Multiple Generation Text Files using Overlapping Tree Structures*

F. WARREN BURTON†
Department of Electrical Engineering and Computer Science, University of Colorado at Denver, Denver, Colorado 80202, *USA*

MATTHEW M. HUNTBACH
Cognitive Studies, University of Sussex, Brighton, U.K.

J. (YIANNIS) G. KOLLIAS
Department of Computer Science, National Technical University of Athens, 9 Heroon Polytechniou Avenue, Zografou Athens (624), *Greece*

When repeatedly editing a text file, one is often faced with a choice of keeping previous generations for backup or deleting previous generations to reduce storage requirements. Since one generation of a text file is often very similar to the previous generation, the above conflict can often be resolved by sharing much of the common information.

We propose using a tree structure to represent a text file. Common subtrees can be shared. Results of an experiment with one file are reported.

INTRODUCTION

In many applications it is necessary to keep a large number of files all of which contain similar information. We might consider for example a collection of files which are all variants of some standard file. In the case of a file which is changing slowly with time it will often be necessary to keep past generations so as to be able to recover from errors. Again one generation of a file is likely to be similar in content to previous generations.

Rather than store repeated copies of the same information it would be possible to store files with shared substructures for those parts which they have in common. The sharing of common substructures has been advocated by Hoare[5] and is usually supported automatically in functional programming languages.[4] Schemes have been presented for modifying a file, leaving the original intact by sharing data.[3,7] This paper reports on some results obtained in practice using such a scheme.

THE METHOD

We use the method of overlapping tree structures described in Ref. 3. Files are represented by a tree structure,[1,2,6] with information common to several files represented by a shared subtree. Each node has a reference count[6] which contains the number of pointers currently referencing it. All nodes with a reference count greater than 1, together with their descendants, contain shared information.

When a file is modified, a duplicate is produced by creating a new pointer to the tree representing the file (and therefore increasing the reference count of the root node by 1). During a modification it is necessary to copy nodes encountered with a reference count greater than 1 since the new copy will be on the path to the modification while the original should remain unchanged. When a node is copied the children which become shared have their reference counts increased. Provided parents are copied before children are considered, only nodes with reference counts greater than 1 need be copied, as other nodes will not be shared and thus may be freely modified.

To see how an insert works with overlapping tree structures, consider the tree shown in Fig. 1 (we are using overlapping binary trees here). Suppose we wish to insert a node containing 'N' in alphabetical order. The root node 'E' is unshared and so does not need to be copied. Node 'G' is also unshared and so may be modified without being copied. Node 'M' though has a reference count of 2 and so is shared and must be copied. Since both the original 'M' and its copy will contain pointers to nodes 'H' and 'P', these nodes will have reference counts of 2 after node 'M' has been copied. Hence node 'P' will be copied when it is encountered. Finally node 'N' will be inserted under the new copy of node 'P'. Fig. 2 shows the structure after the insertion of node 'N' has been completed.

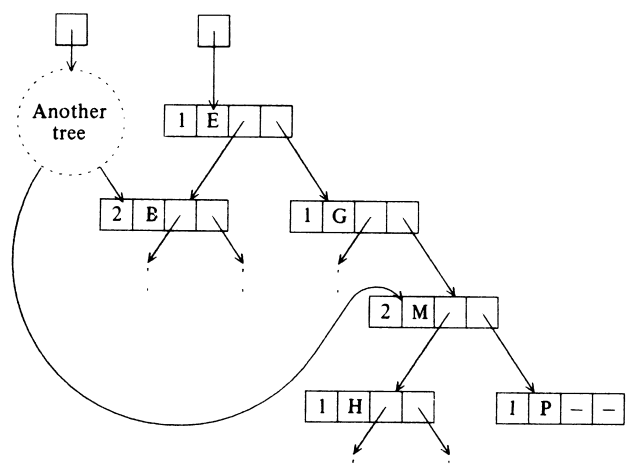

Figure 1. A tree with shared subtrees.

* This work was supported in part by a grant from the United Kingdom Science and Engineering Research Council. In addition, in part this material is based on work supported by the National Science Foundation under Grant No. ECS-8312748.

† Some of this work was done while the first two authors were at the University of East Anglia. Correspondence should be addressed to the first author.

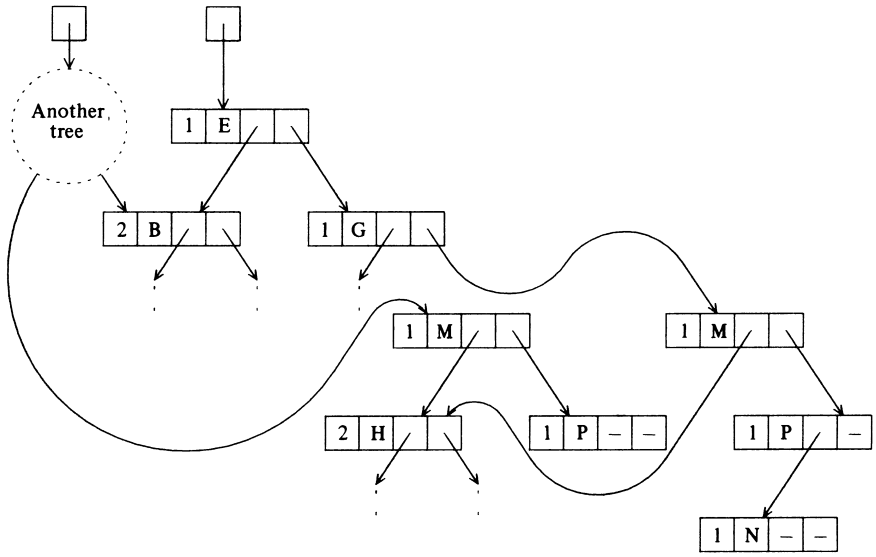

Figure 2. The tree of Fig. 1 after an insertion.

IMPLEMENTATION

In our implementation, files are stored using overlapping B-trees of order 3 (2–3 trees). Each leaf node will store 1 or 2 records, with a record being equivalent to a line of text. Larger leaf nodes could have been used. However, at least one new leaf node must be produced for each update, so larger leaf nodes will tend to reduce the amount of information which can be shared. At any node we keep a count of the number of records in each subtree. This enables lines to be accessed using their position relative to the start of the file as a key. These counts are updated as the file is modified. Files may be accessed sequentially by performing a tree traversal.

We have incorporated this into a simple editing system which allows new files to be created and lines to be added or deleted.

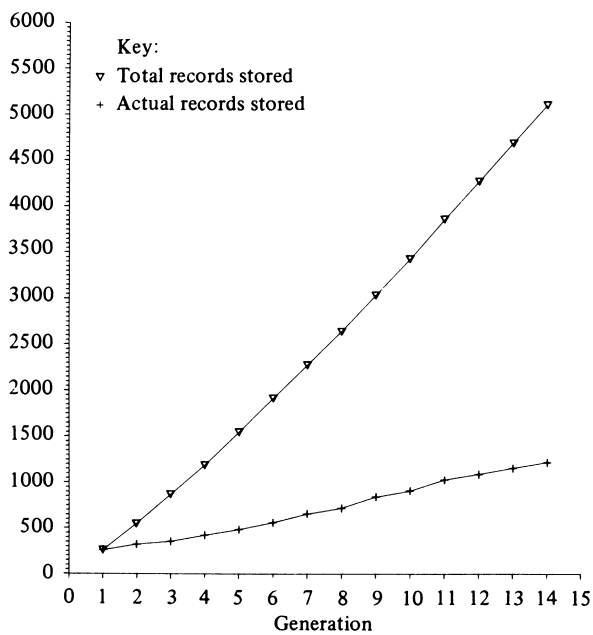

Figure 3. Growth in storage with each new generation.

EXPERIMENTAL RESULTS

The overlapping tree structure editor was used in the preparation of a paper which went through a total of 14 generations before the final version was ready. On average for each generation, about 4 lines out of 5 could be found in the previous generation without change. Each generation was saved, resulting in a final data structure with 14 overlapping trees. After producing a new generation of the paper a count was made of the actual number of records stored (that is counting those shared only once). This was compared with the number of records which would be necessary were each generation to be stored separately. Fig. 3 shows the changes in these two figures as each new generation was added.

It will be noted that adding each new generation requires an average of about one-fifth of the space that would be required were it to be stored without shared substructures. If this system were to be used on a larger scale it should be noted that a B-tree of a higher order may be used to reduce the number of node accesses. This would tend to increase the amount of data which needs to be copied. However, with larger files the proportion of data which is common between generations is also likely to be greater. Analysis of the amount of space saved by using shared substructures is given in Refs. 3 and 7.

ERROR RECOVERY

When a file is updated, existing nodes are left unchanged except for their reference counts. Should a system failure occur during an update then the reference counts may not be correct. There are two solutions to this problem.

Reference counts can be recomputed by traversing all trees after a system failure. Alternatively, if reference counts are always increased before a new pointer is set and decreased after a pointer has been changed, then any error in a reference count will make it overly large. If old generations are sometimes deleted, then some form of garbage collection will be required to remove leaves which are no longer being used but have overly large (greater than zero) reference counts.

Periodic reorganisation may be desirable. This could order the nodes in the most recent generation of a file sequentially, to speed access, and collect unreferenced nodes which have been left behind due to system failures as described above.

CONCLUSION

We have used overlapping tree structures in a practical text-editing situation. We have shown that in this situation a considerable saving of space can be made by storing past generations of a file using this method. Our results indicate that overlapping tree structures may be used with benefit in applications where it is necessary to store large numbers of similar files. Details such as optimal leaf size will depend on the machine being used and the anticipated pattern of use, as well as the relative importance of economy of storage vs speed of access.

REFERENCES

1. R. Bayer and E. McGreight, Organization and maintenance of large ordered indexes. *Acta Information* **1** (3) 173–189 (1972).
2. R. Bayer and K. Unteraner, Prefix B-trees. *ACM Trans. Database Syst.* **6** (1) 174–193 (March 1981).
3. F. W. Burton and J. G. Kollias, Representing multiple generations files using overlapping tree structures. Technical Report. University of East Anglia, Computer Studies.
4. P. Henderson, *Functional Programming. Application and Implementation.* Prentice-Hall, Englewood Cliffs, New Jersey (1980).
5. C. A. R. Hoare, Recursive data structures. *J. of Computer and Inf. Sciences* **4** (2) 105–132 (1975).
6. D. E. Knuth, *The Art of Computer Programming, Fundamental Algorithms*, Vol. 1, Addison-Wesley, Reading, Mass. (1973).
7. J. K. Mullin, Change area B-trees: a technique to aid error recovery. *Computer Journal.* **24** (4) 367–373 (1981).