T-CP/38/5//26769

Local Correction of Helix( ) Lists

IAN J. DAVIS

Reprinted from
IEEE TRANSACTIONS ON COMPUTERS
Vol. 38, No. 5, 1989

Local Correction of Helix(k) Lists

IAN J. DAVIS

***Abstract*—A helix(k) list is a newly defined robust multiply linked list having k pointers in each node. In general, the ith pointer in each node addresses the ith previous node. However, the first pointer in each node addresses the next node, rather than the previous. This paper presents an algorithm for performing local correction in a helix($k \geq 3$) list. Given the assumption that at most k errors are encountered during any single correction step, this algorithm performs correction whenever possible, and otherwise reports failure. The algorithm generally reports failure only if all k pointers addressing a specific node are damaged, causing this node to become disconnected. However, in a helix(3) structure, one specific type of damage that causes disconnection is indistinguishable from alternative damage that does not. This also causes the algorithm to report failure.**

***Index Terms*—Linked lists, local error correction, robust storage structures, software fault tolerance.**

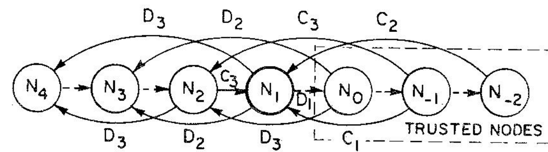


Fig. 1. Pointers used in a correct helix(3) locality.

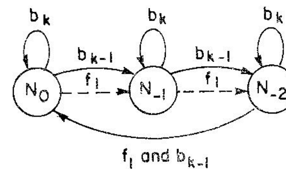


Fig. 2. An empty instance of a helix($k = 3$) structure.

I. Introduction

A HELIX(k) storage structure is a circular multiply-linked list of nodes, in which each node contains k pointers. In general, the ith pointer in each node links that node to the ith previous node. However, the first pointer in each node addresses the next node, rather than the previous (Fig. 1). A particular *instance* of a helix(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 multiply linked list of nodes, and are the only nodes in the instance when the instance is *empty* (Fig. 2). Each node within an instance contains an *identifier* whose value uniquely identifies the instance to which this node belongs. A *count* of the number of nonheader 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*.

Although a helix(k) structure contains considerable redundancy, a small number of well-chosen errors can produce an instance which is not correctable [7]. 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 [4], [5], [8], [9].

A local correction procedure visits all of the components of a storage structure instance in some deterministic order, by following pointers from the headers of the instance, and corrects errors when these are first encountered. Having ensured that a component is correct, this component becomes *trusted.* 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. Informally, these are the constraints that are imposed on a local correction procedure. More precise characterizations of such procedures [3] are too complex to be attempted here.

The local correction procedure described in this paper operates under the assumption that at most k errors occur in any locality. When presented with a set of header nodes whose addresses are known, it proceeds backwards from these header nodes through the helix($k \geq 3$) instance iteratively attempting to identify the correct address of the previous node. This previous node is called the *target.*

Having established the location of the target, back pointers that should address this target can be corrected, as can the forward pointer and identifier in this target. Having performed any necessary corrections, these components become trusted, and the target node becomes the *last* trusted node. Alternatively, having established that no correct pointer addresses the target, it can be reported that the target node is *disconnected.*

II. Notation

Nodes will be labeled 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 labeled 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

Manuscript received September 10, 1986; revised September 28, 1987 and April 22, 1988.

The author is with the Department of Computer Science, University of Waterloo, Waterloo, Ont., Canada N2L 3G1.

IEEE Log Number 8826769.

0018-9340/89/0500-0718$01.00 © 1989 IEEE

appropriate, superscripts will indicate the number of consecutive occurrences of a pointer type within a path.

One method of attempting to identify the target is to use *votes* [3]. 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 labeled 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 labeled D and 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. Each candidate *receives a vote* equal to the number of votes supporting it. If the candidate is not the target, then it is an *incorrect* candidate.

The following votes are used in this paper:

Vote	Path followed	Compared to node or path
C_i, $1 \leq i < k$	$N_{-i} \cdot b_{i+1}$	
C_k	$N_{2-k} \cdot b_k \cdot f_1$	
D_1	$N_n \cdot f_1$	N_0
D_i, $2 \leq i < k$	$N_n \cdot b_i$	$N_0 \cdot b_{i+1}$
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 (Fig. 1) $N_0 \cdot b_2$ is the only backpointer addressing N_2 that is not used by any constructive vote. For helix($k \geq 4$) structures, other backpointers have this property and can be used instead of $N_0 \cdot b_2$ if so desired.

III. Theoretical Results

It is assumed throughout this section that at most k errors occur in any single locality, and that the Valid State Hypothesis [7] holds. This asserts that, in the absence of errors, identifiers and pointers within the instance being corrected contain information that differs from information occurring at the same offset in other nodes within the node space. Without some assumption about the numbers of errors occurring in a locality, and the number of errors seen when invalid components are examined, little can be said about the behavior of any local correction algorithm.

Theorem 1 shows how an algorithm can detect and correct up to k errors in the empty instance. Subsequently, it is assumed that the instance being corrected is not empty. Under this assumption, Theorem 2 shows that the target receives at least k votes, and that incorrect candidates receive at most k votes, if distinct from the last k trusted nodes. Theorem 3 specifies when disconnection of the target can be suspected and, in all but one case, determined. Theorem 4 demonstrates how the target can be identified in all other cases. Collectively, these results can be used to construct a simple, efficient algorithm, that performs local correction whenever possible.

Theorem 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 (Fig. 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 nonempty 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. □

Theorem 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 $2k - r \geq k$ votes, and b) incorrect candidates receive at most $r \leq k$ 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 nonempty 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 others $2k - r$ 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

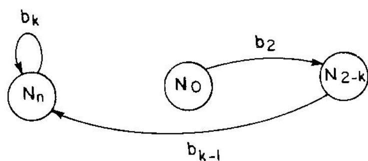


Fig. 3. Configuration if C_{k-2} and D_k share b_{k-1}.

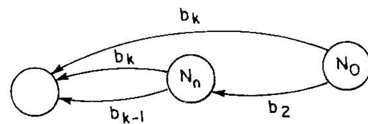


Fig. 4. Configuration if D_{k-1} and D_k share b_{k-1}.

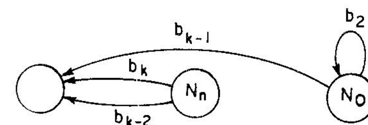


Fig. 5. Configuration if D_{k-2} and D_k share b_{k-1}.

known to be distinct. This error can be shared with at most one of C_{k-2} (Fig. 3), D_{k-1} (Fig. 4), and when $k \geq 4$ D_{k-2} (Fig. 5), since no other vote uses a b_{k-1} pointer. 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. □

Theorem 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 Theorem 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 N_n exists, and this candidate is the target node N_1, then only diagnostic votes contain errors, implying that $N_{2-k} \cdot b_k$ is correct. Conversely, if N_n is not the target, the path $N_0 \cdot b_2 \cdot f_1$ and all paths used by constructive votes incorrectly address 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 $N_0 \cdot b_2 \cdot f_1$ also occurs in the path $N_{2-k} \cdot b_k \cdot f_1$ used by C_k. Thus, $N_2 \cdot f_1$ contains an error but once again $N_{2-k} \cdot b_k$ 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 (Fig. 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.

The pointer $N_n \cdot f_1$ cannot address N_0 since it is known that N_n receives no diagnostic vote. 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 it if 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 (Fig. 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 only if the above path appears incorrect.

Multiple Candidates: If constructive votes agree on a common candidate, but support a different candidate from that addressed by $N_0 \cdot b_2 \cdot f_1$, then the target is connected. Otherwise, since constructive votes disagree, any candidate 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 N_1. Only the diagnostic vote D_1 can use one of these erroneous pointers to support N_n. However, this implies that N_n is N_2, and that $N_2 \cdot f_1$ addresses N_0.

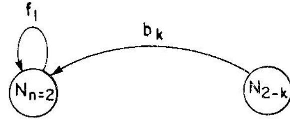


Fig. 6. $N_n \cdot f_1$ being used by C_k to support N_n.

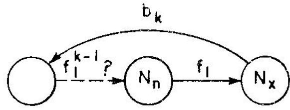


Fig. 7. The path $N_n \cdot f_1 \cdot b_k \cdot f_1^{k-1}$

The statement of the theorem 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 4: If the conditions of Theorem 3 are satisfied, and it has been determined that the target is connected as described in Theorem 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 Theorem 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 Theorem 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 $N_{2-k} \cdot b_k$. □

IV. Conclusions

The above results are the natural progression of ideas first presented in [4]. This earlier work presented an algorithm that corrected mod(k) linked lists [1], [2], [6] by using weighted votes. Mod(k) lists can be derived from helix(k) lists by replacing all the back pointers in a helix(k) list by a single pointer addressing the kth previous node. The mod(k) algorithm is somewhat simpler to implement than the algorithm presented in Appendix A of this paper, but the use of weighted votes resulted in a proof of correctness that was more complex than desired.

Empirical results presented in Appendix B suggest that the algorithm presented in this paper when applied to helix(k) structures is significantly better than earlier algorithms used to correct the similar spiral(k) structure [3]. This is hardly surprising despite the similarities between these two classes of structure. Earlier algorithms operated under the assumption that at most $k - 1$ errors occurred in any locality of the spiral (k) structure, and therefore made no attempt to either detect disconnection or to behave intelligently when k errors occurred in a locality.

Currently, it is unclear how one might evaluate a correction algorithm, or identify the type of behavior that could reasonably be expected from a "good" algorithm. While it seems reasonable to judge an algorithm on its empirical behavior, there seems no way of confidently simulating the types of unknown error that are likely to be encountered in any real environment. If anything, the theoretical behavior of a correction algorithm is of even less use in predicting the practical usefulness of an algorithm. However, theoretical results help identify the types of errors that will be corrected by the algorithm, and may eventually be used to accurately predict the statistical behavior of correction algorithms. This is an open and interesting area for research.

Appendix A

Pseudocode for Correction Algorithm

```
correct_headers();                                   /* Terminate if null instance */
for (count=0; count < max_possible; count=count+1){
    candidates=0;
    for (i=0; i≤k; i=i+1){                           /* Apply constructive votes */
      case "i=0":      Nx=N0·b2·f1;                  /* For simplicity assume Nx exists */
      case "i=k":      Nx=N2-k·bk·f1;
      case "default:"  Nx=N-i·bi+1;
        if (Nx≠any candidate [j]){
          j=candidates; candidate[j]=Nx; vote[j]=0;candidates=candidates+1;
      }
      if (i≠0) vote[j]=vote[j]+1;                    /* Nx=candidate[j] */
    }
```

```
        for (i=0; i<candidates; i=i+1){                    /* Apply diagnostic votes */
          N_x=candidate[i];
          for (j=1; j<=k; j=j+1){
              case "j=1:"        if (N_x·f_1=N_0)                  vote[i]=vote[i]+1;
              case"j=k:"         if (N_x·b_k=N_0·b_2·b_(k-1))      vote[i]=vote[i]+1;
              case "default:"    if (N_x·b_i=N_0·b_(i+1))          vote[i]=vote[i]+1;
          }
          if (N_x=N_j, for any 0>=j>=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 trusted or N_a·id bad or N_a=N_(2-k)·b_k or N_a·f_1=N_a
              or N_a·f_1 b_k·f_1^(k-1)=N_a without cycles) abort(Target disconnected);
          }
          else if (N_a·f_1=N_0 and N_a=N_(1-k)·b_k and N_a=N_(2-k)·b_(k-1)){
              if (k=3)      abort(Target may be disconnected);
              if (N_a=N_(-1)·b_3) abort(Target disconnected);
          }
        case "N_a and N_b have k votes":
          if (N_a·f_1≠N_b·f_1){
              if (N_b·f_1=N_0)       N_a=N_b;
          } else if (N_a=N_(2-k)·b_k) N_a=N_b;
      case "Otherwise": abort(Target disconnected);

      N_a·id=id; N_a·f_1=N_0;                              /* N_a is the target node N_1 */
        for (i=2; i<=k; i=i+1)N_(1-i)·b_i=N_a;             /* Assignments may be unnecessary */
        if (N_a=last header) correct_count();              /* Terminate successfully */
    }
    abort(Algorithm looping);
```

Appendix B

Empirical Results

A. 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. In the spiral(k) storage structure, each node has $k - 1$ pointers that address the next $k - 1$ nodes, and a kth pointer that addresses the kth previous node. The spiral(k) structure is identical to the helix(k) structure in all other respects [3].

In general, k errors in a spiral(k) structure may cause some nodes to be reachable only via forward pointers. Unfortunately, in a spiral(3) structure a different set of $k = 3$ errors may cause these nodes to be reachable only via back pointers. Thus, any local correction algorithm that anticipates three errors in a spiral(3) locality may be unable to perform correction, even though the structure remains connected. By using a helix($k \geq 3$) structure this problem is avoided.

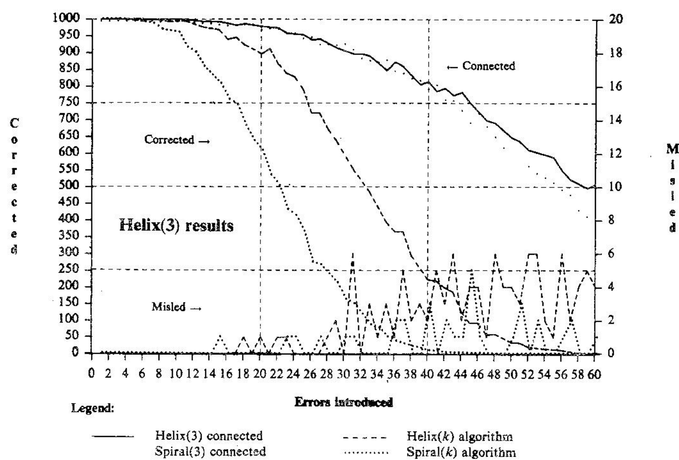
Each instance contained 100 consecutively located nodes plus headers. Increasing number of pointers were randomly selected from within this instance, and modified by adding or subtracting a random number between 1 and 10. Because the instances being considered were small, the probability that errors caused disconnection was high. Because pointers were modified by a small amount, the probability that votes supported common incorrect candidates was high. This Appendix therefore presents pessimistic estimates of the expected behavior of the correction algorithms described in this paper.

The spiral(3) and spiral(4) instances were corrected using the spiral correction algorithm described in [3]. This algorithm uses 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 concludes that this node is the target. Otherwise, the algorithm reports failure.

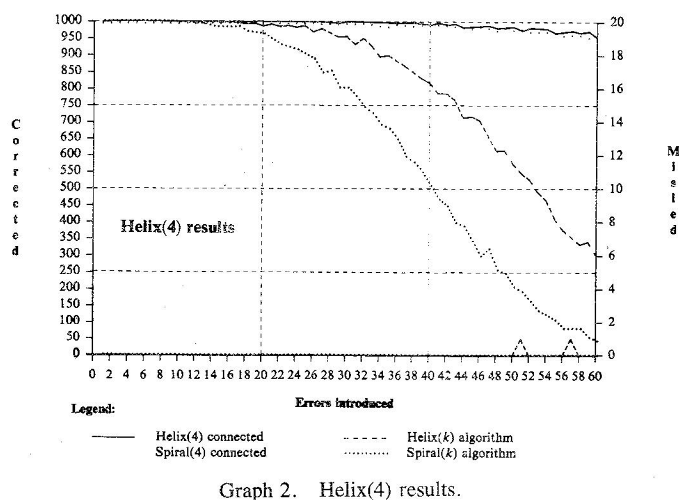
Vote	Path followed	Compared to node
C_i, $1 \leq i < k$	$N_{1+i-k} \cdot b_k \cdot f_i$	
C_k	$N_{1-k} \cdot b_k$	
D_i, $1 \leq i < k$	$N_n \cdot f_i$	N_{1-i}

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 was misled into attempting to apply an incorrect change.



Graph 1. Helix(3) results.



Graph 2. Helix(4) results.

B. Comments

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

The helix(3) structure was corrected 54 percent of the time while the spiral(3) structure was corrected only 36 percent of the time. Similarly, the helix(4) structure was corrected 83 percent of the time, but the spiral(4) structure only 66 percent 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 ten 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 two 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 can be misled when an incorrect candidate receives k votes, while the spiral correction algorithm can be misled only if some incorrect candidate received at least $k + 1$ votes. This becomes particularly significant when constructive votes support nodes outside of the instance being corrected. Given the nature of the diagnostic votes used, and the fact that only components within the instance are damaged, such nodes receive no diagnostic votes from the spiral correction algorithm, but can 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 to those presented earlier for the mod(k) correction algorithm [4]. In order to provide a direct comparison, 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 percent of the time, while the mod(3) structure remained connected only 55 percent of the time. The helix(3) instances were corrected 90 percent of the time, the spiral(3) instances 70 percent of the time, and the mod(3) instances 40 percent of the time. Earlier mod(k) correction algorithms that did not use the techniques presented in this paper consistently corrected 26 percent of such damaged mod(k) instances.

ACKNOWLEDGMENT

The author is greatly indebted to his wife, A. R. Crowe, for the moral, emotional, and financial support that she has provided through this research. She is responsible for motivating me to complete this paper, and assisted in reading early drafts of it. Thanks are also extended to my supervisor, Dr. D. J. Taylor, for the interest he showed in this research, and for the numerous enhancements that he made to this paper. Finally, I would like to thank those individuals who refereed an earlier version of this paper. Their comments have contributed greatly to the final format of this paper.

REFERENCES

[1] J. P. Black, D. J. Taylor, and D. E. Morgan, "An introduction to robust data structures," in *Dig. Papers: 10th Annu. Int. Symp. Fault-Tolerant Comput.*, Oct. 1–3, 1980, pp. 110–112.

[2] ——, "A compendium of robust data structures," in *Dig. Papers: 11th Annu. Int. Symp. Fault-Tolerant Comput.*, June 24–26, 1981, pp. 129–131.

[3] J. P. Black and D. J. Taylor, "Local correctability in robust storage structures," CS-84-44, Dep. Comput. Sci., Univ. of Waterloo, Dec. 1984. *IEEE Trans. Software Eng.*, submitted for publication.

[4] I. J. Davis and D. J. Taylor, "Local correction of mod (k) lists," CS-85-55, Dep. Comput. Sci., Univ. Waterloo, Dec. 1985.

[5] I. J. Davis, "A locally correctable AVL tree," in *Dig. Papers: 17th Int. Symp. Fault-Tolerant Comput.,* July 6–8, 1987, pp. 85–88.

[6] S. C. Seth and R. Muralidhar, "Analysis and design of robust data structures," in *Dig. Papers: 15th Annu. Int. Symp Fault-Tolerant Comput.,* June 19–21, 1985, pp. 14–19.

[7] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Some theoretical results," *IEEE Trans. Software Eng.,* vol. SE-6, pp. 595–602, Nov. 1980.

[8] D. J. Taylor and J. P. Black, "A locally correctable B-tree implementation," *Computer J.,* vol. 29, pp. 269–276, June 1986.

[9] D. J. Taylor and C. J. Seger, "Robust storage structures for crash recovery," *IEEE Trans. Comput.,* vol. C-35, pp. 288–295, Apr. 1986.

Ian J. Davis received the B.Sc. degree in mathematics and computer science from the University of London and the M.Sc. degree in computer science from the University of Toronto., Toronto, Ont., Canada.

He has recently completed his Ph.D. in the field or robust data structures at the University of Waterloo, Waterloo, Ont., Canada. He is a full time employee of Wilfrid Laurier University, and is responsible for the development of a general database system called MSQ which is used in many different applications.