

# A Divide-and-Conquer Algorithm for All Spanning Tree Generation

**Maumita Chakraborty, Ranjan Mehera and Rajat Kumar Pal**

**Abstract** This paper claims to propose a unique solution to the problem of all possible spanning tree enumeration for a simple, symmetric, and connected graph. It is based on the algorithmic paradigm named divide-and-conquer. Our algorithm proposes to perform no duplicate tree comparison and a minimum number of circuit testing, consuming reasonable time and space.

**Keywords** Circuit testing • Connected graph • Divide-and-conquer • Duplicate tree comparison • Simple graph • Spanning tree • Symmetric graph

## 1 Introduction

Divide-and-conquer is a well-known algorithmic approach for solving different problems. It has three phases namely, divide, conquer, and combine [1]. In this paper, we have used this new divide-and-conquer approach in generating all possible spanning trees of a simple, symmetric, and connected graph. All spanning tree generation has been an area, well-explored and well-known for its wide applications in the fields of computer science, chemistry, medical science, biology and many others. The algorithms developed for spanning tree enumeration are mainly targeted towards generating all trees in optimum time and space. Moreover, checking for the

---

M. Chakraborty (✉)

Department of Information Technology, Institute of Engineering  
and Management, Y-12, Block-EP, Sector-V, Salt Lake, Kolkata, India  
e-mail: maumita.chakraborty@gmail.com

R. Mehera

Subex, Inc., 12303 Airport Way, Suite 390, Broomfield, CO 80021, USA  
e-mail: ranjan.mehera@gmail.com

R.K. Pal

Department of Computer Science and Engineering, University of Calcutta,  
JD-2, Sector III, Salt Lake, Kolkata, India  
e-mail: pal.rajatk@gmail.com

duplicate tree and non-tree sequences are two major issues in this area of research. In our algorithm, we claim to eradicate the duplicate tree issue completely and also reduce the number of non-tree sequence generations.

## 2 Literature Survey

Starting from the 60s till date, many algorithms have been devised for computation of all possible trees of a simple, symmetric, and connected graph. Based on the methods being used, three major classifications have been made in this regard, namely: tree testing method [2–11], elementary tree transformation method [12–19], and trees by successive reduction method [20–22].

All the algorithms have some unique features (as well as novelties and limitations) of their own, which may differ even when they are under the same head of classification. In general, *tree testing method* is mainly responsible for generating all possible sequences of the desired length, among which the tree sequences are accepted, and the non-trees are discarded. On the other hand, the second category, namely *elementary tree transformation* starts with an initial BFS or DFS tree and then repeatedly generates one tree from the other by replacement of one or more edges, whose selection criteria is different for different algorithms. Lastly, the *successive reduction method* reduces the graph to trivial subgraphs gradually. Trees of the original graph are then obtained from the trees of the trivial subgraphs. Almost all the algorithms till date can somehow be categorized under any one of the above, as a result of which there is always a scope for devising some new approach in this problem domain.

In the subsequent sections, we discuss the functioning of some of the earlier existing methods of generating all spanning trees in brief, then the development of our algorithm, *DCC\_Trees*, with proper examples and elaborations.

## 3 Existing Techniques for All Spanning Tree Generation

In this section, we present very briefly the underlying working principle of three algorithms which fall under the three existing techniques of all spanning tree generation, as mentioned above.

### 3.1 Trees by Test and Select Method

As mentioned in the earlier section, this method generates all possible sequences of edges; some of them are tree sequences while others are not. J.P. Char had adopted this method of tree generation in his algorithm. Char defines a  $(n - 1)$ -digit

sequence  $\lambda = (\text{DIGIT}(1), \text{DIGIT}(2), \dots, \text{DIGIT}(n-1))$  such that  $\text{DIGIT}(i)$ ,  $1 \leq i \leq n-1$ , is a vertex adjacent to vertex  $i$  in  $G$ . He has also given a *Tree Compatibility Checking* for the sequences as:

**The sequence  $(\text{DIGIT}(1), \text{DIGIT}(2), \dots, \text{DIGIT}(n-1))$  represents a spanning tree of graph  $G$  if and only if for each  $j \leq n-1$ , there exists in  $G$  a sequence of edges with  $(j, \text{DIGIT}(j))$  as the starting edge, which leads to a vertex  $k > j$  [3].**

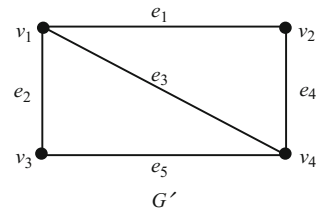
Here we present Char's algorithm [3] at a glance.

Any graph  $G$  is represented by the adjacency lists of its vertices.  $\text{SUCC}(\text{DIGIT}(i))$  is the entry next to  $\text{DIGIT}(i)$  in the adjacency list of vertex  $i$ .

1. Begin
2. Find the initial spanning tree and obtain the initial tree sequence  $\lambda = (\text{REF}(1), \text{REF}(2), \dots, \text{REF}(n-1))$ ;
3. Renumber the vertices of the graph using the initial spanning tree;
4. Initialize  $\text{DIGIT}(i) := \text{REF}(i)$ ,  $1 \leq i \leq n-1$ ;
5. Output the initial spanning tree;
6.  $k := n-1$ ;
7. While  $k \neq 0$  do begin
8. If  $\text{SUCC}(\text{DIGIT}(k)) \neq \text{nil}$  then
9. Begin
10.  $\text{DIGIT}(k) := \text{SUCC}(\text{DIGIT}(k))$ ;
11. if  $\text{DIGIT}(i)$ ,  $1 \leq i \leq n-1$ , is a tree sequence then
12. Begin
13. Output the tree sequence;
14.  $k := n-1$ ;
15. End;
16. End;
17. Else begin
18.  $\text{DIGIT}(k) := \text{REF}(k)$ ;
19.  $k := k-1$ ;
20. End;
21. End;

For the example graph  $G'$  shown in Fig. 1, we can start with an initial tree  $((v_1, v_2), (v_2, v_4), (v_4, v_3))$  obtained by performing BFS on  $G'$ . The vertices  $v_1, v_2, v_4$ , and  $v_3$  are renumbered to  $v_1, v_2, v_3$ , and  $v_4$ , respectively. The adjacency lists of the graph show that there are no more adjacent vertices from  $v_4$  or  $v_2$ . Hence, the next tree sequence will be  $((v_1, v_4), (v_4, v_2), (v_3, v_1))$  and then  $((v_1, v_4), (v_2, v_1), (v_4, v_3))$ , and so on.

**Fig. 1** An example graph  $G'$



### 3.2 Elementary Tree Transformation Method

It has been mentioned earlier that this technique mainly operates by replacing one edge from an existing tree by another suitable edge to give rise to another tree. One such algorithm by Shioura and Tamura [18], following the above working principle, assumes that any spanning tree  $T^0$  be the progenitor of all spanning trees. It then outputs all spanning trees by reversely scanning all children of any spanning tree. They have also defined a useful child-parent relation.

Let  $G$  (consisting of  $V$  vertices and  $E$  edges) be an undirected connected graph consisting of a vertex set  $\{v_1, v_2, \dots, v_V\}$  and an edge set  $\{e_1, e_2, \dots, e_E\}$ . A spanning tree of  $G$  is represented as its edge-set of size  $(V - 1)$ . For any spanning tree  $T$  and any edge  $f \in T$ , the subgraph induced by the edge-set  $T \setminus f$  has exactly two components. The set of edges connecting these components is called the fundamental cut associated with  $T$  and  $f$ , written as  $C^*(T \setminus f)$ . For any edge  $f \in T$  and any arbitrary edge  $g \in C^*(T \setminus f)$ ,  $T \setminus f \cup g$  is also a spanning tree. For any edge  $g \notin T$ , the edge-induced subgraph of  $G$  by  $T \cup g$  has a unique circuit, the fundamental circuit associated with  $T$  and  $g$ . The set of edges of the circuit is denoted by  $C(T \cup g)$ . For any  $g \notin T$  and any  $f \in C(T \cup g)$ ,  $T \cup g \setminus f$  is a spanning tree. Relative to a spanning tree  $T$  of  $G$ , if the unique path in  $T$  from vertex  $v$  to the root  $v_1$  contains a vertex  $u$ , then  $u$  is called an ancestor of  $v$  and  $v$  is the descendant of  $u$ . Similar is the case for edges. A depth-first spanning tree has been defined as a spanning tree such that for each edge of  $G$ , its one incidence vertex is the ancestor of the other. The algorithm at a glance, as given by the authors [18] is as follows.

#### Procedure All-Spanning-Trees ( $G$ )

**Input:** a graph  $G$  with a vertex-set  $\{v_1, v_2, \dots, v_V\}$  and an edge-set  $\{e_1, e_2, \dots, e_E\}$ ;

1. Begin
2. Using depth-first search, do
  - find a depth-first spanning tree  $T^0$  of  $G$ ;
  - sort vertices and edges to satisfy the ancestor-descendant relationship;
3. Output  $T^0$  {" $e_1, e_2, \dots, e_{V-1}$ , tree"};
4. Find-children ( $T^0, V - 1$ );
5. End;

#### Procedure Find-Children ( $T^p, k$ )

**Input:** a spanning tree  $T^p$  and an integer  $k$  with  $e_k < \text{Min}(T^0 \setminus T^p)$ ;

1. Begin
2. If  $k \leq 0$  then return;
3. For each  $g \in \text{Entr}(T^p, e_k)$  do begin {output all children of  $T^p$  not containing  $e_k$ };
4.  $T^c := T^p \setminus e_k \cup g$ ;
5. Output (" $-e_k, +g$ , tree");
6. Find-children( $T^c, k - 1$ );
7. Output (" $-g, +e_k$ ");

8. End;
9. Find-children( $T^p, k - 1$ ); {find the children of  $T^p$  containing  $e_k$ }
10. End;

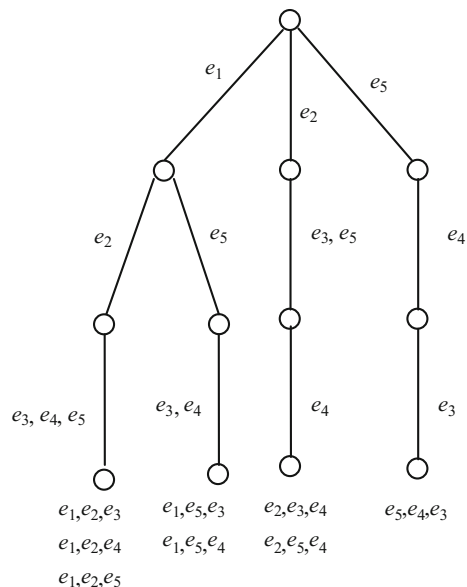
Following the above algorithm,  $T^0 = (e_1, e_4, e_5)$  can be an initial DFS tree for the graph  $G'$  (Fig. 1). The edges  $e_1, e_4$ , and  $e_5$  are to be replaced by other suitable edges of the graph one by one to give rise to its child trees. For example,  $e_1$  can be replaced by  $e_2$  and  $e_3$  to give rise to the trees  $T^{01}(e_4, e_5, e_2)$  and  $T^{02}(e_4, e_5, e_3)$ . Similarly, replacement of  $e_4$  and  $e_5$  will result in some more second generation trees. Once again, the second generation trees like  $T^{01}, T^{02}$ , etc. can also give rise to some other trees by replacement of their edges by some other suitable edges of the graph  $G'$ . This process of tree generation stops as no more child trees are generated.

### 3.3 Successive Reduction Method

This tree generation method operates on the principle of dividing a large graph into smaller subgraphs continuously till the reduced graphs are trivial like edges. Trees of the original graph are then obtained from the trees of the trivial subgraphs. One of the algorithms which rely on this technique is by Winter [22]. Here, a new enumeration algorithm on spanning trees based on the idea of contraction and removal of edges of the graph is presented. First, it constructs all spanning trees containing some selected edge  $e_1$ , then all spanning trees containing another edge  $e_2$ , but not  $e_1$ , and so on.

Figure 2 shows the computation flow tree for generating all spanning trees of the graph  $G'$  (of Fig. 1) using Winter's algorithm. The labels on the edges of the

**Fig. 2** Computation flow tree for Winter's tree generation



computation tree are for those edges of  $G'$  along which contraction takes place. For example, when  $G'$  is contracted along  $e_1$ , it results into two branches (left one indicates further contraction along  $e_2$  and right one indicates further contraction along  $e_5$ ). That is why, the trees generated from the leftmost branch always include  $e_1$  and  $e_2$ , while the next branch always includes  $e_1$  and  $e_5$ , but never includes  $e_2$ . Similarly, contractions on  $G'$  can start from two more edges, namely  $e_2$  and  $e_5$  resulting into two different sets of trees which never include  $e_1$  as one of their edges. The trees generated following a particular sequence of contractions are mentioned at the bottom of each path in the computation flow tree of graph  $G'$ , shown in Fig. 2.

## 4 The New Algorithm (*DCC\_Trees*)

Our objective is to use divide-and-conquer, a new paradigm towards the generation of all possible spanning trees of a graph [1]. A typical divide-and-conquer algorithm solves a problem using following three steps:

Divide: Break the given problem into subproblems of the same type.

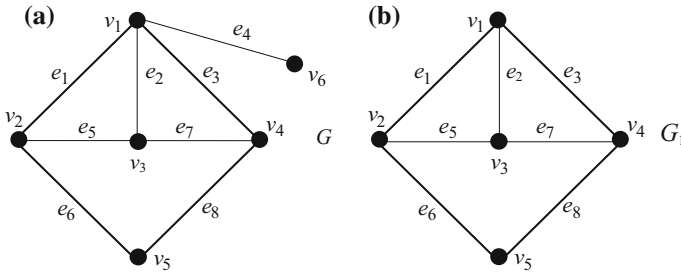
Conquer: Recursively solve these subproblems.

Combine: Appropriately combine the results of the subproblems.

In our algorithm, *DCC\_Trees*, we have used the approach mentioned above to divide a graph into partitions, based on certain criteria/measures, and then merged them in various possible ways to generate different spanning trees. We have defined two types of partitions: primary and secondary, depending on the necessity of their presence in the computed spanning trees. Our algorithm claims to generate no duplicate tree. A very small number of subgraphs generated by the algorithm may lead to circuits, which have also been taken care of subsequently.

### 4.1 Selection of a Reference Vertex and Elimination of Its Pendant Edges

An arbitrary vertex of the given graph is chosen as the reference vertex,  $v_{\text{ref}}$ . If there are one or more pendant edges going out from  $v_{\text{ref}}$  in the graph, then those edges need to be removed. The concept behind this removal is that pendant edges always get included in the resultant spanning trees. Moreover, the removal of pendant edge is necessary, considering the fact that this is a recursive procedure because removal of one pendant edge may give rise to another one and so forth. This step is also required in terms of the completeness of the proposed algorithm. Hence, after generation of all spanning trees with respect to  $v_{\text{ref}}$ , we need to add the pendant edges once again to  $v_{\text{ref}}$  to get  $v_{\text{ref}}$  connected to those vertices which are the other



**Fig. 3** a An example graph,  $G$  b Graph  $G_1$  derived from  $G$  by removing pendant edge  $e_4$

end vertices of those pendant edges. In this regard, let us consider an example graph  $G$  in Fig. 3a. Let us choose any vertex  $v_1$  as  $v_{\text{ref}}$ . From  $v_1$ , there is a pendant edge  $e_4$  going out to  $v_6$ , which is removed for the time being. As a result, the graph formed ( $G_1$ ) is shown in Fig. 3b. We now compute all possible trees of the graph  $G_1$ .

## 4.2 Divide: Decomposition of the Graph

In this phase of the algorithm, we divide the graph into partitions, based on certain criteria/measures. The partitions have been classified into two types: primary and secondary. The formation of primary and secondary partitions is being described in the following two subsections.

**Formation of primary partitions.** All possible combinations of the edges coming out from the reference vertex,  $v_{\text{ref}}$ , taken one to all at a time, form the primary partitions in our algorithm. A primary partition is a necessary or mandatory component of the resultant spanning trees formed from the given graph, as it will ensure the connectivity of  $v_{\text{ref}}$  with the rest of the graph. If there are  $m$  edges coming out of  $v_{\text{ref}}$ , then the number of primary partitions from  $v_{\text{ref}}$  is the sum  ${}^mC_1 + {}^mC_2 + \dots + {}^mC_m$ .

For the example graph,  $G_1$  considered in Fig. 3b, the three edges coming out of  $v_1$  ( $v_{\text{ref}}$  for  $G_1$ ) are  $e_1$ ,  $e_2$ , and  $e_3$ . Thus, the seven ( ${}^3C_1 + {}^3C_2 + {}^3C_3$ ) primary partitions for  $G_1$  are  $\{e_1\}$ ,  $\{e_2\}$ ,  $\{e_3\}$ ,  $\{e_1, e_2\}$ ,  $\{e_1, e_3\}$ ,  $\{e_2, e_3\}$ , and  $\{e_1, e_2, e_3\}$ .

**Formation of secondary partitions.** Once a primary partition is selected, one or more vertices, other than the  $v_{\text{ref}}$ , also get included or visited. Some other unvisited vertices may still be there in the graph. Each edge between a pair of distinct unvisited vertices is considered to be a separate secondary partition. Not only edges, but single left-out vertices can also form separate secondary partitions. After selecting any one primary partition, we find out the secondary partitions, as required. These are the conditional partitions which are subject to change with a change in a primary partition.

Whether we have only edges or vertices or a combination of both as secondary partitions depends on the number of unvisited vertices (after inclusion of the primary partition) as well as the available connections between them, i.e. their adjacency in the given graph.

In this regard, let us find out the secondary partitions for each of the primary partitions of  $G_1$  (Fig. 3b).

- Let us first consider the primary partitions of size one. So, when  $\{e_1\}$  is taken, vertices  $v_3$ ,  $v_4$ , and  $v_5$  remain unvisited. As there is an edge  $e_7$  between  $v_3$  and  $v_4$ , so  $e_7$  is one of the secondary partitions for  $\{e_1\}$ . Now since we have considered edge  $e_7$ , which implies we have already visited vertices  $v_3$  and  $v_4$  and hence edge  $e_8$  can't be considered as a secondary partition, since one end of  $e_8$  is already visited (vertex  $v_4$ ). So now there is a left-out vertex  $v_5$  which has not been included. As a result,  $v_5$  becomes another secondary component. Similarly, for each of the next two primary partitions,  $\{e_2\}$  and  $\{e_3\}$ , we get one edge component and one vertex component as the secondary partitions. For  $\{e_2\}$ , they are either  $e_6$  and  $v_4$ , or  $e_8$  and  $v_2$  respectively. And for  $\{e_3\}$ , the secondary partitions are either  $e_5$  and  $v_5$ , or  $e_6$  and  $v_3$ ,  $e_8$  and  $v_2$  respectively.
- Then, we consider the primary partitions of size two. So, for  $\{e_1, e_2\}$  and  $\{e_2, e_3\}$ ,  $e_8$  and  $e_6$  form the secondary partitions, respectively. However, for  $\{e_1, e_3\}$ , we do not get any edge component, as the left-over vertices  $v_3$  and  $v_5$  are not connected. Hence,  $v_3$  and  $v_5$ , in this case, form two separate secondary partitions.
- Lastly, for the primary partition  $\{e_1, e_2, e_3\}$  of size three, vertex  $v_5$  is the only secondary partition.

### 4.3 Conquer: Searching for Connectors

This section is targeted towards finding out what are the different ways in which the partitions can be joined to give rise to all possible and different spanning trees.

Once we have all the partitions available from the divide phase of *DCC\_Trees*, we search for the connectors which are supposed to join them. For a particular primary partition, the secondary partitions are gradually joined in the reverse order in which they were formed by a set of connectors named as  $CS_1, CS_2, \dots$ , and so on. A pair of partitions along with the connectors from the corresponding connector set gives rise to the subgraphs which gradually lead to the formation of different spanning trees of the graph. This process is continued till we find out the connectors that join the above-formed subgraph with the corresponding primary partition, i.e. those edges which join the primary partition with the rest of the vertices in the given graph. In case any secondary partition,  $s_1$ , does not find a connection with any other secondary partition or  $s_1$  is the singleton secondary partition, we directly join  $s_1$  with the corresponding primary partition with some bridges named as main connectors.  $MC$  is such a set of main connectors.



Let us now find out the set of connectors for the example graph  $G_1$  in Fig. 3b.

- For partition  $\{e_1\}$ ,  $CS_1$  (joining  $e_7$  with  $v_5$ ) =  $e_8$ , and  $MC_1$  (joining  $e_1$  with rest) =  $(e_5, e_6)$ . Similarly, for  $\{e_2\}$ ,  $CS_1$  (joining  $e_6$  with  $v_4$ ) =  $e_8$ , and  $MC_1$  (joining  $e_2$  with rest) =  $(e_5, e_7)$ . Also, for  $\{e_3\}$ ,  $CS_1$  (joining  $e_5$  with  $v_5$ ) =  $e_6$ , and  $MC_1$  (joining  $e_3$  with rest) =  $(e_7, e_8)$ .
- Then, we consider the next set of partitions of size two. For each of  $\{e_1, e_2\}$  and  $\{e_2, e_3\}$ , there is only one secondary partition; hence, no  $CS$ 's. So,  $MC_1$  (for  $\{e_1, e_2\}$ ) =  $(e_6, e_7)$  and  $MC_1$  (for  $\{e_2, e_3\}$ ) =  $(e_5, e_8)$ . However, for the partition  $\{e_1, e_3\}$ , there are two secondary partitions, each of which has to be joined separately with  $\{e_1, e_3\}$ . Hence,  $MC_1$  (joining  $\{e_1, e_3\}$  with  $v_3$ ) =  $(e_5, e_7)$  and  $MC_2$  (joining  $\{e_1, e_3\}$  with  $v_5$ ) =  $(e_6, e_8)$ .
- For the last partition  $\{e_1, e_2, e_3\}$ , the only  $MC_1$  (joining  $\{e_1, e_2, e_3\}$  with  $v_5$ ) =  $(e_6, e_8)$ .

Thus, in the conquer phase, for each primary partition, we find out all possible sets of connectors for all pairs of partitions until we include all the vertices of the graph. Now, we move to combine phase of our algorithm, where we decide how to take different combinations of connectors for each set of partitions to form different tree sequences.

#### 4.4 Combine: Forming the Trees

In this section, we form the required trees of the given graph in two phases.

**Tree generation: Phase 1.** From the earlier two sections, we know that, for each primary partition,  $p_i$ , there can be a set of secondary partitions,  $S_i$ , where  $|S_i| \geq 0$ . To join each such secondary partition of  $S_i$  with one other, there are sets of connectors,  $CS_i$ , and to join them with the primary partitions, we have a set of main connectors,  $MC_i$ , where  $|MC_i| \geq 0$ . We take two partitions and one connector from the corresponding  $CS$ , at a time, to form a subgraph, which is again joined with another partition taking one connector from the next set,  $CS$ . This continues till the primary partition is joined with this gradually growing subgraph using one main connector from set  $MC$ . The largest subgraph thus formed is a spanning tree of the given graph. Again, the same process is repeated with new combinations of connectors at each step, giving rise to a new tree. Taking different combinations of connectors from  $CS$  and  $MC$ , we thus get different trees of the graph. The whole procedure is repeated for each  $p_i$ , thus, yielding more trees in each case. The algorithm claims to generate no circuit or duplicate tree in this phase.

Following the above procedure of Phase 1 tree generation, we get the following trees; here the trees are formed by exactly  $n - 1$  edges of a given graph enclosed within parentheses:

- The partition  $\{e_1\}$  of  $G_1$  yields the following trees:  $(e_1, e_7, e_8, e_5)$  and  $(e_1, e_7, e_8, e_6)$ , where  $e_1$  is the primary partition,  $e_7$  is secondary partition,  $e_8$  is the

connector joining  $e_7$  with  $v_5$ ,  $e_5$  and  $e_6$  are two main connectors joining  $\{e_1\}$  with the rest of the subgraph. Similarly,  $\{e_2\}$  generates the trees  $(e_2, e_6, e_8, e_5)$  and  $(e_2, e_6, e_8, e_6)$ , while  $\{e_3\}$  generates  $(e_3, e_5, e_6, e_7)$  and  $(e_3, e_5, e_6, e_8)$ .

- Trees from partition  $\{e_1, e_2\}$ :  $(e_1, e_2, e_8, e_6)$  and  $(e_1, e_2, e_8, e_7)$ . Trees from  $\{e_1, e_3\}$ :  $(e_1, e_3, e_5, e_6)$ ,  $(e_1, e_3, e_5, e_8)$ ,  $(e_1, e_3, e_7, e_6)$  and  $(e_1, e_3, e_7, e_8)$ . Also, trees from  $\{e_2, e_3\}$ :  $(e_2, e_3, e_6, e_5)$  and  $(e_2, e_3, e_6, e_8)$ .
- Lastly, partition  $\{e_1, e_2, e_3\}$  generates the trees  $(e_1, e_2, e_3, e_6)$  and  $(e_1, e_2, e_3, e_8)$ .

**Tree generation: Phase 2.** This is the next phase in tree generation where we take more than one connector from each set, whether *CS* or *MC*. If there are  $r$  connectors in a connector set, then we take combinations of two or more (up to  $r$ ) connectors at a time. The salient features of this phase are:

1. All the partitions are not included here. Only the primary partition and the corresponding connector combination from *CS* are taken. These now become the mandatory component for this particular case. With the remaining unvisited vertices, we once again form secondary partitions. The mandatory component and the secondary partitions can thus be combined in all possible ways, as described in Phase 1 of tree generation. This is a recursive process where the new combination of two or more connectors is added to the primary partition to form the new mandatory component. Once again secondary partitions are found out and once again they are joined by connectors like before.
2. If there are more than two connectors in any connector set, *CS* or *MC*, we take combinations of two or more such connectors. Now this connector combination is joined with the remaining secondary partition(s) by another set of connectors which should not include any member of the previous connector set, whose members were already considered.
3. While taking combinations of connectors from any connector set, we do not take those combinations which when taken along with the primary partition do not exceed the size of a tree of the given graph.
4. When the main connectors are combined, the primary partition and the main connectors form the only mandatory component at that time, and the same process of finding out secondary components, joining them with the mandatory component in all possible ways is repeated. However, this is the only phase where there is a chance of circuit formation.
5. While combining the main connectors of *MC*, if we find at least two connectors from two different end points of the primary partition, going out to a common vertex, a circuit is formed. Hence, that particular combination of connectors does not yield any tree and thus needs to be discarded.

**Theorem 1** *The algorithm never generates duplicate trees.*

*Proof* According to the algorithm, Phase 1 takes different connectors from different *CS*'s and *MC*'s at a time. Thus, no question of duplication arises. In Phase 2 of tree generation, we take two or more combinations of connectors from the same *CS* or *MC* along with the primary partition to generate more and more trees. This process

may get repeated based on new sets of connectors found at each step. For a particular primary partition, no connector combination from any connector set is allowed to have a member of its own set which joins the remaining secondary component(s). But, if the primary partition changes, and if some connector combination repeats, it does not generate any duplicate tree because of inclusion or exclusion of different edges in the primary partition.  $\square$

**Lemma 1** *Circuits are formed only for primary partitions of size  $k > 1$ .*

*Proof* A primary partition of size  $k > 1$  has  $k$  number of branches going out from  $v_{\text{ref}}$ , and hence,  $k$  end points. There can be  $k$  or even more number of connectors, from  $k$  such end points going out to a common unvisited vertex of the graph. Even if there are at least two different connectors from two end points going out to a common vertex, there is a circuit. Conversely, if  $k = 1$ , i.e. there is a single end point of the primary partition, no question of circuit formation arises.  $\square$

**Lemma 2** *A primary partition of size  $k > 1$ , does not necessarily mean that there is a circuit.*

*Proof* From Lemma 1, it is proved that circuits are formed for only  $k > 1$ , but the reverse is not always true. If there are  $k$  end points of the primary partition and if two or more connectors come out of the same end point but goes to different unvisited vertices of the graph, no circuit is formed. But, if at least two such connectors coming out of different end points go to the same unvisited vertex, a circuit is formed. Hence, Lemma 2 is proved.  $\square$

**Corollary 1** *Joining a primary partition of size  $k > 1$  with a vertex component always yields circuit, while joining it with an edge component may or may not yield circuit.*

*Proof* From Lemmas 1 and 2, we have seen that primary partitions of size  $k > 1$  may or may not generate circuits. If the primary partition is being joined with a vertex component, connector combinations from two or more (up to  $k$ ) end points always go to the vertex, forming a circuit. On the other hand, if the primary partition is being joined with an edge component, then the connectors from the different endpoints may go to the same end-vertex of the edge, forming circuits or to different end vertices of the edge which does not form a circuit.  $\square$

Let us explain the Phase 2 tree generation for the graph  $G_1$  of Fig. 3b.

- For partition  $\{e_1\}$ , only one connector is there in CS. Main connector combination gives  $(e_5, e_6)$ . Only  $v_4$  is left unvisited, which can be joined by both  $e_7$  and  $e_8$ . Thus, trees formed here are:  $(e_1, e_5, e_6, e_7)$  and  $(e_1, e_5, e_6, e_8)$ . Similarly, for partition  $\{e_2\}$ , trees formed are:  $(e_2, e_5, e_7, e_6)$  and  $(e_2, e_5, e_7, e_8)$ , and for partition  $\{e_3\}$ , trees are:  $(e_3, e_7, e_8, e_5)$  and  $(e_3, e_7, e_8, e_6)$ .
- Next, for  $\{e_1, e_2\}$ , no CS is there, as there is a single secondary partition. Main connector combination gives  $(e_6, e_7)$ . Hence, tree formed:  $(e_1, e_2, e_6, e_7)$ . Similarly, for  $\{e_2, e_3\}$ , the tree formed is:  $(e_2, e_3, e_5, e_8)$ . On the contrary, for partition  $\{e_1, e_3\}$ , there are two MC's,  $MC_1$  joining  $v_3$  with  $\{e_1, e_3\}$  and  $MC_2$

joining  $v_5$  with  $\{e_1, e_3\}$ . Since, both  $v_3$  and  $v_5$  are vertex-components, so combination of connectors in  $MC_1$  as well as  $MC_2$  generates circuit, which are thus not considered.

- Lastly, for  $\{e_1, e_2, e_3\}$ , only the vertex  $v_5$  remains unvisited, which is once again a vertex component. Hence, the connectors in  $MC$ , namely,  $e_6$  and  $e_8$ , are not combined, else circuit is formed.

#### 4.5 Rejoining the Pendant Edges

In the last section, we rejoin the pendant edges from  $v_{\text{ref}}$  to all the trees obtained in the previous sections, to get the spanning trees of the original graph. After generating the trees for the graph  $G_1$  (in Fig. 3b) in the earlier sections, we join the pendant edge ( $e_4$ ) from  $v_{\text{ref}}(v_1)$  with them. These give us the following trees for the original graph  $G$  (Fig. 3a):  $(e_1, e_7, e_8, e_5, e_4), (e_1, e_7, e_8, e_6, e_4), (e_2, e_6, e_8, e_5, e_4), (e_2, e_6, e_8, e_6, e_4), (e_3, e_5, e_6, e_7, e_4), (e_3, e_5, e_6, e_8, e_4), (e_1, e_2, e_8, e_6, e_4), (e_1, e_2, e_8, e_7, e_4), (e_1, e_3, e_5, e_6, e_4), (e_1, e_3, e_5, e_8, e_4), (e_1, e_3, e_7, e_6, e_4), (e_1, e_3, e_7, e_8, e_4), (e_2, e_3, e_6, e_5, e_4), (e_2, e_3, e_6, e_8, e_4), (e_1, e_2, e_3, e_6, e_4), (e_1, e_2, e_3, e_8, e_4), (e_1, e_5, e_6, e_7, e_4), (e_1, e_5, e_6, e_8, e_4), (e_2, e_5, e_7, e_6, e_4), (e_2, e_5, e_7, e_8, e_4), (e_3, e_7, e_8, e_5, e_4), (e_3, e_7, e_8, e_6, e_4), (e_1, e_2, e_6, e_7, e_4), and  $(e_2, e_3, e_5, e_8, e_4)$ .$

#### 4.6 Algorithm DCC\_Trees

1. **Begin**
2. Take an input graph  $G = (V, E)$ .
3. Take any vertex from  $V$  as  $v_{\text{ref}}$ .
4. Remove pendant edges from  $v_{\text{ref}}$ , if any, to form graph  $G_1$ .
5. Take all possible combinations of all edges from  $v_{\text{ref}}$ , including one to all edges at a time, forming primary partitions (say,  $P_1, P_2, \dots, P_k$ ).
6.  $\forall P_i, 1 \leq i \leq k$  **begin**
7. Form secondary partitions (say,  $S_1, S_2, \dots, S_w$ ) with either unvisited vertices or edges joining unvisited vertices.
8. For  $j = w$  to 1 **begin**
9. Consider a pair of secondary partitions  $(S_j, S_{j-1})$ .
10. Find out the set of connector(s) (if exist(s)),  $CS$ , joining  $(S_j, S_{j-1})$ .
11. If  $CS = \phi$ , go to Step 15.
12. For each such  $CS$  **begin**
13. Join it with the pair  $(S_j, S_{j-1})$  forming subgraph  $SB$ .
14.  $S_j \leftarrow$  Subgraphs formed in Step 10.
15. **End**
16.  $j \leftarrow j - 1$ .

17. **End**
18. Unjoined secondary partitions (if any) joined to  $P_i$  to form sequences in the tree set  $SP$  of Phase 1.
19. Take combinations of two or more connectors from each connector set,  $CS$  or  $MC$ , along with primary partition, and then form secondary partitions, if required, to generate trees of Phase 2.
20. **End**
21. Add pendant edges of  $v_{\text{ref}}$  to trees generated in Steps 7 through 9 to generate trees of  $G$ .
22. **End**

**Theorem 2** *DCC\_Trees generates all possible spanning trees of a simple, connected graph.*

*Proof* *DCC\_Trees* generates primary partitions from a reference vertex,  $v_{\text{ref}}$ . Primary partitions consider all combinations of edges incident on  $v_{\text{ref}}$ , ensuring inclusion of  $v_{\text{ref}}$  in all the sequences generated as well as considering all the sequences exhaustively with a different set of edges from  $v_{\text{ref}}$ . For each such primary partition, all the edges encompassing adjacent unvisited vertices as well as left-out unvisited vertices, named as secondary partitions, are being considered. Moreover, all the connectors joining the different secondary partitions along with the corresponding primary partition is also considered in this algorithm. If there is no connector between a pair of secondary partitions, then the next secondary partition, in sequence, is considered. Even if there is a secondary partition which has no connection with any other secondary partition being formed, it is directly joined with the primary partition with a set of main connectors. The algorithm thus carries out exhaustive divide and conquer phases, where all components and all possible ways of joining them are considered, ensuring generation of all possible spanning trees of the given graph.

## 4.7 Data Structures and Complexity Issues

The algorithm stores the given graph,  $G$ , in an adjacency matrix, whose storage requirement is  $O(n + m)$ , where  $n$  is the number of vertices and  $m$  is the number of edges of the graph. The primary partitions are being stored in a linked list, one at a time, and hence requires  $O(n)$  space only. Searching for  $v_{\text{ref}}$  in the given graph and removal of its pendant edges can take at most  $O(n)$  time. In the division phase, for a particular primary partition, forming the other partitions can take at most  $O(n)$  time. There can be at most  $\lceil n/2 \rceil$  components having maximum two end points. Thus, searching for the connectors takes at most  $O(n)$  time. Combining such connectors and partitions also require  $O(n)$  time. Now, there can be at most  $O(2^n)$  primary

partitions, when the maximum degree of a vertex is  $n - 1$ , i.e. it is a complete graph. The maximum time taken to generate trees for any one of them is  $O(n)$ . Hence, the worst case time complexity of the algorithm comes out to be  $O(n.2^n)$ .

## 4.8 Salient Features of *DCC\_Trees*

In this section, we like to highlight on some of the key features of the algorithm as follows.

- The algorithm is based on a unique approach never attempted before.
- It guarantees no duplicate tree generation.
- It can generate the spanning trees with a particular edge(s) included, i.e., if we want to find out those spanning trees of a given graph with a particular edge or a group of edges from a particular vertex being always included, then our algorithm can do so efficiently.
- The consistency of the proposed algorithm remains valid irrespective of the selection of starting vertex.
- It is also suitable for parallel processing. Once the primary partitions are ready, trees generated from any one partition are independent of those from another partition, and hence, can be processed in parallel.

## 5 Experimental Results

Even if the computation time required for computing all spanning trees is exponential, we execute the algorithm devised in this paper, *DCC\_Trees*, for graphs whose order as well as size is bounded by some constant, and we can generate all trees for each of such assumed instances in feasible/reasonable amount of time (and space).

In this section, we have incorporated the implementation results of our algorithm for random graph instances having  $|V| = 10$  to  $22$ ,  $|V|$  being the total number of vertices of the graph instances. The implementation has been carried out on an Intel Core i3 quad-core processor running at 2.4 GHz, with 6 GB RAM capacity. A few standard algorithms for generating all possible spanning trees, given by Shioura and Tamura [18], Matsui [15], Mayeda and Seshu [17], Hakimi [13], Char [3], and Winter [22] have also been implemented in the same environment and on same graph instances. Table 1 gives the order and size of the graph instances considered, number of trees generated, and the CPU time taken by all the algorithms to run each instance of the graph.

In Table 1,  $I_i(x, y)$  is the  $i$ -th instance of a graph of order  $x$  (i.e. number of vertices) and size  $y$  (i.e. number of edges). The time taken by each of the algorithms is shown in an *mm-ss* format, where *mm* and *ss* stand for minutes and seconds required to execute the algorithms on the specific instances, respectively. A few observations from Table 1:

**Table 1** Experimental results of computing all spanning trees for random graph instances, having order from 10 to 22

Instances with vertex# and edge#	Number of trees generated	Shioura and Tamura [18]	Matsui [15]	Mayeda and Seshu [17]	Hakimi [13]	Char [3]	Winter [22]	<i>DCC_Trees</i>
		(mm-ss)	(mm-ss)	(mm-ss)	(mm-ss)	(mm-ss)	(mm-ss)	(mm-ss)
$I_1$ (10, 18)	6210	00-00	00-00	00-00	00-00	00-01	00-00	00-00
$I_2$ (10, 15)	636	00-00	00-00	00-00	00-00	00-00	00-00	00-00
$I_3$ (10, 14)	364	00-00	00-00	00-00	00-00	00-00	00-00	00-00
$I_1$ (15, 23)	6054	00-00	00-00	00-00	00-00	00-01	00-00	00-00
$I_2$ (15, 21)	1320	00-00	00-00	00-00	00-00	00-00	00-00	00-00
$I_3$ (15, 21)	2858	00-00	00-00	00-01	00-00	00-00	00-00	00-00
$I_1$ (20, 35)	13100220	09-35	02-20	25-30	08-24	09-41	00-38	00-43
$I_2$ (20, 28)	32854	00-03	00-01	00-05	00-01	00-05	00-00	00-01
$I_3$ (20, 31)	248120	00-17	00-03	00-33	00-10	05-43	00-02	00-05
$I_1$ (22, 32)	616642	01-17	00-13	02-28	01-14	12-04	00-07	00-23

- In the above table, we have shown three instances for each of 10, 15 and 20 vertices and one instance of 22 vertex graph.
- The density of the graphs (actual number of edges in the graph/maximum number of edges possible with given number of vertices) ranges from 0.1 to 0.4 approximately.
- A number of trees generated depend not only on the order and size of the graph but also on the arrangement of edges in the graph.
- Time taken to execute the algorithms depends not only on order and size of the graph but also on the number of trees generated.
- It is observed that the algorithm by Winter [22] gives best results with respect to CPU time compared to all other algorithms considered here. Winter proposes to generate several different tree sequences simultaneously.
- The last column in the table shows the execution time of our algorithm, *DCC\_Trees*, which is found to be better than most of the others. In our algorithm, Phase 1 generates several tree sequences simultaneously, thus consuming much less time than that of Phase 2.

## 6 Applications

Many problems in various fields of science and engineering need to be formulated regarding graphs. Many of them involve various applications of spanning trees like computation of minimum spanning tree or generation of all possible spanning trees from a given graph.

Some of the application areas of spanning trees are as follows:

- *Network Design*: Designing different networks such as phone, electrical, hydraulic, TV cable, computer, road, air traffic, railway, electronic circuits, etc.
- *Approximation Algorithms for NP-hard Problems*: Solving problems like the travelling salesman problem (TSP) having several applications in planning, logistics, and the manufacture of microchips, as well as in DNA sequencing.
- *Cluster Analysis*: Required for routing in mobile ad-hoc networks, identifying patterns in gene expression, document characterization for web search, medical image processing.
- *Image Processing*: For extraction of networks of narrow curvilinear features such as road and river networks from remotely sensed images.
- *Astronomy and Space Sciences*: To compare the aggregation of bright galaxies with faint ones.
- *Biology*: To carry out research in the quantitative description of cell structures in light microscopic images.
- *Molecular Biology*: Frequently used in molecular epidemiology research to estimate relationships among individual strains or isolates.
- *Chemistry*: Used in chemical research for determination of the geometry and dynamics of compact polymers.
- *Archaeology*: For identifying proximity analysis.
- *Bioinformatics*: For micro-array expression of data.
- *Geography*: Used in efficient methods for regionalization of socio-economic units in maps.
- *Binary Spanning Trees*: It is a rooted structure with a parent having 0, 1, or 2 children. Binary trees find applications in language parsing/representation of mathematical and logical expressions, finding duplicates in a given list of numbers. Binary search trees can be used for sorting a list of given numbers.
- *Depth-First Spanning Tree*: An edge  $(v, w)$  that leads to the discovery of an unvisited vertex during a depth-first search is referred to as a tree edge of a graph  $G$ . Collectively the tree edges of  $G$  form a depth-first spanning tree of  $G$ . DFS tree can be used to obtain a topological sorting, to find out the connectedness on a graph and to compute a spanning forest of graph, a cycle in graph and also bi-connected component, if any.
- *Breadth-First Spanning Tree*: BFS algorithm is applied to determine if a graph is bipartite, testing whether a graph is connected, computing a spanning forest of a graph, computing a cycle in a graph or reporting that no such cycle exists, etc. Also used to find the diameter of a tree having applications in network communication systems.
- *Spanning Tree Protocol*: A link layer network protocol that ensures a loop-free topology for any bridged LAN. Also allows a network design to include spare (redundant) links to provide automatic backup paths. *Multiple Spanning Tree Protocol*: Used to develop the usefulness of Virtual LANs (VLANs) further.
- *Broadcast and Peer-to-peer Networks*: Broadcast operation is fundamental to distributive computing. *Flooding Algorithms* construct a spanning tree which is



used for convergecast. Convergecast is collecting information upwards from the spanning tree after a broadcast. A broadcast spanning tree can be built such that each non-leaf peer forwards broadcast messages to its children along the tree.

- *Link Failure Simulation in Network*: One of the most important tools in network capacity planning is the ability to simulate the physical link failure scenario, where the planner would like to cut-off a physical link logically and then try to see how the network behaves due to that failure. During this process the planners would also try to identify all possible alternate routes between the nodes where they have logically torn down the direct physical link, so that they can distribute the entire traffic from that link to the existing path and see whether those links have enough capacity to handle additional load that has been generated due to the physical link failure simulation. Since the above-described process needs to be repeated for all the nodes in the network and hence the necessity of finding all possible spanning trees for the network topology is evident.

## 7 Conclusion

We have given here a new approach towards solving the well-known tree generation problem using the divide-and-conquer technique used in algorithms. The algorithm formulated in this paper is capable of computing all possible spanning trees of a simple, symmetric, and connected graph. The given graph has been divided into a number of partitions, which can be joined by a set of connectors. The selection of different connectors and the way they are being combined with the different partitions give rise to different spanning trees of the graph. Our algorithm does not generate any duplicate tree and also minimizes the formation of the circuit in its tree generation procedure, which has also been taken care of eventually. Till now, we have executed the algorithm on graph instances whose order is at most 22. Currently, we are working on larger graph instances compared to the ones considered in this paper to augment the results published in Table 1. Due to the limitation in a computing environment, some of the algorithms considered in this paper are taking time in weeks for execution.

## References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Divide-and-Conquer: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge, Massachusetts (2009)
2. Berger, I.: The Enumeration of Trees without Duplication. *IEEE Trans. Circuit Theor.* **14**(4), 417–418 (1967)
3. Char, J.P.: Generation of trees, two-trees, and storage of master forests. *IEEE Trans. Circuit Theor.* **15**(3), 228–238 (1968)

4. Gabow, H.N., Myers, E.W.: Finding all spanning trees of directed and undirected graphs. *SIAM J. Comput.* **7**(3), 280–287 (1978)
5. McElroy, M.D.: Algorithm 354: generator of spanning trees [H]. *Commun. ACM* **12**(9), 511 (1969)
6. Naskar, S., Basuli, K., Sen Sarma, S.: Generation of all spanning trees of a simple, symmetric, connected graph. In: National Seminar on Optimization Technique, Department of Applied Mathematics, University of Calcutta, p. 27 (2007)
7. Naskar, S., Basuli, K., Sen Sarma, S.: Generation of All Spanning Trees. Social Science Research Network (2009)
8. Naskar, S., Basuli, K., Sen Sarma, S.: Generation of all spanning trees in the limelight. In: Advances in Computer Science and Information Technology, Second International Conference, CCSIT 2012, Bangalore, vol. 86, pp. 188–192. Proceedings Part III published by Springer (2012)
9. Piekarski, M.: Listing of all possible trees of a linear graph. *IEEE Trans. Circuit Theor.* **12**(1), 124–125 (1965)
10. Sen Sarma, S., Rakshit, A., Sen, R.K., Choudhury, A.K.: An efficient tree generation algorithm. *J. Inst. Electron. Telecommun. Eng.* **27**(3), 105–109 (1981)
11. Trent, H.M.: Note on the enumeration and listing of all possible trees in a connected linear graph. In: Proceedings of the National Academy of Sciences. USA.40, pp. 1004 (1954)
12. Cherkasskii, B.V.: New algorithm for generation of spanning trees. *Cybern. Syst. Anal.* **23**(1), 107–113 (1987)
13. Hakimi, S.L.: On trees of a graph and their generation. *J. Franklin Inst.* **272**(5), 347–359 (1961)
14. Kapoor, S., Ramesh, H.: Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM J. Comput.* **24**(2), 247–265 (1995)
15. Matsui, T.: An algorithm for finding all the spanning trees in undirected graphs. In: METR93-08, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo. 16, pp. 237–252 (1993)
16. Matsui, T.: A flexible algorithm for generating all the spanning trees in undirected graphs. *Algorithmica* **18**(4), 530–543 (1997)
17. Mayeda, W., Seshu, S.: Generation of trees without duplications. *IEEE Trans. Circuit Theor.* **12**(2), 181–185 (1965)
18. Shioura, A., Tamura, A.: Efficiently scanning all spanning trees of an undirected graph. In: Research Report: B-270, Department of Information Sciences, Tokyo Institute of Technology, Tokyo. (1993)
19. Shioura, A., Tamura, A., Uno, T.: An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM J. Comput.* **26**(3), 678–692 (1997)
20. Minty, G.: A simple algorithm for listing all the trees of a graph. *IEEE Trans. Circuit Theor.* **12**(1), 120–120 (1965)
21. Smith, M.J.: Generating spanning trees. In: MS Thesis, Department of Computer Science, University of Victoria (1997)
22. Winter, P.: An algorithm for the enumeration of spanning trees. *BIT Numer. Math.* **26**(1), 44–62 (1986)

Advanced Computing and Systems for Security

Volume Three

Chaki, R.; Saeed, K.; Cortesi, A.; Chaki, N. (Eds.)

2017, XIII, 197 p. 57 illus., Softcover

ISBN: 978-981-10-3408-4