IMPLEMENTATION OF THE BINARY CODING SCHEME AND THE TREE TRAVERSAL ALGORITHMS TO TEST FOR ANCESTOR-DESCENDANT RELATIONSHIPS IN K-ARY TREES

Pervis Fly[1], Natarajan Meghanathan[1*] & Raphael Isokpehi[2]

[1]Department of Computer Science, [2]Department of Biology
Jackson State University, Jackson, MS 39217, USA
*Corresponding Author E-mail: natarajan.meghanathan@jsums.edu

ABSTRACT

This paper discusses the implementation of the binary coding scheme and its comparison with the post-order, pre-order and in-order traversal techniques to test for ancestor-descendant relationships in *k*-ary trees (a tree in which any leaf node has up to *k* children). The approach used is assigning a unique binary code to each node in a tree. The value of the binary code for a node is the concatenation of the code for its parent node (referred to as prefix) and the unique binary representation of the immediate children of the parent node (referred to as suffix). For any two nodes *I* and *J*, if the binary code of node *I* forms the prefix of the code of node *J*, then node *I* is the ancestor for node *J*; otherwise not. This method to test ancestor-descendant relationships has a time complexity of O(1), whereas the tree-traversals incur a complexity of O(*n*).

Keywords: *Binary Coding, Ancestor-Descendant Relationships, K-ary Trees, Tree Traversals*

1. INTRODUCTION

The ancestor-descendant relationship of any two nodes *I* and *J* in a tree or graph can be determined by performing a traversal on a sub-graph with *I* as the root. If *J* is found during this traversal, *J* is considered to be an ancestor of *I* and an ancestor-descendant relationship exists. Performing this traversal will result in a time-complexity of O(*n*) where *n* is the number of nodes in the sub graph involving nodes *I* and *J*. The algorithm *isAncestor*, proposed in an earlier work [1] of one of the authors, is used to efficiently determine an ancestor-descendant relationship using binary codes as identifiers and it results in a time-complexity of O(1). This is because the relationship is quickly determined by comparing the binary code of the node with the lower ID with the node with that of the higher ID.

The term *k*-ary tree [2] is used to define a tree in which any given node has a maximum of *k* children. The binary codes are assigned to each node in the *k*-ary tree by initially assigning the root node a code of 0. Each child of the root node will have a prefix of 0 and a suffix that is determined based on the number of children the root node has. This suffix has a number of bits of $\lceil \log_2 C \rceil$, where *C* is the number of children. Thus, 3 to 4 children will result in each child having a suffix of 2 bits, 5 to 8 children having a suffix of 3 bits, etc. This method is extended to each child. The effectiveness of the binary coding approach is illustrated using an example shown in Figure 1. Here, a traditional approach to determine the ancestor-descendant relationship between any two nodes *I* and *J* on the tree would require making a sub tree with the lower ID (say *I*) as the root and traversing through that whole sub tree, using one of the well-known tree traversal algorithms (pre, post or in-order traversals) [3] that would incur a time-complexity of O(*n*). However, by assigning each node in the tree a unique binary code, based on the above binary code assigning scheme, all it needs to be done is to compare the entire binary code of the node with the lower ID with that of the leading bits of the binary code of the node with the larger ID. In Figure 1, to determine whether *I* and *J* are related through ancestor-descendant relationship, all there needs to be done is to compare node *I*'s binary code of *0000* to the first 4 bits of node *J*'s binary code, *001001*. Since *0000* and *0010* are not identical, node *I* is not an ancestor of node *J*. The binary coding scheme has been studied in several related work (e.g. [5][6]) in the literature.

The rest of the paper is organized as follows: Section 2 presents a brief literature review on related work available in the literature with respect to coding schemes. Section 3 discusses the implementation of the binary coding scheme and the modified versions of the three tree traversal schemes (pre-order, in-order and post-order traversals) for *k*-ary trees. Section 4 presents the simulation results obtained when the implementations were run with 5-ary trees and the number of nodes varied from 100 to 50,000. Section 5 presents the conclusions and discusses future research.

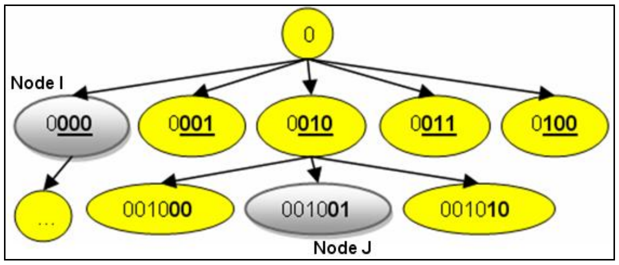


Figure 1. Example – Binary Code Assignment and Determination of Ancestor-Descendant Relationships

2. LITERATURE REVIEW

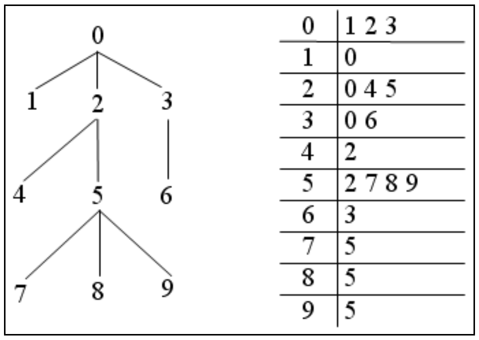
In [7], the authors propose a method to compress the *decimal* codes of trees nodes and use these codes for fast determination of ancestor-descendant relations. However, the worst-case time complexity and space complexity of the compressed decimal codes based method are $O(n+m^2)$ and $O(m)$ respectively, where n amd m represent the total number of nodes and the number of nodes without any grandchildren (referred to as kernel nodes). *Gray* codes are defined as the set of integers 0 ... 2^{n-1} where n is the length of binary string representing the codes and the adjacent integers in this set differ in only a single bit position. In [8], Xiang et. al., proposed an algorithm to generate Gray codes for k-ary trees ($k > 3$) with n internal nodes in 2^{n-1} different ways; however, the issue of determining ancestor-descendant relationships using the Gray codes between any two nodes in a tree has node been addressed. In [9], Gupta propose the idea of assgining (through a pre-order traversal) the left and right branches of the tree with 0s and 1s respectively. However, this method cannot be used to determine ancestor-descendant relationships.

Jun et. al [10] propose the use of Tunstall codes to form the prefix codes for other codes according to a variable-to-fixed length mapping scheme. In [11], the authors introduce the notion of divisibility and primality on k-ary trees and established a relation between the indecomposable prefix codes and prime trees. Neither the use of Tunstall codes nor the use of the indecomposable prefix codes for determining ancestor-descendant relationships has been yet studied. Prufer codes have been used to label free trees (connected a-cyclic undirected graphs) of n nodes using a sequence of length n-2. It takes O(log$n$) time to compute the Prufer codes of an n-node labeled free tree. Though Prufer codes have extensive use in parallel algorithms, their use for determining ancestor-descendant relationships in a tree has not been yet explored.

In [5], the authors explored the use of a binary coding scheme (similar to the BCAD approach implemented in this paper) to index XML data and quickly determine ancestor-descendant relationships in a heirarchy of XML data. In another related work [6], the authors propose a numbering scheme that embeds structural information within XML nodes, encoded as bits. This is more efficient and faster than the Dietz's numbering scheme [12] that utilizes pre- and post- order traversal of a hierarchy of XML data.

3. IMPLEMENTATIONS OF THE BCAD SCHEME AND TREE TRAVERSAL ALGORITHMS FOR K-ARY TREES

The implementation is performed in Java. A TreeMap (2-dimensional data structure) [4] is used to represent the tree; in the TreeMap – there are n keys, with each key representing a node in the tree. Each key (node) has a value which is the list of nodes adjacent to the key (node). For example, in Figure 2, node 2 is adjacent to parent 0 and children 4 and 5, thus the value for key 2 will contain the list {0, 4, 5}. This list will be referred to in the paper as an adjacency list.



0	1 2 3
1	0
2	0 4 5
3	0 6
4	2
5	2 7 8 9
6	3
7	5
8	5
9	5

Figure 2. A Sample 3-ary Tree and its Adjacency List

Starting with the root node, the *k*-ary tree is formed by adding a random number of children from 0 to *k* to the node. This continues in a left-to-right fashion, starting from the root node's leftmost child to the rightmost child. By setting up the tree this way, the leftmost child will always have a lower ID compared to its siblings. Each child has its ID placed in the TreeMap with the parent node as the first value in the adjacency list. For example, referring to Figure 2 again, since node 2 has a parent of 0, the list {0, 4, 5} will imply that the first item in the list, 0, is the parent. Any node that is not marked as the root node will have this implication, as this will be important for performing the traversals on the *k*-ary tree.

Once the tree is formed, first the binary code assignment is performed. The performance of the binary coding scheme-based ancestor descendant relationship determination process, hereafter shortly referred to as BCAD, is compared with the traditional tree traversal algorithms such as pre-order, post-order and in-order traversals. Because the algorithms available for such tree traversals are commonly implemented for binary trees, unique algorithms needed to be developed such that these tree traversals could be performed on a tree with *k* children. Since a TreeMap is used to represent the tree, simulating the traversals by utilizing the adjacency list for each node is necessary. In addition, a one-dimensional list, *visitedList*, is kept to keep track of all nodes visited.

3.1. BCAD Scheme

The BCAD scheme is based of the *isAncestor* algorithm [1] proposed in our earlier work. The algorithm as applied here is as shown through the pseudo code in Figure 2:

```
function isAncestor(int codeI, int codeJ){
    //Input: The complete binary codes of nodes I and J, represented as codeI and codeJ respectively
    //Output: Returns 1 if codeI forms the leading bits of codeJ, otherwise 0.
    int extractedCodeJ;
    Extract first b bits of codeJ, where b is the length of codeI.
    Assign extracted bits to extractedCodeJ.
    if (codeI = = extractedCodeJ)
        return 1;
    else
        return 0;
} //end isAncestor
```

Figure 3. Pseudo Code for Procedure isAncestor of the BCAD Scheme

The *isAncestor* algorithm simply extracts the first *b* bits of the binary code of node *J* where *b* is the number of bits in the complete binary code of node *I* and compares those extracted bits to binary code of node *I*. If the bits compared are identical, then an ancestor-descendant relationship exists; otherwise, not. Note that the *isAncestor* algorithm implicitly assumes that the first argument (binary code of node *I*) could be the binary code of the potential ancestor and the second argument (binary code of node *J*) could be the binary code of the potential descendant and then goes ahead with the comparison. Before calling the *isAncestor* algorithm, the user has to decide the order of the two arguments (i.e., the two nodes) representing the potential ancestor and potential descendant respectively, from left to right.

Given two nodes *I* and *J*, to decide which node would be the potential ancestor and potential descendant, we recommend comparing the length of the binary codes of the two nodes – the node with the larger code length cannot be an ancestor node of the other node (with a smaller code length). Looking back at the procedure described in Section 1 for assignment of binary codes, one can notice that the length of the binary codes of the nodes increases with the number of levels one goes away from the root of the tree.

3.2. Pre-Order Traversal for K-ary Trees

The traditional pre-order traversal functions on a binary tree by using the simple "root, left, right" approach while identifying nodes in the tree. Instead, the modified pre-order traversal for *k*-ary trees uses a "root, left, second left, … , right" approach. In Figure 3, the sequence according to which the vertices are identified illustrates the order in which the vertices are visited according to the pre-order traversal approach. The pseudo code of the modified pre-order traversal algorithm is illustrated in Figure 5.

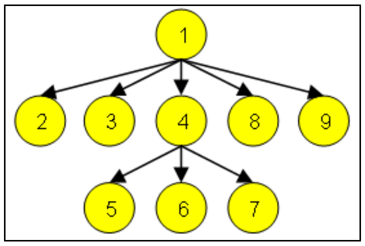


Figure 4. Sequence of Vertices Visited in a 5-ary Tree using Pre-order Traversal

```
1 function pre-Order(int i, int j){
2   //Input: Potential ancestor i and potential descendant j.
3   //Function: Perform pre-order traversal on a subgraph with root i. Return when j has been found or
              when all nodes have been discovered.
4   while (true)
5       boolean moreChildren = false;
6       if (i not in visitedList)
7           add i to visitedList;
8       if (j is in visitedList)
9           return;
10      if (i has children)
11          loop: for (children of i)
12              if (childNode not in visitedList)
13                  moreChildren = true;
14                  i = childNode;
15                  break loop;
16      if (!moreChildren)
17          if (i = = rootNode) return;
18          i = parentNode;
19} //end pre-Order
```

Figure 5. Pseudo Code for Modified Pre-Order Traversal Algorithm for k-ary Trees

The pre-order traversal initially starts by setting a Boolean value *moreChildren* to false (line 5) and adding the root node *i* to *visitedList* (line 7). The children of the root node are then checked from left to right (line 11) to determine if any of the children are in *visitedList*. The first child determined to not be in *visitedList*, which in this case would be the leftmost child, now has *i* pointing to it (line 14). The loop breaks, and since *moreChildren* is now true, the *if* statement on line 16 of Figure 5 is skipped and the while loop continues. Node *i*, which is currently pointing to the leftmost child of the root, is now added to *visitedList*. The children of *i*, if *i* has any children, are then checked just like *i* and its siblings were checked in the previous iteration. If *i* has no children, then *moreChildren* is false and line 16 is true, thus setting *i* as the parent node, or the root node. Since node *i* is pointing back to the root node, the children of the root are checked once again. The leftmost child is currently in *visitedList*, so the next leftmost child is then checked. That child would not be in *visitedList*, so *i* is now pointing to that child. The algorithm stops when node *j* has been found or if, when the flow returns to the root after visiting all the other nodes, all of the root node's children are in *visitedList*.

Unlike the BCAD scheme, with the pre-order traversal (and even in the other two tree traversal schemes), given two nodes *i* and *j*, represented by their node IDs, it would not be easy to determine the potential ancestor and potential descendant nodes. This is because we cannot always say that a node with the smaller ID will be always higher (i.e., closer to the root) than the node with the larger ID. Note that in Figure 4, node 8 is a level higher than node 5 as node 5 is visited earlier than node 8 in a pre-order traversal of the 5-ary tree illustrated in Figure 4.

3.3. In-order Traversal for K-ary Trees

The traditional in-order traversal functions on a binary tree by using the simple "left, root, right" approach while identifying the nodes in the tree. Instead, the modified in-order traversal for *k*-ary trees uses a "left, root, second left, …, right" approach. In Figure 6, the sequence according to which the vertices are identified illustrates the order in which the vertices are visited according to the in-order traversal approach. The pseudo code of the modified in-order traversal algorithm is illustrated in Figure 7.

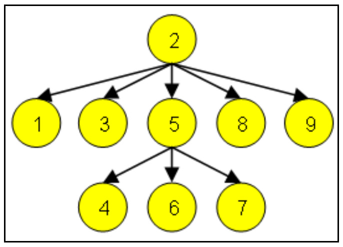


Figure 6. Sequence of Vertices Visited in a 5-ary Tree using In-order Traversal

```
-----------------------------------------------------------------------------------------
1 function in-Order(int i, int j){
2   //Input: Potential ancestor i and potential descendant j.
3   //Function: Perform in-order traversal on a sub graph with root i. Return when j has been found or when all
    nodes have been discovered.
4   while (true)
5       boolean checkOtherChildren = false;
6       boolean moreChildren = false;
7       if (j is in visitedList)
8           return;
9       if (i has children)
10          if (firstChild is in visitedList)
11              if (i is not in visitedList)
12                  add i to visitedList;
13              checkOtherChildren = true;
14          else
15              i  = firstChild;
16      else
17          add i to visitedList;
18          if (i = = root) return;
19          i = parent;
20      if (checkOtherChildren)
21          loop: for (children of i)
22              if (childNode not in visitedList)
23                  moreChildren = true;
24                  i = childNode;
25                  break loop;
26          if (!moreChildren)
27              if (i = = rootNode) return;
28              i = parentNode;
29} //end in-Order
-----------------------------------------------------------------------------------------
```

Figure 7. Pseudo Code for Modified In-Order Traversal Algorithm for k-ary Trees

The in-order traversal initially by checking the first child of the node *i* is currently pointing to (in this case, the root node). This differs from *pre-order* in that the node is not placed in *visitedList* first. The leftmost, or “first”, child of *i* is checked (line 10) to determine if it is in *visitedList*. If it is not (line 14), *i* is now pointing to the leftmost child; the Boolean *checkOtherChildren* is still false, so the while loop is continued. Node *i*, which is currently pointing to the leftmost child, is checked (line 9) to determine if it has a leftmost child that is not already in *visitedList*. If it does (line 10), then the flow will continue as it did in the last iteration. Otherwise (line 11), node *i* will be added to *visitedList* if it is not already there and *checkOtherChildren* will be set to true. (If node *i* has no children, then the flow will go to line 16, where *i* is added to the *visitedList* and the flow will return back to its parent.)
As with *pre-order*, the other children of node *i* are then checked from left to right (line 21) to determine if they are in *visitedList*; the first child not determined to be in *visitedList* will have *i* pointed to that child. This procedure

continues until, like *pre-order*, the control returns to the root, where it is determined to not have any other children that are in *visitedList*, or *j* is found to be in *visitedList*.

3.4. Post-order Traversal for K-ary Trees

The traditional post-order traversal functions on a binary tree by using the simple "left, root, right" approach while identifying the nodes in the tree. Instead, the modified post-order traversal for *k*-ary trees uses a "left, second left, … , right, root" approach. In Figure 8, the sequence according to which the vertices are identified illustrates the order in which the vertices are visited according to post-order traversal. The pseudo code of the modified post-order traversal algorithm is illustrated in Figure 9.

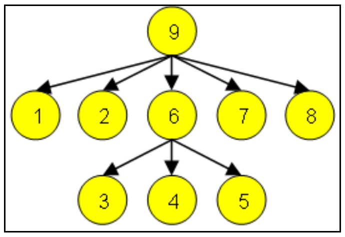


Figure 8. Sequence of Vertices Visited in a 5-ary Tree using Post-order Traversal

```
1 function post-Order(int i, int j){
2    //Input: Potential ancestor i and potential descendant j.
3    //Function: Perform post-order traversal on a subgraph with root i. Return when j has been found or when all
     nodes have been discovered.
4    while (true)
5        boolean moreChildren = false;
6        if (j is in visitedList)
7            return;
8        if (i has children)
9            loop: for (children of i)
10               if (childNode not in visitedList)
11                   moreChildren = true;
12                   i = childNode;
13                   break loop;
14       if (!moreChildren)
15           add i to visitedList;
16           if (i == rootNode) return;
17           i = parentNode;
18} //end post-Order
```

Figure 9. Pseudo Code for Modified Post-Order Traversal Algorithm for k-ary Trees

The *post-Order* traversal is a less complex form of *pre-Order*. Like *in-Order* traversal, the children of the node currently pointing to *i* are checked (line 9) to determine if one of the children is not currently in *visitedList*. If this is true, then *i* is pointed to that child. If all of the children are in *visitedList* (or if *i* never had any children in the first place), then *moreChildren*, flagged false on line 5, stays false; the 'if' condition in line 14 evaluates to true. This means that *i* is added to *visitedList* and *i* is pointed to its parent node. If node *i* is currently the root, then this means that all of the children have been checked and added.

4. SIMULATIONS

After a tree is formed, 1000 node pairs are randomly selected. For each node pair, the BCAD scheme, and the three tree traversal schemes are performed to determine whether an ancestor-descendant relationship exists. For all of these methods, the time taken to find the node pairs is recorded. For each approach, the time taken to determine an ancestor-descendant relationship per pair is the total time taken to determine the ancestor-descendant relationships between the node pairs divided by the number of node pairs found. For the BCAD scheme, the time taken to assign each node a code is combined with the time taken to determine an ancestor-relationship for the coding scheme.

To average out the results and reduce the influence of outliers, 100 5-ary trees are used. For these sets of 100 trees, there are different scenarios created based on the number of trees, which are 1000, 2000, 3000, … , 10000 nodes. To test if any trends continue in larger trees, scenarios of 10000, 20000, …, 50000 nodes are used as well.
Table 1 and Figures 10 and 11 illustrate the performance results. As expected from the theoretical analysis, the time taken to determine an ancestor-descendant relationship for the binary coding scheme is nearly instant compared to the tree traversals, whose time complexity increased linearly as the number of nodes in the tree increases. Even after considering the time taken to assign the binary codes, the run-time complexity of the BCAD scheme increases logarithmically compared to the linear time-complexity of the tree traversals.

Table 1. Performance Comparison of the BCAD Scheme with the Tree Traversals for 5-ary Trees

Number of nodes	Time (in milliseconds) for binary coding scheme and tree traversals				
	BCAD (without binary code assignment time)	BCAD (with binary code assignment time)	Pre-order	In-order	Post-order
1,000	0.02	1.03	2.33	2.23	1.72
2,000	0.04	1.82	8.42	8.23	5.81
3,000	0.06	2.19	17.00	17.29	12.33
4,000	0.09	6.49	27.85	27.10	21.13
5,000	0.11	13.13	31.52	29.20	21.84
6,000	0.14	15.15	55.64	49.16	41.81
7,000	0.16	12.86	62.95	55.61	38.58
8,000	0.21	26.48	69.94	57.80	42.34
9,000	0.24	34.69	79.60	76.06	57.65
10,000	0.22	27.38	91.70	90.20	65.19
20,000	0.46	32.17	214.24	207.39	149.86
30,000	0.85	79.81	346.48	356.48	246.20
40,000	1.50	81.53	351.44	290.52	226.07
50,000	1.60	87.65	650.42	617.89	461.53

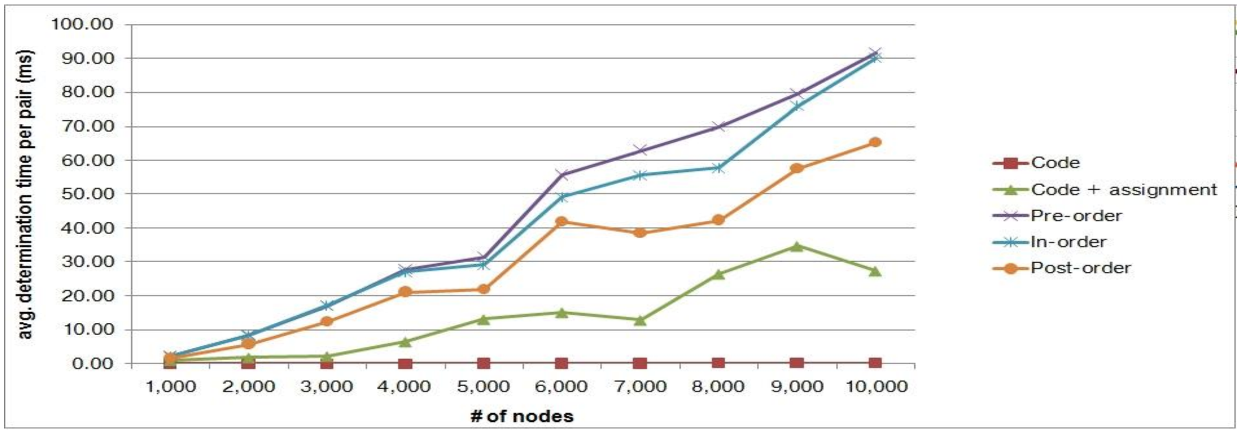


Figure 10. Performance Comparison of the Binary Coding Scheme with the Tree Traversal Schemes for 5-ary Trees with 100 to 10,000 Nodes

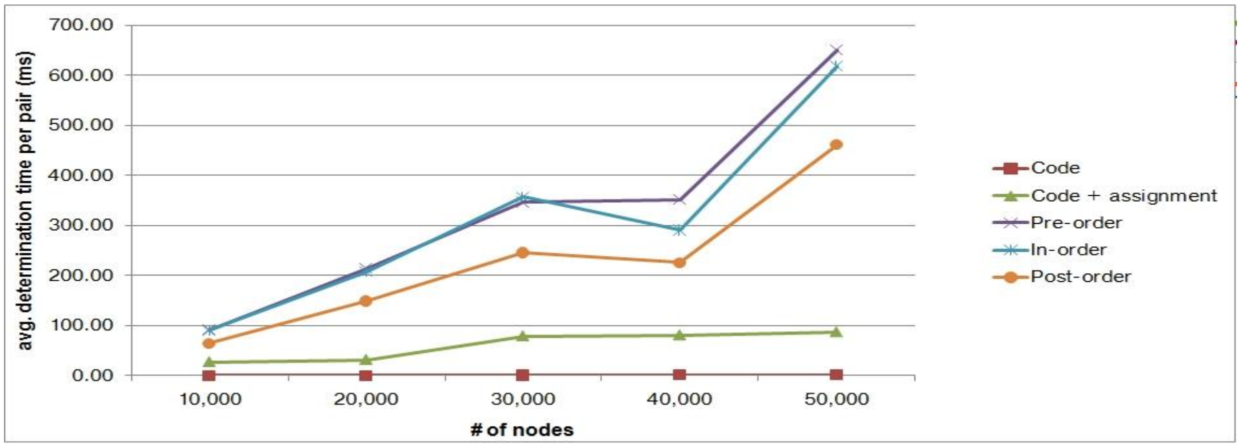


Figure 11. Performance Comparison of the Binary Coding Scheme with the Tree Traversal Schemes for 5-ary Trees with 10,000 to 50,000 Nodes

5. CONCLUSIONS AND FUTURE RESEARCH

We discussed in detail the implementation of the binary coding scheme and its comparison with the classical post-order, pre-order and in-order traversal techniques, modified to test for ancestor-descendant relationships in *k*-ary trees (a tree in which any leaf node has up to *k* children). Our approach is to assign a unique binary code to each node in a tree. Our implementations were tested on 5-ary trees with the number of nodes used being 500, 1000, 2000, 5000, 10000, 20000 and 50000. For each of these values, we formed ten 5-ary trees whose average height varied from 6 to 11. The total number of bits needed to assign unique binary codes for all the nodes in the tree was observed to vary somewhere between 12 to 23 times the number of nodes (i.e., 500 to 50,000) in the tree. The time taken to test for ancestor-descendant relationships between any two nodes (averaged over 1000 potential pairs of nodes) was observed to increase only from 3.6 ms to 82.4 ms, as we increased the number of nodes from 500 to 50,000. On the other hand, the time incurred with the tree-traversal algorithms increased from 12.5 ms to 59,000 ms; the three traversal techniques incurred almost the same time. These observations justify our hypothesis that the binary coding scheme can be used to test ancestor-descendant relationships in O(1) time; whereas, the tree-traversals incur O(*n*) time, where *n* is the number of nodes in the tree.

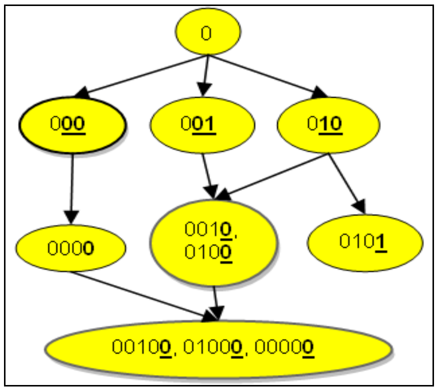


Figure 12. Binary Coded Directed Acyclic Graph (DAG)

Future research will extend the binary coding method to directed acyclic graphs (DAGs) [3]. As with *k*-ary trees, each node will receive a code based on its parent(s). The main issue is that the number of codes could potentially increase exponentially. For example, as Figure 12 shows, if a child receives 2 codes from one parent and one code from another, it would have 3 codes as its identifier. If the child of this node has a separate parent which also has 3 codes, that child would have 6 codes. Finding a more efficient method for code assignment will be necessary. In this pursuit, we plan to investigate the use of transitive reduction on the DAGs [3] to reduce the space complexity of the code assignment as well as the time complexity incurred for determining ancestor-descendant relationships.

6. ACKNOWLEDGEMENTS

This research acknowledges funding support from the National Science Foundation (NSF)—Mississippi EPSCoR Award (EPS-0903787); Research Centers in Minority Institutions (RCMI) — Center for Environmental Health at Jackson State University (NIH-NCRR2G12RR013459); Pittsburgh Supercomputing Center's National Resource for Biomedical Supercomputing (T36 GM008789), and the U.S. Department of Homeland Security Science & Technology Directorate (2007-ST-104-000007; 2009-ST-062-000014; 2009-ST-104-000021). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the funding agency.

7. REFERENCES

[1]. S. Baskiyar and N. Meghanathan, "Binary Codes for Fast Determination of Ancestor-Descendant Relationship in Trees and Directed A-cyclic Graphs," *International Journal of Computers and Applications*, vol. 10, no. 1, pp. 67-71, March 2003.

[2]. P. V. Ramanan and C. L. Liu, "Permutation Representation of K-ary Trees," *Theoretical Computer Science*, vol. 38, pp. 83-98, 1985.

[3]. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms," 3rd Edition, MIT Press, July 2009.

[4]. M. T. Goodrich and R. Tamassia, "Data Structures and Algorithms in Java," 5th Edition, Wiley, February 2010.

[5]. W. Zhang, D. Liu and J. Li, "An Encoding Scheme for Indexing XML Data," *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service*, pp. 525-528, March 2004.

[6]. S. Knight, H. Kim, N. Meghanathan and C. Bland, "Fast Determination of Ancestor-Descendant Relationships using Bit Patterns," *Proceedings of International Conference on Knowledge and Engineering*, pp. 178-182, June 2007.

[7]. J. Aoe, "Efficient Algorithm of Compressing Decimal Notations for Tree Structures," *Proceedings of the 13th International Computer Software and Applications Conference*, pp. 316-323, 1989.

[8]. L. Xiang, K. Ushijima, and T. Changjie, "Efficient Loopless Generation of Gray Codes for K-ary Trees," *Information Processing Letters*, vol. 76, no. 4-6, pp. 169-174, 2000.

[9]. D. K. Gupta, "Generation of Binary Trees from (0-1) Codes," *International Journal of Computer Mathematics*, vol. 42, no. 3-4, pp. 157-162, 1992.

[10]. Y. Jun, F. Fangwei and S. Shiyi, "On the Tunstall Codes in Source Coding," *Acta Mathematicae Applicatae Sinica*, vol. 23, no. 3, pp. 367-376, 2000.

[11]. C. M. Gabriella, D. Guaiana and S. Matanci, "Indecomposable Prefix Codes and Prime Trees," *Proceedings of the 3rd International Conference Developments in Language Theory*, pp. 135-145, 1997.

[12]. P. F. Dietz and D. Sleator, "Two Algorithms for Maintaining Order in a List," *Proceedings of the 19th Annual ACM Conference on Theory of Computing*, pp. 365-372, January 1987.