UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Local Correction of Helix(k) Lists

I. J. Davis

Data Structuring Group
CS-86-30

August, 1986

Local correction of helix(k) lists

I. J. Davis

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

A helix(k) list is a newly defined robust multiply-linked list having k pointers in each node. In general the *i'th* pointer in each node addresses the *i'th* 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 the instance being corrected has a disconnected node. 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.

1. Introduction:

A helix(k) storage structure is a circular multiply-linked list of nodes, in which each node contains k pointers. In general the *i'th* pointer in each node links that node to the *i'th* previous node. However, the first pointer in each node addresses the next node, rather than the previous. Each node contains an identifier field that identifies it as belonging to a specific instance of a helix(k) structure. A count of the number of nodes in the instance is also present. A helix(k) structure has k consecutive headers that allow access to the instance.

The helix(k) storage structure is a variant of the spiral(k) storage structure [3]. In the spiral(k) storage structure each node has $k-1$ pointers that address the next $k-1$ nodes, and a *k'th* pointer that addresses the *k'th* previous node. In general, k errors in a spiral(k) structure may cause some nodes to be reachable only via forward pointers, suggesting that any local correction algorithm must necessarily traverse the instance forwards. 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 all nodes in the structure are connected. By using a helix($k \geq 3$) structure this problem is avoided.

The local correction algorithm described in this paper for a helix($k \geq 3$) structure starts at headers whose addresses are assumed to be correct, and proceeds backwards through the entire instance iteratively identifying the previous node. At each step, paths expected to lead to this previous node are used to vote on the correct value of this previous node. The nodes addressed by these paths receive additional votes when paths proceeding from them appear correct. These votes, and occasionally additional information obtained from the structure being corrected, allow the previous node to be identified whenever possible.

2. Definitions

Each node in a helix(k) structure contains identifier, forward pointer, and $k-1$ back pointer *components*. A count component exists in one of the header nodes. An *error* is an incorrect value in one such component [6].

Initially it is assumed that the addresses of the header nodes within the *instance* being considered are known and can therefore be *trusted*. As correction proceeds, components of the instance become trusted. Any node addressed by a trusted component is trusted. However, trusted nodes may initially contain untrusted components.

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. Back pointers will be labelled b and forward pointers f with subscripts indicating the correct distance spanned by these pointers.

At any correction step, the node N_1 that should immediately precede the trusted nodes will be called the *target*. The target is *disconnected* if no correct pointer in the instance addresses it. Components examined in attempting either to identify the location of the target, or to detect that it is disconnected, define the current *locality*. Local correction requires that the number of untrusted components in any locality be bounded by a constant [3]. When the target has been identified, the identifier and forward pointer in the target, and the back pointers that should address this target, are components of trusted nodes. Since the correct values of these components are known, the values of these components can be corrected if incorrect. Once correct, these components and the target become trusted.

One method of attempting to identify the target is to use *votes* [3,4,7]. 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. 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. 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 with node or path
$C_{1\leq i<k}$	$N_{-i}\cdot b_{i+1}$	
C_k	$N_{2-k}\cdot b_k\cdot f_1$	
D_1	$N_n\cdot f_1$	N_0
$D_{2\leq i<k}$	$N_n\cdot b_i$	$N_0\cdot b_{i+1}$
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. Since at most k errors are assumed to occur in any locality, this ensures that the target lies within the locality being corrected, unless the target is disconnected.

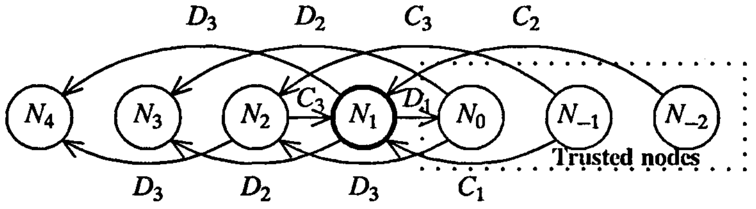


Fig 1. Pointers used in a correct helix(3) locality

3. Theoretical results

It is assumed throughout this section that at most k errors occur in any single locality, and that the Valid State Hypothesis [6] 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 number of errors occurring in a locality, and the number of errors seen when invalid components are examined, little can be said about the behaviour 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 this, any errors in the instance can be trivially corrected.

Proof

In a helix($k\geq 3$) instance at least $2k+1$ components contain values indicating when the instance is empty. Specifically the b_k pointer in each of the k header nodes address themselves, the b_{k-1} pointers in all but the first header point forward one node, the forward pointer in the first header addresses the last, and the count is zero. Given at most k errors, the instance is therefore empty if and only if at least $k+1$ of these components indicate that the instance is empty. □

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 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 some votes must contain only errors present in other votes supporting N_n.

If this 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, this pointer addresses N_0. Since C_k also uses this pointer it supports N_0. But N_0 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-2} and D_{k-1}, since no other vote uses a b_{k-1} pointer. However, 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 share a common b_{k-1} pointer, 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 share a common b_{k-1} pointer, 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 share a common b_{k-1} pointer, 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. □

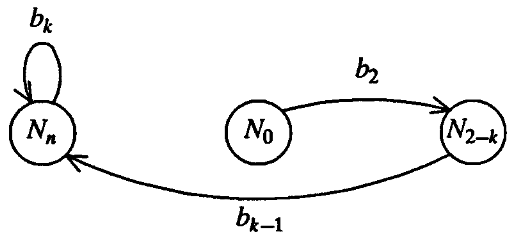


Fig 2. Configuration if C_{k-2} and D_k share b_{k-1}

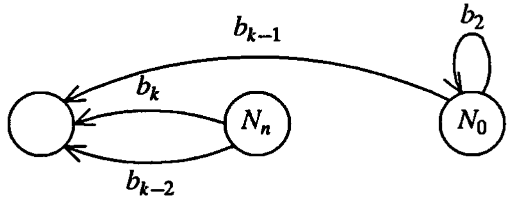


Fig 3. Configuration if D_{k-2} and D_k share b_{k-1}

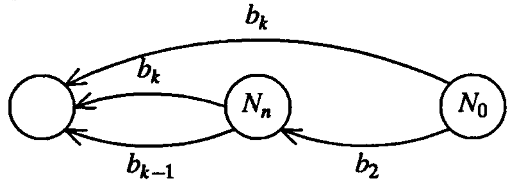


Fig 4. Configuration if D_{k-1} and D_k share b_{k-1}

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 alway 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, then the path $N_0 \cdot b_2 \cdot f_1$ and all paths used by constructive votes address N_n. If N_n is not the target then each path contains an error. Since only k errors occur in the locality, the error in the path $N_0 \cdot b_2 \cdot f_1$ must be shared with C_k, implying that $N_{2-k} \cdot b_k$ is correct. Conversely, if N_n is the target, then only diagnostic votes contain errors, again implying that

$N_{2-k}\cdot b_k$ is correct. Since $N_{2-k}\cdot b_k$ addresses N_2, it can easily be 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 N_n receives k votes, C_k contains exactly this one error and thus $N_n=N_2$, contradiction. Thus $N_n\cdot f_1$ must be correct. Conversely, if N_n is the target N_1, then since all of the diagnostic votes associated with N_{n-1} are damaged $N_n\cdot f_1$ must contain 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 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 all of the above nodes. Then $N_x\cdot b_k$ is correct since it is distinct from all of the k pointers containing errors.

Consider following the path $N_n\cdot f_1\cdot b_k$, and then $k-1$ forward pointers. If $N_n\cdot f_1$ is correct then none of these $k-1$ forward pointers can be $N_2\cdot f_1$, since N_n is not one of the last k trusted nodes, and 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 arrive back at N_n prematurely. Thus N_n is the target if and 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, since all errors occur in pointers correctly addressing N_1, only the diagnostic vote D_1 can support N_n. However, this implies that N_n is N_2, and that $N_2\cdot f_1$ addresses N_0.

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($k\geq4$) 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 utilise 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 $N_n . f_1$ address N_0 then the target is that node not addressed by $N_{2-k} \cdot b_k$. □

4. 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 [1,2,5] 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 k'*th* previous node. The mod(k) algorithm is somewhat simpler to implement than the algorithm presented in this paper, but the use of weighted votes resulted in a proof of correctness that was more complex than desired.

Empirical results presented in the appendix suggest that the algorithm presented in this paper when applied to helix(k) structures is significantly better than earlier algorithms used to correct spiral(k) structures. 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 similar 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 behaviour that could be reasonably be expected from a "good" algorithm. While it seems reasonable to judge an algorithm on its empirical behaviour 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 behaviour 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 error that will be corrected by the algorithm, and may eventually be used to accurately predict the statistical behaviour of correction algorithms. This is an open and interesting area for research.

APPENDIX

Empirical results

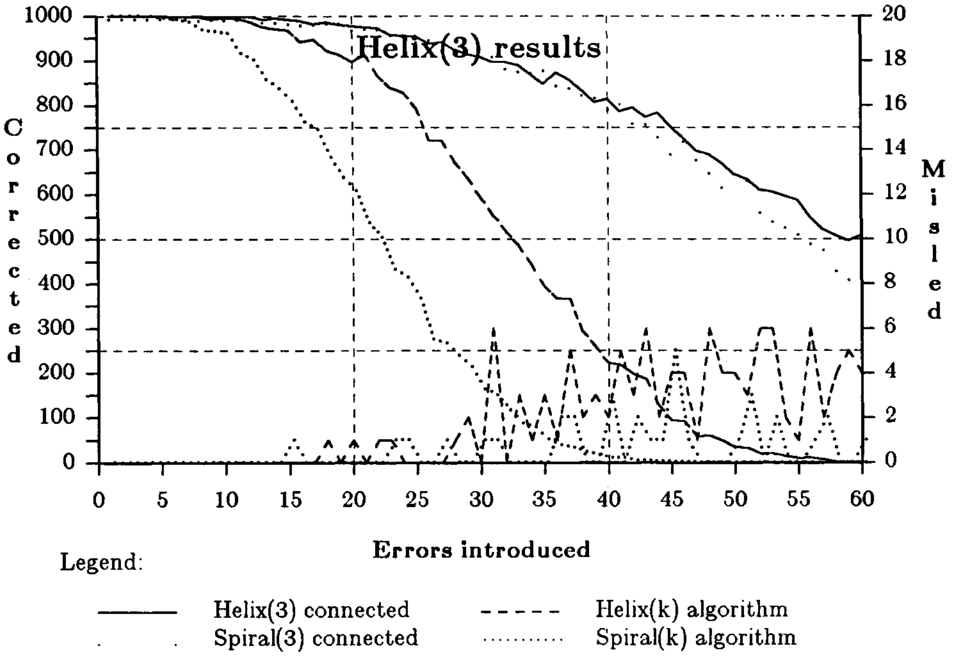
1. Explanation

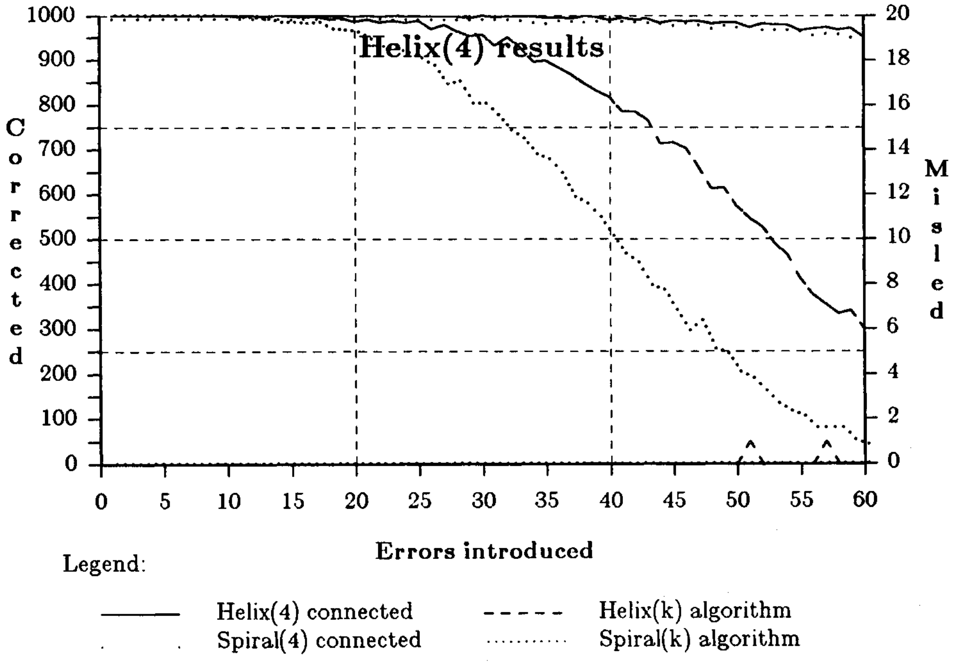
This appendix presents empirical results obtained when "random" errors were introduced into instances of a helix(3), helix(4), spiral(3) and spiral(4) structure. Each instance contained 100 consecutively located nodes plus headers. Increasing numbers of pointers were randomly selected from within this instance, and modified by adding or subtracting a random number between 1 and 10.

The spiral(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 with node
$C_{1 \leq i < k}$	$N_{1+i-k} \cdot b_k \cdot f_i$	
C_k	$N_{1-k} \cdot b_k$	
$D_{1 \leq i < k}$	$N_n \cdot f_i$	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.


Helix(3) results
Corrected
1000
900
800
700
600
500
400
300
200
100
0
Misled
20
18
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25
30
35
40
45
50
55
60
Errors introduced
Legend:
Helix(3) connected
Spiral(3) connected
Helix(k) algorithm
Spiral(k) algorithm


Helix(4) results
Corrected
1000
900
800
700
600
500
400
300
200
100
0
Misled
20
18
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25
30
35
40
45
50
55
60
Errors introduced
Legend:
Helix(4) connected
Spiral(4) connected
Helix(k) algorithm
Spiral(k) algorithm

2. 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 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 it is of some interest to compare the results presented above with 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% of the time, while the mod(3) structure remained connected only 55% of the time. The helix(3) instances were corrected 90% of the time, the spiral(3) instances 70% of the time, and the mod(3) instances 40% of the time. Earlier mod(k) correction algorithms that did not use the techniques presented in this paper consistently corrected 26% of such damaged mod(k) instances.

3. Acknowledgements

The author is greatly indebted to his wife, A. R. Crowe, for the moral, emotional, and financial support that she has provided throughout 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 C. J. Seger for his friendship, and for many discussions relating to robust data structures.

4. References

1. 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 (1-3 October 1980).
2. 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 (24-26 June 1981).
3. 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). Submitted to *IEEE Transactions on Software Engineering*
4. I. J. Davis and D. J. Taylor, Local correction of mod(k) lists, Research report CS-85-55, Waterloo (December, 1985).
5. S. C. Seth and R. Muralidhar, Analysis and design of robust data structures, *Proc. 15th Int. Symp. on Fault Tolerant Computing*, pp. 14-19 (19-21 June 1985).
6. 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).
7. D. J. Taylor and J. P. Black, A locally correctable B-tree implementation, *Computer Journal* **29**(3) pp. 269-276 (June 1986).