# Efficiently Enumerating All Spanning Trees of a Plane 3-Tree

Muhammad Nur Yanhaona[1[0000−0003−2450−3377]], Asswad Sarker Nomaan[2], and Md. Saidur Rahman[2[0000−0003−0112−0242]]

[1] Department of Computer Science and Engineering
BRAC University, Dhaka, Bangladesh
nur.yanhaona@bracu.ac.bd
[2] Graph Drawing and Information Visualization Laboratory
Department of Computer Science and Engineering
Bangladesh University of Engineering & Technology, Dhaka, Bangladesh
1021052002@grad.cse.buet.ac.bd, saidurrahman@cse.buet.ac.bd

**Abstract.** A spanning tree $T$ of a connected, undirected graph G is an acyclic subgraph having all vertices and a minimal number of edges of G connecting those vertices. Enumeration of all possible spanning trees of undirected graphs is a well-studied problem. Solutions exist for enumeration for both weighted and unweighted graphs. However, these solutions are either space or time efficient. In this paper, we give an algorithm for enumerating all spanning trees in a plane 3-tree that is optimal in both time and space. Our algorithm exploits the structure of a plane 3-tree for a conceptually simpler alternative to existing general-purpose algorithms and takes $\mathcal{O}(n+m+\tau)$ time and $\mathcal{O}(n)$ space, where the given graph has $n$ vertices, $m$ edges, and $\tau$ spanning trees. This is a substantial improvement in both time and space complexity compared to the best-known algorithms for general graphs. We also propose a parallel algorithm for enumerating spanning trees of a plane 3-tree that has $\mathcal{O}(n+m+\frac{n\tau}{p})$ time and $\mathcal{O}(\frac{n\tau}{p})$ space complexities for $p$ parallel processors. This alternative algorithm is useful when storing the spanning trees is important.

**Keywords:** Spanning Tree Enumeration · Plane 3-Tree · Time and Space Complexity

## 1 Introduction

A spanning tree of a connected undirected graph G is a subgraph that is a tree spanning all vertices of G. Spanning trees have always been a focus for researchers when dealing with various graph-related problems due to their wide range of applications in areas such as computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids.

The enumeration of all spanning trees of a graph $G$ is to find all possible distinct edge combinations which minimally connect all its vertices without introducing any cycle. Enumerating all possible spanning trees is often crucial in electrical circuits, routing networks, and other network analysis and optimization applications [3]. For example, the current flow of an electrical network is associated with the sum of electrical admittances in its spanning trees [13] [1]. The main issues with spanning tree enumeration are exponential time complexity, costly storage space requirements, and duplicate tree generation. Various algorithms with varying degrees of efficiency exist in the literature addressing these issues [2] [14] [10] [7] [6] [5] [9]. To the best of our knowledge, however, no solution exists that is optimal in all these aspects for any complex graph class.

Char's [2] 1968 algorithm is among the first works on spanning tree enumeration. For a $G$ with $n$ vertices and $m$ edges, it considers all possible combinations of $n-1$ edges as potential spanning trees and gives unique spanning trees as output. The algorithm assigns the vertices of $G$ increasing orders from 1 to $n$, where $n$ is the number of vertices. It next selects the $n^{th}$ vertex as the root of all spanning trees. Then the algorithm repeatedly emits sequences of $n-1$ numbers as a permutation of vertex orders, where the $i^{th}$ number in the sequence is the endpoint of an edge originating from the vertex with order $i$. Although the algorithm avoids duplicate tree generation, its major limitation is that it involves checking if a sequence forms a spanning tree or some other cyclic subgraph. If $\tau'$ and $\tau$ denote the numbers of non-tree and tree sequences of $G$, respectively, then the time complexity of the algorithm is $\mathcal{O}(m+n+\tau+\tau')$. The space required by Char's algorithm is $\mathcal{O}(nm)$. This space overhead is due to a tabular structure that lists all adjacent vertices of each vertex.

The algorithm Rakshit et al. proposed in 1981 [14] is quite similar to Char's algorithm. The privileged reduced incidence-edge structure (PRIES) that the algorithm proposes is effectively the exact structure of [2]. The difference between the two algorithms lies in how they detect non-tree edge sequences. While Char's algorithm repeatedly checks for cycle to prune non-tree edge sequences, Rakshit's algorithm does that by repeatedly removing two "pendant" vertices [4] from a sequence and checking the degrees of remaining vertices. The time and space complexities of this algorithm are similar to Char's algorithm. Both algorithms are superior in time complexity to their contemporary alternative tree enumeration algorithms [10] [7] [6] as they avoid duplicate edge sequence generation and prune some non-tree sequences using the geometric properties of the input graph.

Gabow and Myers [5] and Matsui [9] improve the time complexity of spanning tree enumeration from earlier algorithms by avoiding evaluating candidate

sequences of vertices that cannot form a connected subgraph. Both algorithms recursively generate spanning trees by growing them from smaller sub-trees and checking whether adding a new edge on a growing tree will produce a cycle. Gabow and Myers' algorithm traverses the input graph $G$ in depth-first-order from a single pivot vertex that the algorithm considers the root of all spanning trees of $G$. As it grows a tree from that root vertex, the algorithm removes and restores edges in $G$ in a deterministic order to avoid duplicate tree generation. The algorithm works for both directed and undirected graphs and takes $\mathcal{O}(n + m + m\tau)$ and $\mathcal{O}(n + m + n\tau)$ times for the former and the latter respectively. The vertex and edge count multiplier over the number of spanning trees $\tau$ in this algorithm is due to checking all possible extensions to a new edge or vertex from a single vertex of the tree under construction.

Matsui's algorithm employs that one can construct a spanning tree $T$ of a graph $G$ by swapping an edge $f$ of another spanning tree $T^/$ with an edge $g \notin T^/$ such that $T \bigcup \{g\}$ is a cycle (such a cycle is called a fundamental circuit of $G$). This property allows us to relate all spanning trees of $G$ in a parent-child relationship hierarchy from a single baseline spanning tree where a child spanning tree is the result of replacing a single edge of its parent with another edge, called the pivot edge. The algorithm traverses the hierarchical domain of spanning trees by finding pivot edges using a generic enumeration algorithm proposed by Nagamochi and Ibaraki [11]. The domain traversal order can be breadth-first or depth-first. However, careful edge ordering is essential to avoid returning to an ancestor-spanning tree from a descendent. Then pivot edges discovery from a single spanning tree can take $\mathcal{O}(n)$ time. That gives the total time complexity $\mathcal{O}(n+m+n\tau)$. Both Gabow and Myer [5] and Matsui's algorithm only maintains a single spanning tree and a sorted sequence of edges in memory. Therefore, their space complexity is $\mathcal{O}(n + m)$. One can view these algorithms as optimizations to Read and Tarjan's generic backtracking-based algorithm for listing specific types of subgraphs of a graph [15], which provides $\mathcal{O}(n + m + m\tau)$ time and $\mathcal{O}(n + m)$ space bounds for listing spanning trees.

Onete et al.'s [13] more recent algorithm of spanning tree enumeration achieves the same time and space bounds as Matsui's algorithm. However, the algorithm is quite different and involves no backtracking. Onete et al. found that all and only non-singular submatrices with a specific structure of a reduced incidence matrix of a graph represent spanning trees. Their algorithm utilizes this finding by generating unique submatrices of that kind and spewing corresponding spanning trees in the incidence matrix format. Since checking for non-singularity is a linear process in the number of vertices, the time complexity of their algorithm is also $\mathcal{O}(n + m + n\tau)$.

Kapoor and Ramesh [8] and Shioura and Tamura [16] provide two alternative algorithms to improve the time complexity from $\mathcal{O}(n+m+n\tau)$ to $\mathcal{O}(n+m+\tau)$ when the individual spanning trees need not be output. Instead, their difference from a baseline tree as a list of edge exchanges is sufficient. Both algorithms implicitly traverse the network $S(G)$ formed by all spanning trees of $G$ where the nodes of $S(G)$ are the spanning trees of $G$ and there is an edge between two nodes

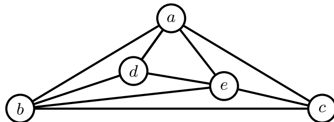



Fig. 1: A plane 3-tree of 5 vertices, $G_5$

of $S(G)$ if the corresponding trees differ by an edge exchange. Both algorithms maintain a dynamic, ordered list of edges associated with fundamental cycles of $G$ in a data structure to optimize the traversal of $S(G)$ and avoid duplicate tree generation. Although the underlying data structures are different in these two algorithms, their maintenance increases algorithms' space complexity to $\mathcal{O}(nm)$. Furthermore, recreating the individual spanning trees from the description of differences sacrifices time optimization.

All the above algorithms involve checking for cycle formation or tracking cyclic relations among edges of the graph, which is the chief contributor to their time or space complexity. Avoiding cycle tracking during spanning tree enumeration of a general graph is challenging. However, alternative approaches that utilize a graph's structure to enumerate spanning trees without cycle tracking can lead to better algorithms for specific graph classes.

In this paper, we consider the problem of enumerating spanning trees of a plane 3-tree. Plane 3-trees are a class of planner graphs formed from repeated triangulation of phases of an initial triangle. Formally, a *plane 3-tree* $G_n$ is a triangulated plane graph $G$ with $n \geq 3$ vertices such that if $n > 3$ then $G$ has a vertex $x$ whose deletion gives a plane 3-tree $G'$ of $n-1$ vertices. Figure 1 shows an example of plane 3-tree $G_5$ containing 5 vertices. Our approach uses the face decomposition structure of a plane 3-tree to order its vertices, then inductively generates all its spanning trees without duplication from that vertex order. The algorithm using breadth-first propagation of inductive expansion has $\mathcal{O}(n+m+\tau)$ time and $\mathcal{O}(n\tau)$ space complexities. When space is not an issue, this algorithm has the advantage of being easily parallelizable. Our second algorithm captures the logic of inductive expansion into a tree alteration scheme like Matsui's to output all spanning trees using dynamic programming (DP) in $\mathcal{O}(n)$ space and $\mathcal{O}(n+m+\tau)$ time.

The structure of the rest of the paper is as follows. Section 2 presents preliminaries on various terms and concepts related to graphs, plane 3-trees, and our algorithms. Section 3 introduces some lemmas and theorems related to properties of spanning trees of a plane 3-tree. Section 4 presents our first and inductive algorithm for spanning tree enumeration. Section 5 describes and analyzes the DP algorithm. The paper concludes with a discussion of future research directions.

## 2 Preliminaries

In this section, we define some terms we will use for the rest of the paper and present some preliminary findings.

Let $G = (V, E)$ is a connected simple graph with vertex set $V$ and edge set $E$. A *subgraph* of $G$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. $G'$ is a *spanning subgraph* of $G$ when $V' = V$. $G'$ is an *induced subgraph* of $G$ when $G'$ contains all edges of $E$ having both endpoints in $V'$. When each edge $(u, v) \in E$ is directional, i.e., $(u, v) \neq (v, u)$ then $G$ is a *directed graph*; otherwise, $G$ is *undirected*. In this paper, we only consider undirected graphs. The *degree* of a vertex $v \in G$, denoted by $degree(v)$, is the number of edges incident to it. The *neighbors* of $v$ are the vertices that share an edge with $v$. That is, if there is an edge $(u, v) \in E$ then $u$ and $v$ are neighbors. We also say $u$ and $v$ are *adjacent* in $G$.

A *tree* $T = (V, E)$ is a connected graph with no cycles. $T$ is *rooted* when one vertex $r \in V$ is distinguished, called the *root*, from others. A *spanning tree* of a graph $G$ is a spanning subgraph that is a tree. We use the term *node* and vertex interchangeably to refer to a vertex of a spanning tree. The unique *path* between two nodes $u, w$ in $T$ is the sequence of edges of the form $(u, v_1), (v_1, v_2), \cdots, (v_i, w)$ that connects $u$ and $w$ in the tree. We use the notations $\rho_{u,v_1,\cdots,w}$ or $\rho_{u,w}$ interchangeably to denote the path between $u$ and $w$. The *length* of a path is the number of edges in it. Let $T$ be a rooted tree with root node $r$. The *parent* $u$ of any node $v$ other than $r$ in $T$ is the immediate predecessor of $v$ on the path from $r$ to $v$. Conversely, $v$ is a *child* of $u$. A *leaf* of $T$ is a node with no children; otherwise, it is an *internal node*. A *sub-tree* $T'$ of $T$ is an induced proper subgraph of $T$. A node $w$ is an *ancestor* of another node $v$ in $T$ when the unique path from $v$ to the root $r$ contains $w$; conversely, $v$ is a *descendent* of $w$. A *fundamental cycle* of a graph $G$ is a cycle formed by adding a non-tree edge $(u, v) \in G$ on its spanning tree

A graph $G$ is *planar* if one can embed it on a plane without any edge crossing. We call such an embedding a plane embedding of $G$. A *plane* graph is a planar graph with a fixed embedding, where the connected regions of a plane embedding are its *faces*. The unbounded region is the *outer face*, and all others are the *inner faces*. The vertices of the outer face are the *outer vertices*, while the rest are *inner vertices*. A plane graph $G$ is a *triangulated plane graph* when every face has precisely three vertices.

A *plane 3-tree* is a triangulated plane graph whose plane embedding is a repeated triangulation of the faces starting from a single outer triangle. Therefore, a *plane 3-tree* $G_n$ is a triangulated plane graph $G$ with $n \geq 3$ vertices such that if $n > 3$ then $G$ has a vertex $x$ whose deletion gives a plane 3-tree $G_{n-1}$ of $n-1$ vertices. We call $x$ the last, or $(n-3)^{th}$, *dividing vertex* of $G_n$. Note that $degree(x) = 3$. We call the three neighbors $u, v, w$ of $x$, the *face vertices* of $x$ in $G_n$ and denotes their set by $\zeta_x$. If $n > 4$ then the last dividing vertex $y$ of $G_{n-1}$ is the $(n-4)^{th}$ dividing vertex of $G_n$. Here $n-3$ and $n-4$ are the *dividing orders* of $x$ and $y$ respectively. Our notion of a dividing vertex is similar to that of a *representative vertex* described in [12].

**Fact 1.** *One can describe the repeated triangulation-based embedding of a plane 3-tree $G_n$ of $n > 3$ vertices unambiguously by an ascending sequence of dividing vertices $v_1, v_2, \cdots, v_{n-3}$ and their face vertex sets $\zeta_{v_1}, \zeta_{v_2}, \cdots., \zeta_{v_{n-3}}$.*

Note that multiple sequences of dividing vertices exist for a plane 3-tree $G_n$ when it has multiple degree-3 vertices. However, all such sequences describe the same plane embedding. For a dividing vertex sequence $d_s = v_1, v_2, \cdots, v_{n-3}$ of a plane 3-tree $G_n$ with $n > 3$, we use $G_1^{im}, G_2^{im}, \cdots, G_{n-3}^{im}$ to represent the sequence of intermediate plane-3 trees we get by introducing the dividing vertices in corresponding order starting from $G_3$ (note that $G_{n-3}^{im} = G_n$). We use the notation $\delta(v, d_s)$ to represent the dividing order of vertex $v$ in $d_s$.

**Fact 2.** *If $w$ is the last dividing vertex among three vertices $u, v, w$ appearing in any dividing vertex sequence of a plane 3-tree $G_n$ of $n \geq 3$ and $u$ and $v$ are adjacent to $w$ then $u$ and $v$ are also mutually adjacent.*

*Proof.* Assume that the dividing order of $w$ is $i$ for $i \leq n-3$. Since repeated triangulation of faces forms $G_n$, the $i^{th}$ dividing vertex $w$ can only have its face vertices $\zeta_w$ as neighbors when we introduce it in $G_i^{im}$. As $u$ and $v$ preceded $w$ in the dividing vertex sequence (or did not appear in the sequence due to being an outer face vertex), $u, v \in G_i^{im}$, which implies $u, v \in \zeta_w$. That is, $u$ and $v$ were part of the same triangular face in an intermediate graph of $G_n$. Therefore, $u$ and $v$ are adjacent. □

We next define some operations on trees that are essential for describing our algorithms. *Edge extension* is the operation of adding a new leaf node to a tree. The reverse operation of edge extension is *edge removal* which delete a leaf and its incident edge. The operation of adding a new vertex of degree 2 in the edge between two tree nodes is *edge subdivision*, and the reverse operation is *path contraction* which removes a node of degree two and connects its two neighbors by an edge. Finally, replacing a path of length two in a tree with a new node connected to all three nodes of the path is *root extension*, and the reverse operation is *root flattening*. Figure 2 illustrates these operations.

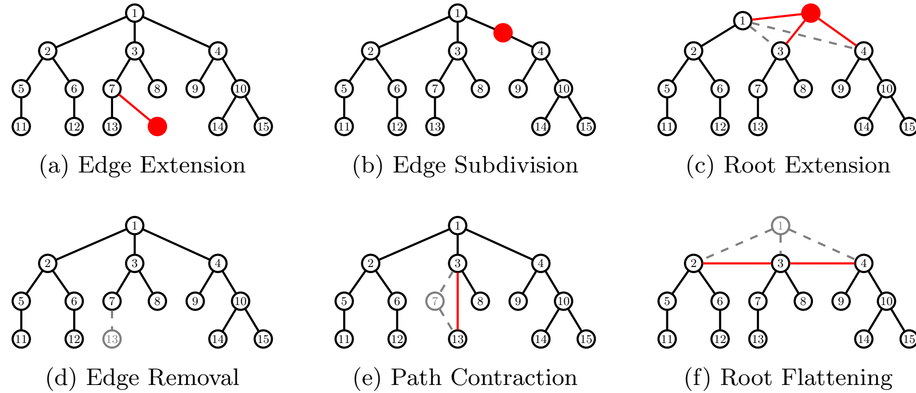



Fig. 2: Tree operations

## 3 On Spanning Trees of a Plane 3-Tree

In this section, we prove some properties of the spanning trees of a plane 3-tree. In subsequent sections, we will use these properties to prove the correctness of our tree enumeration algorithms.

**Lemma 1.** *Any spanning tree $T_n$ of a plane 3-tree $G_n$ with $n > 3$ results from consecutive applications of edge extension, edge subdivision, and root extension starting from a path of length two connecting the outer vertices of $G_n$.*

*Proof.* We prove the property by backtracking from $T_n$ to a path of length two, $T_0$, connecting the outer vertices of $G_n$ by applying the reverse operations of edge extension, edge subdivision, and root extension. Being able to do so implies that we can apply these three operations in the exact reverse order to construct $T_n$ starting from $T_0$.

Assume $d_s = v_1, v_2, \cdots, v_{n-3}$ and $f_s = \zeta_{v_1}, \zeta_{v_2}, \cdots., \zeta_{v_{n-3}}$ are a dividing vertex sequence and corresponding face vertex set sequence describing the embedding of $G_n$. Since $v_{n-3}$ is the last dividing vertex in $d_s$, $degree(v_{n-3}) = 3$ and the node in $T_n$ correspond to $v_{n-3}$ can be adjacent to only one, two, or three other nodes from $\zeta_{v_{n-3}}$.

In case node $v_{n-3}$ is adjacent to only a single node $x \in \zeta_{v_{n-3}}$ in $T_n$. Then $v_{n-3}$ is a leaf. Therefore, we can apply an edge removal on $T_n$ to remove $v_{n-3}$ and the associated edge $(x, v_{n-3})$ to get a spanning tree $T_{n-1}$ for $G^{im}_{n-4}$.

Alternatively, assume node $v_{n-3}$ is adjacent to exactly two nodes $x, y \in \zeta_{v_{n-3}}$ in $T_n$. If we apply path contraction to remove edges $(x, v_{n-3}), (v_{n-3}, y)$ and add $(x, y)$. Then the new tree $T'_{n-1}$ after this modification is also a spanning tree of $G^{im}_{n-4}$. This is true because vertices $x, y$ are adjacent in both $G_n$ and $G^{im}_{n-4}$ (refer to Fact 2).

Finally, if node $v_{n-3}$ is adjacent to all $x, y, z \in \zeta_{v_{n-3}}$ in $T_n$, we apply root flattening to remove $v_{n-3}$ and its incident edges and connect $x, y, z$ using two new edges $(x, y), (y, z)$. This change again produces a spanning tree $T''_{n-1}$ for $G^{im}_{n-4}$ as $x, y, z$ are adjacent in $G^{im}_{n-4}$.

Suppose $T_{n-1}$ is the tree we get after removing the node for $v_{n-3}$ and its incident edges from $T_n$. As $T_{n-1}$ is a spanning tree of another plane 3-tree $G^{im}_{n-4}$ with $n - 1$ vertices, we can repeat one of the three tree contraction operations appropriate for it to remove the node corresponding to $v_{n-4}$ to get a spanning tree of $G^{im}_{n-5}$. If we continue this process, eventually, we will remove all dividing vertices and be left with $T_0$. □

A corollary of Lemma 1 is that:

**Corollary 1.** *Any spanning tree $T_n$ of a plane 3-tree $G_n$ with $n > 3$ is an edge extension, edge subdivision, or root extension of a spanning tree of another plane 3-tree $G_{n-1}$.*

Lemma 1 and its corollary provide a convenient way to generate spanning trees of larger plane 3-trees from that of smaller plane 3-trees. However, we need to know when edge extension, edge subdivision, and root extension create

duplicate trees and when they do not. The following lemmas guide us in that regard. For the sake of notational brevity, we now use $G_{n-1}$ to represent $G^{im}_{v-4}$ of the larger plane 3-tree $G_n$. That is, $G_{n-1}$ is the plane 3-tree that results from removing the last dividing vertex of $G_n$ and its incident edges.

**Lemma 2.** *If two spanning trees $T^1_{n-1}$ and $T^2_{n-1}$ of $G_{n-1}$ are distinct, then an edge extension or edge subdivision to them to add a node for the last dividing vertex $v$ of $G_n$ will always produce two distinct spanning trees $T^1_n$ and $T^2_n$ of $G_n$.*

Lemma 2 is trivially true due to the precondition of distinctness of the spanning trees of $G_{n-1}$. We now have lemma 3.

**Lemma 3.** *If $\zeta_v = \{x, y, z\}$ is the face vertex set of the last dividing vertex $v$ of $G_n$ and $T^1_n, T^2_n$ are two spanning trees of $G_n$ constructed through a root extension from two spanning trees $T^1_{n-1}$ and $T^2_{n-1}$ of $G_{n-1}$ then $T^1_n$ and $T^2_n$ are distinct if and only if at least one of the three sub-trees rooted under nodes $x, y, z$ are different in $T^1_{n-1}$ and $T^2_{n-1}$.*

*Proof.* Note that root extension is possible only when $x, y, z$ form a path $\rho_{x,y,z}$, $\rho_{x,z,y}$ or $\rho_{y,x,z}$ in $T^1_{n-1}, T^2_{n-1}$. The three sub-trees rooted under $x, y, z$ exclude that path. Assume the sub-trees are $T^1_{n-1}(x)$, $T^1_{n-1}(y)$, $T^1_{n-1}(z)$ in $T^1_{n-1}$ and $T^2_{n-1}(x), T^2_{n-1}(y), T^2_{n-1}(z)$ in $T^2_{n-1}$. If any of these sub-trees are pairwise different (i.e., $T^1_{n-1}(i) \neq T^2_{n-1}(i)\ \exists i \in \zeta_v$), the root extension operation will retain the difference in $T^1_n$ and $T^2_n$ as it does not exchange or change sub-trees. On the other hand, if they are pairwise the same sub-trees (i.e., $T^1_{n-1}(i) = T^2_{n-1}(i)\ \forall i \in \zeta_v$), still $T^1_{n-1}$ and $T^1_{n-2}$ can be different due to the difference in the path connecting $x, y, z$. However, root extension removes the path among the three face vertices and connects them via edges to the node for the dividing vertex. Consequently, $T^1_n, T^2_n$ cannot be distinct without some difference in the sub-trees rooted under $x, y, z$ in $T^1_{n-1}$ and $T^2_{n-1}$. (Figure 3 illustrates this case with an example.) □

Lemma 2 and 3 give us insights on how to inductively generate spanning trees of a plane 3-tree without duplication from spanning trees of smaller graphs using a single sequence of dividing vertices. However, edge extension is the only universal operation we can apply to all smaller sub-trees. Edge subdivision or root extension is only possible when the structure of the smaller sub-tree permits it. Note that we worry about this issue despite Lemma 1 showing that any spanning tree of a plane 3-tree is a result of consecutive applications of these three operations because there can be multiple sequences of dividing vertices for a single plane 3-tree. Therefore, using one sequence may allow edge subdivision or root extension at a specific case, while another does not.

Assume $\tau_{n-1} = \{T^1_{n-1}, T^2_{n-1}, \cdots, T^N_{n-1}\}$ is the set of all spanning trees of $G_{n-1}$. Consider any arbitrary $T^i_{n-1}$ that has no pair of nodes from the face vertex set $\zeta_v = \{x, y, z\}$ of the last dividing vertex $v$ of $G_n$ connected by an edge in the tree. We can construct a spanning tree $T_n$ for $G_n$ from $T^i_{n-1}$ where node $v$ is adjacent to two nodes corresponding to its face vertices using the following steps:

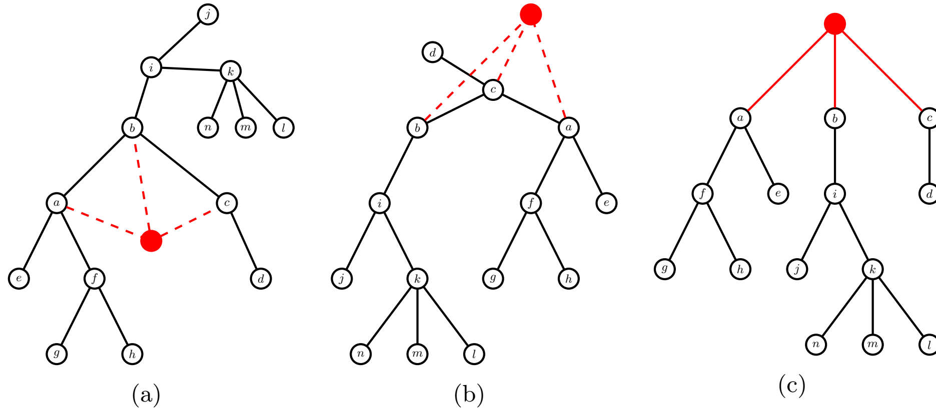


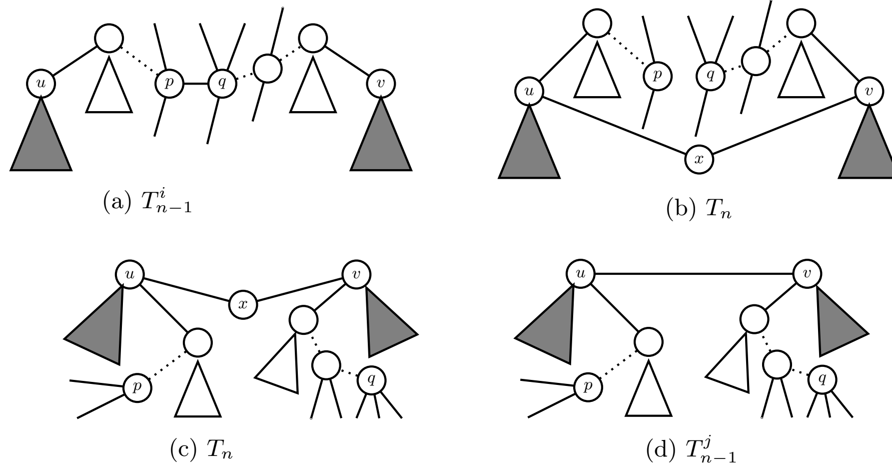

Fig. 3: An illustration of a root extension generating the same spanning tree (c) from two different smaller spanning trees (a) and (b) when the sub-trees rooted under the involved nodes are identical.

1. First, add $v$ by an edge extension.
2. Then connect leaf $v$ with another face vertex node by an edge to form a fundamental cycle.
3. Then remove any one edge from that cycle other than the two edges incident to $v$.

To produce a $T_n$ where $v$ shares an edge with all of $x, y, z$, we can repeat the above process for the remaining face vertex. One can convince thyself that this alternative process generates all spanning trees [9] [16]. The vital concern here is that we can generate spanning trees describable using edge subdivisions and root extensions that we did not generate using those operations in the first place. Let us call this alternative approach of spanning tree generation *cycle breaking*. The following two lemmas prove that a spanning tree generated using cycle breaking is always a replica of another tree inductively generated using edge subdivision or root extension following a single dividing vertex sequence.

**Lemma 4.** *Assume* $\tau_{n-1} = \{T^1_{n-1}, T^2_{n-1}, \cdots, T^N_{n-1}\}$ *is the set of all spanning trees of* $G_{n-1}$ *and* $T^i_{n-1} \in \tau_{n-1}$ *has no edge* $(u, v)$ *such that* $u, v$ *are two face vertices of the last dividing vertex* $x$ *of* $G_n$*. If* $T_n$ *having* $x$ *adjacent to both* $u$ *and* $v$ *is a spanning tree generated from* $T^i_{n-1}$ *using cycle breaking then there is another tree* $T^j_{n-1} \in \tau_{n-1}$ *that we can convert to* $T_n$ *applying an edge subdivision.*

*Proof.* Since $u$ and $v$ are not adjacent in $T^i_{n-1}$, there must be a path $\rho_{u,\cdots,v}$ of length greater than two in between them. Assume $(p, q)$ is the edge we remove to break the cycle that forms when we add edges $(u, x)$ and $(x, v)$ to $T^i_{n-1}$. Figure 4(a) and 4(b) illustrate this conversion. Given $u$ and $v$ are adjacent in $G_{n-1}$, we can subsequently do a path contraction to form another spanning tree $T_{n-1}$ of $G_{n-1}$ which has the edge $(u, v)$. Figure 4(c) and 4(d) illustrate this




(a) $T_{n-1}^i$ (b) $T_n$ (c) $T_n$ (d) $T_{n-1}^j$

Fig. 4: Completeness of edge subdivision

conversion from $T_n^i$ to $T_{n-1}$. Since $\tau_{n-1}$ contains all spanning trees of $G_{n-1}$, this new $T_{n-1} = T_{n-1}^j$ for some $1 \leq j \leq N$ and $j \neq i$. We would get the same $T_n$ by applying an edge subdivision on $T_{n-1}^j$. □

**Lemma 5.** *Assume $\tau_{n-1} = \{T_{n-1}^1, T_{n-1}^2, \cdots, T_{n-1}^N\}$ is the set of spanning trees of $G_{n-1}$ and $T_{n-1}^i \in \tau_{n-1}$ does not have a pair of edges $(x, y), (y, z)$ where $x, y, z$ are the face vertices of the last dividing vertex $v$ of $G_n$. If $T_n$ having $(v, x)$, $(v, y)$ and $(v, z)$ edges is a spanning tree generated from $T_{n-1}^i$ using cycle breaking then there is another tree $T_{n-1}^j \in \tau_{n-1}$ that we can convert to $T_n$ applying a root extension.*

*Proof.* First assume $T_{n-1}^i$ has one of the three edges $(x, y), (y, z), (z, x)$ and that edge is $(x, y)$. Then introducing $(u, x)$ and $(u, y)$ edges creates a fundamental cycle in $T_n$ with $(x, y)$. Cycle breaking must remove the edge $(x, y)$ to keep $u$ adjacent to both $x$ and $y$. Then according to Lemma 4, cycle breaking to make $v$ adjacent to both $y$ and $z$ in $T_n$ is equivalent to an edge subdivision on another tree $T_{n-1} \in \tau_{n-1}$ that has the edge $(y, z)$. We can pick such a $T_{n-1}$ from $\tau_{n-1}$ so that it retains the edge $(x, y)$. We have the choice because the alternative path between $z$ and $y$ has at least three edges due to the absence of the edge $(z, x)$ in $T_{n-1}^i$, which allows us to avoid removing edge $(x, y)$ even if it happens to be a part of the second, larger fundamental cycle. Therefore, we get a $1 \leq j \leq N$ for which $T_{n-1}^j$ has the pair of edges $(x, y), (y, z)$ and applying a root extension to $T_{n-1}^j$ will generate $T_n$.

In case $T_{n-1}^i$ has none of the three edges $(x, y), (y, z), (z, x)$ then we can apply Lemma 4 to discover an intermediate tree $T_{n-1}$ that has one edge among the three. Then we can use the same argument we used above to reach a $T_{n-1}^j$ from $T_{n-1}$ and then show that we can generate $T_n$ from $T_{n-1}^j$ through root extension. □

Lemma 4 and 5 imply that one can add the last vertex of a dividing vertex sequence $d_s = v_1, v_2, \cdots, v_{n-3}$ of $G_n$ to inductively generate all spanning trees of $G_n$ from the spanning trees of $G_{n-1}$ using only edge extension, edge subdivision, and root extension regardless of the dividing order of earlier vertices. However, if $G_n$ has more than one degree 3 vertices, then the last dividing vertex itself can be different between two $d_s^1 = v_1^1, v_2^1, \cdots, v_{n-3}^1$ and $d_s^2 = v_1^2, v_2^2, \cdots, v_{n-3}^2$ describing $G_n$. Then $G_{n-4}^{im} = G_{n-1}$ for the two sequences differ by at least one pair of vertices. Consequently, their set of spanning trees $\tau_{n-1}^1$ and $\tau_{n-1}^2$ has no common tree, which may cause us to miss some spanning trees if we only use one of $d_s^1$ and $d_s^2$ for tree enumeration. The following lemma eliminates this concern by showing that the ordering differences of vertices among two dividing vertex sequences of $G_n$ do not impact spanning tree generation.

**Lemma 6.** *Assume that two dividing vertex sequences $d_s^1 = v_1^1, v_2^1, \cdots, v_{n-3}^1$ and $d_s^2 = v_1^2, v_2^2, \cdots, v_{n-3}^2$ of a plane 3-tree $G_n$ start diverting at index $k$. That is, the sub-sequences $d_{sub}^1 = v_k^1, v_{k+1}^1, \cdots, v_{n-3}^1$ and $d_{sub}^2 = v_k^2, v_{k+1}^2, \cdots, v_{n-3}^2$ are different from their beginning. Assume $\tau_{k-1}$ is the set of all spanning trees of $G_{k-1}^{im}$. Then for each tree $T_1 \in \tau_{k-1}$ and a series of edge extension, edge subdivision, and root extension to $T_1$ in the order of $d_{sub}^1$ forming a spanning tree $T_n$; there is another tree $T_2 \in \tau_{k-1}$ such that an alternative sequence of those operations in the order of $d_{sub}^2$ generates the same spanning tree of $G_n$.*

*Proof.* If some part of the tail end of $d_{sub}^1$ and $d_{sub}^2$ are the same, we can remove that common part from consideration and prove the lemma for a smaller plane 3-tree. Therefore, without the loss of generality, assume that $u = v_{n-3}^1 \neq v_{n-3}^2 = v$. Further, assume $G_{n-1}^1$ and $G_{n-1}^2$ are the plane 3-trees of $n-1$ vertices arising from $G_{k-1}^1$ after adding all but the last dividing vertex from $d_{sub}^1$ and $d_{sub}^2$ respectively. Thus, for any pair of spanning trees $T_{n-1}^1$ of $G_{n-1}^1$ and $T_{n-1}^2$ of $G_{n-1}^2$, $u \notin T_{n-1}^1 \ni v$ and $v \notin T_{n-1}^2 \ni u$. Note that the condition $degree(u) = degree(v) = 3$ holds as both $u$ and $v$ appear as the last dividing vertex in some dividing vertex sequences.

Consider the generation of $T_n$ from $T_{n-1}^1$ by adding $u$ through an edge extension, edge subdivision, or root extension. Since $degree(v) = 3$ and $v$ is already a part of $T_{n-1}^1$, $v \notin \zeta_u$ and and $u \notin \zeta_v$. In addition, node $v$ can be adjacent to at most 3 other nodes in $T_{n-1}^1$. Assume $T_{n-2}^1$ is the tree we get after removing $v$ from $T_{n-1}^1$ using an appropriate edge removal, path contraction, or root flattening. Then we apply the operation on $T_{n-2}^1$ that we would apply to $T_{n-1}^1$ to reach $T_n$. Let us call the tree that results from adding $u$ one step earlier $T_{n-1}^{1'}$. $T_{n-1}^{1'}$ is a spanning tree of $G_{n-1}^2$ as it contains all vertices of $G_n$ except $v$.

If $u$ is added as an edge extension to $T_{n-1}^1$ to form $T_n$ then $T_{n-1}^{1'}$ is the spanning tree of $G_{n-1}^2$ where we can repeat the reverse operation of removing $v$ earlier to generate $T_n$. The same condition holds if $v$ was connected to $T_{n-1}^1$ by an edge extension. If $v$ was connected to $T_{n-1}^1$ by an edge subdivision or root extension and $\zeta_v \bigcap \zeta_u = \emptyset$, then again the same condition holds. If $v$ was adjacent to two vertices $x, y \in \zeta_v$ in $T_{n-1}^1$ and $u$ was added to $T_{n-1}^{1'}$ using an edge subdivision, or root extension involving only one of $x, y$ then again the

same condition holds. On the other hand, the same edge $(x, y)$ cannot undergo two edge subdivisions or both an edge subdivision and a root extension to form $T_n$. Therefore, we covered all cases, and $T_{n-1}^{1'}$ is the spanning tree of $G_{n-1}^2$ that can generate the same $T_n$ using an edge extension, edge subdivision, or root extension.

Since the difference of $v$'s position in $d_{sub}^1$ and $d_{sub}^2$ did not affect the final spanning tree of $G_n$, we can remove $v$ from both sub-sequences and inductively apply the same reasoning for the spanning tree of $G_{n-1}$ and so on. Therefore, the reverse sequence of operations we apply to backtrack the vertices of $d_{sub}^2$ is that alternative sequence for generating $T_n$ from $T_{k-1}$. □

The following corollary elegantly expresses the finding of Lemma 6.

**Corollary 2.** *Any two dividing vertex sequences $d_s^1$ and $d_s^2$ of a plane 3-tree $G_n$ generate the same spanning tree $T_n$ irrespective of their differences if the tree modification operation for each dividing vertex remains unchanged.*

## 4 Inductive Algorithm for Spanning Tree Enumeration

Lemma 1 to 6 give us all the insights we need to generate all spanning trees of a plane 3-tree inductively from spanning trees of smaller constituent plane 3-trees. This section presents and discusses an inductive algorithm for spanning tree enumeration that we will use as the blueprint for a better dynamic programming algorithm in the next section.

**A general outline of the algorithm:** We will take a sequence of dividing vertices, $d_s$, and corresponding face vertex set sequence, $f_s$, of the plane 3-tree $G_n$ as inputs to the algorithm. We initiate a list with the spanning trees for $G_3$. Then we iterate over the dividing vertex sequence. In each iteration, we take one dividing vertex, $v$, and apply edge extension to all trees in the list, edge subdivision to those trees where the operation is permitted, and root extension to those trees where the face vertices of $v$ form a specific path of length two. We store the result of these operations in a new list that replaces the old list at the end of the iteration. When the iterative process completes, the list contains all spanning trees of $G_n$ without duplication. We use root extension with caution due to Lemma 3 showing that some sub-trees under the face vertices of a dividing vertex need to be different to avoid duplicate spanning tree generation. In that regard, if $v$ is the face vertex under consideration at a particular iteration and $x, y, z$ are $v$'s face vertices with $\delta(x, d_s) < \delta(y, d_s) < \delta(z, d_s)$, we apply root extension for $v$ on a tree only if it has the path $\rho_{x,y,z}$. To use this logic for the three outer face vertices, we assign them arbitrary but fixed dividing orders $-2, -1$ and $0$.

Algorithm 1 presents the pseudo-code of the spanning tree enumeration process.

The correctness of Algorithm 1 is evident from the various lemmas of Section 3. Hence, we focus on its running time and space requirement analysis. For that, we first need to understand the cost of generating a dividing vertex sequence and

**Algorithm 1:** Inductive Algorithm for Spanning Tree Enumeration

```
Input: d_s, f_s - a dividing vertex and face vertex set sequence pair of G_n
Output: τ - the set of all spanning trees of plane 3-tree G_n
{x, y, z} ← f_s[0]
τ ← {ρ_{x,y,z}, ρ_{y,x,z}, ρ_{x,z,y}}
for ( i = 0; i < length(d_s); i = i + 1 ) do
    ν ← d_s[i]
    ζ ← f_s[i]
    [α, β, γ] ← sortByDividingOrder(ζ)
    τ' ← ∅
    foreach T ∈ τ do
        foreach μ ∈ {α, β, γ} do
            τ' ← τ' ⋃{T ∪ {(μ, ν)}}
        end
        foreach (μ, ω) ∈ T & {μ, ω} ⊂ ζ do
            τ' ← τ' ⋃{(T − {(μ, ω)}) ∪ {(μ, ν), (ν, ω)}}
        end
        if ρ_{α,β,γ} ∈ T then
            τ' ← τ' ⋃(T − ρ_{α,β,γ}) ∪ {(ν, α), (ν, β), (ν, γ)}
        end
    end
    τ ← τ'
end
```

modifying a spanning tree of the smaller subgraph to produce one for a larger graph/subgraph.

**Dividing Vertex Sequence Generation:** We can generate a $d_s, f_s$ pair for a plane 3-tree $G_n$ by using a graph traversal of $G_n$ starting from any vertex and populating a pair of stacks. Anytime the graph traversal reaches a vertex $v$ of degree three, we push $v$ in one stack and its neighbor set to the other. Then we remove $v$ and its incident edges from the graph and continue the traversal. The traversal ends when we find a vertex with degree two. Then we generate the $d_s, f_s$ pair by popping one element at a time from both stacks and appending the elements in the growing list of dividing vertex and face vertex set sequences. The whole process requires a single traversal of $G_n$ that would take $\mathcal{O}(n+m)$ time where $m$ is the number of edges in $G_n$ and $\mathcal{O}(n+3n) = \mathcal{O}(n)$ space beyond the initial storage for the input graph $G_n$.

**Data Structure for Spanning Trees:** We can represent the spanning trees using a simple array of $n-1$ vertices where the entry at $i^{th}$ index is the other endpoint of an edge incident to the vertex with dividing order $i-1$. We adopt the following approach of modifying a spanning tree for $G^{im}_{j-1}$ to form a spanning tree for $G^{im}_{j}$ to ensure that each entry in the array refers to a distinct edge.

1. If we add the new vertex $v$ with $\delta(v, d_s) = j$ to the spanning tree using an edge extension from existing vertex $x$ then we enter $\delta(x, d_s)$ at index $(j+1)$.

2. If we add $v$ subdividing the edge $(x, y)$ then we investigate index $\delta(x, d_s) + 1$ and $\delta(y, d_s) + 1$ to determine which entry refers to the edge $(x, y)$, replace the current value with $j$ in that entry, and then write the dividing order of the other entry at index $(j + 1)$.
3. Finally, if we use root extension to $x, y, z$ to add $v$ in the spanning tree, then we modify two existing entries using the approach of (2) and add a new entry for the remaining vertex at index $(j + 1)$.

Note that the above approach works even when one of the face vertices of $v$ is the first outer face vertex, which has no entry in the array.

We now prove the following theorem of Algorithm 1 and time complexities.

**Theorem 1.** *Assuming replicating a fixed-size array in memory is a constant time operation, Algorithm 1 enumerates all spanning trees of a plane 3-tree $G_n$ of $n$ vertices, $m$ edges, and $\tau$ spanning trees in $\mathcal{O}(n + m + \tau)$ time and $\mathcal{O}(n\tau)$ space.*

*Proof.* Assuming memory replication of a fixed-length array is a constant time operation, and the storage structure of a spanning tree is as we described above; generating a new spanning tree $T'$ for $G^{im}_{i+1}$ from a spanning tree $T$ of $G^{im}_i$ is a constant time operation. Furthermore, evaluating whether an edge subdivision or root extension is admissible in $T$ also takes a constant time (we can investigate the three indexes for the face vertices of the current vertex in $T$ to determine their adjacency). In addition, the algorithm investigates each $T$ exactly once and generates at least three new trees. Therefore, the algorithm's running time is proportional to the number of spanning trees generated through the iterative process.

If $\tau_{i+1}$ represents the number of spanning trees of $G^{im}_{i+1}$ then $\tau_i \leq \frac{\tau_{i+1}}{3}$ for all $i < n$. Thus, the total number of trees the algorithm generates is $\leq \tau + \frac{\tau}{3} + \frac{\tau}{3^2} + \cdots + \frac{\tau}{3^n} = \frac{\tau(3^{n+1}-1)}{2\times 3^n} < 2\tau$ , where $\tau$ is the number of spanning trees of $G_n$. Thus, the algorithm's running time is $\mathcal{O}(\tau)$. After adding the computation time for generating a dividing vertex and its face vertex set sequences, the overall time complexity of enumerating the spanning trees of an input plane 3-tree $G_n$ becomes $\mathcal{O}(n + m + \tau)$, where $m$ is the number of edges in $G_n$. The algorithm only retains the two input sequences and the set of spanning trees of the current plane 3-tree. Therefore, the space complexity is $\mathcal{O}(n + n\tau) = \mathcal{O}(n\tau)$. □

Suppose we relax the assumption that memory replication of a fixed-length array is a constant time operation. If an array copy takes time linear to the number of elements in the array, then constructing each spanning tree takes $\mathcal{O}(n)$ time. Then the running time of Algorithm 1 becomes $\mathcal{O}(n + m + n\tau)$, which is equivalent to the best-known algorithm for spanning tree enumeration in a general graph. In the next section, we will use dynamic programming to convert the inductive tree generation process into a tree alternation process that avoids array copying and reduces the space required to $\mathcal{O}(n)$. However, one should notice that Algorithm 1 involves no coordination among the elements of the spanning

tree set in each iteration. That makes it perfectly parallelizable. After the generation of the two input sequences and a few iterations of smaller spanning tree generation; different parallel processors can independently continue the inductive process from the initial spanning tree set. That gives the time complexity of $\mathcal{O}(n+m+\frac{n\tau}{p})$ for $p$-processor parallel version of the algorithm. The space requirement in individual processors is $\frac{n\tau}{p}$. Thus the following theorem holds.

**Theorem 2.** *Assuming copying a fixed-size array is a linear time operation to the size of an array, a $p$-processor parallel version of Algorithm 1 enumerates all spanning trees of a plane 3-tree $G_n$ of $n$ vertices, $m$ edges, and $\tau$ spanning trees in $\mathcal{O}(n+m+\frac{n\tau}{p})$ time and $\mathcal{O}(\frac{n\tau}{p})$ space.* □

## 5 DP Algorithm for Spanning Tree Enumeration

Algorithm 1 of the previous section is close to optimal because it involves no duplicate spanning tree generation and churns out new spanning trees in constant time from other intermediate spanning trees. However, it must keep all the intermediate spanning trees of the immediately previous step in memory and involves a tree copying overhead during a new spanning tree generation.

We can eliminate both limitations by converting the inductive tree-generation process into a recursive tree mutation process. Note that if $\tau$ is the set of all spanning trees of the plane 3-tree $G_n$, one can view each spanning tree $T \in \tau$ as a unique configuration of edge extension, edge subdivision and root extension operations. Furthermore, for all $T \in \tau$, these operations are applied on a path of length two connecting the outer face vertices in the same order vertices appear in a dividing vertex sequence. Consequently, we can consider successive trees added in $\tau$ as migration from one configuration to the next. A scheme that can alter between configurations in constant time, traverse all configurations, and only emit unique configurations should emulate Algorithm 1. We only need to maintain a single spanning tree corresponding to the current configuration when traversing the domain of all spanning tree configurations. Suppose we can apply a clever trick of using the list of edges of the current spanning tree as the representation of the underlying configuration. In that case, there is no additional space overhead beyond that of a single spanning tree and two lists for the dividing vertex and face vertex set sequences. That is the logic of our DP algorithm in this section.

First, we need an initial configuration for the described scheme to work. Assume $a, b$, and $c$ are the outer face vertices with our assigned respective dividing orders $-2, -1$, and 0. Then the initial configuration has the path $\rho_{a,b,c}$ and for each dividing vertex $x$ in the sequence $d_s$ the edge $(w, x)$ such that $w \in \zeta_x$ and $\delta(w, d_s)$ is the largest for all of $x$'s face vertices. Figure 5(a) and (b) illustrate a plane 3-tree and its spanning tree for the initial configuration.

If $\zeta_x = \{u, v, w\}$, then alternative configurations for $x$ are to have edge extension from $u$ or $v$, subdividing edges $(u, v), (v, w)$ or $(u, w)$, and root extending the path $\rho_{u,v,w}$. Among them, the admissibility of edge subdivision or root extension depends on the current configuration for $u, v$, and $w$. Let us call moving

to any alternative configuration for $x$ from its default configuration in any tree $T$ a *mutation* and number the mutations in the order we specified from zero to five. Then we can define the following functions.

$$mutateTree(T, v, \zeta_v, i) = \begin{cases} T' & \text{mutation } i \text{ can be applied to } T \text{ for } v \text{ to create } T' \\ \emptyset & \text{otherwise} \end{cases} \tag{1}$$

$$reverseMutation(T', v, \zeta_v, i) = \begin{cases} T & T' \text{ is the result of applying mutation } i \text{ for } v \text{ to } T \\ T' & \text{otherwise} \end{cases} \tag{2}$$

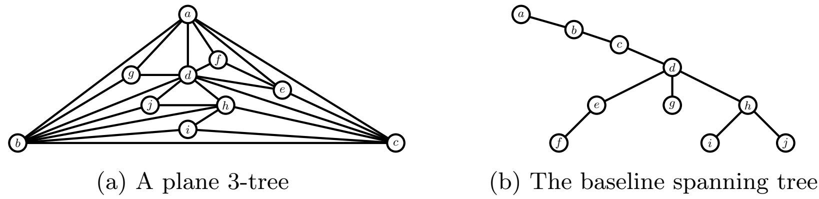
If we use the data structure described in Section 4 for the spanning trees, then both *mutateTree* and *reverseMutation* functions should take constant time. Both require a constant number of index checking and entry updates in the list of edges sorted by the dividing order of vertices.

To permit configuration changes for the initial path connecting the outer face vertices, we only need to support an alternative edge extension and an edge subdivision for the outer vertex with order zero as an exceptional case. Finally, note that the *mutateTree* function updates the argument tree $T$ when the mutation is admissible; otherwise, it leaves $T$ unchanged. Now we can define our DP algorithm for spanning tree enumeration of a plane 3-tree, as illustrated in Algorithm 2.

Now we prove the following theorem on the correctness and time and space complexities of Algorithm 2.

**Theorem 3.** *Algorithm 2 enumerates all spanning trees of a plane 3-tree $G_n$ with $n$ vertices, $m$ edges, and $\tau$ spanning trees without duplication in $\mathcal{O}(n+m+\tau)$ time and $\mathcal{O}(n)$ space.*

*Proof.* Algorithm 2 first generates a spanning tree of $G_n$ for the initial configuration, updates the dividing vertex and face vertex set sequences to include one outer face vertex, and then invokes the function *generateSpanningTrees* that implements the DP algorithm. Therefore, it suffices that we only analyze the *generateSpanningTrees* function for the correctness and complexity of the DP scheme.



(a) A plane 3-tree (b) The baseline spanning tree

Fig. 5: An example plane 3-tree and its baseline spanning tree

```
Algorithm 2: DP Algorithm for Spanning Tree Enumeration
Input: d_s, f_s - a dividing vertex and face vertex set sequence pair of G_n
Output: τ - the set of all spanning trees of plane 3-tree G_n
ζ_0 ← f_s[0]
[α_0, β_0, γ_0] ← sortByDividingOrder(ζ_0)
T ← []
T[0] ← δ(α_0, d_s)
T[1] ← δ(β_0, d_s)
for ( i = 0; i < length(d_s); i = i + 1 ) do
    ζ ← f_s[i]
    [α, β, γ] ← sortByDividingOrder(ζ)
    T[i + 2] ← δ(γ, d_s)
end
d_s ← [γ_0].append(d_s)
f_s ← [{α_0, β_0}].append(f_s)
generateSpanningTrees(d_s, f_s, 0, T)

Function generateSpanningTrees(d_s, f_s: List, i: Integer, T: Array) is
    if i == length(d_s) then
        output(T)
    else
        ν ← d_s[i]
        ζ ← f_s[i]
        for ( m_i = 0; m_i ≤ 5; m_i = m_i + 1 ) do
            T' ← mutateTree(T, ν, ζ, m_i)
            if T' ≠ ∅ then
                generateSpanningTrees(d_s, f_s, i + 1, T')
                T ← reverseMutation(T', ν, ζ, m_i)
        end
        generateSpanningTrees(d_s, f_s, i + 1, T)
    end
end
```

Each time *generateSpanningTrees* output a spanning tree, it is for a different configuration of mutations of vertices from $d_s$. Hence, the algorithm only outputs distinct spanning trees of $G_n$. The function removes mutation in the reverse order it applies them. Therefore, a mutation of any vertex is reversible during a backtracking step if it was admissible during the recursive unfolding. The only question is whether or not *generateSpanningTrees* accommodates all possible mutations of each vertex. The answer lies in the choice of default configuration for dividing vertices.

Assume that $u$ is the vertex with dividing order $i$. Anytime *generateSpanningTrees* tries to apply a mutation for $u$ from its default configuration, all vertices $v$ with $\delta(v, d_s) > \delta(u, d_s)$ are attached via an edge extension to their respective face vertex with the highest dividing order. The critical characteristic of edge extension from a vertex $x$ to $y$ where $\delta(x, d_s) < \delta(y, d_s)$ is that it does not preclude

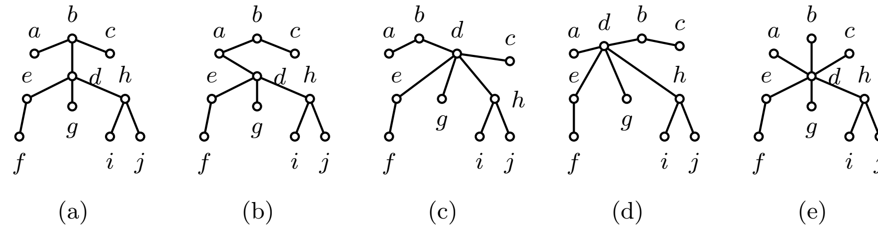



Fig. 6: Possible mutations of vertex d from the baseline tree of Fig 5(b)

any mutation of $x$ involving vertices with lower dividing orders. Figure 6 illustrates this fact for vertex $d$ of the plane 3-tree of Figure 5(a) (Notice that one mutation for $d$, i.e., subdividing the edge $(a, c)$ is not admissible from the initial configuration as that requires a mutation of $c$.). Consequently, admissible mutations for $u$ depend only on the current configuration of vertices that appeared earlier in $d_s$. Therefore, applying a mutation or skipping all mutations for $u$ and then progressing to the next vertex in Algorithm 2 is equivalent to generating all possible spanning trees of $G_i^{im}$ from a fixed spanning tree of $G_{i-1}^{im}$ in Algorithm 1. Thus, Algorithm 2 is behaviorally equivalent to Algorithm 1 and correctly enumerates all spanning trees of $G_n$ without duplication.

Algorithm 2 only maintains a single spanning tree of $G_n$ and two sequences for dividing vertices and their face vertex sets. Hence, we can derive from the analysis of Algorithm 1 that Algorithm 2's time complexity is $\mathcal{O}(\tau)$ and space complexity is $\mathcal{O}(n)$. Combining that with the initial cost of generating the dividing vertex and face vertex set sequences, the overall cost of spanning tree enumeration of a plane 3-tree $G_n$ using dynamic programming is $\mathcal{O}(n + m + \tau)$ in time and $\mathcal{O}(n)$ in space, where $m$ is the number of edges and $\tau$ is the number of spanning trees. □

## 6 Conclusions

In this paper, we propose an algorithm for enumerating all spanning trees of a plane 3-tree $G_n$ of $n$ vertices, $m$ edges, and $\tau$ spanning trees in $\mathcal{O}(n + m + \tau)$ time and $\mathcal{O}(n)$ space. Our algorithm substantially improves both time and space bounds of the best general-purpose spanning tree enumeration algorithm that takes $\mathcal{O}(n + m + n\tau)$ time and $\mathcal{O}(nm)$ space for this specific graph class. The central idea underlying this improvement is identifying a few low-cost extension/mutation operations that we can inductively use to extend spanning trees of smaller induced subgraphs of a plane 3-three to that of larger subgraphs. In our case, there are only three such constant time operations. We proved that these operations could generate all spanning trees of a plane 3-tree and their application avoids duplicate tree generation.

There are several avenues for future research from our current findings. First, one can investigate the possibility of a similar spanning tree enumeration tech-

nique for other graph classes, such as regular graphs with small vertex degrees. Second, one can try to relate our tree extension/mutation operations with edit distance computation for pair of spanning trees of a plane 3-tree or prove properties related to spanning trees. The most exciting future research direction is to prove that if a set of fundamental operations exist that can express any spanning tree of a graph, then all their applicable permutations on a sorted sequence of that graph's vertices enumerate all spanning trees. If that assumption is correct, our specific spanning tree enumeration algorithm for plane 3-tree will generalize. Then the problem of efficient spanning tree enumeration will become the problem of finding such fundamental operations for different graph classes.

## References


1. Brooks, R., Smith, C., Stone, A., Tutte, W.: Determinants and current flows in electric networks. Discrete Mathematics **100**(1), 291–301 (1992). https://doi.org/10.1016/0012-365X(92)90648-Y
2. Char, J.: Generation of trees, two-trees, and storage of master forests. IEEE Transactions on Circuit Theory **15**(3), 228–238 (1968). https://doi.org/10.1109/TCT.1968.1082817
3. Chen, W.K.: Topological analysis for active networks. IEEE Transactions on Circuit Theory **12**(1), 85–91 (1965). https://doi.org/10.1109/TCT.1965.1082396
4. Deo, N.: Graph Theory with Applications to Engineering and Computer Science (Prentice Hall Series in Automatic Computation). Prentice-Hall, Inc., USA (1974)
5. Gabow, H.N., Myers, E.W.: Finding all spanning trees of directed and undirected graphs. SIAM Journal on Computing **7**(3), 280–287 (1978). https://doi.org/10.1137/0207024
6. Hakimi, S., Green, D.: Generation and realization of trees and k-trees. IEEE Transactions on Circuit Theory **11**(2), 247–255 (1964). https://doi.org/10.1109/TCT.1964.1082276
7. Hakimi, S.: On trees of a graph and their generation. Journal of the Franklin Institute **272**(5), 347–359 (1961). https://doi.org/10.1016/0016-0032(61)90036-9
8. Kapoor, S., Ramesh, H.: Algorithms for enumerating all spanning trees of undirected and weighted graphs. SIAM Journal on Computing **24**(2), 247–265 (1995). https://doi.org/10.1137/S009753979225030X
9. Matsui, T.: An algorithm for finding all the spanning trees in undirected graphs (1998)
10. Mayeda, W., Seshu, S.: Generation of trees without duplications. IEEE Transactions on Circuit Theory **12**(2), 181–185 (1965). https://doi.org/10.1109/TCT.1965.1082432
11. Nagamochi, H., Ibaraki, T.: A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. Algorithmica **7**(5&6), 583–596 (1992), https://doi.org/10.1007/BF01758778
12. Nishat, R.I., Mondal, D., Rahman, M.S.: Point-set embeddings of plane 3-trees. In: Brandes, U., Cornelsen, S. (eds.) Graph Drawing. pp. 317–328. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
13. Onete, C.E., Onete, M.C.C.: Enumerating all the spanning trees in an un-oriented graph - a novel approach. In: 2010 XIth International Workshop on Symbolic and Numerical Methods, Modeling and Applications to Circuit Design (SM2ACD). pp. 1–5 (2010). https://doi.org/10.1109/SM2ACD.2010.5672365

14. Rakshit, A., Sarma, S.S., Sen, R.K., Choudhury, A.: An efficient tree-generation algorithm. IETE Journal of Research **27**(3), 105–109 (1981). https://doi.org/10.1080/03772063.1981.11452333
15. Read, R., Tarjan, R.: Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. Networks **5**(3), 237–252 (Jan 1975). https://doi.org/10.1002/net.1975.5.3.237
16. Shioura, A., Tamura, A.: Efficiently scanning all spanning trees of an undirected graph. Journal of the Operations Research Society of Japan **38**(3), 331–344 (1995). https://doi.org/10.15807/jorsj.38.331