault tolerance,” IEEE Transactions on Software Engineering SE-6(6), pp. 585-594 (November 1980). D. J. Taylor, D. E. Morgan, and J. P. Black, “Redundancy in data structures: Some theoretical results,” IEEE Transactions on Software Engineering SE-6(6), pp. 595-602 (November 1980). D. J. Taylor and J. P. Black, “Principles of data structure error correction,” IEEE Transactions on Computers C-31(7), pp. 602-608 (July 1982). D. J. Taylor and J. P. Black, “Guidelines for storage structure error correction,” Proc. 15th Int. Symp. on Fault-Tolerant Computing, pp. 20-22, Ann Arbor, Michigan (June 19-21, 1985). D. J. Taylor and C. J. Seger, “Robust storage structures for crash recovery,” IEEE Transactions on Computers C-35(4), pp. 288-295 (April 1986). D. J. Taylor, Local correction of arbitrary linked lists, Unpublished paper, University of Waterloo, Ontario, Canada (April 1986). D. J. Taylor and J. P. Black, “Experimenting with data structures,” Software−Practice and Experience 16(5), pp. 443-456 (May 1986). D. J. Taylor and J. P. Black, “A locally correctable B-tree implementation,” Computer Journal 29(3), pp. 269-276 (June 1986). D. J. Taylor and M. L. Wright, “Backward error recovery in a UNIX environment,” Proc. 16th Annual Int. Symp. on Fault-Tolerant Computing Systems, pp. 118-123 (July 1986). D. J. Taylor, “Crash recovery for binary trees,” Proc. 17th Int. Symp. on Fault-Tolerant Computing, pp. 80-84, Pittsburgh (July 6-8, 1987). N. Theuretzbacher, “Votrics: Voting triple modular computing system,” 16th Annual Symposium on Fault Tolerant Computing, pp. 144-150 (July 1986). A. B. Tonik, “Checkpoint, restart and recovery: Selected annotated bibliography,” FDT Bulletin of the ACM SIGMOD 7(3 and 4), pp. 72-76 (1975). P. G. Trei, P. L. Zweig, and A. Sullivan, “Gentlemen prefer platinum to bonds -- 32 billion dollar overdraft,” ACM SIGSOFT Software Engineering Notes 11(1), pp. 3-7 (Jan 1986). J. S. M. Verhofstad, “Recovery techniques for database systems,” ACM Computing Surveys 10(2), pp. 167-196 (June 1978). B. J. Walker, R. A. Kemmerer, and G. J. Popek, “Specification and verification of the UCLA "UNIX" kernel,” Communications of the ACM, pp. 118-131 (February 1980). L. Weissman and G. M. Stacey, “An interface system for improving reliability of software systems,” Record, IEEE Symp. on Computer Software Reliability, pp. 136-142, New York

City (April 30-May 2, 1973). J. H. Wensley *et al*, “SIFT: Design and analysis of a fault-tolerant computer for aircraft control,” Proc. IEEE 66(10), pp. 1240-1255 (October 1978). M. V. Wilkes, “On preserving the integrity of data bases,” The Computer Journal 15(3), pp. 191-194 (August 1972). K. Yoshihara, Y. Koga, and T. Ishihara, “A robust data structure scheme with checking loops,” Digest of Papers: 13th Annual Int. Symp. on Fault Tolerant Computing, pp. 241-248, Milan, Italy (June 28-30, 1983). D. Zobel, “The deadlock problem: A classifying bibliography,” Operating Systems Review 17(4), pp. 6-15 (October 1983).