The Tree of Trees: on methods for finding all non-isomorphic tree-realizations of degree sequences

by

Samuel Stern

A thesis submitted to the faculty of Wesleyan University in partial fulfillment of the requirements for the Degree of Bachelor of Arts with Departmental Honors in Computer Science

Middletown, Connecticut 04/19/2017
Acknowledgements

To Professor Danny Krizanc, without whose guidance and curiosity this project would never have gotten started, much less finished.

To Professor James Lipton and Professor Victoria Manfredi, for their insight and expertise.

To my parents, for their support and encouragement.

To my friends, for putting up with me for as long as it took to finish this project.

To the Starbucks and Cafe Sofias of the world, for many productive hours of scones and wifi.
Abstract

A degree sequence for a graph is a list of positive integers, one for every vertex, where each integer corresponds to the number of neighbors of that vertex. It is possible to create sequences that have no corresponding graphs, as well as sequences that correspond to multiple distinct (i.e., non-isomorphic) graphs. A graph that corresponds to a given degree sequence is called a realization of that sequence, and those degree sequences corresponding to trees are referred to as tree-realizable, or tree-able, sequences. In this thesis, we present a program that determines the number of tree realizations for any tree-able sequence, along with the sequences with the most tree-realizations (for fixed sequence lengths).
## Contents

1. Introduction  
   1.1 Definitions  
   1.2 Related results  
   1.3 Overview of results  
2. The algorithm  
   2.1 Proof of correctness  
   2.2 Order of runtime  
   2.3 Improvements on the basic algorithm  
   2.4 The tree of trees  
3. Results and discussion  
4. Conclusion  

Bibliography  

Appendix A: Sequence generation  
Appendix B: Java implementations
1. Introduction

One of the more interesting areas of research in graph theory is that of graph reconstruction. Graphs in general have many characteristics – edge count, vertex count, degree sequence, graph order, connectedness, the presence of spanning trees – which can differ between otherwise similar graphs. Considerable effort has been focused on the question of which of these characteristics can be used to uniquely define a graph. Any characteristic with this property can be used to reconstruct the graph, with no other information required.

A degree sequence is one characteristic of a graph: a list of positive integers, one for each vertex, such that each represents the number of neighbor vertices possessed by that vertex. It is generally known that degree sequences, by themselves, cannot be used to uniquely reconstruct a graph. While certain degree sequences do uniquely determine a graph, other sequences have hundreds of distinct corresponding graphs; still other sequences have no corresponding graphs at all. Some work has been done in the field of determining relative realization count [1]: in other words, determining whether one sequence has more realizations than another. We provide an algorithm to determine how many distinct realizations exist for a given sequence.

1.1. Some definitions and preliminaries. We begin with some definitions, following from those in [2]. A graph is a collection of vertices and edges, which are unordered pairs of vertices. A vertex set $V$ and edge set $E$ uniquely determine a graph, and so a graph $G$ can be represented as $(V, E)$. In a graph $G = (V, E)$, if the edge $(u, v)$ is contained in $E$, then the vertices $u$ and $v$ are neighbors of each other. In this thesis, we will represent graphs both pictorially and in
adjacency-list form. (Adjacency-list form utilizes a list of lists, one list per vertex, such that each nested list contains the neighbor vertices of that specific vertex.)

A **path** of length $k$ in a graph $G = (V, E)$ from a vertex $u$ to another vertex $v$ is a sequence of distinct vertices $(v_0, v_1, \ldots, v_k)$ such that $u = v_0$, $v = v_k$, and the edge represented by $(v_{i-1}, v_i)$ is contained within $E$ for $i = 1, 2, \ldots, k$. A **cycle** in a graph $G = (V, E)$ is a path $(v_0, v_1, \ldots, v_k)$ such that $v_0 = v_k$ and the path contains at least three edges. A **tree** is a graph with no cycles—in other words, any two given vertices are linked by only one path.

The **neighborhood** of a vertex $v$ is the set of vertices $\{v_1, v_2, \ldots, v_d\}$ such that $v$ and $v_i$ are neighbors for all $i$ in $1, 2, \ldots, d$. We define the **degree** of a given vertex as its number of neighbors, or the size of its neighborhood. We define the **degree sequence** of a graph as a list of the degrees of each of its vertices (usually in either ascending or descending order), represented as $D = (d_1, d_2, \ldots, d_n)$.

While the degree sequences of graphs may vary, it has long been acknowledged that degree sequences of trees all have something in common: for $n$ being the number of vertices, the elements of a tree degree sequence always sum to $2(n-1)$. The reverse is also folklore:

**Lemma 1.** If a sequence of positive integers is of length $n$, and its elements sum to $2(n-1)$, then that degree sequence can only be realized as a tree.

Thus we have some limits on what a given degree sequence can “look like”. We define a **realization** of a degree sequence as any specific graph that matches the sequence. Only degree sequences that satisfy the predicate of Lemma 1 can be
realized as trees (see Fig. 1.1), and these sequences can be realized only as trees. We refer to these sequences as **tree-able** sequences, or tree sequences. Lemma 1 allows us to generate tree-able degree sequences knowing that those sequences can be realized in at least one way. However, the lemma offers no “upper bound” on how many realizations a sequence might have. For example, the sequence \([1, 1, 1, 2, 2, 3]\) is tree-able but has more than one distinct realization (see Fig. 1.2). It may be useful to have a method for determining how many “distinct” realizations a tree-able degree sequence has. Our criterion for two trees being “distinct” is that they not be isomorphic. Following from the definition in [2]:

**Isomorphism.** Two graphs \(A\) and \(B\), with vertex and edge sets \(V_A, E_A\) and \(V_B, E_B\) respectively, are isomorphic if there exists a bijection \(C\), mapping vertices in \(V_A\) to vertices in \(V_B\), such that, for all pairs \((u, v)\) of vertices in \(V_A\), \((u, v)\) is in \(E_A\) if and only if the pair \((C[u], C[v])\) is in \(E_B\).

For a degree sequence \(D\), we use \(T(D)\) to denote the set of distinct (i.e., non-isomorphic) realizations of \(D\). We use \(M(n)\) to denote the maximum of \(|T(D)|\) over all \(D\) of length \(n\).
Trees.

Fig. 1.1: An attempt at realizing $[1, 1, 1]$, which doesn’t satisfy Lemma 1.

Fig. 1.2: Two distinct (non-isomorphic) realizations of the tree-able sequence $[1, 1, 1, 2, 2, 3]$. All vertices above are labeled with their degree.
1.2. Related results. Ours is only the most recent research into questions related to graph determination and realizability by degree sequences. Among the most fundamental results in the field is the Erdős-Gallai theorem [3]. It may be stated as follows:

**Theorem.** A sequence of non-negative integers \( d_1 \geq d_2 \geq \ldots \geq d_n \) has a corresponding graph (with \( n \)-many vertices) if and only if \( d_1 + \ldots + d_n \) is even and:

\[
\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{j=k+1}^{n} \min(d_j, k)
\]

is true for every \( k \) between 1 and \( n \), inclusive.

The Havel-Hakimi algorithm [4] [5] is similar in terms of scope and results, but not method. It forms a central part of our research.

**Havel-Hakimi.** Let \((d_1, \ldots, d_n)\) be a sequence of integers, where \( d_1 = m < n \), and \( d_1 \geq d_2 \geq \ldots \geq d_n \). The sequence is realizable as a graph if and only if the sequence \((d_2−1, d_3−1, \ldots, d_{m+1}−1, d_{m+2}, \ldots, d_n)\) contains non-negative integers and is realizable as a graph.

Research has also been done into relating the number of realizations of different degree sequences [1]. The main result was to establish that, if a degree sequence \( D_1 \) “majorizes” another sequence \( D_2 \), according to a formula laid out in [1], then \( D_1 \) has more realizations than does \( D_2 \) (though this does not account for isomorphisms). Kapoor et al. [13] researched the characterization of simple graphs and trees with \( n \)-many vertices based on their degree sets (which are degree sequences with duplicates removed). Winkler et al. [14] [15] [16] [17] proved
other results related to degree sets and trees. Other results have been proven on
the topic of characterizing the degree sequences of specific classes of graphs, such
as split graphs [19] [20], unicycle graphs [21], cactus graphs [22], Halin graphs
[23] and 2-trees [6].

1.3. Overview of results. In this thesis, we present an algorithm which,
given a tree-able sequence $D$ of length $n$, computes $T(D)$ in $O((n^2)!^n)$ time. As
part of this, we use an algorithm for tree isomorphism removal based on the al-
gorithm of Aho, Hopcroft and Ullman [12], as well as an updated version which
encodes more information than that described by [12]. We then discuss improve-
ments to our original algorithm which allow it to run more efficiently, computing
every $T(D_n)$ for some $n$ in $O(c_n(n^9))$, where $c_n$ is the number of non-isomorphic
trees on $n$ vertices, and is bounded by $2.956^n/(n^{5/2})$ [18]. With these, we compute
$M(n)$, and the distributions of the various $T(D_n)$, up to $n = 18$. This involves
an independently-discovered algorithm for generating all tree-able sequences of
length $n$, which is similar to the work done by Dimitrov [8] but may also be of
interest in and of itself. (This algorithm runs in $O(n!(c_n))$ time, with $n$ being the
length of all the sequences to be generated; even faster versions are postulated in
[9].) We finally provide a Java implementation of our code.
2. The algorithm

The basic idea that underlies our algorithm is not new. It may be viewed as a series of logical extensions to the Havel-Hakimi algorithm [4], [5] originally developed in 1955. That recursive algorithm serves to decide if a degree sequence is realizable, or in other words, if there exists a graph that matches the sequence.

Recall:

**Havel-Hakimi.** Let \((d_1, \ldots, d_n)\) be a sequence of integers, where \(d_1 = m\), and \(d_1 \geq d_2 \geq \ldots \geq d_n\). The sequence is realizable as a graph if and only if the sequence \((d_2 - 1, d_3 - 1, \ldots, d_{m+1} - 1, d_{m+2}, \ldots, d_n)\) consists of non-negative integers and is realizable as a graph.

(We note that Havel-Hakimi is designed to apply not only to trees, but to graphs in general. For our purposes, we will be considering it only in relation to tree-able sequences as defined in Lemma 1.)

Essentially, Havel-Hakimi “builds” a hypothetical graph of the sequence, vertex by vertex. For a number removed from the degree sequence of value \(m\) (for each “vertex” created), \(m\)-many other numbers in the sequence are reduced by 1 (and so edges are created between the new vertex and \(m\)-many others.) Eventually, one entire graph will be built, and the remaining sequence will contain nothing but zeroes, or else negative numbers (and therefore be obviously non-graph-realizable). The algorithm is recursive, in the sense that the sequence \((d_2 - 1, d_3 - 1, \ldots, d_{m+1} - 1, d_{m+2}, \ldots, d_n)\) is a strictly smaller instance of the original problem.
We note that even in cases when the sequence is known to correspond to a graph (for example, sequences that are tree-able by Lemma 1), the Havel-Hakimi algorithm yields only one possible realization of the graph. Whether there exist others, or what they might be, is outside the algorithm’s purview. Yet it is easy to see that any possible graph of a degree sequence can be constructed from the sequence, in a Havel-Hakimi-like style. Since a graph (and/or tree) is a collection of vertices and the edges between them, and the vertex count is fixed by the length of the given degree sequence, it follows that one way to build all possible graphs involves trying every possible set of edges, i.e., of connections between vertices. Intuitively, an algorithm to do so must decrease a vertex’s “remaining degree” by 1 every time it binds a new edge to that vertex. In the end, each vertex will have as many edges as the original sequence depicted, and the sequence of “remaining degrees” will have been decreased to a list of zeroes.

Our algorithm is recursive, much like Havel-Hakimi, but a single call can return multiple results. The idea is that, instead of trying only one possible “way” of making connections, we try them all, and consider them separately. The “base case” of the algorithm is a sequence containing all zeroes (or, in its updated form, a sequence of \((n - 2)\)-many zeroes and two ones).

This design can be complicated to execute, as well as time-inefficient. Moreover, there is no guarantee that the graphs returned will be non-isomorphic, as per the definition in Chapter 1. Intuitively, given two graphs, if one can be moved around in the plane so that it looks like the other, they are isomorphic, and not distinct from each other as far as most of graph theory is concerned (see Fig. 2.1). Ideally, our algorithm for generating all possible graphs of a sequence wouldn’t count isomorphisms as separate. The problem of graph isomorphism has been
the subject of considerable research [10] [11], and as of yet, there is no known polynomial-time algorithm for determining whether two graphs are isomorphic. However, for several subclasses of graphs, such algorithms do exist, and one such subclass is that of trees. As such, from this point, we will focus primarily on trees, rather than graphs in general.

Aho, Hopcroft, and Ullman [12] devised a linear-time algorithm for tree-isomorphism; we follow the description in [7]. The insight was to consider rooted trees, which are ordinary trees with one vertex labeled as the root. As long as one’s choice of root is consistent between trees ([12] also provided a method of finding a “central” vertex in any tree), it is possible to create a “description” of any tree, inductively representing the number of “child” vertices at each level of the rooted tree, so that isomorphic trees will have the same “description.” More specifically, the algorithm selects either one or two “central” vertices by finding the two vertices in any given tree that are farthest apart (a process which runs in time linear relative to the sequence length), then selects the “midpoint”, or the two midpoints, on the path between the “farthest pair”. If there is only one center, the algorithm selects it as the so-called root, and envisions all of its neighbors as its children, each of which may have children of their own. (Note that this is just a conceptual alteration; the tree remains unchanged.) The algorithm finishes by encoding (into a string) the number of child vertices at every level; [12] proved that this is an isomorphism invariant between rooted trees, and can be used as an isomorphism invariant for trees in general. (If there are two centers, each is given a turn as the root, and two such strings are built. Any tree isomorphic to a two-centered tree will itself also have two centers, and the pairs of descriptor strings will match.)
The most relevant detail is that the “descriptions”, or canonical names, can be written as special patterns of two (arbitrary) characters (see Fig. 2.2). [12] used ones and zeroes; we will use a’s and b’s (for now).
Isomorphic trees (vertices labeled with arbitrary letters).

Fig. 2.1. Some isomorphic realizations of the tree-able sequence $[1, 1, 1, 2, 3]$.

Canonical names.

Fig. 2.2. Some trees and their AHU-style canonical names (root vertices in red.)
What we’ve described so far is the most basic form of our algorithm, which we call Algorithm A. Given a degree sequence, the steps are as follows:

(0) Check to see if the sequence is tree-able by verifying that (sum of its \( n \)-many degrees) = 2\((n - 1)\).

(1) Recursively form bonds. At every level, try every possibility, each with its own recursive call.

(2) Once the recursion is finished generating tree possibilities, create canonical names for each of them.

(3) Create equivalence classes based on which trees share canonical forms.

(4) Remove every member but one from each class. One of each kind of non-isomorphic tree will remain.

(5) Return the list of non-isomorphic trees.

In Section 2.1, we delve into the specifics, and prove that, for a degree sequence of length \( n \), the running time of this algorithm is \( O((n^2)!^n) \). We also discuss some improvements that may give the algorithm an asymptotically better running time. (The specific step of tree isomorphism testing runs in time proportional to \( O(R^2) \), in the worst case, for \( R \) being the number of possibly-isomorphic realizations that the above algorithm generates. Relating \( R \) to the input sequence length \( n \) is less straightforward and is also discussed in 2.1.)

With this done, we go on to find the degree sequence of length \( n \) with the most non-isomorphic realizations (denoted \( M(n) \)). We’ve successfully done so for \( n \) up to 18. We thus form a new sequence, \( RM(n) \) (short for realization-maximum sequence of length \( n \)), which does not, as of yet, appear in the Online Encyclopedia of Integer Sequences (accessible at https://oeis.org/).
2.1. Proof of correctness. We will now prove the correctness, and determine the asymptotic order of runtime, of this algorithm. To restate, given a ordered degree sequence of length $n$:

(0) Check to see if the sequence is tree-able by verifying that (sum of its $n$-many degrees) $= 2(n - 1)$.

(1) Recursively form bonds. At every level, try every possibility, each with its own recursive call.

(2) Once the recursion is finished generating tree possibilities, create canonical names for each of them.

(3) Create equivalence classes based on which trees share canonical forms and which do not.

(4) Remove every member but one from each class. One of each kind of non-isomorphic tree will remain.

(5) Return the list of non-isomorphic trees.

Step 0 is straightforward (and, in our final model, isn’t needed at all). We’ve already discussed the algorithm for Step 2. Steps 3 and 4 are straightforward (if time-costly), and Step 5 is just a formalism. Step 1 is the following recursion:

(1) Recursively form bonds.

   (a) Find all pairs of numbers in the sequence which represent a valid potential bond. For all such pairs:

      (i) Decrease both numbers by 1.

      (ii) Perform a recursive call on the resulting sequence.

      (iii) Accept, as a return value, all trees generated by the diminished sequence.
(iv) In all trees returned, add the *valid* bond pair chosen in (a).

(v) Add these updated trees to the group of trees to return.

(b) Return all trees yielded by (iv).

We immediately run into the question of what constitutes a *valid* bond. Concerns include: we don’t want to form bonds with zero-degree vertices (those already have as many neighbors as they need) even though all but two vertices will eventually end up with degree zero. We also don’t want to form more than one bond between the same two vertices (such graphs are outside the scope of this thesis). Checking for zeroes is straightforward, but avoiding double-bonds might seem to require some way of remembering, between recursive calls, what bonds have already been formed. It would be possible to implement this, but it would also be complex, not to mention costly in terms of memory space. Instead, we present a simple rule for validity which solves both problems, without requiring extra memory space:

**Bond validity.** At every level of recursion, only bonds that join a leaf (vertex of degree 1) and a non-leaf (vertex of degree $> 1$) are valid. If the only remaining non-zero vertices are two leaves, however, they may be validly bonded.

It must be proven that: the rule never allows bonds for zero-degree vertices; the rule never bonds two vertices more than once; and that, for any given degree sequence, all possible realizations can be formed under the rule. The first proposition is evident.

*Claim.* No two vertices $m$ and $n$ may be bonded more than once.
Proof. Assume to the contrary that, in some recursive call, there exist two vertices \( m \) and \( n \) which have already been bonded to each other once, and could be bonded to each other again, both times under the validity rule. Note that, for the first bond to occur, either \( m \) or \( n \), or both, must have been of degree 1. After the first bond was formed, the degrees of \( m \) and \( n \), or \( d(m) \) and \( d(n) \), were reduced by 1 each. Since either \( d(m) = 1 \) or \( d(n) = 1 \) before the bond was formed, it follows that either \( d(m) \) or \( d(n) \) was zero afterwards. The rule doesn’t allow for bonds to be formed with zero vertices, in any recursive call. This contradicts the premise. \( \Box \)

The inductive tree lemma. We claim that, given a degree sequence \( D \) of length \( n \) which is realizable only as a tree (i.e., the sum of \( D \)'s elements is \( S_D = 2(n-1) \)), any bond formed under the validity rule creates a new sequence \( D' \) (along with a leading zero which we disregard), which is also realizable only as a tree (i.e., \( S_{D'} = 2(n-2) \); see Fig. 2.3.).

Proof. Consider \( D_t = [d_1, d_2, \ldots, d_n] \) with \( d_1 \leq \ldots \leq d_n \). The sum of all elements \( = s_t = 2(n-1) \).

As a consequence of Lemma 1, \( D_t \) is guaranteed to contain at least two 1’s. Without loss of generality, consider the bonds formed from \( d_1 \) to some \( d_m, m \geq 2 \). The resulting sequence \( D'_t \) is of the form \([0, d_2, d_3, \ldots, d_m-1, d_{m+1}, \ldots, d_n]\). Ignoring the leading zero, the sequence is of length \((n-1)\), and \( s'_t = s_t - 2 \) (because of the bond being formed). We must now show that Lemma 1 holds for \( D'_t \).

\[
2(\text{length}(D'_t) - 1) = 2([n - 1] - 1) \\
2(\text{length}(D'_t) - 1) = s_t - 2
\]
Therefore, by Lemma 1, $D'_t$ is only realizable as a tree. □

With the inductive tree lemma, we can prove the final claim required for the validity rule to be usable.

**Claim.** All possible realizations of the tree sequence $D_t = [d_1, d_2, ..., d_n]$ can be generated by forming only those bonds permitted under the validity rule.

**Proof.** As noted in the beginning of Chapter 2, a graph is merely a collection of vertices and edges, and for a given degree sequence, the vertex count is fixed. Only the edges vary. What we are trying to prove is that all possible edge-collections can be generated under the validity rule.

**Base case.** $D_t = [1, 1]$. We state without proof that this is the lexicographically smallest tree sequence, other than the tree consisting of one vertex and no bonds at all.

The only possible edge here is from $d_1$ to $d_2$. Under the second part of the validity rule, this bond may be formed. Therefore, all possible edge collections can be formed under the validity rule.

**Induction step.** Assume that all possible edge collections of any $D_t = [d_1, d_2, ..., d_n]$, of length $n$, are formable using only bonds permitted by the validity rule. To prove: All possible edge collections of any $D_u = [d_1, d_2, ..., d_{n+1}]$, of length $(n + 1)$, are formable using only bonds permitted by the validity rule.
The only edges allowed are those from those $d_x$ which equal 1 to those $d_y$ which don’t. Without loss of generality, we can take $D_u$ to be non-decreasing and consider $d_1$, which is known to equal 1. Consider any bond formed between $d_1$ and some $d_k$ such that $d_k > 1$. Forming that bond yields $D_{u'} = [0, d_2, \ldots, d_k - 1, \ldots, d_{n+1}]$. By the tree induction lemma, $D_{u'}$ is itself a tree-able sequence of length $n$ (once it’s re-sorted and the leading zero is removed). By our induction hypothesis, all edge collections of the tree-sequence $D_{u'}$ are realizable under the validity rule. The only bond formable in $D_u$ but not in $D_{u'}$, under any rule, is the bond between $d_1$ and $d_k$. This bond is already known to be formable under the validity rule, which means all new edge collections involving that particular bond can be generated under the validity rule. Note that all possible edge collections of $D_u$ involve one $d_1$-to-$d_m$-style bond, along with one complete edge collection of everything else. We’ve proven that all such combinations are possible under the validity rule. $\square$
Fig. 2.3. The validity rule in action. (All vertices labeled by degree.)
2.2. Order of runtime. We now determine the running time of the algorithm overall. To restate:

(0) Check to see if the sequence is tree-able by verifying that (sum of its \( n \)-many degrees) = 2(n - 1).

(1) Recursively form bonds. At every level, try every possibility, each with its own recursive call.

(2) Once the recursion is finished generating tree possibilities, create canonical names for each of them.

(3) Create equivalence classes based on which trees share canonical forms and which do not.

(4) Remove every member but one from each class. One of each kind of non-isomorphic tree will remain.

(5) Return the list of non-isomorphic trees.

The runtime may vary greatly even within this model, for different ways of implementing the steps. For a sequence of length \( n \), Step 0 can be implemented in \( O(n) \) (and, in our final model, it isn’t needed at all). We implemented Step 2 in the \( O(n) \) method of [12]. Step 4 is \( O(1) \); Step 5 is merely a formalism. Far and away, the most time-costly segments are Steps 1 and 3. To restate Step 1:

(1) Recursively form bonds.

(a) Find all pairs of numbers in the sequence which represent a valid potential bond. For all such pairs:

(i) Decrease both numbers by 1.

(ii) Perform a recursive call on the resulting sequence.
(iii) Accept, as a return value, all trees generated by the diminished sequence.

(iv) In all trees returned, add the valid bond pair chosen in (a).

(v) Add these updated trees to the group of trees to return.

(b) Return all trees yielded by (iv).

We adopt the convention of timing notation used in [2]: specifically, we use $T_1(n)$ to refer to the time taken by the entirety of Step 1 on an input of size $n$. Since Step 1 is a recursion, $T_1(n)$ must be defined recursively.

Realistically, we don’t have to try every pair of numbers for (a): simply trying all leaves with all non-leaves suffices. However, in the worst case, we have $n/2$-many leaves and $n/2$-many non-leaves. This results in trying each of $n/2$-many leaves with each of $n/2$-many non-leaves, for $(n/2)^2$-many pairings. This is asymptotically indistinguishable from trying all $n(n - 1)/2$ possible pairings in the sequence. By design, the worst-case for possible leaf-nonleaf pairings is also the worst case for how many valid pairs will be found, and by extension, for how many times steps (i)-(iv) will occur.

Step (i) is constant-time. Step (ii) is a recursion on a sequence of length $(n - 1)$, as we’ve established. Step (iii) is merely a formalism. Steps (iv) and (v), although constant-time in and of themselves, must be applied to every value returned by the call in (ii). Let $r_n$ denote the total number of results from (ii), for a sequence of length $n$. There is a sufficiently large constant $C$ such that the overall running time satisfies:

$$T_1(n) \leq (n^2/4)[C + T_1(n - 1) + r_{n-1}(C)].$$
Before we can even solve the recurrence, we need a value for \( r \), or henceforth \( r_{n-1} \) (since it depends on the total number of results from a call of size \( n - 1 \)).

By definition, \( r_{n-1} \) is the number of realizations returned by one particular call to a sequence of size \( n - 1 \). Of course, that number will be the number of realizations returned by all the calls to sequences of size \( n - 2 \). This forms its own recurrence, with the worst-case being as follows:

\[
  r_2 = 1 \\
  r_n = ([n - 1]/2)^2(r_{n-1})
\]

As discussed, the \(([n - 1]/2)^2\) comes from the theoretical worst-case number of calls of size \([n - 1]\): \((n - 1)/2\)-many leaves paired with \((n - 1)/2\)-many non-leaves. Of course, not all such calls will actually attain the worst-case; some of them may have as few as \((n - 1)\)-many leaves paired with only one non-leaf, for \((n - 1)\)-many calls in total. That said, determining the exact distribution of best-vs.-worst-cases, and their effects on their respective recursive calls, falls outside the scope of this thesis.

By inspection, the number of calls returned for a sequence of length \( n \) is proportional to \(((n - 1)/2)2!\). (In practice, some elements in the factorial sequence are missing, not to mention that \((n - 1)\) isn’t always even, but the asymptotic analysis is the same.)

We can finally solve the recurrence of \( T_1 \). We have:

\[
  T_1(n) \leq (n^2/4)[C + T_1(n - 1) + ((n - 1)/2)2!(C)] \\
  T_1(2) = C \text{ (since we hard-coded the ‘base case’)}
\]
Evidently, this approach is superpolynomial (and this covers only one of the five parts of our original algorithm). To see our improvements on this basic approach, see sections 2.3 and 2.4; here we continue with an analysis of the exact running time.

Claim: \( T_1(n) = O((n^2)!^n) \)

Proof:

Base case: \( n = 2 \).

\[ T_1(2) = \text{some integer constant } C. \]
\[ \leq C((n^2)!^n) \text{ for all } n \geq 2 \]

Induction step: Assume \( T_1(n) \) is \( O((n^2)!^n) \).

Prove: \( T_1(n+1) \) is \( O(((n+1)^2)!^{n+1}) \)

\[ T_1(n+1) \leq ((n+1)^2/4)[C + T_1(n) + (n/2)!^2(C)] \]
\[ \leq ((n+1)^2/4)[C + C((n^2)!^n) + (n/2)!^2(C)] \text{ by hypothesis} \]
\[ \leq [C/4]((n+1)^2)((n^2)!^n) + C(n^2) + CO(n^2!) \text{ by algebra} \]
\[ \leq [C/4]((n+1)^2)((n^2)!^{n-1}) + C(n^2) + CO(n^2!) \text{ by algebra} \]
\[ < [C/4](((n+1)^2)!^{n+1}) \text{ for any } C \geq 4. \] \( \square \)

Steps 2 and 3, the creation and comparison of canonical names, depend on how many realizations are created. As previously discussed, the worst-case number of potentially isomorphic realizations, for an initial sequence of length \( n \), is capped by \( ((n - 1)/2)!^2 \). With an \( O(n) \) method of creating canonical names, we have total name creation taking time \( O(n((n-1)/2)!^2) \). The theoretical worst-case for comparison involves all names being distinct, corresponding to all the realizations
being isomorphic. (This never quite happens, but similar cases occur.) For \( r \) being the number of realizations, the total comparison count is \( r(r - 1)/2 \). With worst-case \( r \) being known, the runtime is \(((n - 1)/2)^2!\). As noted, Step 0 runs in \( O(n) \), and steps 4 and 5 run in time \( O(C) \). None of these dwarf \( T_1 \), so the total runtime is as follows, for a sequence of length \( n \):

\[
O((n^2)!^n) .
\]

It should come as little surprise that algorithm A requires large amounts of time, to say nothing of memory space, to run. Prior to discussing the algorithm’s uses and our results (see Chapter 3), we discuss some improvements.

**2.3. Improvements on the basic algorithm.** The factorials in the runtime expression come from the sheer bulk of potentially-isomorphic realizations generated, which then take time to build off of in step 1, and to create names for in step 2 and compare in step 3. What is intriguing is that most of them are usually unnecessary; the final group of realizations tends to contain at least as many isomorphisms as not. If there were a way to ‘nip these in the bud’, to eliminate those realizations at an earlier stage, the recurrence would take less time, and the final canonical name creation/isomorphism removal would be much quicker, if necessary at all.

Recall that the Havel-Hakimi algorithm, on which our algorithm is based, is a recurrence which creates a single tree by forming one bond and creating a smaller tree from what remains, and so on. Algorithm A tries every possible bond and every possible subtree. The question arises: need we try every single subtree? Could we remove the isomorphic subtrees early on, during one of the subcalls, and still have the same result?
The answer, in theory, is yes. An isomorphism is merely a bijection between two graphs, or trees, in this case. If two trees are isomorphic already, and one new vertex \( v \) is bonded to the “same” vertex on both trees, then the resulting two trees have a new bijection: the original bijection \( B \) trivially extended so that \( B(v_1) = v_2 \). More generally, if two of the sub-trees are isomorphic, adding the same chain of new bonds to each will result in two isomorphic final trees. Therefore, getting rid of one of the isomorphic sub-trees has no effect on the final tree-group, save to get rid of an isomorphism in advance.

This doesn’t work as described, though. The first problem is that running the isomorphism-removal code at every step, rather than only at the end, causes some tree-possibilities to never be generated. In other words, too many possibilities are culled this way. The reason has to do with the fuzziness of our definition of adding bonds to the “same” place in both trees. Our algorithm doesn’t use “platonic” trees; ours are represented in adjacency-list form. A tree is an array of lists, one for each vertex, with each list consisting of that vertex’s neighbors. Each vertex is represented by a number, and the numbering between isomorphic trees can differ. As a result, the “original sequences” of two isomorphic trees can vary. In any subcall, the vertices of a three-vertex tree will always be of degrees 1, 1, and 2, even though, for example, those same vertices may have been of degrees 3, 6 and 7 in the original sequence. All trees with the degree sequence \([1, 1, 2]\) are isomorphic, but in a subcall, dropping all but one such tree can close off possible realizations.

Ideally, to perform isomorphism removal in a given subcall, we would want to have some awareness of what the original sequence was before it was reduced. Equivalently, we want to keep track of the bonds the program already made before
reaching that particular subcall. Keeping track of the original sequence between subcalls can be achieved by adding an extra parameter to the function, representing a version of the original not to be altered. To make use of this information, we devised a way of creating canonical names which is slightly modified from the algorithm in [12]: instead of using 1’s and 0’s, or a’s and b’s, we encode original degree values into the canonical name itself. (The canonical name for the hypothetical in the last paragraph would be 37b6bb, or something similar, instead of aababb.) The result is a different kind of name which contains the distinct patterns used to check for isomorphism, as well as information about the initial sequence. This kind of name can be used to obtain a stronger guarantee than isomorphism, something like isomorphism up to original degree. For our purposes, we’ll call the process itself in-line isomorphism-removal.

The steps of Algorithm B, the algorithm for in-line isomorphism removal, would be as follows:

(0) Check to see if the sequence is tree-able by verifying that (sum of its \( n \)-many degrees) = 2(\( n \) - 1).

(1) Recursively form bonds. At every level, including the first:
   (a) Try every possible valid bond, each with its own recursive call.
   (b) Once all the calls are finished generating tree possibilities, create canonical-original names for each of them.
   (c) Create equivalence classes based on which trees are isomorphic up to original degree and which are not.
   (d) Remove every member but one from each class.
   (e) Return the list of non-isomorphic-up-to-original-degree trees.
We have not been able to prove the correctness of this theoretical algorithm, nor have we been able to design a correct implementation. We have, however, implemented an algorithm which generates canonical names for detecting isomorphism up to original degree; see the Appendix section devoted to our code.

2.4. The tree of trees. We note that Algorithm A, and the theoretical algorithm B, do a lot of repeated calculations. For a degree sequence of length 5, here $D_5$, finding $T(D_5)$, the number of non-isomorphic trees generated by $D_5$, requires some knowledge of all $D_4$ of length 4, and so on. When we want to find $T(D)$ for a single specific $D$, this has little significance. However, we might want to compute every $T(D_n)$ for ascending $n$ (which, for example, we would have to do in order to compute the sequence of $M(n)$ values). In this case, it makes sense to save all the $T(D_{n-1})$ values so as not to have to recalculate them. This approach is an example of dynamic programming. We refer to this version of the algorithm as Algorithm C.

We are faced with the problems of finding a way to store all the values of $T(D_{n-1})$, and of finding the “correct” way to make use of them, when the time comes. By Lemma 1, if we start with a tree-able degree sequence $D_n = (d_1, d_2, \ldots, d_n)$, and we form a single bond between a leaf and a non-leaf, then the resulting sequence $D_{n-1} = (0, d_2, d_3, \ldots, d_k - 1, \ldots, D_n)$ is also tree-able. This recurrence has been the basis of every algorithm proposed so far. The 0 in $D_{n-1}$ is the leaf of the bond, and the $d_k - 1$ is the non-leaf. Once the bond has been made, it’s not hard to rearrange the sequence so that the 0 is first. We may also move $(d_k - 1)$ so that $D_{n-1}$ is in nondecreasing order once more. (For our purposes, we can guarantee that $D_n$ is sorted to begin with. It takes at most two “swaps” to get $D_{n-1}$ back in sorted order: one for the reduced leaf and one for the reduced
non-leaf.) Once $D_{n-1}$ is ordered, if the leading zero is ignored, we know that $D_{n-1}$ is a tree-able sequence. Assuming for now that we have an algorithm that tries every tree-able sequence of length $n$ (which we’ll discuss in Appendix A), and assuming that we run our tree-building algorithm on all $D_2, D_3, \ldots, D_n$, we know that we’ve already computed the value of $T(D_{n-1})$ by the time we want to compute the value $T(D_n)$. If we’ve stored all the values of $T(D_{n-1})$, we can look up all of the realizations corresponding to the sequence $D_{n-1}$ and add the single bond already formed. Thus we obtain all of the realizations corresponding to the sequence $D_n$.

If all values of the form $T(D_{n-1})$ are stored, then values of the form $T(D_{n-2})$ are never needed to compute $T(D_n)$. All dynamic programming approaches involve using the $T(D_{n-1})$ values to compute the $T(D_n)$ values, storing each of the new values as they are computed. They then use these $T(D_n)$ to compute all $T(D_{n+1})$ (while leaving the $T(D_{n-1})$ values to be garbage-collected). Thus, the “brute force” solution is to store every value of $T(D_{n-1})$, but this is a burden on both running time and memory space. (In theory, it’s still faster than algorithm A, but due to memory constraints, the end results are not much more useful.) A slightly more sophisticated solution is to remove all the isomorphisms from the collection $T(D_{n-1})$, and store only those that remain. In theory, this still leaves enough information to compute all realizations of degree sequences of the form $D_n$. Intuitively, given the $T(D_{n-1})$ values, we try all the ways of adding one leaf “all the places a leaf can go”, and remove the isomorphisms, leaving us with $T(D_n)$.

However, we can’t determine “all the places a leaf can go” from $T(D_{n-1})$ alone. We’ve established that $D_{n-1}$, given a leading zero, is $D_n$ with one bond already formed, and two or fewer swaps out of its original order. Since the leading zero
represents the bonded leaf, our task is to find all possible candidates for \((d_k - 1)\), the bonded non-leaf, in \(D_{n-1}\). Intuitively, then, we’re looking for all values \(d_j\) in \(D_{n-1}\) such that \(d_j + 1\) is present in \(D_n\) (and \(d_j > 0\)). We can find all these by inspection on \(D_n\). For all \(d_m\) in \(D_n\) such that \(d_m > 1\), we search \(D_{n-1}\) for all occurrences of \(d_m - 1\). Having found these, we create new tree realizations by bonding leaves to all \(d_m - 1\). Filtering out all the isomorphic realizations leaves us with \(T(D_n)\).

It remains to be proven that this approach creates exactly the right number of possibilities, prior to removal of isomorphisms. In other words, we prove that it neither creates any realizations that don’t correspond to the initial sequence, nor fails to create any of those that do.

**Claim:** Algorithm C as described creates all of the possible realizations of a given degree sequence \(D_n\) (but possibly more).

**Proof:** We use induction.

**Base case:** \(n = 2\). This case is hard-coded into our algorithm; there is only one possible realization.

**Induction step.** Assume that the approach creates, and stores, all possible realizations of any and all degree sequences \(D_n\). To prove: the approach creates all possible realizations of the form \(D_{n+1}\).

Clearly, any tree realization is the product of adding one bond to a pre-existing tree realization (with the base-case being a one-vertex tree.) As a result, all possible realizations of the form \(D_{n+1}\) are the result of adding one bond to a tree
realization of the form $D_{n-1}$. By assumption, we have already created, and have access to, all such realizations. For all $d_m > 1$ in $D_n$, the algorithm as described searches $D_{n-1}$ for all occurrences of $d_m - 1$ and adds “bonds” to these occurrences (creating a new copy of the realization for every new bond formed). Due to our selection of the various $d_m$, we know that the algorithm finds all possible places to form new bonds. As a result, we know that all tree realizations of the form $D_{n+1}$ are created. □

**Claim:** Algorithm C as described does not create any realizations that do not match the original degree sequence $D_n$.

**Proof:** We use induction.

**Base case:** $n = 2$. This case is hard-coded into our algorithm; there is only one possible realization.

**Induction step:** Assume that, when creating and storing the realizations of any degree sequence of the form $d_n$, the approach does not create or store any realizations that do not match that $d_n$. To prove: the approach does not create any realizations that do not match any given sequence of the form $d_{n+1}$.

The approach is known to create only those realizations that are formed from realizations of the form $d_n$ by adding a single bond, between an as-yet nonexistent leaf and some element in the sequence. That element $d_j$ must satisfy the constraint that $d_j + 1$ is present in $D_n$. The entire sequence $d_n$ must satisfy the constraint that all its other elements are present in $d_{n+1}$, and that the only elements contained by $d_{n+1}$ and not $d_n$ are one leaf and $d_j + 1$. By assumption, the only realizations created for a sequence $d_n$ are those that match the sequence. By the constraints,
the only bonds which may be added are those that add the missing leaf, and add one to $d_j$, which causes the new sequence to be identical to $d_{n+1}$. Thus all realizations formed this way are known to match $d_{n+1}$. □

We’ve described the method in which Algorithm C uses the stored results, and we’ve proven that this method is correct. We still need to design a way to actually store these results. We will need accurate storage and fast retrieval. The realizations themselves are stored as Java *arrayLists*, with each *arrayList* representing a single realization in adjacency-list form. These *arrayLists* are double-nested, and since a single degree sequence can have multiple realizations, they must be stored in an *arrayList* of their own. In other words, for any degree sequence, we will need to store and retrieve collections of realizations in the form of triple-nested *arrayLists*, in a structure that recognizes degree sequences as keys. For this purpose, we defined our own structure called a *treeOfTrees*. As implied by the name, the structure is a tree. Each branch contains pointers to either another branch or a leaf, and each leaf contains the realizations of a specific degree sequence. A *treeOfTrees* is stored in its calling context as a pointer to its root. To perform a look-up of a degree sequence $D_n$ in a *treeOfTrees* $t_n$, the last element of the sequence $D_n$, also known as $d_n$, is keyed to the root of $t_n$. $t_n$ will have at most one pointer to a child which is labeled with the number $d_n$. (The root has no label.) We showed above that all realizations of the form $D_n$ will have been created (and inserted into the tree) before they are needed, so the root will have exactly one pointer-child labeled as $d_n$. This child $t_{n-1}$ is searched for a pointer to a child with label $d_{n-1}$, and so on. Eventually, the look-up function will reach a leaf containing all the realizations of $d_n$. Insertion and retrieval, for a degree sequence of length $n$, are $O(n)$. 
We can now analyze the running time of Algorithm C. However, the analysis will be different from those for the previous algorithms. For Algorithm A (or, in theory, even Algorithm B), it makes sense to talk about the time taken to compute the realizations of a single $D_n$. This will recursively involve the time taken for some sequences $D_{n-1}$. In the case of this dynamic programming algorithm, the end goal is not the computation for a single degree sequence $D_n$, but the computation for all such sequences. It’s impossible to compute the realizations of any $D_n$ without having first computed and stored the results for every possible $D_{n-1}$, for reasons which will soon be evident. As a result, though this algorithm accomplishes essentially the same task as the previous two, we need to develop a ‘neat’ way of comparing their running times. For this algorithm, we will provide the time taken to compute all degree sequences of length $n$, as a function of the time taken to compute all sequences of length $n - 1$. The steps are as follows:

(-1) Build the treeOfTrees containing all realizations of degree sequences of the form $d_{n-1}$.

(0) Generate all tree-able sequences of length $n$. For each such sequence:

(1) Find all potential bonds in $d_n$ which bond vertices of degrees 1 and $k > 1$. For each such bond:

(a) Form the bond, by setting the leaf to degree 0 and the non-leaf to degree $k - 1$.

(b) Rearrange the sequence so that it is once again in sorted order (discarding the degree-0 entry).

(c) Search treeOfTrees$_{n-1}$ for an entry corresponding to the rearranged, diminished sequence.

(d) Retrieve all realizations corresponding to the rearranged, diminished sequence. For all such realizations:
(i) Undo the rearrangements of $b$ (recreating the degree-0 entry in the process).

(ii) Find all degrees in the sequence which equal $k - 1$. For each such degree:

(A) Form a copy of the realization selected in $d$.

(B) Form a bond between the selected degree and the degree-0 entry, so that the degree-0 entry becomes a leaf again, and the degree $k - 1$ entry is once again of degree $k$.

(C) Add the updated shallow copy to the list of realizations to be returned.

(2) Once the recursion is finished generating realizations, create canonical names (general, not up-to-original-degree) for each realization in the list.

(3) Create equivalence classes based on which trees share canonical forms and which do not.

(4) Remove every member but one from each class. One of each kind of non-isomorphic tree will remain.

(5) Add the list of non-isomorphic tree realizations to $treeOfTrees_n$, under the index of the original sequence $d_n$.

(6) Repeat steps 1 through 5 until all realizations of all sequences of length $n$ have an entry in $treeOfTrees_n$.

In Step 0, we are assuming the existence of some method to generate every tree-able sequence of length $n$. This particular form of the algorithm won’t work without it; if we fail to compute the realizations for some $d_{n-1}$, then the lookup in (c) might return nothing for some sequence $d_n$. After designing an algorithm to compute all tree-able sequences of length $n$, we discovered that this topic has been
researched extensively [8] [9]. As far as we are aware, our method is essentially the same as one already developed and described. (For those interested, we include our method in Appendix A, along with a proof of its correctness.) However, we do not include the running time of the generation algorithm in the recursion. It isn’t necessary to generate the sequences ‘as we go’. In other words, Algorithm C could first generate all sequences up to a certain length, store them, then refer to the stored sequences as needed (in constant time) while the main body of the algorithm runs. In this case, the time taken to generate all sequences of length $n$ is consistently dwarfed by the time taken to compute all realizations of those sequences. (No version of our algorithm has yet been able to compute the realizations for all the sequences of length 19, regardless of how much time we allow, but we can generate all sequences up to length 40 in less than a second.)

Our current Java implementation does, in fact, generate the sequences as they’re needed, but this contributes a negligible amount to the running time. That said, the number of length-$n$ sequences generated, denoted $c_n$, is an important factor in the runtime analysis, as it determines how many times steps 1 through 5 will occur. Values of $c_n$ have been calculated up to $n = 70$ [8], but relating them to $n$ is more difficult. We return to this later.

Step (a) runs in constant time. For a sequence of length $n$, step (b) can be made to run in time proportional to $O(n)$. (This is possible because the diminished sequence is, at most, two ‘swaps’ out of sorted order; it’s faster to find them by inspection than to utilize a general sorting technique.) Step (c) is $O(n)$ in theory (see the code in Appendix B). Step (i) is constant time, though it requires us to save the swapping information from (b). Step (A) runs in $O(n)$ time. Steps (B) and (C) are constant-time. Step 2 runs in time proportional to however many
realizations were generated thus far, denoted $r$. Specifically, step 2 is $O(nr)$ since the canonical-name-making algorithm is $O(n)$. Step 3 runs in time proportional to $r^2$, since every canonical name might need to be compared to every other name (though in practice this never happens). Step 4 is constant-time. Step 5, in the worst case, is $O(nr)$ (with all $r$ realizations remaining and $O(n)$ time to insert each one into $treeOfTrees_n$). Step 6 is just a formalism. We will say that the number of (potentially isomorphic) realizations generated from a single length-$n$ sequence is an upper bound for the number of (non-isomorphic) realizations generated from one of its length-$(n-1)$ subsequences. Thus, we can say that the number of realizations retrieved in (d) is proportional to $r$ (though Step (d) itself is constant-time).

In Step 1, finding the bonds is $O(n)$. The number of bonds found, though, could be proportional to $n^2$. The algorithm can be modified so that the bond count is proportional to $n$ without changing the results. (This is done, for example, by disregarding any potential bond between a leaf and a degree-$q$ vertex if a bond between any other leaf and any other degree-$q$ vertex has already been selected and evaluated. We’ve yet to prove the correctness of this, so our Java implementation uses the $n^2$ method). We don’t have an easy way to calculate the worst-case number of realizations in (d) (though algorithms have been designed for this task). However, since this algorithm uses all possible results of the form $D_{n-1}$, rather than some of them, to calculate the results for the various $D_n$, it makes sense to talk about the average-case number of realizations, rather than the worst case. For all the sample sizes we’ve been able to compute, the average number of realizations for an arbitrary degree sequence of length $n$ is roughly proportional to $n$. (For larger $n$, the $D_n$ with the maximum number of realizations are always outliers;
the majority of the $D_n$ have fewer than $n$ representations. See Chapter 3, Figure 4.) The number of degrees in Step (ii) which might equal $k - 1$ is proportional to the sequence length $n$, in the worst case (though ii itself is linear-time).

With this, we can finally provide a worst-case value for $r$, the number of realizations generated so far, based on our average-case value for how many realizations a degree sequence of length $n - 1$ is likely to have. There are $n^2 - n = O(n^2)$ potential bonds in step 1 (each of which calls for one set of $D_{n-1}$ realizations), and there are $(n-2) = O(n)$-many possible $D_n$-copies that each one of those $d_{n-1}$-realizations might create in (ii). If the average call to a sequence $D_{n-1}$ returns $n$-many realizations in total, the algorithm could create as many as $n^2(n)(n) = n^4$ realizations of $d_n$, possibly containing isomorphisms, in a single step.

With $r = n^4$, the recurrence is as follows:

$$T(2) = C$$

$$T(n) = T(n - 1) + c_n (\text{Step}_1 + n^2(\text{Step}_a + \text{Step}_b + \text{Step}_c + \text{Step}_d + r(\text{Step}_i + \text{Step}_i + n(\text{Step}_A + \text{Step}_B + \text{Step}_C)))) + \text{Step}_2 + \text{Step}_3 + \text{Step}_4 + \text{Step}_5)$$

Many of these values are constant. For $C$ being the greatest of them, the following is true:

$$T(n) \leq T(n - 1) + c_n (n + n^2(C + n + n + C + r(C + n + n(n + C + C))) + nr + r^2 + C + nr)$$

$$T(n) \leq T(n - 1) + c_n (n + n^2(n + n^4(n + n(n))) + n(n^4) + (n^4)^2 + n(n^4))$$

$$T(n) \leq T(n - 1) + c_n (n + n^2(n + n^4(n + n^2)) + n(n^4) + (n^4)^2 + n(n^4))$$
\[ T(n) \leq T(n-1) + c_n(n + n^2(n + n^5 + n^6) + n(n^4) + (n^4)^2 + n(n^4)) \]

\[ T(n) \leq T(n-1) + c_n(n + n^3 + n^7 + n^8 + n^5 + n^8 + n^5) \]

\[ T(n) \leq T(n-1) + c_n(n^8) \]

By inspection, then, \( T(n) \) is \( O(c_n(n^9)) \).

It remains to find an exact value for \( c_n \), the number of sequences \( D_n \) which exist for some length \( n \), and which this algorithm realizes. But first: we finally have a ‘neat’ way of comparing Algorithm C with algorithm A. Algorithm A only computes the realizations of one \( D_n \), but it can be trivially extended to compute the realizations of all \( D_n \). This is done by generating all such sequences and iterating over them. To accomplish this task, Algorithm A (and, in theory, Algorithm B) take time proportional to their single-sequence time multiplied by \( c_n \), the number of sequences available. In other words, they are \( O(c_n(n^2)!^n) \) for this task. This is the same task that Algorithm C is used for, which means that Algorithm C, at \( O(c_n(n^9)) \), is much faster. There is no distinct way for Algorithm C to do single-sequence calculations. In order to get one sequence’s result, this algorithm would calculate the results for the desired sequence, and all others of the same length, then print one particular set of realizations from the current treeOfTrees.

There are a few ways to relate \( c_n \) to \( n \). We can establish a (weak) theoretical upper bound as follows:

Finding the number of sequences \( D_n \) which exist for some length \( n \) is a partition problem, in which we are asked to partition \( 2(n - 1) \) degrees into \( n \) slots.
Combinatorically, we have \((n - 1)\)-many choices to make (after which the \(n\)th slot will be fixed by our previous choices). As noted above, we can’t choose any single element to be “too large”, lest we run out of degrees too early. For each choice, our options are anything between 1 and \((n - 1)\), inclusive, for \((n - 1)\)-many options total. Choosing between these options \((n - 1)\)-many times yields at most \((n - 1)^{(n-1)}\)-many possibilities.

As mentioned earlier, Knuth [18] proved a stronger, if more complex, upper bound for \(c_n\). We observe that the number of sequences of length \(n\) must be less than or equal to the number of non-isomorphic trees on \(n\)-many vertices. (There will usually be more than one such realization per sequence, but by Lemma 1, there must exist at least one per sequence.) Thus \(c_n\) is bounded above by the number of non-isomorphic trees on \(n\)-many vertices, which in turn is bounded by \(2.956^n/(n^{5/2})\).

Thus we can write the running time as \(O(2.956^n(n^{13/2}))\). This Algorithm \(C\) is the asymptotically fastest we’ve produced so far.
3. Results and discussion

We have calculated $RM(n)$, the maximum number of non-isomorphic realizations attainable by any degree sequence of length $n$, for $n$ up to 18. For $n = 0$ and $n = 1$, there is only one degree sequence and one corresponding tree (the empty tree and a one-node tree respectively).

<table>
<thead>
<tr>
<th>$n$</th>
<th>$RM(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>73</td>
</tr>
<tr>
<td>13</td>
<td>174</td>
</tr>
<tr>
<td>14</td>
<td>364</td>
</tr>
<tr>
<td>15</td>
<td>759</td>
</tr>
<tr>
<td>16</td>
<td>1859</td>
</tr>
<tr>
<td>17</td>
<td>4177</td>
</tr>
<tr>
<td>18</td>
<td>8715</td>
</tr>
</tbody>
</table>
This sequence does not, as of yet, appear in the On-Line Encyclopedia of Integer Sequences (https://oeis.org/).

We have also computed the sequences which attain these maxima. Starting at $n = 2$, they are as follows:

<table>
<thead>
<tr>
<th>$n$</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{1,1}</td>
</tr>
<tr>
<td>3</td>
<td>{1,1,2}</td>
</tr>
<tr>
<td>4</td>
<td>{1,1,1,3} and {1,1,2,2}</td>
</tr>
<tr>
<td>5</td>
<td>{1,1,1,1,4}, {1,1,1,2,3}, and {1,1,2,2,2}</td>
</tr>
<tr>
<td>6</td>
<td>{1,1,1,2,2,3}</td>
</tr>
<tr>
<td>7</td>
<td>{1,1,1,2,2,2,3}</td>
</tr>
<tr>
<td>8</td>
<td>{1,1,1,1,2,2,3,3}</td>
</tr>
<tr>
<td>9</td>
<td>{1,1,1,1,2,2,2,3,3}</td>
</tr>
<tr>
<td>10</td>
<td>{1,1,1,1,1,2,2,2,3,4} and {1,1,1,1,2,2,2,2,3,3}</td>
</tr>
<tr>
<td>11</td>
<td>{1,1,1,1,1,1,2,2,2,2,3,4}</td>
</tr>
<tr>
<td>12</td>
<td>{1,1,1,1,1,1,1,2,2,2,2,3,3,4}</td>
</tr>
<tr>
<td>13</td>
<td>{1,1,1,1,1,1,1,2,2,2,2,2,3,3,4}</td>
</tr>
<tr>
<td>14</td>
<td>{1,1,1,1,1,1,1,2,2,2,2,2,2,3,3,4}</td>
</tr>
<tr>
<td>15</td>
<td>{1,1,1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,4}</td>
</tr>
<tr>
<td>16</td>
<td>{1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,4}</td>
</tr>
<tr>
<td>17</td>
<td>{1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,4}</td>
</tr>
<tr>
<td>18</td>
<td>{1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,3,3,3,4}</td>
</tr>
</tbody>
</table>

We note that there appears to be a similarity in the sequences for higher values of $n$, though we’ve yet to find a determining pattern. The only values of
\( n \) for which multiple distinct sequences attained the maximum were 4, 5, and 10. Whether there exist others is currently unknown. We also note that, for \( n = 4 \) and 5, no tree has more than one realization. Only \( n = 10 \) has a split maximum where the maxima are greater than 1.

We’ve also calculated the distributions of the realizations: in other words, we know how many sequences of length \( n \) have one realization, how many have two, how many have three, and so on. See Figure 4 for plots of the distributions for \( n \) between 6 and 12. Our Java implementations for Algorithms A and C both output the distribution results for every \( n \) as it is calculated (see our documentation).
Fig. 4. The distributions of the sequences and their realizations.

**x-axis:** the number of realizations the sequences have.

**y-axis:** the number of sequences with $x$-many realizations.
We have sample timing results for Algorithms A and C, specifically the time taken to calculate all realizations of size \( n \), for ascending \( n \). (Both algorithms ultimately exceeded our IDE’s capacity for memory space.) All times are given in milliseconds elapsed since the overall start time.

<table>
<thead>
<tr>
<th>Sequence length</th>
<th>Time of A</th>
<th>Time of C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>39</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>115</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>568</td>
<td>40</td>
</tr>
<tr>
<td>11</td>
<td>3664</td>
<td>81</td>
</tr>
<tr>
<td>12</td>
<td>38545</td>
<td>153</td>
</tr>
<tr>
<td>13</td>
<td>(crashed at around 301000)</td>
<td>553</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>1218</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>2473</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>7457</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>27878</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>128358</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>(crashed at around 672000)</td>
</tr>
</tbody>
</table>

These times were obtained on a Windows 10 Enterprise 2015 (a 3.40 GHz processor).
4. Conclusion

We have presented two algorithms for determining the number of non-isomorphic tree realizations of any tree-able sequence. Both run in super-polynomial time, though the second is far faster than the first in practice. For $n$ up to 18, we’ve found the sequences of length $n$ which have at least as many realizations as any other. We’ve calculated the list of these maximum-realization counts, denoted $RM(n): \{1, 1, 1, 1, 2, 3, 5, 9, 17, 33, 73, 174, 364, 759, 1859, 4177, 8715\}$. This does not, as of yet, appear in the On-Line Encyclopedia of Integer Sequences. Ways of moving forward might include providing a functional implementation of Algorithm B, the algorithm with in-line isomorphism removal (which is unlikely to outstrip Algorithm C, but might be faster for specific sub-cases). Having listed out the $D_n$ of various $n$ with the most realizations of all sequences of their size, and having noticed the beginnings of a pattern in the list, we are curious to know if the pattern continues on for higher values of $n$. If so, there might be a way of proving that the pattern always holds. We would also like to find out if any $D_n$ of higher $n$ turn out to have two (or more) maximum-realization sequences; so far, we’ve found this to be true only of $n = 4, 5, \text{ and } 10$. 
Bibliography


Appendix A: Sequence generation

At a relatively early stage in this project, we found the need for a method of generating all tree-able sequences of length $n$. The code we developed to do so is below, along with proofs of correctness and running time. We discovered only later that this problem has been looked into before, by Dimitrov [8], as well as Kelleher and O’Sullivan [9], among others.

Our code is as follows:

```java
public static ArrayList<ArrayList<Integer>> seqMaker(
    int nodeCount, int totalDegrees, int prevCap) {
    // generates all tree-able degree sequences with
    // as many nodes as nodeCount
    // totalDegrees should originally be set to 2 * (nodeCount - 1)
    // and prevCap should originally be set to nodeCount, though it later
    // varies independently of nodeCount
    // nodeCount <= totalDegrees < 2(nodeCount - 1) over all invocations
    // also all trees have at least two leaves
    ArrayList<ArrayList<Integer>> conseguir = new ArrayList<ArrayList<Integer>>();
    ArrayList<Integer> container = new ArrayList<Integer>();
    if (nodeCount == totalDegrees) {
        // if this invocation of seqMaker has reached a point
        // where the remainder of the sequence can only be filled
        // with leaves (i.e. if we have have exactly as many degrees
        // remaining as we have places to put them)
        int degreeCount = 0;
        while (degreeCount < totalDegrees) {
            container.add(1);
            degreeCount++;
        }
        conseguir.add(container); // only one way of filling in the rest,
        return conseguir; // so only one sub-realization is returned
    }
   ...
```
int capClone = prevCap;
ArrayList<ArrayList<Integer>> holdStuff; // this will hold our options
ArrayList<Integer> holdNest; // this will hold a single option

// capClone is essentially a way of conveying to the subcalls what's already been tried. We don't want any subcall trying any degrees larger than whatever value of [capClone] we pass to it
while (capClone > 1) {
  if (!((totalDegrees - capClone) < (nodeCount - 1))) {
    // basically: in the recursive calls, there's going to be exactly [nodeCount - 1]-many slots to be filled, so we need at least [nodecount - 1]-many degrees left to fill them. The recursive calls must not try to include a single degree which is "too big",
    // and we choose capClone accordingly to prevent it.
    holdStuff = seqMaker((nodeCount - 1), (totalDegrees - capClone), capClone);
    // get the recursive result in [holdStuff]
    int holdSize = holdStuff.size();
    int holdCount = 0;
    while (holdCount < holdSize) {
      holdNest = new ArrayList<Integer>();
      singleClone(holdStuff.get(holdCount), holdNest);
      // [holdStuff] will consist of a list of possibilities
      // and we use [holdNest] to create independent clones
      // for each of them
      holdNest.add(capClone); // and, on to the end of each, we add the single degree which we selected in the context of this call. Then we add [holdNest] to the return group conseguir.add(holdNest);
      holdCount++;
    }
  }
  capClone--;
}
The algorithm is structured as a recursion, such that each recursive call may create multiple subcalls. Each recursive call has the sequence “built” one entry further than does its parent; eventually, all subcall-chains will terminate when the sequence-chains reach the desired length. Each recursive call is created with three parameters. One is nodeCount, which denotes how many entries have yet to be filled. (In other words, it’s the desired sequence length minus the number of recursive calls that have occurred before this one). For the initial call, it is set to the desired sequence’s length. The second is totalDegrees, which denotes how many of the possible degrees in the sequence have already been “used”. It is initially set to $2(n - 1)$, for $n$ being the desired sequence length; it is strictly decreasing between a call and its subcalls. The final parameter is prevCap, which is there mainly because we wanted the sequences to be generated in sorted order (specifically non-decreasing order). No subcall is allowed to create degrees larger than were entered in its parent call (as this might cause a decrease in the sequence), and prevCap, initially set to $n$, is used so that each subcall can tell the lowest degree entered so far, which it may not exceed.

The steps are as follows:

1. Check to see if totalDegrees is equal to nodeCount. If it is, fill the remainder of the sequence with leaves and return it.
2. Otherwise, while prevCap is greater than 1 (with totalDegrees guaranteed greater than nodeCount):
(a) Decide if the current value of \(\text{prevCap}\) is a valid choice for the next degree in the sequence. If it is:

(i) “Select” the current value of \(\text{prevCap}\) as the next degree in the sequence.

(ii) Perform a recursive call to determine all ways of filling in the rest of the sequence (in light of our having already selected \(\text{prevCap}\)). For all such ways:

(A) Add \(\text{prevCap}\) as the newest value.

(B) Add the updated sequence to the group of sequences to be returned.

(3) Return the group of sequences generated thus far.

If \(\text{totalDegrees}\) ever becomes less than \(\text{nodeCount}\) (or, in other words, if we’re left with more spaces in the sequence than remaining degrees with which to fill them), then we cannot complete the sequence. The validity check in a only allows subcalls for which \(\text{totalDegrees}\) is equal to, or greater than, \(\text{nodeCount}\). (If the two are equal, in any recursive call, the only way to fill the remainder of the sequence is by allocating one degree to each slot and returning this lone possibility, as is done in 1.) The opposite problem is also prevented. In theory, we might select a degree which is ‘too small’, thereby forcing later calls to use larger degrees to finish the sequence (thereby ruining the non-decreasing character of the sequence). However, the sequence is not allowed to ‘select’ leaf vertices; leaves may be added only via 1. The smallest ‘selection’ the algorithm may make is to select a degree-2 as its first choice (and, with this being the new \(\text{prevCap}\), it will choose degree-2 vertices in all subsequent calls until 1 is invoked). It is generally known that, for any \(n\), there exists a tree-able sequence of length \(n\) consisting
of \((n - 2)\)-many vertices of degree 2, and two leaves. Therefore, we know that making even the smallest possible degree choice in the earliest possible moment will lead to a valid result.

The recursive call in (ii) is updated to “reflect” our choice of prevCap’s current value as the next node in the sequence, by decreasing the remaining nodeCount by 1, decreasing the remaining degrees to be allocated by prevCap itself, and by having the recursive call not allocate any degrees higher than prevCap (to preserve the sequence’s non-decreasing character). The chosen sequence element is added on to the end of all the possibilities returned by the recursive call, rather than the beginning.

To compute all sequences of length \(n\), an outside context should call the algorithm with nodeCount and prevCap equal to \(n\), and totalDegrees equal to \(2(n - 1)\). Henceforth, we will assume that this is true of the initial call.

**Generation lemma I:** All sequences generated by a call to the algorithm have a degree sum of whatever value of totalDegrees that particular call was given.

**Base case:** totalDegrees = nodeCount

In this case, the algorithm returns a list of nodeCount-many leaves, as per line 1. These leaves sum to totalDegrees.

**Induction step:** Assume that, in cases where totalDegrees = nodeCount + \(k\), \(0 < k < m\) inclusive, for some integers \(k\) and \(m\), all sequences generated by a call to the algorithm have a degree sum of whatever value of totalDegrees that particular call was given. To prove: in any case where totalDegrees = nodeCount
+ $m + 1$, all sequences generated by a call to the algorithm have a degree sum of whatever value of $totalDegrees$ that particular call was given.

By definition, Line 1 doesn’t apply. As per the validity check in (a), any recursive calls will select a new degree $d$, and so create a new value of $totalDegrees - d$, to use going forward, which is less than $m + 1$ but greater than $nodeCount - 1$. By the induction hypothesis, these recursive calls will return sequences with a degree sum of $totalDegrees - d$. With the new degree $d$ added on, the total degree sum is $totalDegrees - d + d = totalDegrees$. □

Therefore, if all outside calls to the algorithm are given with $nodeCount$ and $prevCap$ equal to some $n$, and $totalDegrees$ equal to $2(n - 1)$, we know that the sequences returned will have a degree sum of $2(n - 1)$. By Lemma 1, this means that all sequences returned will be valid tree-able sequences.

**Generation lemma II:** The algorithm generates every possible sequence of length $nodeCount$ and degree sum $totalDegrees$.

**Base case:** $nodeCount = totalDegrees$

There is only one possible sequence of length $nodeCount$ with degree sum $totalDegrees$, which is a sequence of leaves. Line 1 guarantees that this sequence will be returned.

**Induction step.** Assume that, in cases where $totalDegrees = nodeCount + k$, $0 < k < m$ inclusive, for some integers $k$ and $m$, the algorithm generates every possible sequence of length $nodeCount$ and degree sum $totalDegrees$. To prove:
in any case where \( \text{totalDegrees} = \text{nodeCount} + m + 1 \), the algorithm generates every possible sequence of length \( \text{nodeCount} \) and degree sum \( \text{totalDegrees} \).

As long as \( \text{totalDegrees} > \text{nodeCount} \), there is no possible sequence where the highest value is a leaf. (This would require the entire sequence to be made of leaves, which would not allow it to sum to \( \text{totalDegrees} \).) The algorithm tries every possible candidate for the next node in the sequence, as long as it isn’t a leaf (not in this case, at least) and as long as it leaves enough degrees to complete the sequence. Thus, all degree candidates ranging between 2 and \( (\text{totalDegrees} - \text{nodeCount} + 1) \), inclusive, are tried in some recursive call. (The only way some of these candidates may be prohibited by \( \text{prevCap} \) is if they’ve been tried in some other recursive call.) Any selected degree candidate \( d \) will lead to another recursive call with parameters \( \text{totalDegrees} - d \) and \( \text{nodeCount} - 1 \).

Thanks to the restrictions on \( d \), \( \text{totalDegrees} - d = \text{totalDegrees} - 2 \) at most, and at least \( \text{totalDegrees} - d = \text{totalDegrees} - (\text{totalDegrees} - \text{nodeCount} + 1) = \text{nodeCount} - 1 = (\text{the new value of nodeCount} + 0) \). Therefore, the induction hypothesis applies to every recursive case. Thus we have that every recursive call returns every possible sequence of its length and its value of \( \text{totalDegrees} \). We’ve already established that every possible candidate for the ‘first’ chosen degree is tried, which leads to every possible recursive call being tried. Therefore, if the recursive calls return every possibility, we have that the algorithm generates every possible sequence of length \( \text{nodeCount} \) and degree sum \( \text{totalDegrees} \).

Therefore, if all outside calls to the algorithm are given with \( \text{nodeCount} \) and \( \text{prevCap} \) equal to some \( n \), and \( \text{totalDegrees} \) equal to \( 2(n - 1) \), we know that every possible tree-able sequence of length \( \text{nodeCount} \) will be generated.
Prior to proving the next lemma, we would like to point out part of the purpose of \texttt{prevCap}. In the algorithm, its value is copied into another variable \texttt{capClone}, which is used to select all possible values of the next candidate degree, which may never exceed \texttt{capClone}. Whatever degree value is selected becomes that particular recursive call’s value of \texttt{prevCap}, and so guarantees that all subsequent degrees chosen will never exceed \texttt{capClone}. Thus we have that the sequences will be in sorted order.

\textit{Generation lemma III}: For given values of \texttt{nodeCount} and \texttt{totalDegrees}, the algorithm never generates more than one copy of the same sequence.

\textit{Base case: \texttt{nodeCount} = \texttt{totalDegrees}}

There is only one possible sequence in this case, and it is returned by line 1.

\textit{Induction step}. Assume that, in cases where \texttt{totalDegrees} = \texttt{nodeCount} + \textit{k}, \textit{0 < k < m} inclusive, for some integers \textit{k} and \textit{m}, the algorithm never generates more than one copy of the same sequence. \textit{To prove:} in any case where \texttt{totalDegrees} = \texttt{nodeCount} + \textit{m} + 1, the algorithm never generates more than one copy of the same sequence.

Within a single step, the algorithm never generates two identical recursive calls. Every recursive call will get a \texttt{nodeCount} value which is one less than that which the calling context was given, but they will have different \texttt{totalDegrees} values, since each call will be the result of a different ‘candidate’ degree \textit{d} being ‘tried’, or in other words, subtracted from the calling context’s value of \texttt{totalDegrees}. By the logic established in the previous proof, all of these recursive calls are examples of the cases to which the induction hypothesis applies, so we know that
no individual recursive call returns duplicate results. In the calling context, each sequence returned by the same recursive call will receive the same ‘new’ degree value $d$ (which still doesn’t make them identical). Sequences from two different recursive calls will receive two separate ‘new’ degrees $d_1$ and $d_2$ in Step A. Once these are appended, two such sequences may have the same degree sum (as, in the end, they all will). However, we know that $d_1$ does not equal $d_2$, and we also know that $d_1$ and $d_2$ are the greatest degrees in their sequences (so far). Thus, the sequences will have different maximum-values and cannot be identical. Therefore we have that no two sequences in any calling context, regardless of whether they are from the same recursive call or different recursive calls, can be identical. □

To restate the steps of the sequence generation algorithm:

(1) Check to see if $totalDegrees$ is equal to $nodeCount$. If it is, fill the remainder of the sequence with leaves and return it.

(2) Otherwise, while $prevCap$ is greater than 1 (with $totalDegrees$ guaranteed greater than $nodeCount$):

(a) Decide if the current value of $prevCap$ is a valid choice for the next degree in the sequence. If it is:

(i) “Select” the current value of $prevCap$ as the next degree in the sequence.

(ii) Perform a recursive call to determine all ways of filling in the rest of the sequence (in light of our having already selected $prevCap$). For all such ways:

(A) Add $prevCap$ as the newest value.

(B) Add the updated sequence to the group of sequences to be returned.
(3) Return the group of sequences generated thus far.

The check in step 1 is constant time, and the potentially-resulting operation is linear in \( n \), the length of the input. In step 2, `prevCap` may be as high as \( n \). (The highest degree selectable is \( (n - 1) \).) The check in step \( a \) is constant-time. Step \( i \) is a formalism of constant time. Step \( A \) is constant-time, as are step \( B \) and step 3. The number of ‘ways’ of filling in the rest of the sequence, however, is harder to derive. In the best case, there may only be one way to fill the remainder (in which case Step 1’s check will create the return sequence in the recursive call). On the other hand, the worst case number of returned realizations may be proportional to \( c_{n-1} \), the number of sequences \( D_{n-1} \) which exist for some length \( n - 1 \). As in Chapter 3, the value is bounded by \( 2.956^n / (n^{5/2}) \).

The recurrence is as follows:

\[
T(2) = C, \text{ since Step 1 will always be invoked.}
\]

\[
T(n) = Step_1 + n(Step_a + Step_i + Step_{ii} + c_n(Step_A + Step_B)) + Step_3
\]

\[
T(n) = n + n(C + T(n-1) + c_n(C + C)) + C
\]

\[
T(n) = n + n(C + T(n-1) + c_n) + C
\]

\[
T(n) = n + Cn + nT(n-1) + n(c_n) + C
\]

\[
T(n) = n + nT(n-1) + n(c_n) + C
\]

By inspection, then, the running time is \( O(n!(c_n)) \), or in other words, \( O(n!(2.956^n)) \). While this would appear inefficient, the algorithm always outperforms our actual
tree-realization algorithms. For theoretical methods of constructing faster generation algorithms, see Kelleher and O’Sullivan [10].
Appendix B: Java implementations

This project comprises just over 1,500 lines of Java code. The full file can be found at:
https://github.com/manofdalsaca/thesis-code/blob/master/GraphCode.java

Below, we reproduce some sections of particular interest.

The `treeOfTrees` class, and the external tree-searching function:

```java
class treeOfTrees {
    // This is a data structure for storing realizations of tree-able degree
    // sequences, indexed by the sequences themselves. Insertion and retrieval
    // should be linear-time, relative to the length of the sequence.
    public ArrayList<treeOfTrees> offspring;
    // the list of children in the tree. Will be null if this node
    // contains a sequence and its corresponding realizations.
    public int childID;
    // the integer representing what "number" in the sequence corresponds to
    // this node. For leaves, will always be the first node in the corresponding
    // sequence.
    public ArrayList<ArrayList<ArrayList<Integer>>> listing;
    // for a leaf, the list of realizations corresponding to the degree sequence
    // represented by the leaf. Null for all others.

    public treeOfTrees(int newChildID) {
        // for creating a branch node, rather than a leaf.
        // Will only be called once by an outside context, for the root;
        // the other branches will build themselves via bindNewSequence.
        offspring = null;
        // will be filled in later
        listing = null;
        childID = newChildID;
    }

    public treeOfTrees(int[] newSeqID, ArrayList<ArrayList<ArrayList<Integer>>> toStore) {
```
// for creating a leaf node, rather than a branch. Should not need to
// be called at all by an outside context; will be called instead by
// bindNewSequence.
offspring = null;
listing = toStore;
childID = newSeqID[0];
}

public void addChild(treeOfTrees toAdd) {
    // add the specified node as a child to the current node
    if (this.offspring == null) {
        this.offspring = new ArrayList<treeOfTrees>();
    }
    this.offspring.add(toAdd);
}

public void bindNewSequence(int[] newSequence, int curIndex,
    ArrayList<ArrayList<ArrayList<Integer>>> newStorage) {
    // For inserting a sequence into its proper position in the tree,
    // and its realizations along with it.
    // Inputs: the sequence, and a value representing the current index in
    // the sequence (initially curIndex.length - 1, eventually 0), and of
    // course the list of representations to be inserted for [newSequence].
    if (curIndex == 0) {
        // if we've reached the correct position in the tree and it's
        // time to create the storage leaf
        treeOfTrees nuovoChild = new treeOfTrees(newSequence, newStorage);
        this.addChild(nuovoChild);
        return;
    }
    // otherwise: we need to find the next child branch to recurse on
    // or, if there is none yet, create such a child branch
    int toFindOrCreate = newSequence[curIndex];
    treeOfTrees novaChild;
    if (this.offspring == null) { // if no child branches exist yet
        novaChild = new treeOfTrees(toFindOrCreate);
        novaChild.bindNewSequence(newSequence, curIndex - 1, newStorage);
this.addChild(novaChild);
return;
}

int toSearch = this.offspring.size();
int searchCount = 0;
boolean doneVal = false;
while (searchCount < toSearch && !doneVal) {
    if (this.offspring.get(searchCount).childID == toFindOrCreate) {
        // if the desired child branch does exist and we’ve found it
        doneVal = true;
        this.offspring.get(searchCount).bindNewSequence(newSequence,
                   curIndex - 1, newStorage);
    }
    searchCount++;
}

if (!doneVal) { // if some child branches already exist, but not the
    // one we were looking for
    novaChild = new treeOfTrees(toFindOrCreate);
    novaChild.bindNewSequence(newSequence, curIndex - 1, newStorage);
    this.addChild(novaChild);
}

public static ArrayList<ArrayList<ArrayList<Integer>>> searchTheTree(
    int [] orderedSeq, treeOfTrees toFind, int curPlace) {
    // the linear-time retrieval algorithm for class treeOfTrees,
    // to return the realizations corresponding to the desired degree
    // sequence. We decided to have this function be separate from class
    // treeOfTrees.
    // Inputs: the sequence to search for, and the treeOfTrees to search,
    // and a placeholder value (initially 0, eventually the length of
    // [orderedSeq] - 1)
    // Outputs: the realizations corresponding to [orderedSeq], or
// a [null] pointer if [orderedSeq] hasn't been properly inserted into
//[toFind]. (This will never happen in the contexts we’ve created.)

if (toFind.listing != null) {return toFind.listing;}

// if we're already at the leaf
// [offspring] is of the type ArrayList<treeOfTrees>
int offCount = 0;
int offMany = toFind.offspring.size();
int toMatch = orderedSeq[curPlace];
while (offCount < offMany) {
    if (toMatch == toFind.offspring.get(offCount).childID) {
        return searchTheTree(orderedSeq,
                           toFind.offspring.get(offCount), curPlace - 1);
    }
    offCount++;
}
System.out.println("Child not found error"); //should never get here
System.out.println("811");
return toFind.listing;
The main recursive body of the dynamic programming algorithm (Algorithm C):

```java
class TreeNode { // A class representing a node in the tree.
    int value;
    TreeNode left, right;
    public TreeNode(int value) {
        this.value = value;
        left = right = null;
    }
}
```

```java
public static ArrayList<ArrayList<ArrayList<Integer>>> treeThree(
    ArrayList<ArrayList<Integer>> inputs,
    int[] degreeSeq, int[] origSeq, treeOfTrees reference) {
    // Our (correct) implementation of Algorithm C.
    // Takes a degree sequence and returns its realizations.
    // inputs: the degree sequence and an empty tree holder
    // and the original sequence, not modified between calls
    // not to mention a tree-of-trees so that some previously computed
    // information can be saved
    // outputs: the list of all non-isomorphic trees matching the
    // input degree sequence
    int counter = 0;
    int countup;
    int[] outClone;
    ArrayList<ArrayList<ArrayList<Integer>>> sendback =
        new ArrayList<ArrayList<ArrayList<Integer>>>();
    // to contain the results. You’ll see
    while (counter < degreeSeq.length && degreeSeq[counter] == 0) {
        counter++;// finding the first non-zero entry in the degree sequence
    }
    if (counter == degreeSeq.length) {
        // if the entire degree sequence is zeroes
        sendback.add(inputs);
        // will always be adding a blank array list, while it calls downwards
        return sendback;
    }
    ArrayList<ArrayList<Integer>> baseClone;
    if (counter == (degreeSeq.length - 2)) {
        // if almost the entire degree sequence is zeroes
        // and the remainder is a trivial base case which we can hardcode
        baseClone = new ArrayList<ArrayList<Integer>>());
```
cloneMake(inputs, baseClone);
if (degreeSeq[counter] != 1) {System.out.println("Error");
    System.out.println("971");
}
if (degreeSeq[counter + 1] != 1) {System.out.println("Error");
    System.out.println("974");
}
baseClone.get(counter).add(counter + 1);
baseClone.get(counter + 1).add(counter);
sendback.add(baseClone);
return sendback;
}

ArrayList<ArrayList<Integer>> resultClone =
    new ArrayList<ArrayList<Integer>>(());
int resultCount;
countup = counter + 1;
int firstSwap = -1;
int secondSwap = -1;
int targetDegree;
boolean swapped;
int[] manyTargets;
int targetCount;
int curTarget;
boolean fini;
while (countup < degreeSeq.length) {
    if (isValidPair(degreeSeq, counter, countup)) {
        // if we've found a valid link to make
        outClone = degreeSeq.clone();
        outClone[counter] = outClone[counter] - 1;
        outClone[countup] = outClone[countup] - 1;
        // form the link in our degree sequence
        targetDegree = outClone[countup]; // and keep track of the
        // newly reduced degree of the node we just bonded to
        ArrayList<ArrayList<Integer>> results =
// treeThree(inputs, outClone, origSeq);

treeLookUp(outClone, reference);

// and get back all the ways of making trees out of what’s left
resultCount = 0;
while (resultCount < results.size()) {
    // for all the ways of making trees out of what’s left:
    resultClone = new ArrayList<ArrayList<Integer>>();
    cloneMake(results.get(resultCount), resultClone);
    manyTargets = howManyOfSize(resultClone, targetDegree, counter);
    targetCount = 0;
    fini = false;
    while (targetCount < manyTargets.length & !fini) {
        resultClone = new ArrayList<ArrayList<Integer>>();
        cloneMake(results.get(resultCount), resultClone);
        curTarget = manyTargets[targetCount];
        if (curTarget == -1) { fini = true; }
        if (!fini) {
            resultClone.get(curTarget).add(counter);
            resultClone.get(counter).add(curTarget);
            sendback.add(resultClone);
            targetCount++;
        }
    }
    resultCount++;
}
}
countup++;
return sendback;
The main recursive body of the brute-force algorithm:

```java
public static ArrayList<ArrayList<ArrayList<Integer>>> treeTwo(
    ArrayList<ArrayList<ArrayList<Integer>>> inputs, 
    int[] degreeSeq, int[] origSeq) {
    // the function for Algorithm A (and the framework for B).
    // Inputs: the degree sequence and an empty tree holder
    // and the original sequence, not modified between calls
    // Outputs: the list of all non-isomorphic trees matching the
    // input degree sequence
    int counter = 0;
    int countup;
    int[] outClone;
    ArrayList<ArrayList<ArrayList<Integer>>> sendback =
        new ArrayList<ArrayList<ArrayList<Integer>>>();
    // to contain the results. You’ll see
    while (counter < degreeSeq.length && degreeSeq[counter] == 0) {
        counter ++; // finding the first non-zero entry in the degree sequence
    }
    if (counter == degreeSeq.length) {
        // if the entire degree sequence is zeroes
        sendback.add(inputs);
        // will always be adding a blank arrayList, while it calls downwards
        return sendback;
    }
    ArrayList<ArrayList<Integer>> baseClone;
    if (counter == (degreeSeq.length - 2)) {
        // if almost the entire degree sequence is zeroes
        // and the remainder is a trivial base case which we can hardcode
        baseClone = new ArrayList<ArrayList<Integer>>();
        cloneMake(inputs, baseClone);
        if (degreeSeq[counter] != 1) {System.out.println("Error");}
        if (degreeSeq[counter + 1] != 1) {System.out.println("Error");}
        baseClone.get(counter).add(counter +1);
        baseClone.get(counter+1).add(counter);
    }
```
sendback.add(baseClone);
return sendback;
}

ArrayList<ArrayList<Integer>> resultClone =
    new ArrayList<ArrayList<Integer>>()
int resultCount;
countup = counter + 1;
while (countup < degreeSeq.length) {
    if (isValidPair(degreeSeq, counter, countup)){
        // if we've found a valid link to make
        outClone = degreeSeq.clone();
        outClone[counter] = outClone[counter] - 1;
        outClone[countup] = outClone[countup] - 1;
        // form the link in our degree sequence
        ArrayList<ArrayList<ArrayList<Integer>>> results =
            treeTwo(inputs, outClone, origSeq);
        // and get back all the ways of making trees out of what's left
        resultCount = 0;
        while (resultCount < results.size()) {
            // for all the ways of making trees out of what's left:
            resultClone = new ArrayList<ArrayList<Integer>>()
            cloneMake(results.get(resultCount), resultClone);
            // make a clone of that particular way
            resultClone.get(counter).add(countup);
            resultClone.get(countup).add(counter);
            // and add the pairing to that particular way
            sendback.add(resultClone);
            // and add the updated way to the return clump
            resultCount++;
        }
    }
}
}

countup++;
// sendback = fastRemove(sendback, true, origSeq);
// Include the above line to invoke our implementation of Algorithm B.
// Without it, this will run as Algorithm A. The implementation of B
// is still error-prone; any attempts at bug-fixing are encouraged.

return sendback;
The code for removing isomorphisms (both ways):

```java
public static ArrayList<ArrayList<ArrayList<Integer>>> fastRemove(
    ArrayList<ArrayList<ArrayList<Integer>>> givens,
    boolean strictVersion, int[] primaSeq) {
    // a function to remove all isomorphisms from a collection of tree
    // realizations of a given degree sequence. Named because it runs
    // much more quickly than our original function for this purpose.
    // Inputs: an ArrayList containing trees, themselves represented as
    // arrayLists, and a boolean determining whether we want removal
    // up-to-original-degree or removal in general ([false] for general)
    // and the original sequence (which is only used if [strictVersion] is
    // true, but we require it regardless)
    // Outputs: a similar ArrayList containing trees, but with all
    // isomorphisms removed, under whatever paradigm was selected by
    // [strictVersion]
    ArrayList<ArrayList<ArrayList<Integer>>> emptyStart =
        new ArrayList<ArrayList<ArrayList<Integer>>>();
    String[] allUniques = new String[10];
    String[] uniqueDoubles = new String[10];
    int uniqueCount = 0;
    int realizeCount = 0;
    int emptyCount;
    boolean isAnIsomorphism;
    ArrayList<ArrayList<Integer>> currentRealization;
    String currentRep;
    String currentRep2;
    ArrayList<ArrayList<Integer>> toCheck;
    String stringCheck;
    String stringCheck2;
    while (realizeCount < givens.size()) {
        // for every tree in the input ArrayList
        isAnIsomorphism = false;
        emptyCount = 0;
```
currentRealization = givens.get(realizeCount);
if (strictVersion) {
    currentRep = makeARep(currentRealization, primaSeq);
    currentRep2 = makeOtherRep(currentRealization, primaSeq);
} else {
    currentRep = makeBasicRep(currentRealization, primaSeq);
    currentRep2 = makeOtherBasicRep(currentRealization, primaSeq);
}
stringCheck = "Q";
if (realizeCount == 0) {emptyCount = allUniques.length;}
while (emptyCount < uniqueCount && (!isAnIsomorphism)) {
    //for every tree we're planning to return as non-isomorphic
    stringCheck = allUniques[emptyCount];
    stringCheck2 = uniqueDoubles[emptyCount];
    if (stringCheck.equals(currentRep) ||
        stringCheck2.equals(currentRep) ||
        stringCheck.equals(currentRep2)) {
        //if the current tree is isomorphic to one we're already
        //planning to return, under the conditions requested:
        //flag it as such
        isAnIsomorphism = true;
    }
    emptyCount++;
}
if (!isAnIsomorphism) {
    //if the tree isn't isomorphic to anything we've seen so far:
    emptyStart.add(currentRealization);
    allUniques[uniqueCount] = currentRep;
    uniqueDoubles[uniqueCount] = currentRep2;
    uniqueCount++;
    if (allUniques.length - 1 == uniqueCount) {
        allUniques = stringDouble(allUniques);
        uniqueDoubles = stringDouble(uniqueDoubles);
    }
    }
    realizeCount++;
    }
    return emptyStart;
    }