Finding All Breadth First Full Spanning Trees in a Directed Graph

Hoda Khalil
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
hodakhalil@cmail.carleton.ca

Yvan Labiche
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
labiche@sce.carleton.ca

***Abstract*— This paper proposes an algorithm that is particularly concerned with generating all possible distinct spanning trees that are based on breadth-first-search directed graph traversal. The generated trees span all edges and vertices of the original directed graph. The algorithm starts by generating an initial tree, and then generates the rest of the trees using elementary transformations. It runs in O(E+T) time where E is the number of edges and T is the number of generated trees. In the worst-case scenario, this is equivalent to O ($E + \frac{E^n}{N^n}$) time complexity where N is the number of nodes in the original graph. The algorithm requires O(T) space. However, possible modifications to improve the algorithm space complexity are suggested. Furthermore, experiments are conducted to evaluate the algorithm performance and the results are listed. (*Abstract*)**

Keywords—directed graph; trail; spanning tree; breadth first traversal.

I. INTRODUCTION

Directed graphs (digraphs) and their spanning trees have many applications in different fields such as representing street maps, flow networks with valves in the pipes, electrical networks, modeling computer programs where vertices are instructions and edges are execution sequences, theory of language, game theory, and social groups dynamics [3, 15]. Finding an efficient procedure for generating all spanning trees of a graph is a very important practical problem [15].

Breadth first spanning trees are a subset of all spanning trees that satisfy breadth-first-search (BFS). Many graph analysis algorithms are based on BFS graph traversal [17]. Other applications of BFS include finding neighboring locations in a Global Positioning System (GPS) system, Garbage collection, crawlers in search engines, finding the shortest paths; i.e. paths containing minimum number of edges, and small world networks properties analysis [17]. Nevertheless, algorithms that are specific to generating all breadth first traversal spanning trees are not available in the literature to the best of our knowledge.

In this paper, the concept of full spanning trees that cover all edges as well as all vertices is introduced. The paper proposes an algorithm that produces all distinct breadth first traversal trees of an input directed graph. The algorithm has time and space complexity of O(T) where T is the number of generated trees.

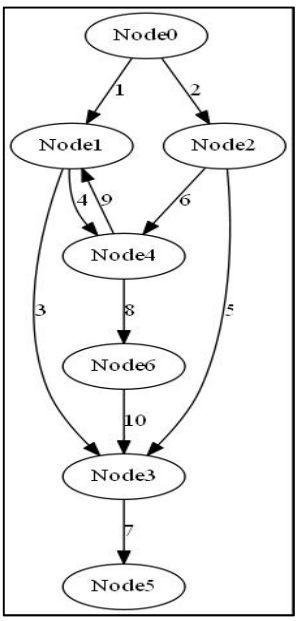

Fig. 1. Directed Graph G

The suggested algorithm is verified against many experimental directed graphs as well as directed graphs representing real life case studies. It has been used to generate breadth first test suites for the software systems representing the case studies. Section II of the document encapsulates the background, motivation, definitions needed to present the solution, while section III summarizes work that is similar to the provided solution. Section IV is the hypothesis on which the developed algorithm is based. V covers the algorithm steps and complexity. In section VI, a step by step example to contextualize the explanation of the algorithm steps to the reader is provided. Then, the experiments conducted to validate the algorithm and measure its effectiveness is provided in VII. Finally, the conclusion of the paper is in VIII.

II. BACKGROUND

A directed graph (digraph) is a graph where edges have directions. A directed graph G consists of a set of vertices $V=\{v_1, v_2,\ldots\}$, a set of edges $E = \{e_1, e_2,\ldots\}$, and a mapping Ψ that maps every edge onto an ordered pair of vertices = (v_1, v_2) [15]. Vertices are also referred to as nodes. A directed graph can be represented by an Incidence Matrix, which is anmatrix A

=[a_{ij}] whose rows corresponds to vertices and columns corresponds to edges such that a_{ij}=1 if jth edge is incident out of ith vertex, a_{ij} = -1 if jth edge is incident into ith vertex, and a_{ij}=0 otherwise [15]. Another way of representing directed graphs is by diagrams where vertex v_1 is an initial vertex if edge e_k is incident out of v_1, and is terminal if v_1 is a terminal vertex if edge e_k is incident into v_1 [15].

Before discussing the proposed algorithm, some terminology must be formally defined. We start by defining a walk as a sequence of edges in a graph G, where there are vertices v_1, …, v_n such that e_i= $v_{i-1}v_i$ for i=1,..,n. A walk for which v_i are pairwise distinct is called a path. A trail on the other hand, is a walk for which e_j are pairwise distinct [9].

The spanning tree of a connected graph G with n vertices are subsets of n-1 edges that form a connection between all vertices of G and equivalently contains no cycles [11]. In some applications, a spanning tree that covers all edges as well as all vertices is required [3]. The breadth-first spanning trees generation algorithm introduced in this paper is originally developed to produce test suites for software artifacts. Therefore, the desired generated trees are supposed to cover all edges and all vertices, since edges in this case represent operations in the System Under Test (SUT). Hence, the concept of the full spanning tree is introduced. A *full spanning tree* of a graph G is a subgraph that contains all the edges (consequently all the vertices) in G and has no cycles. While the traditional spanning tree is a collection of paths, the full spanning tree is a collection of trails. For the sake of simplicity, throughout the rest of the document "spanning trees" is used to refer to full spanning trees.

A Breadth First Traversal of a graph is defined as follows: Given a graph G = (V, E) and an initial vertex *s*, BFS systematically explores the edges of *G* to find every vertex reachable from *s*. It computes the smallest number of edges from *s* to each reachable vertex. It also produces a "breadth-first tree" with root *s* that contains all reachable vertices [4].

What the devised algorithm is trying to achieve is not just one breadth first traversal of the input digraph, but rather a collection of all the possible breadth first traversals of that graph. The motivation behind creating all possible breadth first traversal trees in the case of generating test suites for a SUT is the need to compare the effectiveness of all possible breadth first traversal trees test suites of the original state diagram.

In this paper, diagrams are used to visualize digraphs, where nodes represent vertices and edges are represented by arrows whose tail is at the initial node of that edge and whose head is at the terminal node. Node 2 in Fig. 1 is the initial vertex for edge 6, while Node 4 is the terminal vertex for the same edge. Fig. 1. is used throughout the paper to explain the proposed algorithm.

III. Similar Work

Usually there are many spanning trees for a connected graph. A reasonable way to generate spanning trees of a graph is to start with one spanning tree, add an edge to it to form a circuit, then remove any other branch to break that circuit. This is called elementary tree transformation [15].

One of the very early attempts to generate all spanning trees of a graph is Char's algorithm that generates all combinations of n-1 edges, then examines each combination to decide whether it forms a tree or not [2]. The algorithm runs in O(E+N+N(T+T')) where T is the number of generated trees, while T' is the number of generated combinations of edges that do not form trees [8].

Mayeda and Seshu develop an algorithm that is based on elementary graph transformations and generates the trees without duplicates. In this work, the idea of starting with a root spanning tree to generate all spanning trees by exhaustive search when replacing each edge is introduced [14].

Minty introduces another idea by generating a partial tree and adding one edge at a time to that tree. If the added edge converts the tree to a graph by forming a cycle, all the edges forming that cycle are removed from the graph. The algorithm's time complexity is O(N×E+N×E×T) where T is the total number of generated trees, while the space complexity is O(N×E) [8].

Hakimi introduces in his research two methods for generating the trees of a given undirected graph. The first algorithm produces duplicate tree [7], while the second has high time complexity as it is not the result of elementary transformations [5].

In [6], an algorithm for generating all spanning trees of directed and undirected graphs is introduced. The time complexity for this algorithm is O(N+E+E×T) and the space is O(N+E),

Matsui's algorithm is based on the idea of generating one spanning tree of an undirected graph and then adding an edge to form a cycle. Removing another edge of the cycle introduces a new spanning tree. The algorithm runs in O(N+E+T) time, and O(N+E) space complexity [13].

Another improvement in generating spanning trees of graphs is by Kapoor and Ramesh who present three algorithms for enumerating all spanning trees in directed, undirected, and weighted graphs. They use the concept of computation tree where each node represents one spanning tree of the undirected graph. They use edge replacement to transform a tree and add a child tree to the computation tree. The first algorithm handles undirected graphs and runs in O(N+E+T) with space requirement of O(N2×E). The second algorithm is for directed graphs and runs at time complexity of O(TN+N3). The third algorithm generates spanning trees of weighted graphs in O(N×E+T×logN) time and O(N×E+T) space [10].

In [16] an algorithm that uses an initial minimum spanning tree and generates the rest of the trees in order of increasing cost using edge exchange is presented. Time complexity is O(T×E×logE+T^2) and space complexity is O(T×E) [16].

An algorithm that requires O((N+E) × (T+T')) where T' is the set of generated graphs that contain cycles and thus do not qualify as trees is introduced in [1]. The algorithm is based on classifying the graph edges into two categories; pendant edges that are not part of a graph cycle, and circuit edges. Pendant edges are included in all trees. Circuit edges parts of cycles in the graph and are analyzed to obtain all possible trees [1].

Knuth in volume 4 of his famous series "The Art of Computer Programming", lists the steps to visit all the spanning trees of a directed graph represented by a doubly linked list. He uses a technique called "dancing links" to remove and restore items from and to doubly linked lists to produce a new tree. So, the idea is very similar to swapping edges of an initial tree to get the rest of the spanning trees [11]. The complexity of the algorithm is not calculated.

Through researching work that covers generating all spanning trees we claim that to the best of our knowledge there is no available work in the literature that generates all BFS distinctive trees. The algorithms surveyed expresses the complexity in terms of the number of generated trees which is an output not input parameter. The algorithm proposed in this paper runs in O(E+T) if expressed in terms of the output which is better time complexity than any of the mentioned pervious algorithms. However, the solution proposed can be improved in terms of space complexity as described in section V.C.2). It is also worth mentioning that there is another difference between the algorithm proposed here and previous algorithms as the generated trees span all edges as well as all vertices. All the trees generated by the proposed algorithm are distinctive.

IV. HYPOTHESIS

A directed graph may have many BFS spanning trees. The proposed algorithm is based on the hypothesis that the reason behind having more than one spanning tree of a single directed graph with a known original node is having nodes with more than one incoming edge; i.e. nodes with indegree > 1. We refer to each of those nodes as a Multi Incoming Edges Node (*MIEN*).

```
Algorithm 1: CreateInitialBreadthTree
Input: A directed graph G with root node n_root
Output: A breadth first spanning tree T

BFST(G)
  n_root ← GET_INITIAL_NODE(G)
  T ← SET_ROOT(n_root)
  //Q is a queue of all graph nodes that has not been yet processed
  ENQUEUE (Q, n_root)
Do
  n ← DEQUEUE(Q)
    //If n is a leaf, then there is no need to explore its outgoing edges
  do if (not LEAF(n))
          n_current ← Copy(n)
          //Getting all of which n is the initial vertex,
          edges ← GET_OUTGOIN_EDGES (G, n)
          //Marking n to know that it has been already added to the
          output tree
          SET_BRANCHED (n, true)
          for each e ∈ edges
                  //Getting the node that is a terminal vertex of e
                  n_d ← GET_DESTINATION_NODE(e)
                  //add the edge e to T such that n_current is the initial
                  node of e and n_d is the terminal node
                  ADD (T, n_current , e, n_d)
                  //If n_d has not been marked as processed
                  previously, add it to the Q
                  If (not BRANCHED (n_d)
                          Add (Q, n_d)
          End loop
  Endif
While (not EMPTY(Q))
```

Figu. 2. Creating Initial Tree

```
Create Sister Tree
Input: A tree t, two MIEN nodes : n_sec and n_origina
Output: A tree identical to the input tree except for the subtrees rooted
at n_sec, n_original. The two subtrees are switched in the new t_new.
CreateSisterTree( (t, n_sec, n_original)
        //Create an exact copy of the input graph
        t_new ← Clone(t)
        //Swap the edges whose terminal nodes n_sec, n_original. This
        means exchanging the parents of the two MIEN nodes.
        t_new ← Swap_Leading_Edges(t_new ,n_sec, n_original )
```

Fig. 3. Creating Sister Tree

Nodes 1,3, and 4 in Fig. 1 are MIEN nodes, while nodes 0, 2, 5, and 6 are not.

An important observation is that if the MIEN node is a leaf that does not have outgoing edges, then it is not significant for creating a different traversal tree, since the traversal path terminates at this node anyway and it will never form a root of a subtree.

To produce two different BFS trees, the graph should have two edges originating from two nodes on the same level (at the same distance from the initial node) and leading to the same non-terminal (not a leaf) destination node. However, that destination node must be on a different level that is further from the root node (deeper) than the two source nodes.

V. THE ALGORITHM

The algorithm starts by creating an initial BFS tree. Then from that tree, a collection of all other possible unique breadth

```
Algorithm 2: CreateBreadthColony
Input: A directed graph G with root node n_root
Output: C: A colony of all breadth first spanning trees of the input
graph.
Calls: Create Initial Breadth Tree of Fig. 2, and Create Sister Tree of
Fig. 3.

CREATE_BREADTH_COLONY (G)
//Creating the first breadth first spanning tree.
 t_i ← BFST (G)
//Adding the initial tree to the list of breadth first spanning trees
 ADD (C, t_i)
 nodes ← GET_ALL_MIEN_NODES(t_i)
//Iterate on all nodes with more than one incoming edge
 for (each n ∈ nodes)
   TreesToUpdate ← SIZE(C)
   n_original ← n
//If this node is in the position where it was first hit by traversal
   If (not HAS_DUPLICATE_PREFIX (n_original))
     n_sec ← ADD_PREFIX(n_original)
     do
       If (SAME_LEVEL (n_sec, n_original)
         //Iterate on the trees that has the current original MIEN node
         n_original in the first position it was hit
         for (each t ∈ TreesToUpdate)
           t_sis ← CREATE_SISTER_TREE (t, n_sec, n_original)
           ADD (C, t_sis)
         End for
       n_sec ← x+ n_sec
      //Exit the loop when there are no more other
      possible positions for this MIEN node nodes n_original
      while (not GET_NODE (t, n_sec) ≠ null)
    Endif
End for
```

Fig. 4. Creating Breadth Trees Colony

first traversal trees, that are modified copies of the first tree, are generated.

A. `Creating the Initial Breadth First Traversal Tree

The first step of the BFS Trees (BFST) generation algorithm is to create an initial breadth first traversal tree. Fig. 2 is the pseudocode for the conventional algorithm used for this step. The algorithm starts from the initial node of the input directed graph and adds that node to a first in, first out data structure (FIFO) referred to as Q in Fig. 2. Then, the algorithm starts consuming node by node from Q. For each consumed node, the algorithm iterates on the outgoing edges and inspects one edge at a time. The destination node n_d of the currently processed edge is retrieved. Then, if n_d has been added to the resulting tree T before, this indicates that the n_d is a MIEN node and that this is not the first time the current traversal reaches n_d. In such case, n_d is marked as MIEN and a prefix 'x' is added to it to indicate that it is a copy of a previously traversed node.

The colony creation algorithm of Fig. 4 makes use of this data. This will be explained in detail in the following section. Then the algorithm adds n_d to the resulting tree T. If n_d has never been processed (i.e. branched before), n_d gets added to the FIFO data structure Q in order to process it at a later stage. This way, this node will never be processed before all the nodes that are already in Q gets processed. At any point in time, the stored nodes in Q are closer to the root of the tree than the node being added at the moment by the BFST algorithm. When a node n_1 is referred to as being closer to the root of the graph than another node n_2, this means that to reach n_1 from the root node, fewer edges are needed than the edges needed to reach n_2.

Finally, the breadth first traversal algorithm terminates when all the nodes in Q are consumed. The following section describes how to make use of BFST to generate all possible breadth first traversal trees of the original tree to construct a breadth first colony of trees.

B. Constructing a Breadth First Colony

The algorithm concerned with generating all unique BFS trees, the pseudocode of which is shown in Fig. 4, starts by calling the BFST of Fig. 2 algorithm to create an initial BFS tree. After creating the initial tree, the algorithm loops on all MIEN nodes of that tree that were identified in the first step of the algorithm while creating the initial BFS tree. For each MIEN node ($n_{original}$), the algorithm loops on all of the copies of that node. In other words, the algorithm loops on all the different ways to reach that node. If a copy n_{sec} is at the same level of $n_{original}$, the subtrees whose roots are $n_{original}$ and n_{sec} gets swapped by calling the algorithm in Fig3 to create a new tree and that new tree gets added to the list of trees constructing the colony C. If the two copies of the same node are not at the same level in the initial breadth first traversal tree, no switching takes place, and we move to the next copy of the node to examine it. After examining all the copies and performing the required switches, another node of the original tree is examined until all the nodes belonging to the initial tree are consumed.

It is worth noting that based on the BFS performed in the first step, the order by which nodes are consumed from the queue is based on their distance from the root. So, no child node is consumed before its parent. Otherwise, children will be swapped in trees where parents are not swapped yet and this may result in having duplicate trees and missing BFS trees.

C. Complexity Analysis

1) Time Complexity

Theorem V.c.1: BFST runs in O(E) time.

Proof: The algorithm in Fig. 2 has two nested loops. The outer loop has number of iterations equal to the number of nodes, while the inner loop has number of iterations equal to the outer degree of each node. Therefore, the complexity of the algorithm is O(N×D) where D is the average outdegree of the vertices. However, N×D is the average number of edges outgoing of each node multiplied by the number of nodes, which is simply the total number of edges in the graph. Thus, N×D = E where E is the total number of edges in the graph. This proves Theorem c.1.

This is also logically intuitive since the BFST visits each edge of the original graph G once.

Theorem V.c.2: The algorithm CreatingAllTrees in Fig. 4. has time complexity of O ($E+\frac{E^n}{N^n}$) or in function of the output trees O(E+T) where T is the number of generated trees.

Proof: The algorithm calls algorithm 1 above, the time complexity of which is O(E) from Theorem V.c.1, then it goes through three nested loops. The most outer loop iterates on MIEN nodes that are not leaves. Assuming worst case, where all nodes in the graph are non-leaf MIEN nodes, the number of iterations for this loop will be N where N is the total number of nodes in the original graph. The middle loop loops on all copies of each MIEN node. Given the previous assumption that all nodes are MIEN, the worst-case scenario for the number of iterations for this loop will be the average number of edges per node. This number is equal to $\frac{E}{N}$. The most inner loop iterates on all previously created trees. This number is always $< \frac{E}{N}$, which is the average number of edges. Hence, the total complexity of the algorithm is O ($E + \frac{E^n}{N^n}$). This can be logically deduced since the total number of iterations the algorithm goes through is equal to the number of generated trees. This number is at the maximum equal to all combinations of copies of MIEN nodes. Assuming that each MIEN node has $\frac{E}{N}$ (average indegree of a node) copies. Then, all possible combinations are $\frac{E^n}{N^n}$ as suggested by theorem V.c.2.

2) SPACE COMPLEXITY

Theorem D.2.1: The resulting trees of the algorithm occupy $\frac{E^n}{N^n}$ space.

Proof: As explained in the previous example the number of trees generated is equal to the maximum number of all combinations of ingoing edges per node. The average number of ingoing edges for each node is $\frac{E}{N}$. Therefore, all possible combinations are equal to $\frac{E^n}{N^n}$.

However, the space occupied can be drastically improved as each tree differ from the other by only a swapped edge and therefore a swapped subtree. A modified algorithm can store the

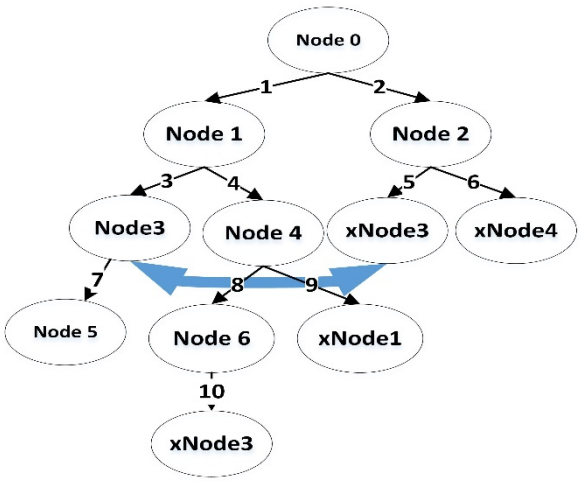

Fig. 5. BFS Tree #1 of G

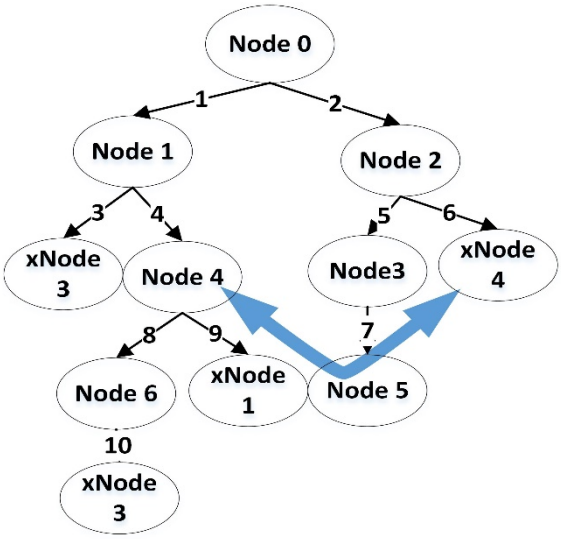

Fig. 6. BFS Tree #2 of G

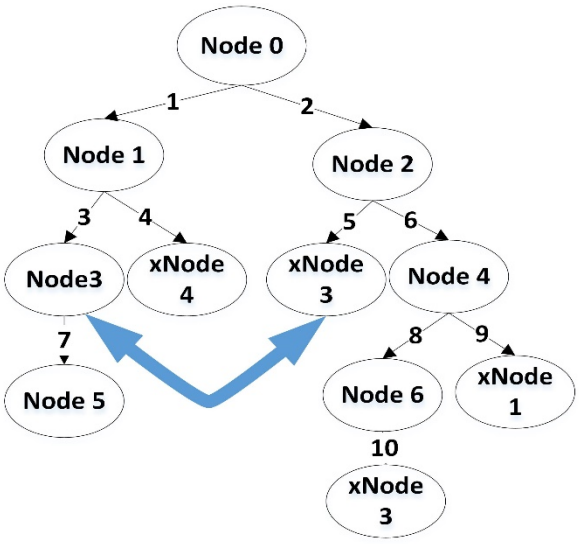

Fig. 7. BFS Tree #3 of G

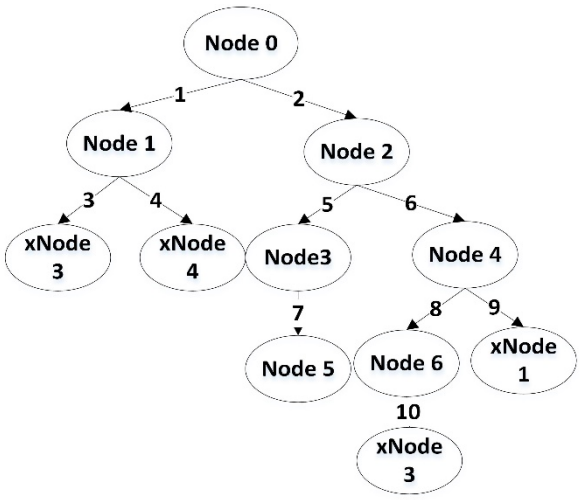

Fig. 8. BFS Tree #4 of G

difference between the trees only rather than the whole set of trees. This will result in a space complexity of $O(\frac{E}{N})$.

VI. Example

To illustrate how the algorithm works, Fig. 1 shows a directed graph G that has seven nodes and ten edges. Three of the seven nodes are MIEN nodes. Nodes 3 and 4 are non-leaf MIEN nodes in the original graph of Fig. 1 that affect the number of possible spanning trees of G.

If we assume that Fig. 5 is the initial spanning tree produced by the algorithm in Fig. 2, and Node 3 is the first MIEN node to be processed. Swapping the subtree rooted at Node 3 and its copy xNode3 that is on the same level results in the subtree rooted at Node 3 including Node 5 to be descendent of Node 1 rather than Node 2 as shown in the second spanning tree of G shown in Fig. 6.

There are no other copies of Node 3 to be switched with the current position of Node 3 to produce a new tree since the third copy of Node 3 is at a further distance from the root and therefore in a breadth first traversal algorithm the third copy of Node 3 can never be branched at that position. Therefore, the algorithm moves to the second MIEN Node, which is in this case Node 4 that is a child of Node 1. By swapping Node 4 and xNode 4 (in Fig. 5) that are on the same level, a new tree gets produced as shown in Fig. 7.

Next step is providing another copy of the Tree in Fig. 6 in order to have the combination where both Node 3 and Node 4 are descendants of Node 2. This swap will produce the fourth spanning tree shown in Fig. 8.

Although Node 1 is a MIEN node in the original graph of Fig. 1, no swapping is possible since the two possible positions of Node 1 are not on the same level in the tree and hence any breadth first traversal will reach Node 1 as a root first.

VII. Experimentaion

To facilitate the process of validating the algorithm proposed in this research with empirical studies, we developed a tool that

automatically generates breadth first suites and examines the resulting trees to verify their uniqueness. The tool has been used to experiment with small case studies as well as four main real life case study systems.

The breadth first traversal trees generation algorithm proposed is generic and could be applied to any directed graph. However, it was originally introduced to generate test suites for software based on state machines modeling. The state machine representing the SUT is the directed graph being input to the algorithm, while the trails of the generated trees are the test cases used to test the SUT.

The real-life case studies used during experimentation are of various characteristics and from different fields. The four case studies are a data structure representing an ordered set, an embedded controller represented by a cruise control simulator, an Automatic Telling Machine (ATM), and a Videocassette Recorder (VCR). This is in addition to many other case studies similar to Fig. 1. Table 1 is a summary of the sizes of the different case studies, the number of generated trees, and the execution time for the proposed algorithm using a machine that runs Microsoft® Windows 10, with 8 GB RAM and Intel® core i5 processor.

TABLE 1 EXECUTION TIME FOR GENERATING ALL BFS TREES

CAS STUDY	ORDERED SET	VCR	CRUISE CONTROL	ATM
NO. OF NODES	5	17	5	10
NO. OF EDGES	17	65	29	22
NO. OF BF TREES	2	24	3	9
GENERATION TIME (MS)	1	6	<1	<1

VIII. CONCLUSION

The proposed algorithm is of interest in many applications in diverse areas such as social networks, finding shortest paths, crawlers of search engines, GPS navigation systems, and garbage collection.

The presented algorithm provides a solution for the problem of finding all possible unique BFS traversal trees of a directed graph in O ($E+\frac{E^n}{N^n}$) time complexity, which is equivalent to O(E+T) where T is the number of generated BFS trees. Experiments that were carried to measure the effectiveness and efficiency of the algorithm has been summarized in the paper.

To the best of our knowledge, there is no available algorithm other than the one presented in this paper that generates all breadth first full spanning trees. Although the literature has many algorithms producing all spanning trees of a graph, the provided algorithm is of very efficient time complexity and it could be generalized to handle undirected graphs as well as directed graphs. However, there is room for improvement in terms of the space complexity as currently the algorithm stores all generated trees while with minor modification it can store the difference between the generated trees only which will result in remarkable space saving possibility.

REFERENCES

[1] M. Chakraborty, R. Mehera, and R. K. Pal, "Divide-and-Conquer: An Approach to Generate All Spanning Trees of a Connected and Undirected Simple Graph," *Adv. Intell. Syst. Comput.*, vol. 3, pp. 65–84, 2014.

[2] J. Char, "Generation of Trees, Two-Trees, and Storage of Master Forests," *IEEE Trans. Circuit Theory*, vol. 15, no. 3, pp. 228–238, 1968.

[3] G. Chartrand, *Introductory Graph Theory*. 2012.

[4] T. Cormen, R. Rivest, and C. Leiserson, *Introduction to Algorithms*, Third. Cambridge, Mass., United States: McGraw-Hill, 2009.

[5] W. Duplications, "Generation of Trees," no. July, pp. 7–8, 1969.

[6] H. N. Gabow and E. W. Myers, "Finding All Spanning Trees of Directed and Undirected Graphs," *SIAM J. Comput.*, vol. 7, no. 3, pp. 280–287, 1978.

[7] S. L. Hakimi, "On trees of a graph and their generation," *J. Franklin Inst.*, vol. 272, no. 5, pp. 347–359, 1961.

[8] R. Jayakumar, K. Thulasiraman, and M. N. Swamy, "Complexity of Computation of a Spanning Tree Enumeration Algorithm," *IEEE Trans. Circuits Syst.*, vol. 31, no. 10, pp. 853–860, 1984.

[9] D. Jungnickel, *Graphs, Networks and Algorithms*, vol. 53, no. 9. 2008.

[10] S. Kapoor and H. Ramesh, "Algorithms for enumerating all spanning-trees of undirected and weighted graphs," *Proc. 2nd Work. Algorithms {&} Data Struct.*, no. 519, pp. 461–472, 1991.

[11] D. E. Knuth 1938, *The art of computer programming*. 2005.

[12] A. Mathematics, "On Directed Trees and Directed k-Trees of a Digraph and their Generation Author (s): Wai-Kai Chen Source : SIAM Journal on Applied Mathematics , Vol . 14 , No . 3 (May , 1966), pp . 550-560 Published by : Society for Industrial and Applied Mathematics," vol. 14, no. 3, pp. 550–560, 2016.

[13] T. Matsui, "An algorithm for finding all the spanning trees in undirected graphs," no. METR93-08, 1993.

[14] W. Mayeda and S. Seshu, "Generation of Trees Without Duplications," *IEEE Trans. Circuit Theory*, vol. 12, no. 2, pp. 181–185, 1965.

[15] Narsingh Deo, *Graph Theory with Applications to Engineering and Computer Science*. 2017.

[16] K. Sörensen and G. K. Janssens, "An algorithm to generate all spanning trees of a graph in order of increasing cost," *Pesqui. Operacional*, vol. 25, no. 2, pp. 219–229, 2005.

[17] M. Then, M. Kaufmann, and F. Chirigati, "The More the Merrier: Efficient Multi-Source Graph Traversal," *Proc. ...*, pp. 449–460, 2014.