A Flexible Edge Search Algorithm for Graceful Tree Labeling

Michael Arnold[1] and Fabio Vitor[2]

Department of Mathematical and Statistical Sciences
University of Nebraska at Omaha
Omaha, NE, USA
email[1]: https://orcid.org/0009-0001-5262-9949
email[2]: ftorresvitor@unomaha.edu

July 10, 2024

Abstract

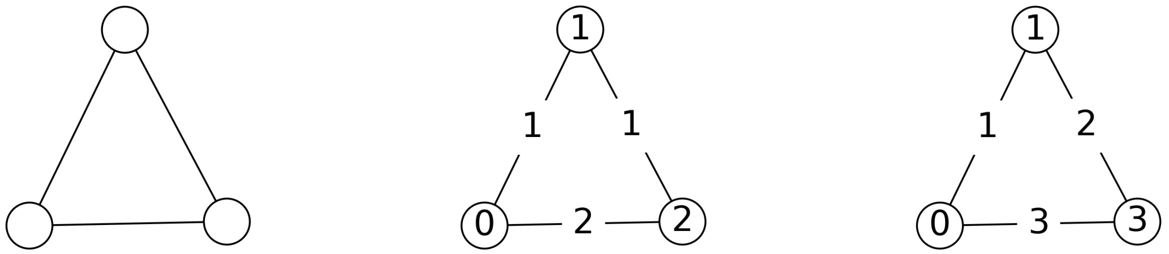
Graceful labeling is a famous problem in graph theory. Conjecture says that all trees are graceful (GTC). The Edge Search Algorithm developed by Horton [16] can solve graceful labeling for unrooted trees. However, it cannot accurately solve graceful labeling for rooted trees. This paper presents the Flexible Edge Search Algorithm, which can quickly find a solution to the graceful labeling problem in rooted trees. In addition, it accurately finds which rooted trees are impossible to solve for. Computational experiments indicate that all such impossible roots are in trees of the same class: the scorpion.

1 Introduction

Graceful labeling is a famous problem in graph theory. Formally, let $G = (V, E)$ be a graph where $V = \{1, 2, ..., n\}$ is the set of nodes and $E = \{1, 2, ..., m\}$ is the set of edges. For a graph to have a graceful labeling, the nodes must be labeled with distinct integers and the edges must labeled with the absolute difference of the two neighboring nodes. If the set of edge labels is $\{1, ..., m\}$, then the labeling is called graceful. Also, the graph itself is graceful, and the node with the smallest label is a graceful root.

This property of being graceful is also called β-labeling and belongs to the larger study of graph labeling, a topic introduced by Rosa in 1966.[24] Labelings are considered a basic and naturally occurring property of graphs. Researchers have proposed the use of graph labeling, such as the suggestion by Aziz et. al. to use the graceful property for choosing addresses in a network or keys in a database[1], or by Sun et. al. to apply the idea to password generation[25].

Figure 1 demonstrates the idea of graceful labeling. Figure 1a shows an unlabeled graph. The attempt in Figure 1b fails to be graceful because two edges have the same label. On the other hand, Figure 1c is graceful because there are no such repeats.



(a) An unlabeled graph.

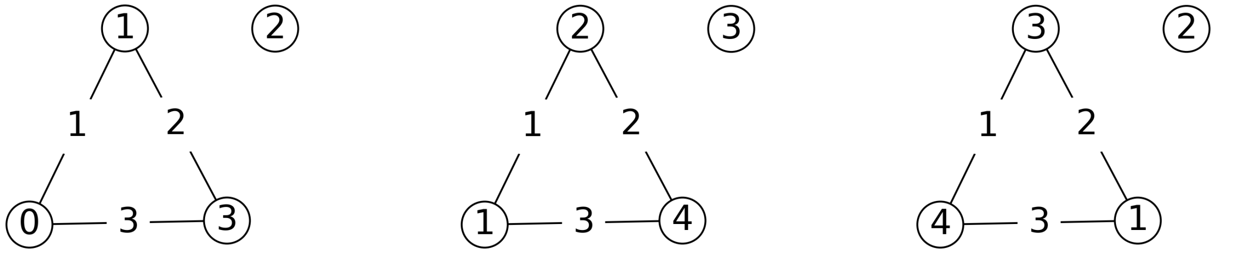
(b) A failed attempt.

(c) A graceful labeling.

Figure 1: Variations from an unlabeled graph.

Note that in Figure 1c, the node labels are not consecutive. Labeling the nodes consecutively is not required and in general, node labels in graceful graphs are not consecutive. But this property can be useful - for instance, a computer algorithm that finds labelings is easier to implement if all of the node labels are known beforehand. It is easy to satisfy this condition by adding a node that is not connected to other nodes. Figure 2a demonstrates this idea where an arbitrary node is added. The graph is still graceful while now also having consecutive node labels.

Node labels, as a set, can also be flexible. Figure 2b shows a labeling that is isomorphic to 2a but with all node labels increased by a constant, in this case by 1. Figure 2c depicts another isometric case, with the same set of node labels applied in reverse order, called the *complementary* labeling. Observe that all of the examples in Figure 2 are graceful labelings.



(a) Added node for consecutive node labels.

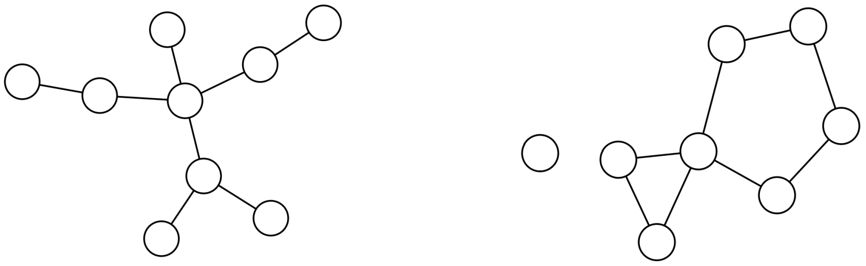
(b) Different node labels, but isometric.

(c) Reversed order of node labels.

Figure 2: A few isomorphic solutions.

Proving the graceful-ness of a graph or class of graphs can be deceptively challenging. Even when a graceful labeling exists for a single graph, finding one by hand can be difficult. A partial solution can appear to be much closer to a perfect solution than it actually is. Proving that a given graph is not graceful can be computationally intensive. And proving that a given class of graph is graceful involves a specialized proof in each case. Small changes to a graph (adding a leaf, for example) can dramatically alter the set of solutions. All told, graceful labeling is a hard combinatorial problem.

One critical class of graphs in graph theory are trees. A tree is a connected graph with $n-1$ edges and no cycles, meaning there is a unique path to eventually reach any node from any other node. Tree graphs often resemble a tangle of branches, but are similar to a real tree in that removing any edge results in exactly two disconnected pieces. Figure 3 shows the distinction between a graph that is not a tree and one that is a tree.

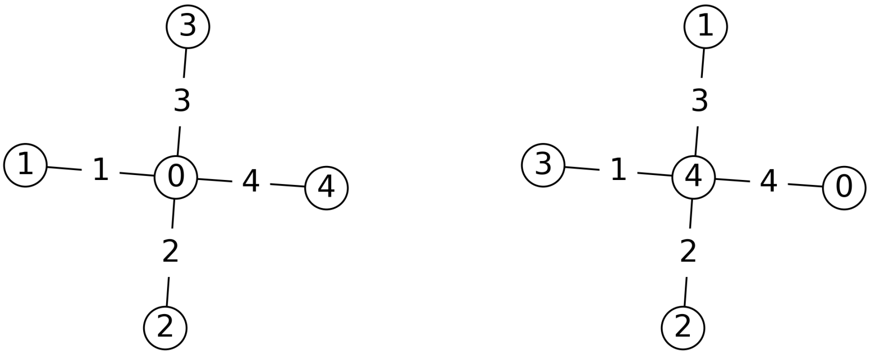


(a) This graph is a tree: it is connected and there are no cycles.

(b) A graph that is not a tree: one node is disconnected from the rest and it has cycles.

Figure 3: Examples of a tree and a non-tree.

A conjecture by Ringel and Kotzig says that all trees are graceful,[1] as first noted by Rosa[24]. Indeed, most trees permit many solutions each, as the number of nodes increases, the number of solutions[11] greatly outpaces the number of trees[10]. However, some types of trees are more restrictive. For instance, stars, a type of tree with one center node connected to all other nodes, permit only one solution up to isomorphism (see Figure 4a and Figure 4b).



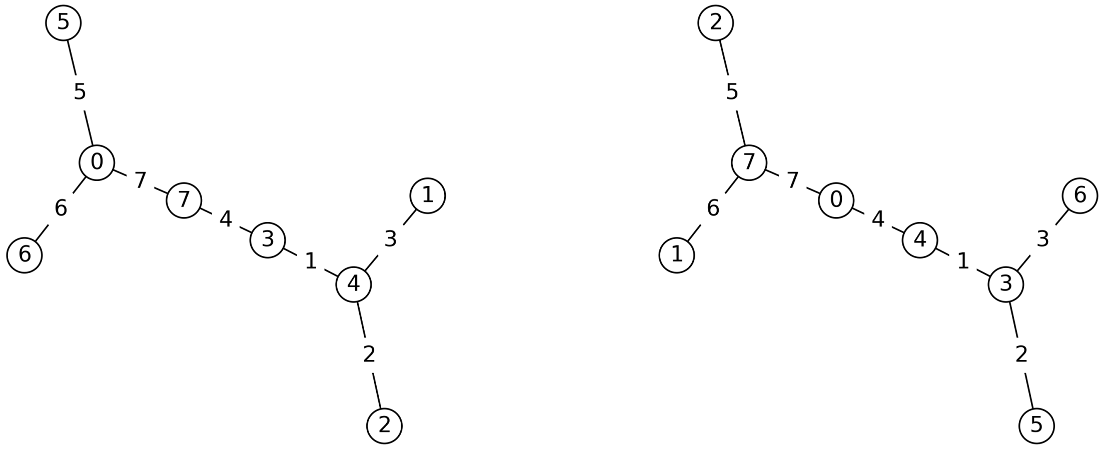
(a) A star solution.

(b) An isomorphic solution.

Figure 4: A star solution and its complement.

A complementary solution for a given labeling can also be seen by using an adjacency matrix. Consider the labeling in Figure 5a. Its adjacency matrix can be seen in Figure 6a. That matrix may be flipped along the reverse diagonal to obtain the matrix in Figure 6b. This matrix indicates the solution in Figure 6b. Notice that each matrix follows the rule that each forward diagonal contains exactly one edge, other than the longest.

[1] Called the Ringel–Kotzig conjecture, or the Graceful Tree Conjecture or just GTC.



(a) A graceful labeling.

(b) The complementary solution.

Figure 5: A graceful labeling and its complementary solution.

	0	1	2	3	4	5	6	7	
0	.					x	x	x	0
1		.			x				1
2			.		x				2
3				.	x			x	3
4		x	x	x	.				4
5	x					.			5
6	x						.		6
7	x			x				.	7
	0	1	2	3	4	5	6	7	

(a) An adjacency matrix.

	0	1	2	3	4	5	6	7	
0	.				x			x	0
1		.						x	1
2			.					x	2
3				.	x	x	x		3
4	x			x	.				4
5				x		.			5
6				x			.		6
7	x	x	x					.	7
	0	1	2	3	4	5	6	7	

(b) The complementary matrix.

Figure 6: The adjacency matrices for the above trees.

Trees with duplicate branches may have fewer solutions. Horton[16] found a 'chandelier' that required a long time to solve (Figure 7). The solution mimics some of the symmetry found in the original tree. Because there is no obvious reason why a tree must have a graceful labeling, a tree could conceivably be so restrictive that it does not permit any graceful labelings at all. The search for a counter-example is one reason to check every tree.

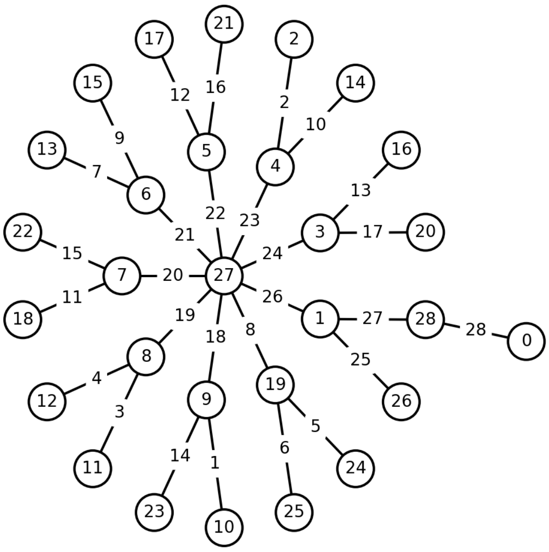


Figure 7: A labeling found by Horton for a tree with many symmetries.

For most trees, one of the most effective methods of finding a graceful labeling is the Edge Search Algorithm developed by Horton[16] and later improved by Fang [12]. Fang called his version "Backtrack". Both names describe the algorithm well. The algorithm always has a partial solution in mind. During any iteration, the algorithm finds all edges that could use the largest unused label. It tries this label in each one. If none of the edges are valid for that label, then it assumes it made a mistake and removes its most recent labeling. It then continues in a different direction.

The Edge Search Algorithm can go very fast or very slow. A solve can take as few as n iterations, but the number of iterations can rise exponentially if the starting parameters are chosen poorly or unluckily. To reduce the number of iterations, Horton and Fang each had a strategy. They both stop the algorithm after a certain number of steps, and make adjustments. Horton starts with a low threshold for the number of iterations and then retries with a different starting point and a higher threshold, while Fang starts with a high threshold and then abandons Edge Search in favor of a heuristic method.

This paper's primary contribution is an algorithm that improves upon Horton's method by using new strategies. The method developed for the paper is referred to as the Flexible Edge Search Algorithm. The most important new part is in the flexible target list, which allows edge labels to be tried in non-descending order. The algorithm also rules out edge-label matches which have either obvious right answers or obvious wrong answers. And the algorithm varies the priority of trying each edge depending on its location.

One of the findings of the computational experiments performed on a set of trees is that most roots were graceful, and most trees had all graceful roots. All of the trees with non-graceful roots had only one such root up to isomorphism, and all of these trees can be placed into a single class called "scorpion". Another contribution of this paper is a new characterization of the scorpion class and an algorithm for generating all graceful scorpions.[2]

The remainder of this paper is organized as follows. Section 2 presents a brief literature review on the advancements of graceful labeling for trees. Section 3 describes the Edge Search Algorithm

[2] To be distinguished from the scorpion graphs used by Best, et. al.[5]

and shows an example to demonstrate the algorithm. Section 4.1 describes the Flexible Edge Search Algorithm developed for this paper while Section 4.2 shows some other improvements to the Edge Search Algorithm. Section 5 discusses the results of trying each strategy and Section 6 characterizes the resulting graph labelings and the roots which failed to be graceful. Finally, Section 7 concludes the paper and provides a topic for future research.

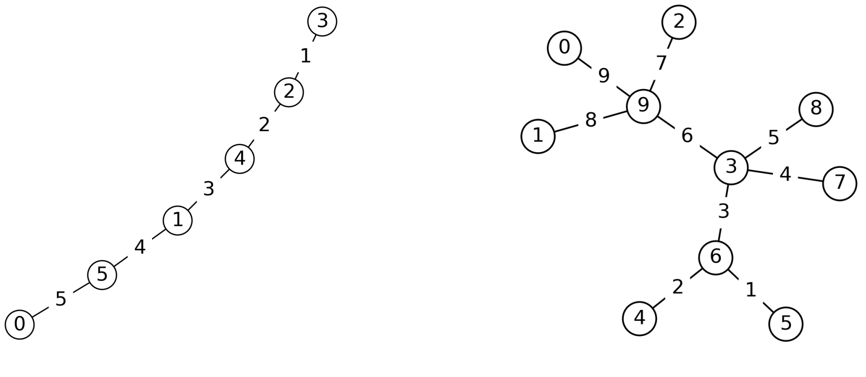
2 Literature Review on Graceful Tree Labeling

The most comprehensive review of graph labeling is the regularly updated survey by Gallian [13]. Surveys about graceful trees in particular have been done by Edwards and Howard in 2006 [9] and by Robeva in 2011 [22]. Algorithms for finding graceful trees were surveyed in 2022 by Brankovic and Reynolds [6].

Rosa introduced graceful labeling in 1966 [24], although the name "graceful labeling" was coined by Solomon W. Golomb in 1972 [15]. The conjecture that all trees are graceful is also called the Ringel–Kotzig conjecture, and was once called a "disease" by Kotzig himself [18]. The desire to prove the GTC originates from attempts to prove Ringel's 1963 conjecture about complete graphs [21]. This prior conjecture states that complete graphs with an odd number of nodes, say $2n+1$, can be deconstructed into $2n+1$ identical copies of any tree of $n+1$ nodes. A partial proof of the 1963 conjecture was submitted by both Keevash et. al. in 2020 [19] and Montgomery et. al. in 2021, [20] and revisions of a GTC proof have been proposed by Gnang between 2018 and 2022 [14].

Algorithms have been used to prove the GTC up to a certain number of nodes. Aldred and McKay in 1998 combined hill-climbing with tabu search to verify trees of 27 nodes or fewer [2], while Horton in 2003 used a backtrack method called the Edge Search Algorithm to improve this to 29 nodes or fewer [16]. Most recently, Fang in 2010 improved this to 35 nodes by combining the Edge Search Algorithm with a probabilistic search [12].

Specific classes of trees are known to be graceful. Stars are graceful, by assigning the smallest or largest label to the center. Paths, which resemble a line made up of edges, are graceful, by starting at one end and alternating between assigning the smallest or largest remaining labels (see Figure 8a). Caterpillars (a tree where removing all the leaves will produce a path) are graceful (see Figure 8b) because they can be constructed by adding a leaf to the smallest or largest node label, not necessarily alternating [24].

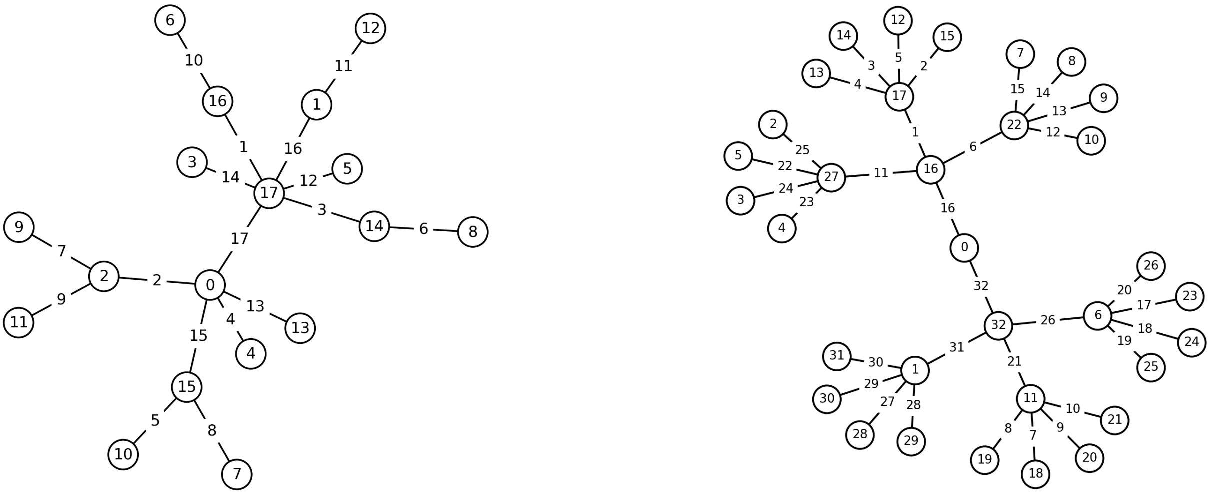


(a) A path.

(b) A caterpillar.

Figure 8: Some simple tree types are known to be graceful.

Many more elaborate types of trees are known to be graceful. Trees of diameter five or less are graceful [17] (the diameter of a graph is the longest direct path between two nodes); see Figure 9a. This was later improved by Wang et. al. to diameter seven or less [27], but only for trees with a perfect matching (a perfect matching is a selection of the edges such that each node is only selected once). Symmetrical trees, which have the condition that every node of the same distance from the center has the same degree, are graceful; Bermond noted this is always possible [4] while Rofa has proposed a more direct algorithm [23]; see Figure 9b.



(a) A tree with diameter five. This labeling was found by Hrnciar.

(b) A symmetrical tree. The labeling comes from Rofa's algorithm.

Figure 9: More complex trees require more elaborate algorithms to label.

There is likely some fact about any graph which will cause it to be graceful or not, but this factor is unknown. One can further this distinction between graceful and non-graceful graphs by adding requirements to a graceful graph and discovering which cases become non-graceful. Using rooted trees is one option. This requires the root to take on the lowest possible node label. The root is graceful if it allows a graceful labeling. A tree which is graceful at every possible root was first called 0-*rotatable* by Chung and Hwang in 1981 [8] and also later named 0-*ubiquitous* by Van Bussel in 2003 [26]. The terms k-*rotatable* and k-*ubiquitous* apply to any label k.

However, some trees have non-graceful roots. In all known cases, these non-graceful roots are in trees that fit into a class that resembles the constellation Scorpius (see the example in Figure 10). In this paper, these trees are referred to as *scorpions*. Cahit described several classes of these scorpions in a 2002 working paper [7], although this paper generalizes these to a single class in Section 6. Observe that Van Bussel [26] and Anick [3] also found this class of trees, grouped into scorpions without tails (D) and scorpions with tails (D'). Van Bussel provided a condition for when a scorpion would or would not be graceful. Anick conducted a computer search and analysis, finding conditions where a tree may or may not be k-ubiquitous. His search first found all graceful graphs up to a certain order and selected all of those that were trees.

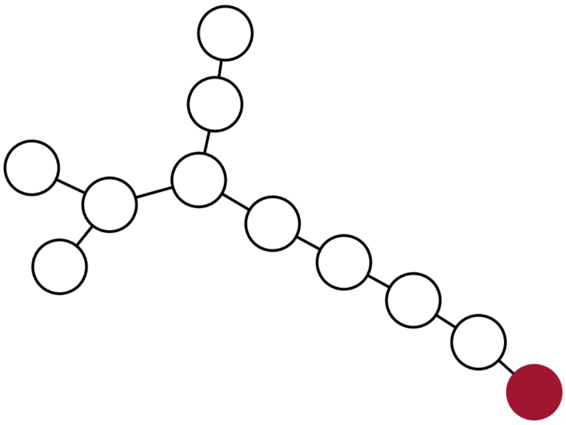


Figure 10: A scorpion tree with a 5-node tail, a 2-node claw and a 1-node claw. The red root has no graceful solutions.

3 The Edge Search Algorithm

When finding a graceful labeling by hand, it is easy to put labels on nodes and then check if the resulting edge labels are valid. The Edge Search Algorithm reverses this process. If one stipulates that a given edge label must be used next, can then a node that satisfies this condition be found?

The Edge Search Algorithm works by recursively asking this question. When it is successful at finding one more label, it moves deeper into its current solution. When it fails, it takes a step back and moves to the next possible edge, or if no more edges remain to be checked, it takes another step back and does the same. Therefore, the Edge Search Algorithm is considered a depth-first method.

The Edge Search Algorithm is mostly deterministic, except for some random choices during the initialization step. For example, the starting node (or the root node) is chosen at random. This may be an issue because some trees have unsolvable roots. That is, choosing a node to have the lowest label can already rule out all some solutions. There is no known way to tell if a root is unsolvable in general, so the algorithm can waste a significant amount of time on bad roots.

Another random choice during the initialization step is the strategy for edge priority. That is, the order edges are labeled in first. The edges have a set priority over one another, determined randomly before labeling begins. This priority can significantly impact the solving time. Sometimes this can mean hitting a limit on the number of iterations and trying a different set of starting conditions. But this shuffling does not change whether a tree is solvable overall.

If the Edge Search Algorithm cannot find a solution after a certain number of iterations, then it must stop and start again using a different set of parameters. For instance, the algorithm can start from a different root node, or use a special strategy for edge priority, or some other change in strategy. Observe that the limit on the number of iterations must be determined carefully as setting a high number of attempts can waste time and setting a low number can result in unnecessary extra steps.

Algorithm 1 presents the steps of the Edge Search method in detail. Algorithm 2 and 3 are subroutines called within Algorithm 1 to apply an edge label and remove a failed edge label, respectively. Observe that the current edge label the algorithm tries to place is called *target*, and each recursive call is for the next label in the *target_list*. There are also many lists that track which labels are used or unused, and the label state of each edge. The most important edges are the edges in *active* that are each connected to one labeled node and one unlabeled node. Each call to the Edge Search Algorithm checks if any active edge can take on the current target. To illustrate the algorithm, Example 1 is presented.

Algorithm 1 : The Edge Search Algorithm

```
Require: A target_list of edge labels, a set of unused node labels, a set of current node labels,
    a set of edges fully labeled, a list of half-labeled active edges, a list of inactive edges with no
    labels, the current number of iterations, and an iteration_limit;
Ensure: Return result as either “success”, “impossible”, or “hit limit”, and the number of
    iterations;
if target_list = “empty” then
    return result ← “success” and iterations;
iterations ← iterations + 1;
if iterations > iteration_limit then
    return result ← “hit limit” and iterations;
target ← first edge label in the target_list;
applicable(iterations) ← “empty”;
for each edge in active do
    if the first node label from the edge in active ± target is in unused then
        note this new_label;
        add the edge to applicable(iterations);
for each edge in applicable(iterations) do
    call Apply New Label with the edge, the new_label, an empty list of movers and all the lists
    in the current solution state;
    copy target_list to new_target_list without the first edge label;
    call the Edge Search Algorithm with new_target_list and solution;
    if result is “success” then
        return result ← “success” and iterations;
    else
        call Remove New Label with the edge, the new_label, the list of movers and all the lists
    in the current solution state;
return result ← “impossible” and iterations;
```

Algorithm 2 : Apply New Label

```
Require: A solution tree containing node lists unused and labels, and edge lists labeled, active
  and inactive; an active edge and a new_label for its node; an empty list of movers.
  apply the new_label from unused to the unlabeled node in edge;
  move new_label from labels to unused;
  move the edge from active to labeled;
  note all inactive edges connected to the newly labeled node by adding them to movers;
  move all edges in movers from inactive to active;
  return solution;
```

Algorithm 3 : Remove New Label

```
Require: A $solution$ tree containing node lists $unused$ and $labels$, and edge lists $labeled$, $active$
    and $inactive$; an active $edge$ and the $new\_label$ for its node; a list of $movers$.
  remove the $new\_label$ from the node in $edge$;
  move $new\_label$ from $unused$ to $labels$;
  move the edge from $labeled$ to $active$;
  move all edges in $movers$ from $active$ to $inactive$;
  return $solution$;
```

Example 1. *Consider the tree depicted in Figure 11 with a randomly selected root.*

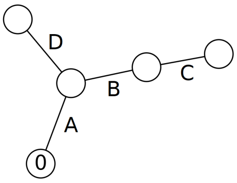


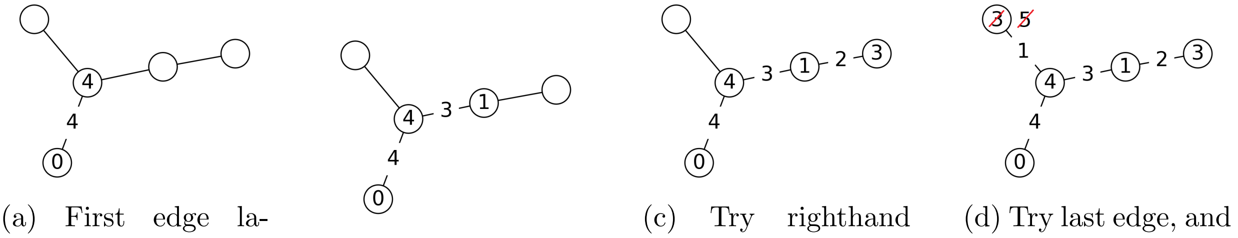
Figure 11: Root labeled 0.

Let *target_list* = $(4,3,2,1)$, *unused* = $\{1,2,3,4\}$, *labels* = $\{0\}$, *labeled* = $()$, *active* = (A), *inactive* = (B,C,D), *iterations* = 0, and *iteration_limit* = M where M is sufficiently large. Since the *target_list* is not "empty", then *iterations* = 1 and *target* becomes 4, which is the first in the *target_list*. Furthermore, *applicable*(1) is empty.

There is only one *active* edge, "A", and its only labeled node has the label 0. The *target* value is 4, so the algorithm checks whether $0-4=-4$ or $0+4=4$ are in the set of *unused* labels, which is true for the second case. Therefore, *applicable*(1) = (A). Given that the list of *applicable* edges has only "A", the following updates are made: *unused* = $\{1,2,3\}$, *labels* = $\{0,4\}$, *labeled* = (A), *active* = (B,D), *inactive* = (C), and *new_target_list* = $(3,2,1)$. The resulting tree is presented in Figure 12a and the algorithm is called again with the updated *target_list*.

With *iterations* = 2, the *target* is 3, and *applicable*(2) starts empty. There are two *active* edges, "B" and "D", and the algorithm must check if either are applicable. They both have their first node label as 4. Because *target* = 3, the algorithm checks if either $4+3=7$ or $4-3=1$ are in *unused*. The latter is, so *applicable*(2) = (B,D). The edge order is predetermined before the algorithm begins, as discussed in Section 4. Let us say "B" is chosen first. Thus, the following updates are made: *unused* = $\{2,3\}$, *labels* = $\{0,1,4\}$, *labeled* = (A,B), *active* = (C,D), *inactive* = $()$, and *new_target_list* = $(2,1)$. The algorithm is called once again with *target_list* = $(2,1)$. Figure 12b depicts the resulting tree.

The process is repeated for *iterations* = 3 and *target* = 2. In this case, *applicable*(3) = (C,D) since $1+2=3$ and $4-2=2$, respectively. Note that "D" is applicable for both *iterations* = 2 and *iterations* = 3. Let us assume edge "C" is tried first. Before the next call of the Edge Search Algorithm, the following parameters are: *unused* = $\{2\}$, *labels* = $\{0,1,3,4\}$, *labeled* = (A,B,C), *active* = (D), *inactive* = $()$, and *new_target_list* = (1). Figure 12c demonstrates this tree. Observe that the next *iterations* = 4 and *target* = 1. There is only one *active* edge left, "D", but neither $4+1=5$ nor $4-1=3$ are in the set of *unused* labels. Consequently, edge "D" is not applicable and *iterations* = 4 has no applicable edges at all. Figure 12d shows this case.



(a) First edge labeled.

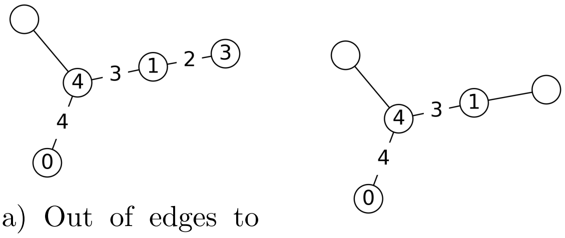
(b) Try middle edge.

(c) Try righthand edge.

(d) Try last edge, and fail.

Figure 12: After four iterations, the fourth cannot find an applicable label.

The Edge Search Algorithm thus goes back to the previous call (*iterations* = 3) and the corresponding tree is shown in Figure 13a, which is identical to Figure 12c (since this is where the algorithm left off). Since *result* from *iterations* = 4 is "impossible", the following is updated: *unused* = $\{2,3\}$, *labels* = $\{0,1,4\}$, *labeled* = (A,B), *active* = (C,D), *inactive* = $()$. The resulting tree after this attempt is presented in Figure 13b.

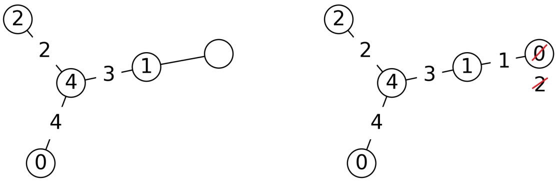


(a) Out of edges to try.

(b) Remove the label.

Figure 13: Backtracking out of the fourth iteration and preparing to continue the third.

Since there is still one more applicable edge to be checked in *iterations* = 3, "D", then the following is updated: *unused* = $\{3\}$, *labels* = $\{0,1,2,4\}$, *labeled* = (A,B,D), *active* = (C), *inactive* = $()$. Figure 14a shows this tree. The *new_target_list* = (1) as the Edge Search Algorithm is called once again, and in this case for *iterations* = 5. One can see that there are no applicable edges as shown in Figure 14b.

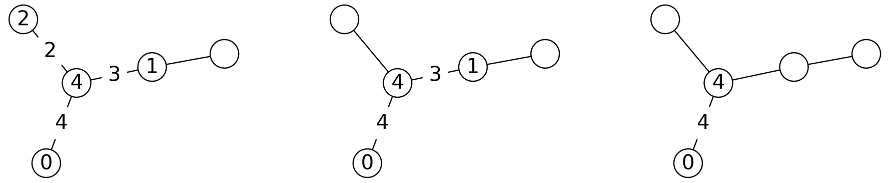


(a) Try upper-left edge.

(b) Try righthand edge, and fail.

Figure 14: Trying the second applicable edge in the third iteration, but later failing to label the final edge in iteration five.

The algorithm thus goes back to *iterations* = 3 and the corresponding tree is shown in Figure 15a. Again, this is the same as Figure 14a. Since labeling edge "C" in *iterations* = 5 did not work out, then labeling edge "D" in *iterations* = 3 also does not work. Therefore, this labeling is reversed as shown in Figure 15b. Since there are no more applicable edges to try, the Edge Search Algorithm returns to *iterations* = 2 and its tree is demonstrated in Figure 15c (same as Figure 12a).



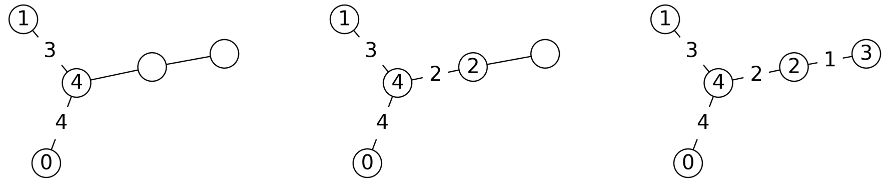
(a) Out of edges to try.

(b) Remove another label; out of edges.

(c) Remove another label.

Figure 15: Return to iteration three and remove another label, then return to iteration two and remove another label.

Remember that there is one remaining applicable edge in *iterations* = 2, "D", which is labeled with *target* = 3 (Figure 16a). Continuing, edge "B" is labeled with *target* = 2 in *iterations* = 6 (Figure 16b) and edge "C" is labeled with *target* = 1 in *iterations* = 7 (Figure 16c). Since the *target_list* is "empty", the algorithm returns *result* as "success" and so this is a graceful labeling.



(a) Try upper-left edge.

(b) Try middle edge again.

(c) Try righthand edge.

Figure 16: Continuing with iteration 2, label an edge and continue to iterations 6 and 7, both successful.

One should notice in this example that the list of edge priorities does have an impact on the number of iterations. Suppose that the priority is "A", "D", "B", "C". In this case, the Edge Search Algorithm solves this graceful labeling problem within only 4 iterations instead of 7. Furthermore, the Edge Search Algorithm cannot find all graceful labelings for a given tree. Indeed, there are some roots that have no findable solution via the Edge Search Algorithm. The following section presents the Flexible Edge Search Algorithm, which can find graceful labelings for all graceful roots.

4 The Flexible Edge Search Algorithm

The Edge Search Algorithm discussed in Section 3 may take too long to solve on certain trees. To mitigate this, Horton made some improvements to the original version of the algorithm. For instance, it restarts itself after failing too often, with increasing tolerance. Each restart also begins with the computer's list of edges shuffled around, so that different edges are tried first. Finally, an additional version did not waste time checking roots isomorphic to bad roots, although Horton found this feature to be slower in some cases; this paper does not include this feature for comparison purposes in Section 5.

Both versions of the Edge Search Algorithm developed by Horton are designed to solve all trees regardless of root. Therefore, they do not know which roots are graceful or not. However, it may be useful to find which roots are not graceful.

The Flexible Edge Search Algorithm developed for this paper is able to prove which roots are graceful and which roots are not. This can be done by finding solutions that require a non-descending target list. To efficiently do this, the Flexible Edge Search Algorithm permutes from one target list to the next. It uses the minimum depth to determine which targets never need to change and it uses the maximum depth to determine which permutations can be skipped over. It uses two different methods of permuting the target list. And during a solve, it skips over applicable edges that emulate an earlier target list. All these features are discussed in Section 4.1.

Furthermore, the Flexible Edge Search Algorithm has a root priority, it does not check roots that are isomorphic to graceful roots, it has a method to determine which edge to try first, and it classifies edges and does not check on more than one edge from each class. These are discussed in Section 4.2.

4.1 Flexible Target List

In the original Edge Search Algorithm, the largest edge label was assigned first, and then the second largest, and so on. In this way, the list of edge targets is simply $(n, n-1, ..., 1)$ for $n+1$ nodes and n edges. However, one can assign them in any order. This would provide more flexibility, permitting finding solutions that could not be found with the original target list (see Figure 17). The downside is that this could potentially check $n!$ possible target lists. However, several considerations can cut down on this growth in solving time and they are discussed in the next sections.

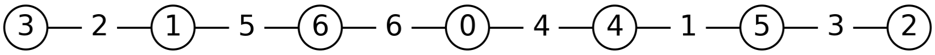


Figure 17: A labeling found with a non-descending target list. The 1 edge label is placed before the 3 edge label.

4.1.1 Minimum Depth

In all graceful labelings, the largest edge labels are connected, so the first few targets never need to be tried in different order. Let us call the number of targets which never need to be permuted as the *minimum depth* of the root.

Theorem 1 shows that the first 3 targets in the target list are connected. This will imply that the minimum depth of any graceful labeling is at least 3.

Theorem 1. *Every graceful labeling with 3 or more edges has edge labels n, $n-1$ and $n-2$ on connected edges.*

...	**n-2**	**n-1**	**n**	
	n-2	*n-1*	*n*	**0**
		n-2	*n-1*	**1**
			n-2	**2**
				⋮

Figure 18: The possible selections for the three largest edge labels.

Proof. Consider the adjacency matrix of a graceful labeling (see Figure 18 representing the possibilities). One can see that the connection $\textcircled{0}\frown\textcircled{n}$ exists, being the only way to obtain edge label n; say $\textcircled{0}\,\overline{n}\,\textcircled{n}$. Then either $\textcircled{0}\,\overline{n-1}\,\textcircled{n-1}$ or $\textcircled{1}\,\overline{n-1}\,\textcircled{n}$.

Let us consider the first case, which altogether means $\textcircled{n}\,\overline{n}\,\textcircled{0}\,\overline{n-1}\,\textcircled{\text{n-1}}$. Then either $\textcircled{0}\,\overline{n-2}\,\textcircled{n-2}$, $\textcircled{1}\,\overline{n-2}\,\textcircled{n-1}$, or $\textcircled{2}\,\overline{n-2}\,\textcircled{n}$.

In the first case, the three edges form a star with $\textcircled{0}$ at the center.

In the second case, the three edges form a path with $\textcircled{n}\,\overline{n}\,\textcircled{0}\,\overline{n-1}\,\textcircled{n-1}\,\overline{n-2}\,\textcircled{1}$.

In the third case, the three edges form a path with $\textcircled{2}\,\overline{n-2}\,\textcircled{n}\,\overline{n}\,\textcircled{0}\,\overline{n-1}\,\textcircled{n-1}$.

For the prior case of $\textcircled{1}\,\overline{n-1}\,\textcircled{n}$, one can obtain a similar set of three possibilities by similar reasoning. In all cases, the three edge labels n, $n-1$ and $n-2$ are connected. □

In some cases, even more targets do not change. If the starting root is at the end of a leaf with label $\textcircled{0}$, then the other node in that leaf is $\textcircled{\text{n}}$. One can remove the leaf and turn $\textcircled{\text{n}}$ into a proper root with label $\textcircled{0}$ by taking the complement. Solving this new smaller tree is identical to solving the remainder of the original tree. For trees with a long path ending in a leaf, the entire path can be obviated if the root is chosen as the endpoint of that path, because all the choices along the path are forced. Theorem 2 first proves a general statement about attaching leaves, which is then used by Theorem 3 to show a fact about minimum depth.

Theorem 2. *A root is graceful if and only if a new leaf attached to it is graceful in the induced tree.*

Proof. We first show that attaching a leaf to a graceful root results in the leaf being graceful. Say A is the leaf that will be attached to graceful root B. We know that in the original tree T, there is a graceful labeling where B has label 0. So then say A is attached to B to form T'. If $n-1$ is the number of edges in T, then n is the number of edges in T'. Apply label n to A and so $\textcircled{B}\frown\textcircled{A}$ is labeled as $\textcircled{n}\,\overline{n}\,\textcircled{0}$. Taking the complement, the edge is now labeled as $\textcircled{0}\,\overline{n}\,\textcircled{n}$, and B has label 0 and is the graceful root of the complementary solution.

We next show that if a leaf is graceful, then its adjacent node is graceful. The leaf is again A and has label 0, in tree T'. Its only attached edge has B on the other end. This edge $\textcircled{A}\frown\textcircled{B}$ is labeled $\textcircled{0}\,\overline{n}\,\textcircled{n}$. But since there are no other nodes connected to $\textcircled{0}$, then the only way of creating $\overline{n-1}$ is $\textcircled{n}\,\overline{n-1}\,\textcircled{1}$ (as illustrated in Figure 18). Since B is labeled n, then $\textcircled{B}\,\overline{n-1}\,\textcircled{1}$. If A is removed to form T, then the largest edge label becomes $n-1$. Reducing all node labels by 1 results in $\textcircled{B}\,\overline{n-1}\,\textcircled{0}$ or $\textcircled{n-1}\,\overline{n-1}\,\textcircled{0}$. Taking the complement, $\textcircled{0}\,\overline{n-1}\,\textcircled{n-1}$ or $\textcircled{B}\,\overline{n-1}\,\textcircled{n-1}$ and so B is a graceful root in T. □

Theorem 3. *Say the root of a gracefully labeled tree T' is a leaf A connected to node B. Then say T is the tree with A removed and with root B. Then the minimum depth of T' = 1 + minimum depth of T.*

Proof. Because T is graceful, it therefore has a minimum depth m. Because the m largest edge labels are connected in T, and because there is only one possible way to gracefully connect A and B in T', then the $m+1$ largest edge labels are connected in T'. □

4.1.2 Maximum Depth

But what about maximum depth? Well, sometimes while attempting to find a solution with a given target list, the algorithm never even tries the last few targets. With this information, it is no use to simply re-arrange those last few. One needs to jump ahead in the list of permutations that could be tried next. This can be done by counting the maximum number of targets that the algorithm was ever able to try while solving. This is called *maximum depth*.

However, this can be further improved. Say a star exists at the end of a tree and the furthest solution only assigns labels to 2 out of 5 leaves in that star. Thus, the 2 that were assigned could not have been part of a valid solution. So one can discount not only unlabeled edges, but also all edges that are isomorphic to an unlabeled edge. This is called *classwise depth*, and the simpler method may be called *naive depth*.

During preliminary computational experiments, *classwise depth* required 10% fewer iterations than naive depth, but almost all of these gains were made obsolete by the randomly permuted target list, discussed in Section 4.1.3. Calculating *classwise depth* also requires more time within each iteration, so therefore *classwise depth* is only used when the number of edges exceeds the number of edge classes by five or more.

Note that while minimum depth is known before solving, maximum depth is specific for each target list. Note also that a faster solving time is achieved by a large minimum depth and a small maximum depth.

4.1.3 Permuting the Target List

It was earlier mentioned having to re-arrange the target list. For parts of the target list that must change, there are two sensible ways to check through all permutations. One method prefers to jumble the start of a list, and the other prefers to jumble the end. So the list (10,9,8, ..., 3,2,1) could be iterated to (9,10,8, ..., 3,2,1) with the first method or (10,9,8, ..., 3,1,2) with the second. These are called ‘reverse’ and ‘negate’, respectively.

These two methods can quickly find solutions that exist on either end of the big list of permutations, but for some trees the solutions are all closer to the middle of the permutation list. This could take a long while to locate. So, after $2n^2$ calls to the Flexible Edge Search Algorithm, a less precise strategy can be enabled: a randomly shuffled target list (subject to minimum depth). This is not always ideal. For roots that are actually impossible, checking random solutions only wastes time that could be spent progressing through the permutation list. This method “random” is tried only when the total iteration count is even and only within a given iteration interval. Because random solutions tend to be either easy to find or non-existent, the random shuffle is again disabled after n^3 iterations. These limits of $(2n^2, n^3)$ proved slightly better than limits of $(n^2, 2n^2)$, (n^2, n^3), or (n^3, n^4).

4.1.4 Descending Targets within an Edge Class

Since the target list can consider edge targets out of their usual order, it may label edges of the same class with labels in a way that replicates an earlier target list. But this is never necessary, so the Flexible Edge Search Algorithm does not try to apply a larger edge label than already exists within a particular edge class.

4.2 Other Features of the Flexible Edge Search Algorithm

Although the most important difference between the Edge Search Algorithm and the Flexible Edge Search Algorithm is the flexible target list, there are other improvements that can be made. For each of them, preliminary computational experiments were performed to assess their value in reducing the solving time. The results of these experiments are discussed in Section 5.

4.2.1 Root Priority

Even though this paper focuses on solving every root, one may wish to predict which roots solve in the fewest iterations. One method may be to choose roots near or far from the center. The center is a valid choice for stars, while an endpoint of a leaf furthest from the center is the best choice for paths. If forced to choose one strategy, it was found that endpoints are slightly better on average.

4.2.2 Similar Roots

Even when finding results for every root, there is no need to check every root. Each graceful labeling has a root labeled $\textcircled{0}$, and that root is connected to the edge labeled $\overset{\frown}{n}$, and the other node connected to that edge is labeled $\textcircled{n}$. This graceful labeling has a complementary solution with those two nodes labeled with $\textcircled{n}$ and $\textcircled{0}$, respectively. So every graceful labeling reveals two valid roots, and once a solution with one root is found, there is no need to check its complementary root. Additionally, the edge connecting these two nodes may be isomorphic to neighboring edges, and all nodes connected to these edges are also marked as successful and skipped over for checking. Horton did a similar check that ruled out many roots that were isomorphic to bad roots. The Flexible Edge Search Algorithm does not try to rule out bad roots because using the flexible target list allows almost all roots to be graceful, as mentioned later in Section 6.

4.2.3 Edge Priority

When both the Edge Search and Flexible Edge Search Algorithms have multiple active edges to choose from, they can either choose whichever edge is the first in the list, choose randomly,[3] or use some other criteria. One helpful criteria was found by noting how central each node was. The strategy can either choose to prefer central nodes or endpoints. This paper denotes these strategies as "near" and "far", respectively. While "far" does much better than "near", switching between the two after each attempt ("switching") is better than either one alone. Randomizing the edge priority did slightly worse than switching.

This paper's research also tried a similar set of strategies that look at each edge's distance from the chosen root rather than from the tree's center. This did not yield an improvement. Additionally, switching the strategy after each iteration proved to be faster than switching half as often.

4.2.4 Applicable Edge Classes

Each edge was classified relative to its first node and its subtrees. Two edges are in the same class only if they share a node and have isomorphic subtrees. Often, all active edges of the same class are applicable, but if one of them fails to take a label, they would all fail. Therefore, each iteration of the Flexible Edge Search Algorithm tests no more than one edge of each class.

[3] As used by Horton. "The adjacency list of every node was scrambled." pg 29

4.3 The Complete Flexible Edge Search Algorithm and Numerical Examples

The following algorithms include the Flexible Edge Search Algorithm (the main algorithm) and the most relevant subroutines. The main algorithm itself is called by a testing function which can rotate through all possible roots, and which itself is called by another function which proceeds through different trees.

The main algorithm is a recursive function which attempts to find an edge for one target and then call itself again for the next target. The algorithm can stop either when it has finished applying targets (a "success"), when it exhausts all possible edges to label ("impossible"), or if it runs for too long ("hit limit"). It calls the subroutine "Find Applicable" to create a list of applicable edges for the target and tries each edge in turn. For each edge, it calls "Apply New Label" before calling another instance of itself. After that instance finishes, it will either be successful and send that success back to the function that called it, or call "Remove New Label" and try the next applicable edge, if any. If the applicable edges are exhausted, it will return "impossible".

Algorithm 4 : Flexible Edge Search Algorithm

```
Require: A target_list of edge labels and a solution object containing a set of unused node labels,
    a set of current node labels, a set of edges fully labeled, a list of half-labeled active edges, a list
    of inactive edges with no labels, the current number of iterations, and an iteration_limit;
Ensure: Return result as either "success", "impossible", or "hit limit", and the number of
    iterations;
if target_list = "empty" then
    return result ← "success";
iterations ← iterations + 1;
if iterations > iteration_limit then
    return result ← "hit limit";
target ← first edge label in the target_list;
applicable(iterations) ← Find Applicable(target, solution);
for each case in applicable(iterations) do
    call Apply New Label with target, case and solution;
    copy target_list to new_target_list without the first edge label;
    call the Flexible Edge Search Algorithm with new_target_list and solution;
    if result is "success" or "hit limit" then
        return result;
    else
        call Remove New Label with case and solution;
return result ← "impossible";
```

The "Find Applicable" subroutine is called by each instance of the main algorithm in order to decide which edges to try the target on. This subroutine implements two features that take edge class into consideration. It does not add more than one edge of each class to the list and it does not add any edge class to the list if the target is larger than the smallest label currently applied to that class.

Algorithm 5 : Find Applicable

```
Require: A target edge label and a partial solution tree;
Ensure: A list of applicable edges;
  for each edge in active_edges do
      note the edge_class of that edge;
      if the edge_class is already in the list of such classes then
          continue to the next edge;
      note the current maximum label for the edge_class;
      if target is larger than the current maximum label then
          continue to the next edge;
      if the first node label from the edge in $active \pm target$ is in unused then
          note this new_label;
          create a new applicable case object using the edge and the new_label;
          add case to the list of applicable edges;
  if edge strategy is “near” then
      sort applicable by decreasing edge distance
  else if edge strategy is “far” then
      sort applicable by increasing edge distance
  return applicable;
```

The “Apply New Label” and “Remove New Label” subroutines are called by the main algorithm each time it tries an applicable edge, although the second will be skipped if the labeling was part of a successful solution. Several variables are changed to reflect the new labeling, but also a list of *movers* is noted. These are edges that move from being *inactive* to *active*, and will move back if the labeling is not successful. In addition, the new *target* becomes the new minimum edge label for the applicable edge class, in cases where the target list strategy is not random. This is also undone if the labeling fails.

Algorithm 6 : Apply New Label

```
Require: An applicable case, a solution tree, and a target;
  apply the new_label from unused to the unlabeled node in edge;
  move new_label from labels to unused;
  move the edge from active to labeled;
  note all inactive edges connected to the newly labeled node by adding them to movers;
  move all edges in movers from inactive to active;
  if the target strategy for this solution is not “random” then
      set the minimum edge label for this edge class to target;
  return solution;
```

Algorithm 7 : Remove New Label

```
Require: An applicable case and a solution tree;
  remove the new_label from the node in edge;
  move new_label from unused to labels;
  move the edge from labeled to active;
  move all edges in movers from active to inactive;
  if the target strategy for this solution is not “random” then
      set the minimum edge label for this edge class to its previous value;
  return solution;
```

The “Iterate List Strategy”, “Skip List”, and “Iterate List” subroutines relate to the flexible target list. The “Iterate List Strategy” is called by the same testing function that calls the main algorithm, after each such call that returned “impossible”. Its main purpose is to find the next appropriate target list that will use the target strategy that was used right before. To obtain this, the “Iterate List Strategy” subroutine itself calls two other subroutines, “Skip List” and “Iterate List”. Observe that “Skip List” is only used by the “negate” strategy because they both iterate on the end of the list. It then changes the strategy to “reverse” or “negate”, depending on which was used previously. It also changes the *target_list* in use to the target list tracked by that strategy.

Apart from the *target_list*, another variable called *is_last* is tracked for each strategy to indicate if either strategy has iterated past the last possible target list. This indicates that there is no target list that can find a solution. When either target strategy reaches this point, that root is marked as “impossible”.

Algorithm 8 : Iterate List Strategy

```
Require: A rooted tree and a failed solution;
Ensure: Return is_last as “True” or “False”;
  if the target_strategy is “negate” then
      call Skip List with the tree’s target_list and the maximum_depth of the solution;
  call Iterate List with the target_list, the minimum_depth, and the current target_strategy, which
  returns a new target_list and is_last;
  if the target_strategy is “negate” then
      $target\_list(negate) \leftarrow target\_list$;
      $is\_last(negate) \leftarrow is\_last$;
  if the target_strategy is “reverse” then
      $target\_list(reverse) \leftarrow target\_list$;
      $is\_last(reverse) \leftarrow is\_last$;
  if either of $is\_last(negate)$ or $is\_last(reverse)$ is “True” then return “True”;
  else return “False”;
```

The “Skip List” subroutine uses the *maximum_depth* from the previous search to decide which permutations can be skipped over. It does this by placing any targets after the maximum depth in increasing order. Both iteration strategies in “Iterate List” will then find the next permutation that ignores these final targets.

Algorithm 9 : Skip List

```
Require: A $target\_list$ and a $depth$;
  $list\_start \leftarrow$ the $target\_list$ before the $depth$ index;
  $list\_end \leftarrow$ the $target\_list$ after and including the $depth$ index;
  sort $list\_end$ in increasing order;
  $target\_list \leftarrow list\_start$ concatenated with $list\_end$;
  return $target\_list$;
```

The "Iterate List" subroutine uses two possible methods, depending on the target list strategy. In either case, it splits the list based on the *minimum_depth*. For the "negate" strategy, it negates all the numbers in the target list and then finds the next permutation in lexicographic order, before negating the numbers again. For the "reverse" strategy, it reverses the list and then finds the next permutation, before reversing the list again. Both also note if the previous list was the last possible permutation, and return this fact as the *is_last* variable.

Algorithm 10 : Iterate List

```
Require: A $target\_list$, $minimum\_depth$ and a $target\_strategy$;
Ensure: A $target\_list$ and $is\_last$ as "True" or "False";
  $split\_at \leftarrow minimum\_depth + 3$;
  $list\_start \leftarrow$ the $target\_list$ before the $split\_at$ index;
  $list\_end \leftarrow$ the $target\_list$ after and including the $split\_at$ index;
  $is\_last \leftarrow$ "False";
  if the $target\_strategy$ is "negate" then
      negate all numbers in the $target\_list$;
      set the $target\_list$ to the next lexicographic permutation;
      set $is\_last$ to "True" if this was the last permutation;
      negate all numbers in the $target\_list$;
  else if the $target\_strategy$ is "reverse" then
      reverse the numbers in the $target\_list$;
      set the $target\_list$ to the next lexicographic permutation;
      set $is\_last$ to "True" if this was the last permutation;
      reverse the numbers in the $target\_list$;
  $target\_list \leftarrow list\_start$ concatenated with $list\_end$;
  return $target\_list$ and $is\_last$;
```

To demonstrate the Flexible Edge Search Algorithm, Examples 2 and 3 are presented. Example 2 shows how the target list changes. Example 3 depicts how the "Find Applicable" subroutine considers edge classes.

Example 2. *Consider the path tree in Figure 19.*

The root node is not a leaf, so the minimum depth for this root is 3. The Flexible Edge Search Algorithm first tries *target_list* = $(6, 5, 4, 3, 2, 1)$ Notice that this is the same *target_list* considered by the the original Edge Search Algorithm. The algorithm is able to assign four labels to edges (see Figure 20), but no more than this. After 21 iterations, the algorithm exhausts all possibilities and concludes that it is impossible to gracefully label this rooted tree with the given target list. At this point, the original Edge Search Algorithm would give up on this root. But the Flexible Edge Search Algorithm can move on to another *target_list*.

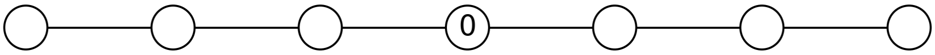


Figure 19: A path with six edges and the root in the center.

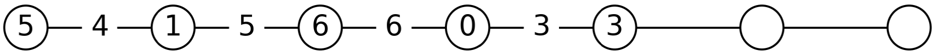


Figure 20: The maximum depth reached using the default target list.

To obtain the next list, it first considers the maximum depth reached in the previous attempt, and then use one of two methods to permute the list. Because the maximum depth was 4, we know that the fifth target never worked and we can skip over any *target_list* that agrees on the first five targets in that order. Any *target* after the maximum depth is placed in ascending order. The *target_list* changes from $(6, 5, 4, 3, 2, 1)$ to $(6, 5, 4, 3, 1, 2)$. The next permutation is guaranteed to change the fifth target.

The first permutation strategy is "negate". First, the list is divided in half, with the targets within the minimum depth in the first portion, and the rest in the second portion. Because the minimum depth was 3, this creates the two lists $(6, 5, 4)$ and $(3, 1, 2)$. Then all the numbers in the second list are negated to obtain $(-3, -1, -2)$. Next, the next permutation in lexicographic order is obtained, which is $(-2, -3, -1)$ (this may be more intuitively seen by adding a constant of 3 to all numbers). This list is negated again to obtain $(2, 3, 1)$ and rejoined to the other list resulting in $(6, 5, 4, 2, 3, 1)$.

Having obtained the next *target_list* via "negate", the algorithm sets this list aside. There is another permutation strategy which uses its own independent target list, called "reverse". Because "reverse" evolves from the same starting *target_list* as "negate", there is no need to test that *target_list* again. Therefore, during the initialization step of the algorithm, the *target_list* for "reverse" is permuted once before the first test begins.

To permute with "reverse", the algorithm again separates the starting list $(6, 5, 4, 3, 2, 1)$ in two, using the already known minimum depth. The first list is again $(6, 5, 4)$ and the second list is $(3, 2, 1)$. Because there is no maximum depth to consider, the algorithm only calls the "Iterate List Strategy" subroutine. Since the strategy is "reverse", the order of the second list is reversed to obtain $(1, 2, 3)$. The next list in lexicographic order is $(1, 3, 2)$. After reversing again to obtain $(2, 3, 1)$, this list is rejoined to the first creating the list $(6, 5, 4, 2, 3, 1)$.

Notice that this is the same list created using the "negate" strategy! Because one strategy starts by re-arranging the start of a list, and the other starts by re-arranging the end of a list, they may agree on the entire list at most once as they independently progress. However, each strategy is not simply the opposite process of the other, because the maximum depth creates different skipping opportunites for each.

Since the algorithm is still on the "reverse" strategy, it tests its version of the *target_list* = $(6, 5, 4, 2, 3, 1)$. After 31 iterations, this list also fails to solve. The maximum depth reached is 5, as depicted in Figure 21.

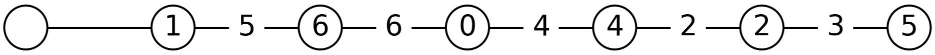


Figure 21: The maximum depth reached using the second target list.

The "reverse" strategy thus permutes again. Because the strategy is "reverse", the algorithm does not call "Skip List". The algorithm again splits the list, this time into $(6, 5, 4)$ and $(2, 3, 1)$. Reversing the second list results in $(1, 3, 2)$, which permutes lexicographically to $(2, 1, 3)$ and reverses again to $(3, 1, 2)$. Rejoining, the outgoing list is therefore $(6, 5, 4, 3, 1, 2)$.

Again, because the algorithm most recently tried "reverse", it now tries "negate". But, because the list was incidentally the same, the result is the same: 31 iterations and maximum depth of 5. To permute this list, there is the same trivial skipping step, and the same splitting of $(6, 5, 4)$ and $(2, 3, 1)$. Negating $(2, 3, 1)$ results in $(-2, -3, -1)$, which permutes to $(-2, -1, -3)$ and negates again to $(2, 1, 3)$. The full list becomes $(6, 5, 4, 2, 1, 3)$.

Having found the next list for "negate", the algorithm switches back to the list for "reverse" and attempts to solve. This list was $(6, 5, 4, 3, 1, 2)$ and it succeeds in finding a solution. In fact, it is also lucky in choosing the edge order and solves in just 6 iterations. The final result is the same tree shown in Figure 17.

Example 3. *Consider the tree in Figure 22.*

The solve of this tree requires 146 iterations across 3 target lists. Let us consider the algorithm's progress through the second *target_list* = $(9, 8, 7, 5, 6, 4, 3, 2, 1)$ (note the positions of targets 5 and 6), beginning at iteration 53.

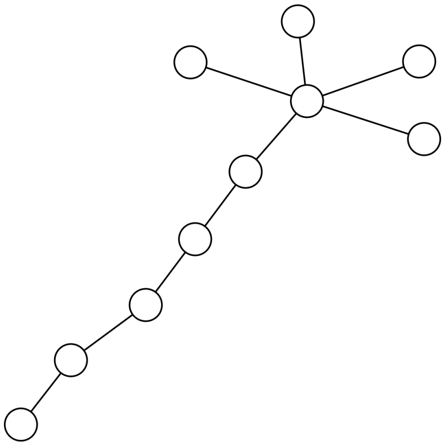


Figure 22: A tree with ten nodes and some isomorphic edges.

Figure 23 shows the partial solution at the start of iteration 53. The *target* edge label is 5. The *applicable*(53) edges would be (A,B,C,D,E), except that A,B,C,D are all of the same class. Therefore, only the first ("A") is tried, and it is tried first.

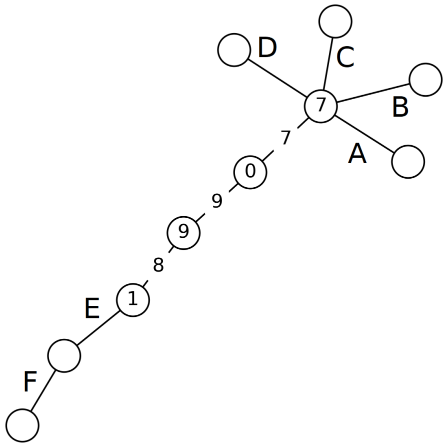


Figure 23: The start of iteration 53 with three edges already labeled.

Figure 24 shows the tree at the start of iteration 54. As the next *target* is 6, the edges "B", "C" and "D" would all be eligible, except that edge "A" is of the same class and has a lower label (5). Labeling one of these with 6 would replicate a partial solution from the earlier *target_list* where 6 was tried before 5. Because all of those partial solutions were already ruled out, then none of "B", "C" or "D" are truly applicable. As there are no more applicable edges, the algorithm exits iteration 54 without a successful solution.

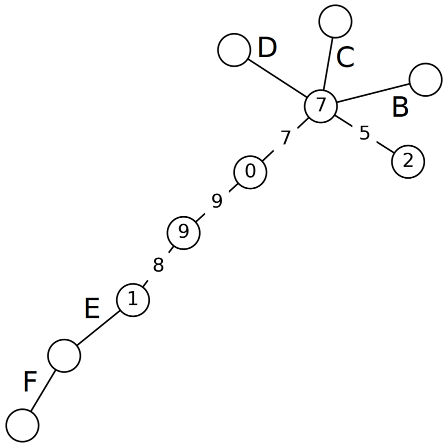


Figure 24: Labeling edge 'A' during iteration 54.

Back to iteration 53, since "A" was impossible, it then moves on to edge "E". Figure 25 shows the tree during iteration 55. As there are no applicable edges in this iteration either, the algorithm also exits this iteration and returns to iteration 53, which has now checked all of its applicable edges. The algorithm backtracks again and continues for several more iterations before finding the target list "impossible".

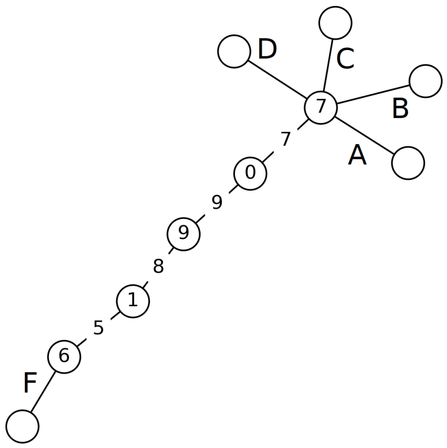


Figure 25: Labeling edge 'E' during iteration 55.

Because this *target_list* has no solutions, the outcome would be the same no matter which algorithm was chosen. However, because the algorithm considered the edge classes, it was able to eliminate possibilities more quickly. For this example, the second *target_list* requires 61 iterations to find it impossible, compared to 918 iterations required if not considering edge classes. Albeit, before that point the algorithm would give up on the mentioned target lists and start trying randomized target lists.

The question remains on whether the Flexible Edge Search Algorithm and all its features finds graceful labelings faster than the Edge Search Algorithm. The next section presents a computational study to answer this question.

5 Computational Study

This section presents a comprehensive computational study to compare the CPU time and number of iterations to solve graceful labeling on trees using the Flexible Edge Search Algorithm versus the Edge Search Algorithm. The study also compares the impact of each of the features discussed in Sections 4.1 and 4.2 in the CPU time and iteration count.

Both the Flexible Edge Search and the Edge Search Algorithms were implemented in Python version 3.10. Computational experiments were performed using an Intel(R) Xeon(R) CPU E31245 @ 3.30GHz processor with 32.0 GB of RAM.

The algorithms were tested on all 7,741 trees with 15 nodes. Observe that testing all 123,867 trees with 18 nodes would take approximately 30 minutes with the Flexible Edge Search Algorithm and a few hours with the Edge Search Algorithm. Testing trees with 20 nodes would take a few hours and a few days, respectively. So testing the set of all trees with 15 nodes strikes a good balance. It provides a large variety of trees, but the solve times are fast enough to allow for many tests. In addition, each test on each tree would end if it reached 10^5 iterations and would be marked as a failure.

To generate the trees, this paper used the Next Tree algorithm by Wright [28]. Next Tree outputs level sequences, such as the examples in Figure 26. Each sequence is unique and is transformed into an *edgelist* representing a unique tree.

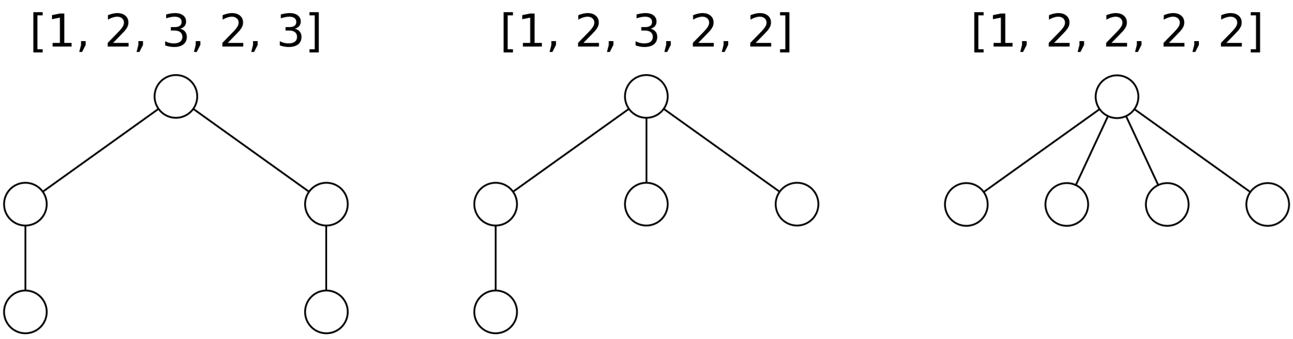


Figure 26: The level sequences and trees with $n = 5$.

5.1 Comparison to the Edge Search Algorithm

Just as the Edge Search Algorithm ("Horton") was created to test *unrooted* trees, the Flexible Edge Search Algorithm ("Flexible") was created to test *rooted* trees. However, it is still possible to conduct both tests on each algorithm. The results of all four tests are presented in Table 1.

Algorithm	Test	Average Iterations	Total CPU Time (s)	Excess Failures
Horton	Trees	344	13.8	0
Flexible	Trees	81	30.7	-
Horton	Roots	12,995	454.5	409
Flexible	Roots	607	75.7	-

Table 1: Comparing Horton's algorithm to this paper's algorithm.

Observe that when testing unrooted trees, "Horton" required more iterations, but solved in less CPU time. This is most likely due to the extra computation required by "Flexible" to classify each edge. However, "Flexible" performed much better when testing all roots, requiring 95% less iterations and approximately $\frac{1}{6}$th of the CPU time. In addition, "Horton" failed to find solutions before the cutoff time for roots in 409 trees.

The following sections present the computational results of several tests that disable one or more features. The goal of these tests is to show the impact of each one on the Flexible Edge Search Algorithm.

5.2 Class Features

Among the features noted in Section 4, several of them consider the class of each edge. Recall that two edges are in the same class if they are isomorphic and directly connected. Many variations on the Flexible Edge Search Algorithm were compared. "Ignore all class features" did not spend any time classifying each edge and did not use any related features. "Try similar roots" tested roots of the same edge class, even if the result was already known. "Try duplicate classes" allowed multiple instances of the same class into the same list of applicable edges. "Try non-descending targets" allowed edge labels to be applied within a class in cases that were already tried by an earlier target list. Finally, the results for the full algorithm are also presented for comparison. Table 2 presents the results of these tests. Note that another feature related to edge class, the maximum depth, is examined in another set of tests.

Test	**Average Iterations**	**Total CPU Time (s)**	**Excess Failures**
Ignore all class features	5,258	272.6	129
Try similar roots	871	96.9	1
Try duplicate classes	4,091	311.5	87
Try non-descending targets	753	80.6	0
Use all class features	607	75.7	-

Table 2: Comparison showing the effect of features related to edge class.

Observe that the largest benefit from a single feature seems to arise from ruling out duplicate classes when forming the list of applicable edges. Figure 27 shows an example of a tree which requires far fewer iterations because of this feature. Note also that the test which ignored all class features saved some time by not finding the edge classes at all.

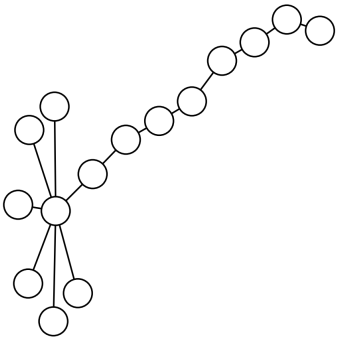


Figure 27: In a test allowed to run indefinitely, ruling out duplicate classes reduces the iteration count for this tree from 384,000 to 1,500.

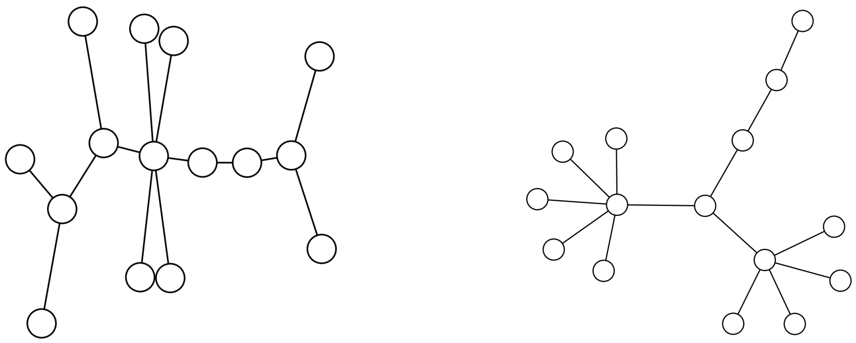
5.3 Minimum and Maximum Depth

As previously mentioned, the *classwise* maximum depth is a class feature, but the maximum depth itself can be calculated in a naive way by counting the maximum number of edges that were ever labeled during a solution attempt. Or the concept can be ignored entirely by assuming *maximum_depth* = n. Similarly, the minimum depth can be either found or assumed. It can be calculated based on the structure surrounding a given root (called the “true” minimum depth) or partially ignored by assuming *minimum_depth* = 3, as is true in the worst case. Or it can be completely ignored by assuming *minimum_depth* = 0. The results of testing all four of these cases compared to the full algorithm are presented in Table 3.

Test	Average Iterations	Total CPU Time (s)	Excess Failures
Max depth = n	752	76.9	0
Naive max depth	736	79.9	0
Min depth = 0	786	90.6	2
Min depth = 3	622	76.4	0
True min depth, classwise max depth	607	75.7	-

Table 3: Comparing the effects of depth-related methods.

Notice that while finding the most accurate versions of the maximum and minimum depth does save some time, the largest difference is between ignoring minimum depth and assuming *minimum_depth* = 3, even though it is larger in some cases. The time spent finding the "true" minimum depth does not reduce the solve time very often. Similarly, finding the maximum depth with respect to class does reduce the number of iterations, but the time loss seems to be roughly equal to the time gained. The differences may be concentrated in specific cases; Figures 28a and 28b show two examples of trees with exceptionally large benefits from each feature.



(a) Using true minimum depth in this case reduced iterations from 4.9 million to 4,800.

(b) Using classwise maximum depth in this case reduced iterations from 6.6 million to 33,000.

Figure 28: The two depth strategies reduce the number of different target lists that need to be checked, but save little time on average.

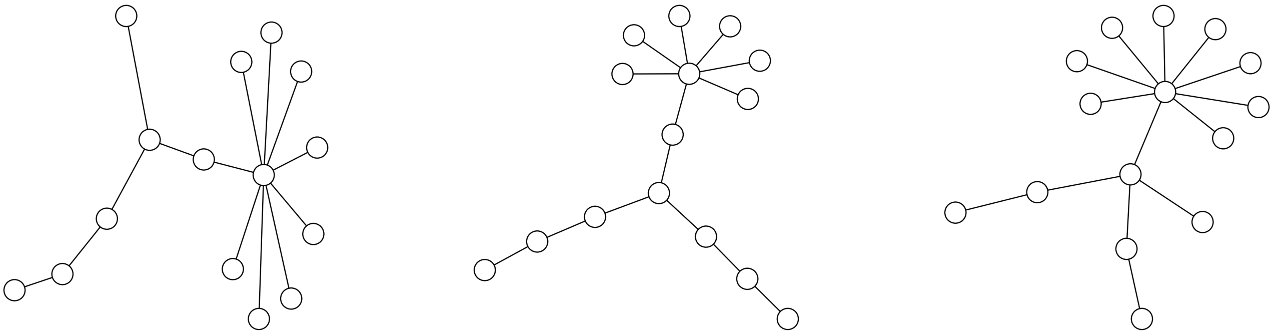
5.4 Target List Strategy

For roots that do not have a solution with the default target list, the target list must be permuted in some way, using the "reverse" or "negate" methods. Additionally, both methods can be tried half of the time, or some times the target list can be randomized, with consideration for the minimum depth. The results of testing these variations can be found in Table 4.

Test	Average Iterations	Total CPU Time (s)	Excess Failures
Reverse only	887	90.2	4
Negate only	719	86.1	2
Reverse, negate	789	87.2	3
Reverse, random	964	102.9	10
Negate, random	950	96.9	13
Reverse, negate, random	607	75.7	-

Table 4: Comparing strategies for permuting the target list.

Although each strategy can be much better than the other for a given tree (see Figure 29), "negate" proved better more often than "reverse". Additionally, switching between two proved worse than "negate" alone, but mixing in some "random" target lists made "negate" much worse. Surprisingly, using all three strategies proved to have the best result.



(a) Case where the 'reverse' strategy is best.

(b) Case where the 'negate' strategy is best.

(c) Case where switching between all strategies is best.

Figure 29: Cases where the different target list strategies outperform the others.

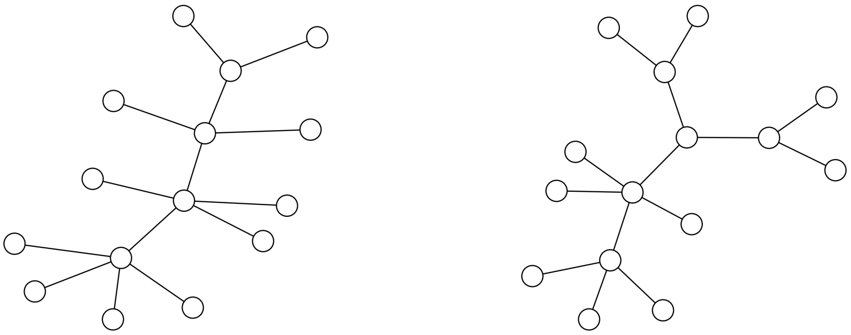
5.5 Edge Priority

For trees with many branches, the algorithm must often decide which branch to try labeling first. As discussed, two methods are "near" and "far", both using the distance from the tree center to assess an edge. However, the "switching" strategy involves using each, depending on whether the iteration count for the root is even or odd. Table 5 presents the results of five different strategies pertaining to this.

Test	Average Iterations	Total CPU Time (s)	Excess Failures
No edge priority	1922	170.4	9
Far	885	95.3	4
Near	1922	169.5	9
Random	1891	168.3	10
Switch far/near	607	75.7	-

Table 5: Comparing strategies for permuting the target list.

Observe that "no edge priority" seems to have the same inferior result as "near"; this is probably due to how Next Tree generates the edgelists from the center of each tree. The "random" method is similar to what Horton mentions in his paper; it does not perform much better than no strategy. However, "far" reduces the time and iterations by about half, and "switching" does better still. "Far" does better than "near" by more than 1000 iterations in 18% of cases (see example Figure 30a), each differed by less than 1000 iterations in 78% of cases and 'near' beat 'far' by more than 1000 iterations in 3% of cases (see example Figure 30b).



(a) Case where "far" strategy solves in 176 iterations while "near" takes 97,000.

(b) Case where "near" strategy solves in 139 iterations while "far" takes 25,000.

Figure 30: The "near" and 'far" strategies can have different effects even on similar trees.

5.6 Summary

Overall the optimizations came in two magnitudes: those that allowed the algorithm to terminate in a reasonable time at all, such as ruling out duplicate classes in the applicable list or considering the edge priority; and those that gave smaller but noticeable improvements, such as not testing similar roots, *classwise* depth or the strategy for permuting the target list.

The recommendation is that for solving graceful labeling on unrooted trees, the Edge Search Algorithm should be used. On the other hand, for solving graceful labeling on rooted trees, the Flexible Edge Search Algorithm is much faster and should be used. Based on computational experiments, the best version of the Flexible Edge Search Algorithm uses all class features, it considers both the minimum and maximum depth, it permutes the target list using the "reverse", "negate", and "random" strategies, and it considers the edge priority by switching between both the "near" and "far" methods. The next section discusses some other findings of the Flexible Edge Search Algorithm.

6 Other Findings and Discussion

On the computational experiments, the Flexible Edge Search Algorithm determined which roots had a graceful solution, and those that were impossible to solve for. Trees with impossible roots were all of the same class: consider a tree with only two edges, attach stars to each endpoint (called claws) and attach a path to the center (called the tail). Because of the resemblance to the constellation Scorpius, this paper calls these trees *scorpions*.

Every non-graceful root was found to be at the end of a scorpion tail, or at the center of a scorpion itself if the tail had length zero. When such a root is not graceful, this node is called the *stinger*. Some scorpions have a claw that is isomorphic to its tail - meaning that there can be two isomorphic stingers. But there were no trees with an impossible root anywhere else, and no tree had more than two impossible roots, and at most one up to isomorphism (see Figure 31). Bear in mind that these are empirical results only, and a non-graceful root may exist in some other form.

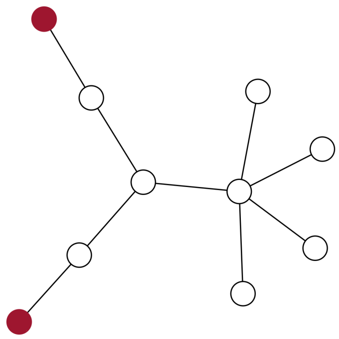


Figure 31: This scorpion has a claw that is isomorphic to its tail, and a stinger at the end of each.

6.1 Graceful Scorpions

Some scorpions are graceful, and all known graceful scorpions fit into one subclass. Cahit surmised that there were apparently several such subclasses [7] and thoroughly described them, while Van Bussel noted only one such subclass by providing a criteria for when they exist [26]. He conjectured that all non-graceful rooted trees are of this form. This class was later illustrated and discussed by Anick [3]. Figure 32 provides a modified description of the single subclass that describes all known graceful scorpions; several examples follow. This section also provides an algorithm for generating all graceful scorpions of this form.

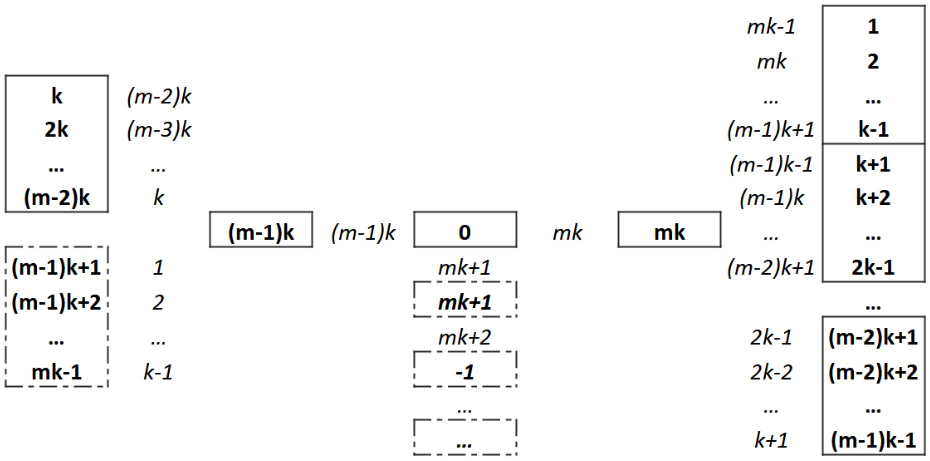
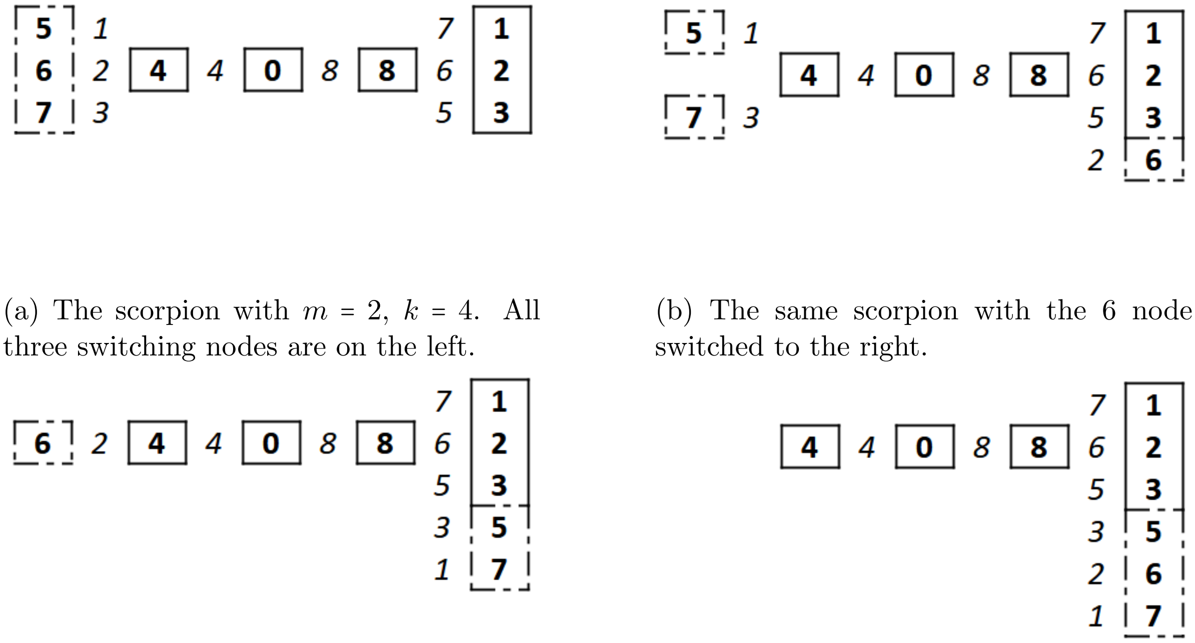


Figure 32: The general form of all known graceful scorpions. Note that the 0-node here may not be the root, unless the tail is length 0. Italics indicate edge labels.

Note a few features. There are a total of $mk + 1$ nodes, not including the tail. The righthand claw can be divided into $m - 1$ sub-claws of $k - 1$ nodes each. Another sub-claw of $k - 1$ nodes can be found on the lower-left; this one is special in that edge pairs from it may be switched over to the other side. Nodes and edges in this claw are called *switchers* and this claw is the *switching claw*. The highest and lowest labels can be switched as a pair, and so on. If there are an odd number of edges in the claw (when $k - 1$ is odd), the middle label can be switched on its own. Lastly, there are $m - 2$ additional edges on the left, one for each of the other sub-claws after the first two; this special sub-claw will be empty if m is less than 3. Altogether the number of nodes is $(m - 1)(k - 1) + k - 1 + m - 2 + 3 = m(k - 1) + m + 1 = mk - m + m + 1 = mk + 1$.

Apart from using m and k, the number of nodes in the left claw, tail, and right claw can be denoted with (l,t,r), respectively, as used by Anick. This notation does not necessarily indicate the labeling pattern itself, as can be seen in Figures 34 and 35.

Consider a few scorpion examples. Figure 33 shows different scorpions all based on the parameters $m = 2$, $k = 4$. This results in a subclaw size of $k - 1 = 3$. There is $m - 1 = 1$ subclaw that is always on the right. Because $m - 2 = 0$, there are no additional edges on the left. As always, there is also one subclaw that could be on the left or the right, or split between the two sides. Figure 33a presents this special subclaw on the left, but the connection $\textcircled{6}\overset{\frown}{2}\textcircled{4}$ on the left could be switched over to the right and become $\textcircled{8}\overset{\frown}{2}\textcircled{6}$ (Figure 33b). Furthermore, the pair of connections $\textcircled{5}\overset{\frown}{1}\textcircled{4}$ and $\textcircled{7}\overset{\frown}{3}\textcircled{4}$ could be switched, as a pair, to the right and become $\textcircled{8}\overset{\frown}{1}\textcircled{7}$ and $\textcircled{8}\overset{\frown}{3}\textcircled{5}$ (Figure 33c). Because there are both a single switcher and a pair of switchers, this allows for this (3,0,3) scorpion to become a (2,0,4), a (1,0,5) or even a (0,0,6) (Figure 33d), and therefore all of the mentioned scorpions are graceful.



(a) The scorpion with $m = 2$, $k = 4$. All three switching nodes are on the left.

(b) The same scorpion with the 6 node switched to the right.

(c) Switching the node pair of 5 and 7 instead. The nodes exchange edge labels.

(d) Switching all three nodes to the right, emptying the left side.

Figure 33: Four different trees can be created based on the parameters $m = 2$, $k = 4$.

But not all scorpions are as accommodating. Consider the case with $m = 2$ and $k = 5$, shown in Figure 34. In this case, there is no single switcher, so while this (4,0,4) scorpion can become a (2,0,6) or a (0,0,8), this scorpion cannot become a (3,0,5) or (1,0,7). Indeed, the (1,0,7) scorpion has a stinger in the center.

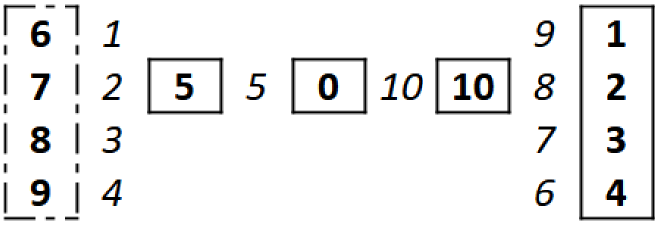


Figure 34: Graceful scorpion with $m = 2$, $k = 5$.

But the (3,0,5) case is not impossible. Consider $m = 5$ and $k = 2$. The subclaw size is $k - 1 = 1$ and there are $m - 1 = 4$ subclaws on the right. Because $m - 2 = 3$, there are 3 extra edges on the left. Figure 35 presents this scorpion as a (4,0,4) with a single switcher; switching the only edge in the switching subclaw allows a (3,0,5) graceful scorpion. Observe also that Figures 34 and 35 present two ways to create a (4,0,4) graceful scorpion.

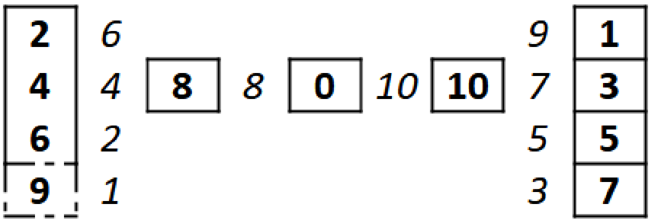


Figure 35: Graceful scorpion with $m = 5$, $k = 2$.

Finally, Figure 36 presents a larger graceful scorpion with $m = 10$, $k = 10$, and a tail of unknown length. This tree is not yet in standard form. The largest edge label is at the end of the tail, and the node at the end of the tail has node label 0 in standard form. Observe that the length of the tail does not affect the labeling pattern in the rest of the tree, other than the offset added to the node labels.

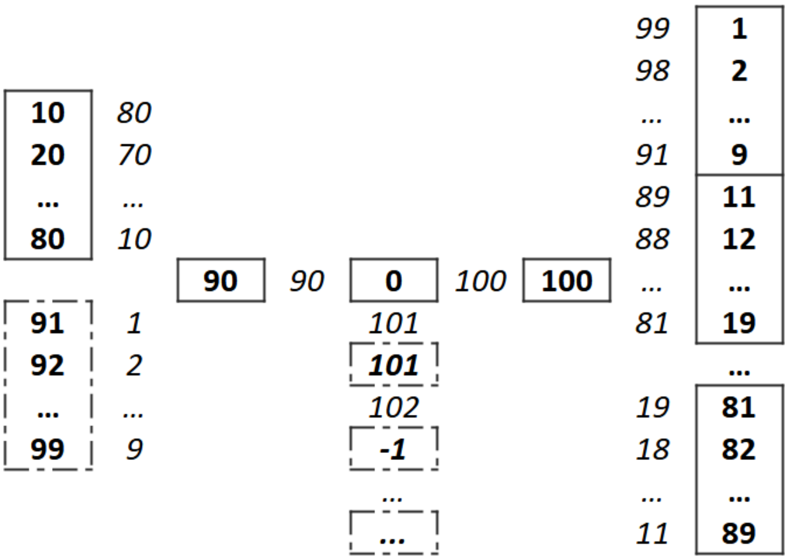


Figure 36: Graceful scorpion with $m = 10$, $k = 10$.

6.2 Generating and Testing Scorpions

Does the form given by Figure 32 predict all graceful scorpions? To find data related to this question, all such scorpions without tails were generated using Algorithm 11, which lists all possible claw sizes a scorpion could have based on the variables m and k mentioned earlier. They were then tested alongside all other tailless scorpions, for 50 nodes or fewer. Empirically, all of the scorpions that were predicted to be graceful actually were, and all other scorpions had an impossible root in the center.

Algorithm 11 : Theorize Graceful Scorpions

```
Require: Values $m$ and $k$ indicating a scorpion with $mk + 1$ nodes;
Ensure: A list of scorpions represented by two claw sizes;
  $unswitched \leftarrow [m - 2, m(k - 1)]$;
  $switched \leftarrow [m - 2 + k - 1, (m - 1)(k - 1)]$;
  $current \leftarrow unswitched$;
  create an empty list of scorpions;
  while $current[0] < switched[0]$ do
      if $current[0] \leq current[1]$ then
          append current to scorpions;
      else if $current[0] > current[1]$ then
          append the reverse of current to scorpions;
      if $k - 1$ is even then
          Add 2 to $current[0]$;
          Subtract 2 from $current[1]$;
      else if $k - 1$ is odd then
          Add 1 to $current[0]$;
          Subtract 1 from $current[1]$;
  return scorpions;
```

Observe that if k is odd, then the subclaw size $k - 1$ is even and only pairs of edges can be switched. But if k is even, then $k - 1$ is odd and the middle edge can be switched on its own. The cases where $k - 1$ is even create gaps so that many scorpions cannot be generated with this algorithm. If these gaps are not filled by a different combination of m and k, then the scorpion will have a stinger in the center, according to the empirical results.

Table 6 shows the results for each n, including how many scorpions were graceful for all roots, how many had a non-graceful root, and running totals for each. This confirms and extends the results obtained by Van Bussel. Overall, it appears that the number of tailless scorpions with a stinger approaches the number with none. Algorithm 11 predicts that for $n \leq 100$, 47% of tailless scorpions have stingers.

n	Tailless Scorpions	All Roots Graceful	Stinger Root	% Total No Stinger	% Total with Stinger
3	1	1	0	1.00	0.00
4	1	1	0	1.00	0.00
5	2	2	0	1.00	0.00
6	2	1	1	0.83	0.17
7	3	3	0	0.89	0.11
8	3	1	2	0.75	0.25
9	4	4	0	0.81	0.19
10	4	3	1	0.80	0.20
11	5	4	1	0.80	0.20
12	5	1	4	0.70	0.30
13	6	6	0	0.75	0.25
14	6	1	5	0.67	0.33
15	7	5	2	0.67	0.33
16	7	4	3	0.66	0.34
17	8	8	0	0.70	0.30
18	8	1	7	0.64	0.36
19	9	9	0	0.68	0.32
20	9	1	8	0.62	0.38
21	10	10	0	0.66	0.34
22	10	5	5	0.65	0.35
23	11	7	4	0.64	0.36
24	11	1	10	0.60	0.40
25	12	12	0	0.63	0.37
26	12	4	8	0.61	0.39

Table 6: Results of testing all tailless scorpions for $n \leq 50$.

n	Tailless Scorpions	All Roots Graceful	Stinger Root	% Total No Stinger	% Total with Stinger
27	13	8	5	0.61	0.39
28	13	6	7	0.60	0.40
29	14	14	0	0.63	0.37
30	14	1	13	0.59	0.41
31	15	14	1	0.61	0.39
32	15	1	14	0.58	0.42
33	16	16	0	0.61	0.39
34	16	7	9	0.60	0.40
35	17	10	7	0.60	0.40
36	17	5	12	0.58	0.42
37	18	18	0	0.60	0.40
38	18	1	17	0.57	0.43
39	19	11	8	0.57	0.43
40	19	8	11	0.57	0.43
41	20	20	0	0.59	0.41
42	20	1	19	0.56	0.44
43	21	19	2	0.58	0.42
44	21	1	20	0.55	0.45
45	22	22	0	0.57	0.43
46	22	9	13	0.57	0.43
47	23	13	10	0.57	0.43
48	23	1	22	0.55	0.45
49	24	24	0	0.56	0.44
50	24	5	19	0.55	0.45

Table 7: Results of testing all tailless scorpions for $n \leq 50$ - continued.

7 Conclusions and Further Research

This paper presents the Flexible Edge Search Algorithm, designed to solve graceful tree labeling for rooted trees. The primary feature of the algorithm is the flexible target list, which allows the edge targets to be tried in non-descending order. However, this greatly increases the possible solve time, so several other features are used to cut this down. Some features consider the class of each edge, which was found based on whether two edges were isomorphic and adjacent. The largest benefit came from ruling out edges of the same class at each iteration. There were also considerations of minimum and maximum depth, and different strategies for choosing the next edge to label and how to find the next target list.

The Flexible Edge Search Algorithm was able to solve for all roots much more quickly than its predecessor, and more accurately. While the Edge Search Algorithm had 409 instances of either exceeding 100,000 iterations or of erroneously marking roots as non-graceful, the Flexible Edge Search Algorithm never required more than 10,000 iterations and averaged just 607, which is about 40 per root or just 3 per edge per root. As mentioned, the most beneficial feature was ruling out duplicate edges; using the “switching” edge priority helped second most, followed by ruling out similar roots, and using all of the different methods of changing the target list in tandem.

In empirical testing, all of the trees with non-graceful roots looked remarkably similar, as has been noted in work by Van Bussel and Anick. This paper introduces the name *scorpion* for these trees and offers a new description of their structure, and a simple algorithm for predicting which scorpions are not graceful. It is still not known whether these scorpions represent every case of a non-graceful root, but all scorpions with up to 50 nodes conform to this same pattern.

The topic of stingers presents an area of future research. Theorem 2 showed that the graceful property (or lack thereof) is transferred when a leaf is added or removed. But what happens to a non-graceful root itself when a leaf is added? Empirically, two stingers were never adjacent, and attaching a leaf to a stinger always caused the stinger to become graceful in the induced tree. This leads to Conjecture 1, which would provide a more specific step towards the GTC.

Conjecture 1. *Attaching a leaf* $\textcircled{B}$ *to a non-graceful root* $\textcircled{A}$ *will have* $\textcircled{A'}$ *be graceful in the induced tree.*

References

[1] Md Momin Al Aziz et al. "Graceful labeling of trees: Methods and applications". In: *2014 17th International Conference on Computer and Information Technology (ICCIT)*. IEEE. 2014, pp. 92–95.

[2] REL Aldred and Brendan D McKay. "Graceful and harmonious labellings of trees". In: *Bull. Inst. Combin. Appl* 23 (1998), pp. 69–72.

[3] David Anick. "Counting graceful labelings of trees: A theoretical and empirical study". In: *Discrete Applied Mathematics* 198 (2016), pp. 65–81.

[4] Jean-Claude Bermond and Dominique Sotteau. "Graph decompositions and G-designs". In: *5th British Combinatorial Conference, 1975, Congressus Numerantium 15, Utilitas Math Pub.* 1976, pp. 53–72.

[5] M Best, Peter van Emde Boas, and LENSTRA HW JR. "A Sharpened Version fo the Aanderaa-Rosenberg Conjecture". In: (1974).

[6] Ljiljana Brankovic and Michael J Reynolds. "Computer search for graceful-like labelling: A survey." In: *Electronic Journal of Graph Theory & Applications* 10.1 (2022).

[7] I Cahit. "On Zero-Rotatable Small Graceful Trees: Caterpillars". In: *Science Direct Working Paper* S1574-0358 (2002), p. 04.

[8] FRK Chung and FK Hwang. "Rotatable graceful graphs". In: *Ars Combin* 11 (1981), pp. 239–250.

[9] Michelle Edwards and Lea Howard. "A survey of graceful trees". In: *Atlantic Electronic Journal of Mathematics* 1.1 (2006), pp. 5–30.

[10] *Entry A000055 in The On-Line Encyclopedia of Integer Sequences*. 2024. URL: https://oeis.org/A337274.

[11] *Entry A337274 in The On-Line Encyclopedia of Integer Sequences*. 2024. URL: https://oeis.org/A337274.

[12] Wenjie Fang. "A computational approach to the graceful tree conjecture". In: *arXiv preprint arXiv:1003.3045* (2010).

[13] Joseph A Gallian. "A dynamic survey of graph labeling". In: *Electronic Journal of combinatorics* 1.DynamicSurveys (2018), DS6.

[14] Edinah K Gnang. "A proof of the Kotzig-Ringel-Rosa Conjecture". In: *arXiv preprint arXiv:2202.03178* (2022).

[15] Solomon W Golomb. "How to number a graph". In: *Graph theory and computing*. Elsevier, 1972, pp. 23–37.

[16] Michael Horton. "Graceful trees: statistics and algorithms". PhD thesis. University of Tasmania, 2003.

[17] Pavel Hrnčiar and Alfonz Haviar. "All trees of diameter five are graceful". In: *Discrete Mathematics* 233.1-3 (2001), pp. 133–150.

[18] Charlotte Huang, Anton Kotzig, and Alexander Rosa. "Further results on tree labellings". In: *Util. math., 21c* 3148 (1982).

[19] Peter Keevash and Katherine Staden. "Ringel's tree packing conjecture in quasirandom graphs". In: *arXiv preprint arXiv:2004.09947* (2020).

[20] Richard Montgomery, Alexey Pokrovskiy, and Benny Sudakov. "A proof of Ringel's conjecture". In: *Geometric and Functional Analysis* 31.3 (2021), pp. 663–720.

[21] G Ringel. "Problem 25, Theory of Graphs and its Applications (Proc. Int. Symp. Smolenice 1963)". In: *Czech. Acad. Sci., Prague* (1963).

[22] Elina Robeva. "An extensive survey of graceful trees". In: *Undergraduate Honours Thesis, Standford University, USA* (2011).

[23] Rafael I Rofa. "A Graceful Algebraic Function Labelling of Rooted Symmetric Trees". In: *arXiv preprint arXiv:2109.09511* (2021).

[24] Alexander Rosa et al. "On certain valuations of the vertices of a graph". In: *Theory of Graphs (Internat. Symposium, Rome)*. 1966, pp. 349–355.

[25] Hui Sun, Xiaohui Zhang, and Bing Yao. "Construction of New Graphical Passwords with Graceful-Type Labellings on Trees". In: *2018 2nd IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. IEEE. 2018, pp. 1491–1494.

[26] Frank Van Bussel. "0-Centred and 0-ubiquitously graceful trees". In: *Discrete mathematics* 277.1-3 (2004), pp. 193–218.

[27] Tao-Ming Wang et al. "Infinitely many equivalent versions of the graceful tree conjecture". In: *Applicable Analysis and Discrete Mathematics* (2015), pp. 1–12.

[28] Robert Alan Wright et al. "Constant time generation of free trees". In: *SIAM Journal on Computing* 15.2 (1986), pp. 540–548.