A Pivot Gray Code Listing for the Spanning Trees of the Fan Graph

Ben Cameron, Aaron Grubb ✉, and Joe Sawada

University of Guelph, Guelph, Canada {ben.cameron,agrubb,jsawada}@uoguelph.ca

Abstract. We use a greedy strategy to list the spanning trees of the fan graph, F_n, such that successive trees differ by pivoting a single edge around a vertex. It is the first greedy algorithm for exhaustively generating spanning trees using such a minimal change operation. The resulting listing is then studied to find a recursive algorithm that produces the same listing in $O(1)$-amortized time using $O(n)$ space. Additionally, we present $O(n)$-time algorithms for ranking and unranking the spanning trees for our listing; an improvement over the generic $O(n^3)$-time algorithm for ranking and unranking spanning trees of an arbitrary graph.

Keywords: spanning tree · greedy algorithm · fan graph · combinatorial generation.

1 Introduction

This paper is concerned with the algorithmic problem of listing all spanning trees of the fan graph. Applications of efficiently listing all spanning trees of general graphs are ubiquitous in computer science and also appear in many other scientific disciplines [3]. In fact, one of the earliest known works on listing all spanning trees of a graph is due to the German physicist Wilhelm Feussner in 1902 who was motivated by an application to electrical networks [7]. In the 120 years since Feussner's work, many new algorithms have been developed, such as those in the following citations [1, 4, 6, 8, 9, 12–17, 19–22, 24].

For any application, it is desirable for spanning tree listing algorithms to have the asymptotically best possible running time, that is, $O(1)$-amortized running time. The algorithms due to Kapoor and Ramesh [14], Matsui [16], Smith [22], Shioura and Tamura [20] and Shioura et al. [21] all run in $O(1)$-amortized time. Another desirable property of such listings is to have the *revolving-door* property, where successive spanning trees differ by the addition of one edge and the removal of another. Such listings where successive objects in a listing differ by a constant number of simple operations are more generally known as *Gray codes*. The algorithms due to Smith [22], Kamae [13], Kishi and Kajitani [15], Holzmann and Harary [12] and Cummins [6] all produce Gray code listings of spanning trees for an arbitrary graph. Of all of these algorithms, Smith's is the only one that produces a Gray code listing in $O(1)$-amortized time. A stronger notion of a Gray code for spanning trees is where the revolving-door makes strictly local changes. More specifically, we would like the differing edges to share a common endpoint. Such a Gray code property, which we call a *pivot Gray code*, is not given by any previously known algorithm. This leads to our first research question.

Research Question #1 Given a graph G (perhaps from a specific class), does there exist a pivot Gray code listing of all spanning trees of G? Furthermore, can the listing be generated in polynomial (ideally constant) time per tree using polynomial space?

A related question that arises for any listing is how to *rank*, that is, find the position of the object in the listing, and *unrank*, that is, return the object at a specific rank. For spanning trees, an $O(n^3)$-time algorithm for ranking and unranking a spanning tree of a specific listing for an arbitrary graph is known [5].

Research Question #2 Given a graph G (perhaps from a specific class), does there exist a (pivot Gray code) listing of all spanning trees of G that can be ranked and unranked in $O(n^2)$ time or better?

An algorithmic technique recently found to have success in the discovery of Gray codes is the greedy approach. An algorithm is said to be *greedy* if it can prioritize allowable actions according to some criteria, and then choose the highest priority action that results in a unique object to obtain the next object in the listing. When applying a greedy algorithm, there is no backtracking; once none of the valid actions lead to a new object in the set under consideration, the algorithm halts, even if the listing is not exhaustive. The work by Williams [23] notes that some very well-known combinatorial listings can be constructed greedily, including the binary reflected Gray code (BRGC) for binary strings, the plain change order for permutations, and the lexicographically smallest de Bruijn sequence. Recently, a very powerful greedy algorithm on permutations (known as Algorithm J, where J stands for "jump") generalizes many known combinatorial Gray code listings including many related to permutation patterns, rectangulations, and elimination trees [10, 11, 18]. However, no greedy algorithm was previously known to list the spanning trees of an arbitrary graph.

Research Question #3 Given a graph G (perhaps from a specific class), does there exist a greedy strategy to list all spanning trees of G? Moreover, does such a greedy strategy exist where the resulting listing is a pivot Gray code?

In most cases, a greedy algorithm requires exponential space to recall which objects have already been visited in a listing. Thus, answering this third question would satisfy only the first part of **Research Question #1**. However, in many cases, an underlying pattern can be found in a greedy listing which can result in space efficient algorithms [10, 23].

To address these three research questions, we applied a variety of greedy approaches to structured classes of graphs including the fan, wheel, n-cube, and the compete graph. From this study, we were able to affirmatively answer each of the research questions for the fan graph. It remains an open question to find similar results for other classes of graphs.

1.1 New Results

The *fan graph* on n vertices, denoted F_n, is obtained by joining a single vertex (which we label v_∞) to the path on $n-1$ vertices (labeled $v_2, ..., v_n$) – see Fig. 1.

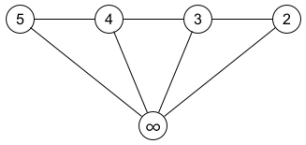

Fig. 1: F_5

Note that we label the smallest vertex v_2 so that the largest non-infinity labeled vertex equals the total number of vertices. Let $\mathbf{T}_n$ denote the set of all spanning trees of F_n. We discover a greedy strategy to generate $\mathbf{T}_n$ in a pivot Gray code order. We describe this greedy strategy in Section 2. The resulting listing is studied to find an $O(1)$-amortized time recursive algorithm that produces the same listing using only $O(n)$ space, which is presented in Section 3. We also show how to rank and unrank a spanning tree of the greedy listing in $O(n)$ time in Section 3, which is a significant improvement over the general $O(n^3)$-time ranking and unranking that is already known. We conclude with a summary in Section 4.

2 A Greedy Generation for $\mathbf{T}_n$

With our goal to discover a pivot Gray code listing of $\mathbf{T}_n$, we tested a variety of greedy approaches. There are two important issues when considering a greedy approach to list spanning trees: (1) the labels on the vertices (or edges) and (2) the starting tree. For each of our approaches, we prioritized our operations by first considering which vertex u to pivot on, followed by an ordering of the endpoints considered in the addition/removal. We call the vertex u the *pivot*.

Our initial attempts focused only on pivots that were leaves. As a specific example, we ordered the leaves (pivots) from smallest to largest. Since each leaf u is attached to a unique vertex v in the current spanning tree, we then considered the neighbours w of u in increasing order of label. We restricted the labeling of the vertices to the most natural ones, such as the one presented in Section 1.1. For each strategy we tried all possible starting trees. Unfortunately, none of our attempts lead to exhaustive listings. Applying these strategies on the wheel, n-cube, and complete graph was also unsuccessful.

By allowing the pivot to be any arbitrary vertex, we experimentally discovered several exhaustive listings for $\mathbf{T}_n$ for n up to 12 (testing every starting tree for $n = 12$ took about eight hours). One listing stood out as having an easily defined starting tree as well as a nice pattern which we could study to construct the listing more efficiently. It applied the labeling of the vertices as described in Section 1.1 with the following prioritization of pivots and their incident edges:

> Prioritize the pivots u from smallest to largest and then for each pivot, prioritize the edges uv that can be removed from the current tree in increasing order of the label on v, and for each such v, prioritize the edges uw that can be added to the current tree in increasing order of the label on w.

Since this is a greedy strategy, if an edge pivot results in a spanning tree that has already been generated or a graph that is not a spanning tree, then the next highest priority edge pivot is attempted. Let $\textsc{Greedy}(T)$ denote the listing that results from applying this greedy approach starting with the spanning

tree T. The starting tree that produced a nice exhaustive listing was the path $v_\infty, v_2, v_3, \ldots, v_n$, denoted P_n throughout the paper. Fig. 2 shows the listings $\textsc{Greedy}(P_n)$ for $n = 2, 3, 4, 5$. The listing $\textsc{Greedy}(P_6)$ is illustrated in Fig. 3. It is worth noting that starting with the path $v_\infty, v_n, v_{n-1}, \ldots, v_2$ or the star (all edges incident to v_∞) did not lead to an exhaustive listing of $\mathbf{T}_n$.

As an example of how the greedy algorithm proceeds, consider the listing $\textsc{Greedy}(P_5)$ in Fig. 2. When the current tree T is the 16th one in the listing (the one with edges $\{v_2v_\infty, v_2v_3, v_3v_4, v_5v_\infty\}$), the first pivot considered is v_2. Since both v_2v_3 and v_2v_∞ are present in the tree, no valid move is available by pivoting on v_2. The next pivot considered is v_3. Both edges v_3v_2 and v_3v_4 are incident with v_3. First, we attempt to remove v_3v_2 and add v_3v_∞, which results in a tree previously generated. Next, we attempt to remove v_3v_4 and add v_3v_∞, which results in a cycle. So, the next pivot, v_4, is considered. The only edge incident to v_4 is v_4v_3. By removing v_4v_3 and adding v_4v_5 we obtain a new spanning tree, the next tree in the greedy listing.

To prove that $\textsc{Greedy}(P_n)$ does in fact contain all trees in $\mathbf{T}_n$, we demonstrate it is equivalent to a recursively constructed listing that we obtain by studying the greedy listings. Before we describe this recursive construction we mention one rather remarkable property of $\textsc{Greedy}(P_n)$ that we will also prove in the next section: If X_n is last tree in the listing $\textsc{Greedy}(P_n)$, then $\textsc{Greedy}(X_n)$ is precisely $\textsc{Greedy}(P_n)$ in reverse order.

3 An $O(1)$-amortized time Pivot Gray Code Generation for $\mathbf{T}_n$

In this section we develop an efficient recursive algorithm to construct the listing $\textsc{Greedy}(P_n)$. The construction generates some sub-lists in reverse order, similar to the recursive construction of the BRGC. The recursive properties allow us to provide efficient ranking and unranking algorithms for the listing based on counting the number of trees at each stage of the construction. Let t_n denote the number of spanning trees of F_n. It is known that

$$t_n = f_{2(n-1)} = 2\frac{((3-\sqrt{5})/2)^n - ((3+\sqrt{5})/2)^{n-2}}{5-3\sqrt{5}},$$

where f_n is the nth number of the Fibonacci sequence with $f_1 = f_2 = 1$ [2].

By studying the order of the spanning trees in $\textsc{Greedy}(P_n)$, we identified four distinct stages S1, S2, S3, S4 that are highlighted for $\textsc{Greedy}(P_6)$ in Fig. 3. From this figure, and referring back to Fig. 2 to see the recursive properties, observe that:

- The trees in S1 are equivalent to $\textsc{Greedy}(P_5)$ with the added edge v_6v_5.
- The trees in S2 are equivalent to the reversal of the trees in $\textsc{Greedy}(P_5)$ with the added edge v_6v_∞.

The trees in S3 and S4 have both edges v_6v_5 and v_6v_∞ present.

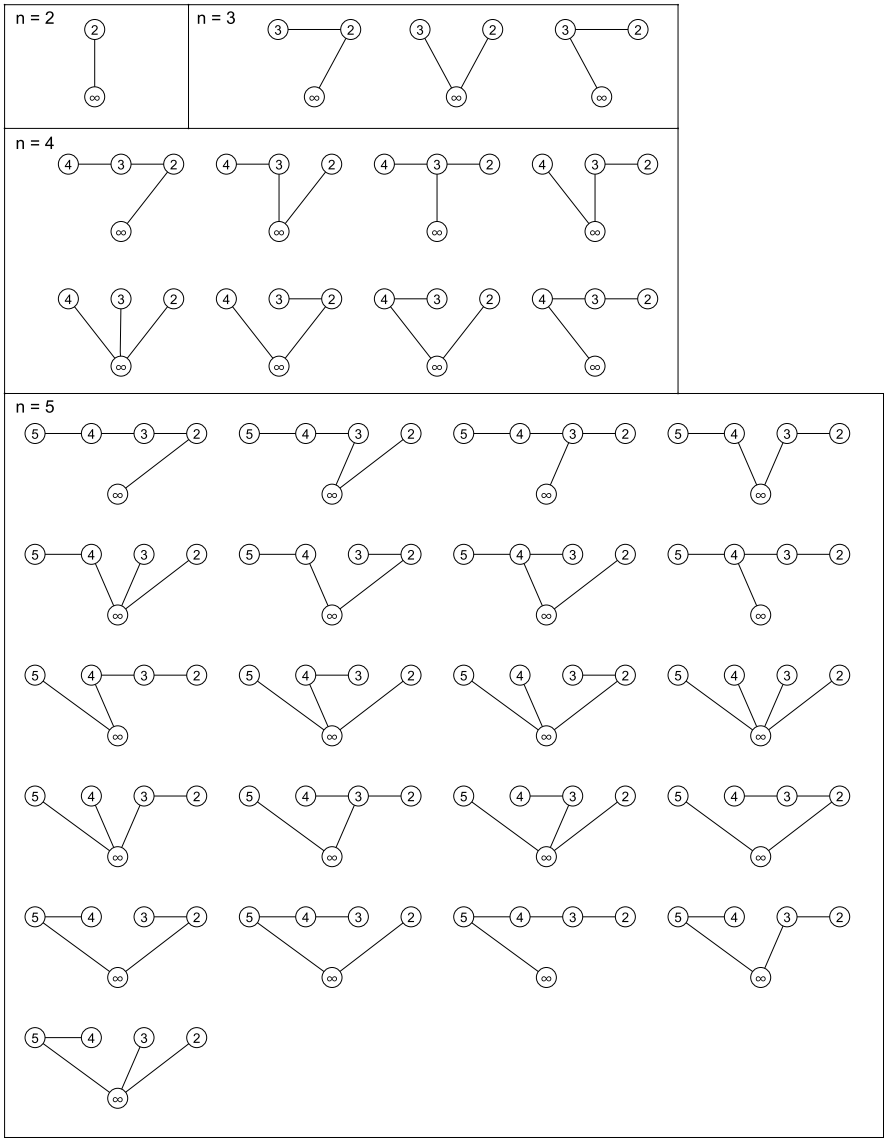

Fig. 2: GREEDY(P_n) for $n = 2, 3, 4, 5$. Read left to right, top to bottom.

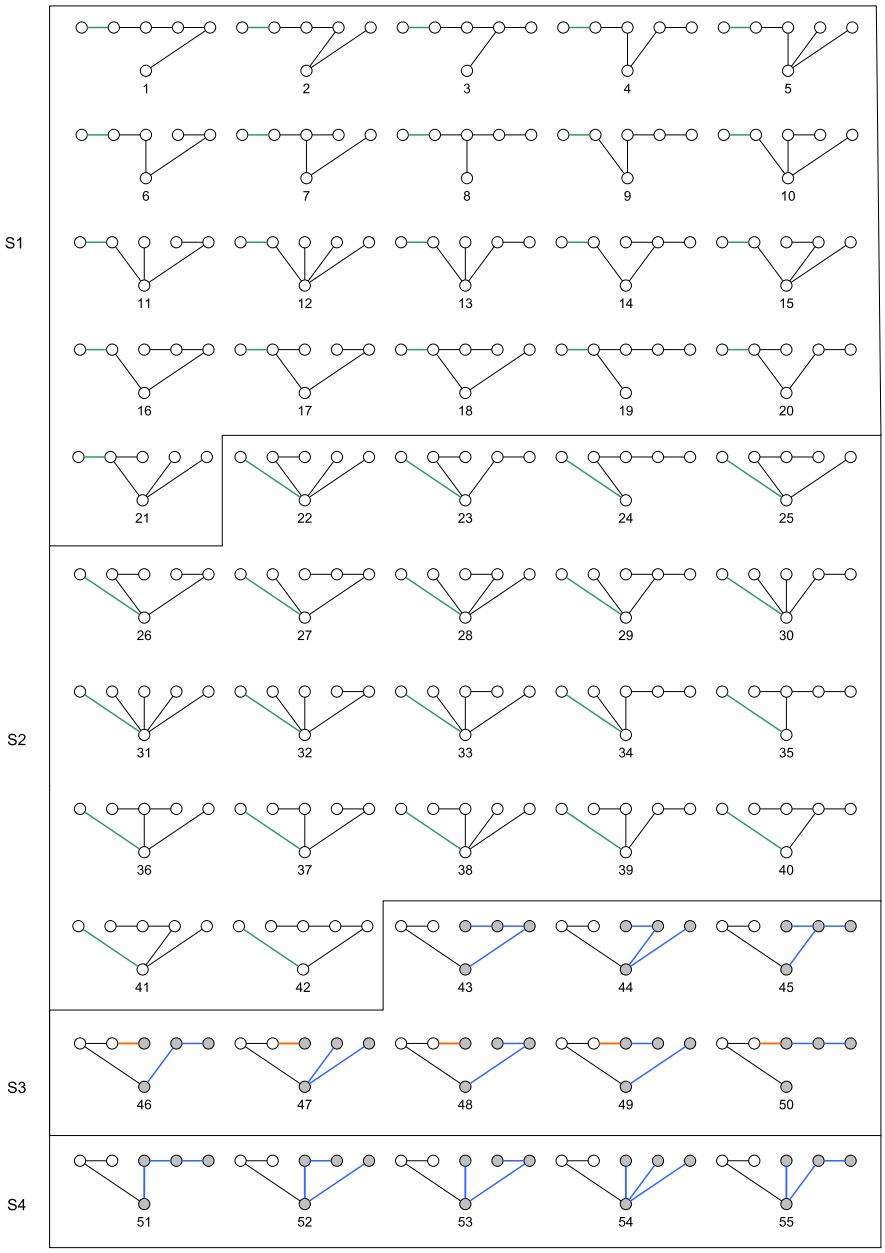

Fig. 3: GREEDY(P_6) read from left to right, top to bottom. Observe that S1 is GREEDY(P_5) with v_6v_5 added, S2 is the reverse of GREEDY(P_5) with v_6v_∞ added, S3 is GREEDY(P_4) with v_6v_5 and v_6v_∞ added, except the edge v_4v_∞ is replaced by v_4v_5, and S4 is the last five trees of GREEDY(P_4) in reverse order (v_4v_∞ is now present) with v_6v_5 and v_6v_∞ added.

– In S3, focusing only on the vertices v_4, v_3, v_2, v_∞, the induced subgraphs correspond to $\textsc{Greedy}(P_4)$, except whenever v_4v_∞ is present, it is replaced with v_4v_5 (the last five trees).
– In S4, focusing only on the vertices v_4, v_3, v_2, v_∞, the induced subgraphs correspond to the trees in $\textsc{Greedy}(P_4)$ where v_4v_∞ is present, in reverse order.

Generalizing these observations for all $n \geq 2$ leads to the recursive procedure $\textsc{Gen}(k, s_1, \mathit{varEdge})$ given in Algorithm 1, which uses a global variable T to store the current spanning tree with n vertices. The parameter k indicates the number of vertices under consideration; the parameter s_1 indicates whether or not to generate the trees in stage S1, as required by the trees for S4; and the parameter *varEdge* indicates whether or not a variable edge needs to be added as required by the trees for S3. The procedure $\textsc{RevGen}(k, s_1, \mathit{varEdge})$, which is left out due to space constraints, simply performs the operations from $\textsc{Gen}(k, s_1, \mathit{varEdge})$ in reverse order. For each algorithm the base cases correspond to the edge moves in the listings $\textsc{Greedy}(P_2)$ and $\textsc{Greedy}(P_3)$.

Let $\mathcal{G}_n$ denote the listing obtained by initializing T to P_n, printing T, and calling $\textsc{Gen}(n, 1, 0)$. Let L_n denote the last tree in this listing. Let $\mathcal{R}_n$ denote the listing obtained by initializing T to L_n, printing T, and calling $\textsc{RevGen}(n, 1, 0)$. Thus, $\mathcal{R}_n$ is the the listing $\mathcal{G}_n$ in reverse order.

Algorithm 1

```
1:  procedure Gen(k, s_1, varEdge)
2:      if k = 2 then                                              ▷ F_2 base case
3:          if varEdge then T ← T − v_2v_∞ + v_2v_3; Print(T)
4:      else if k = 3 then                                         ▷ F_3 base case
5:          if s_1 then
6:              if varEdge then T ← T − v_3v_2 + v_3v_4; Print(T)
7:              else T ← T − v_3v_2 + v_3v_∞; Print(T)
8:          T ← T − v_2v_∞ + v_2v_3; Print(T)
9:      else
10:         if s_1 then
11:             Gen(k − 1, 1, 0)                                   ▷ S1
12:             if varEdge then T ← T − v_kv_{k−1} + v_kv_{k+1}; Print(T)
13:             else T ← T − v_kv_{k−1} + v_kv_∞; Print(T)
14:         RevGen(k − 1, 1, 0)                                    ▷ S2
15:         T ← T − v_{k−1}v_{k−2} + v_{k−1}v_k; Print(T)
16:         Gen(k − 2, 1, 1)                                       ▷ S3
17:         if k > 4 then T ← T − v_{k−2}v_{k−1} + v_{k−2}v_∞; Print(T)
18:         RevGen(k − 2, 0, 0)                                    ▷ S4
```

Our goal is to show that $\mathcal{G}_n$ exhaustively lists all trees in $\mathbf{T}_n$ and moreover, the listing is equivalent to $\textsc{Greedy}(P_n)$. We accomplish this in two steps: first we show that $\mathcal{G}_n$ has the required size, then we show that $\mathcal{G}_n$ is equivalent to

GREEDY(P_n). Before doing this, we first comment on some notation. Let $T - v_i$ denote the tree obtained from T by deleting the vertex v_i along with all edges that have v_i as an endpoint. Let $T + v_iv_j$ (resp. $T - v_iv_j$) denote the tree obtained from T by adding (resp. deleting) the edge v_iv_j. For the remainder of this section, we will let T_n denote the tree T specified as a global variable for GEN and REVGEN, and we let $T_{n-1} = T - v_n$ and $T_{n-2} = T - v_n - v_{n-1}$.

Lemma 1. *For $n \geq 2$, $|\mathcal{G}_n| = |\mathcal{R}_n| = t_n$.*

Proof. This result applies the Fibonacci recurrence and straightforward induction by counting the number of trees recursively generated in each stage S1, S2, S3, S4 as described earlier in this section. The base cases for $n = 2, 3, 4$ are easily verified by stepping through the algorithms. A formal proof is omitted due to space constraints. □

To prove the next result, we first detail some required terminology. If $T \in \mathbf{T}_n$, then we say that the operation of deleting an edge v_iv_j and adding an edge v_iv_k is a *valid* edge move of T if the result is a tree in $\mathbf{T}_n$ that has not been generated yet. Conversely, if the result is not a tree in $\mathbf{T}_n$, or the result is a tree that has already been generated, then it is not a *valid* edge move of T. We say an edge v_iv_j is *smaller* than edge v_iv_k if $j < k$. An edge move $T_n - v_iv_j + v_iv_k$ is said to be *smaller* than another edge move $T_n - v_xv_y + v_xv_z$ if $i < x$, if $i = x$ and $j < y$, or if $i = x$, $j = y$, and $k < z$.

Lemma 2. *For $n \geq 2$, $\mathcal{G}_n =$ GREEDY(P_n) and $\mathcal{R}_n =$ GREEDY(L_n).*

Proof. By induction on n. It is straightforward to verify that the result holds for $n = 2, 3, 4$ by iterating through the algorithms. Assume $n > 4$, and that $\mathcal{G}_j =$ GREEDY(P_j) and $\mathcal{R}_j =$ GREEDY(L_j) for $2 \leq j < n$. We begin by showing $\mathcal{G}_n =$ GREEDY(P_n), breaking the proof into each of the four stages for clarity.

S1: Since $n > 4$ and $s_1 = 1$, GEN($n-1, 1, 0$) is executed. By our inductive hypothesis, $\mathcal{G}_{n-1} =$ GREEDY(P_{n-1}). These must be the first trees for GREEDY(P_n), as any edge move involving v_nv_{n-1} or v_nv_∞ is larger than any edge move made by GREEDY(P_{n-1}). Since GREEDY(P_{n-1}) halts, it must be that no edge move of T_{n-1} is possible. So GREEDY(P_n) must make the next smallest edge move, which is $T_n - v_nv_{n-1} + v_nv_\infty$. Since T_n is a spanning tree, it follows that $T_n - v_nv_{n-1} + v_nv_\infty$ is also a spanning tree (and has not been generated yet), and therefore the edge move is valid. At this point, GEN($n, 1, 0$) also makes this edge move, by line 13.

S2: REVGEN($n-1, 1, 0$) ($T_{n-1} = L_{n-1}$) is then executed. By our inductive hypothesis, $\mathcal{R}_n =$ GREEDY(L_{n-1}). Since GREEDY(L_{n-1}) halts, it must be that no edge moves of T_{n-1} are possible. At this point, $T_{n-1} = P_{n-1}$ because REVGEN($n-1, 1, 0$) was executed. The smallest edge move now remaining is $T_n - v_{n-2}v_{n-1} + v_nv_{n-1}$. This results in $T_n = P_{n-2} + v_nv_{n-1} + v_nv_\infty$, which is a spanning tree that has not been generated. So, GREEDY(P_n) must make

this move. GEN$(n, 1, 0)$ also makes this move, by line 15. So, $\mathcal{G}_n$ must equal GREEDY(P_n) up to the end of S2.

S3: Next, GEN$(n-2, 1, 1)$ starting with $T_{n-2} = P_{n-2}$ is executed. Since $varEdge = 1$, $v_{n-2}v_{n-1}$ is added instead of $v_{n-2}v_\infty$. GREEDY(P_n) also adds $v_{n-2}v_{n-1}$ instead of $v_{n-2}v_\infty$ since $v_{n-2}v_{n-1}$ is smaller than $v_{n-2}v_\infty$ and this edge move results in a tree not yet generated. Other than the difference in this one edge move, which occurs outside the scope of T_{n-2}, GEN$(n-2, 1, 0)$ and GEN$(n-2, 1, 1)$ (both starting with $T_{n-2} = P_{n-2}$) make the same edge moves. Since we also know that $\mathcal{G}_{n-2} =$ GREEDY(P_{n-2}) by the inductive hypothesis, it follows that $\mathcal{G}_n$ continues to equal GREEDY(P_n) after line 16 of GEN$(n, 1, 0)$ is executed. We know that $T_{n-2} = L_{n-2}$ after GEN$(n-2, 1, 0)$. However, $T_{n-2} = L_{n-2} - v_{n-2}v_\infty + v_{n-2}v_{n-1}$ instead because GEN$(n-2, 1, 1)$ was executed ($varEdge = 1$). It must be that no edge moves of T_{n-2} are possible because GREEDY(P_{n-2}) (and GEN$(n-2, 1, 1)$) halted. The smallest edge move now remaining is $T_n - v_{n-2}v_{n-1} + v_{n-2}v_\infty$. This results in $T_{n-2} = L_{n-2}$. Also, $T_n = T_{n-2} + v_nv_{n-1} + v_nv_\infty$ is a spanning tree since T_{n-2} is a spanning tree of F_{n-2}. So GREEDY(P_n) makes this move. GEN$(n, 1, 0)$ also makes this move, by line 17, and thus $\mathcal{G}_n =$ GREEDY(P_n) up to the end of S3.

S4: Finally, REVGEN$(n-2, 0, 0)$ starting with $T_{n-2} = L_{n-2}$ is executed. By our inductive hypothesis, $\mathcal{R}_{n-2} =$ GREEDY(L_{n-2}). From the recursive definition of REVGEN, it is clear that REVGEN$(n-2, 0, 0)$ and REVGEN$(n-2, 1, 0)$ make the same edge moves until REVGEN$(n-2, 0, 0)$ finishes executing. So, by the inductive hypothesis, the listings produced by REVGEN$(n-2, 0, 0)$ and GREEDY(L_{n-2}) are the same until this point, which is where GEN$(n, 1, 0)$ finishes execution. By Lemma 1 we have that $|\mathcal{G}_n| = t_n$. Therefore, GREEDY(P_n) has also produced this many trees, and each tree is unique. Thus, it must be that all t_n trees of F_n have been generated. Thus, GREEDY(P_n) also halts.

Since $\mathcal{G}_n$ and GREEDY(P_n) start with the same tree, produce the same trees in the same order, and halt at the same place, it follows that $\mathcal{G}_n =$ GREEDY(P_n). It is relatively straightforward to show that $\mathcal{R}_n =$ GREEDY(L_n) by using similar arguments as above. This proof is omitted due to space constraints. □

Since $\mathcal{G}_n$ is the reversal of $\mathcal{R}_n$, we immediately obtain the following corollary.

Corollary 1. *For $n \geq 2$,* GREEDY(P_n) *is equivalent to* GREEDY(L_n) *in reverse order.*

Because GREEDY(P_n) generates unique spanning trees of F_n, Lemma 1 together with Lemma 2 implies our first main result. This result answers **Research Question #3** and the first part of **Research Question #1** for fan graphs.

Theorem 1. *For $n \geq 2$, $\mathcal{G}_n =$* GREEDY(P_n) *is a pivot Gray code listing of $\mathbf{T}_n$.*

To efficiently store the global tree T, the algorithms GEN and REVGEN can employ an adjacency list model where each edge uv is associated only with the smallest labeled vertex u or v. This means v_∞ will never have any edges

associated with it, and every other vertex will have at most 3 edges in its list. Thus the tree T requires at most $O(n)$ space to store, and edge additions and deletions can be done in constant time. Our next result answers the second part of **Research Question #1** for fan graphs.

Theorem 2. *For $n \geq 2$, $\mathcal{G}_n$ and $\mathcal{R}_n$ can be generated in $O(1)$-amortized time using $O(n)$ space.*

Proof. For each call to $\textsc{Gen}(n, s_1, varEdge)$ where $n > 3$, there are at most four recursive function calls, and at least two new spanning trees generated. Thus, the total number of recursive calls made is at most twice the number of spanning trees generated. Each edge addition and deletion can be done in constant time as noted earlier. Thus each recursive call requires a constant amount of work, and hence the overall algorithm will run in $O(1)$-amortized time. There is a constant amount of memory used at each recursive call and the recursive stack goes at most $n-3$ levels deep; this requires $O(n)$ space. As mentioned earlier, the global variable T stored as adjacency lists also requires $O(n)$ space. □

3.1 Ranking and Unranking

We now provide ranking and unranking algorithms for the listing $\mathcal{G}_n$ of all spanning trees for the fan graph F_n.

Given a tree T in $\mathcal{G}_n$, we calculate its rank by recursively determining which stage (recursive call) T is generated. We can determine the stage by focusing on the presence/absence of the edges v_nv_{n-1}, v_nv_∞, $v_{n-2}v_\infty$, and $v_{n-2}v_{n-1}$. Based on the discussion of the recursive algorithm, there are t_{n-1} trees generated in S1, t_{n-1} trees generated in S2, t_{n-2} trees generated in S3, and $t_{n-2} - t_{n-3}$ trees generated in S4. S3 is partitioned into two cases based on whether $v_{n-2}v_{n-1}$ ($varEdge$) is present. For the remainder of this section we will let $T_{n-1} = T - v_n$ and $T_{n-2} = T - v_n - v_{n-1}$.

For $n > 1$, let $R_n(T)$ denote the rank of T in the listing $\mathcal{G}_n$. If $n = 2, 3, 4$, then $R_n(T)$ can easily be derived from Fig. 2. Based on the above discussion, for $n \geq 5$:

$$R_n(T) = \begin{cases} 2t_{n-1} + 2t_{n-2} - R_{n-2}(T_{n-2}) + 1 & \text{if } e_1, e_2, e_3 \in T \\ 2t_{n-1} + R_{n-2}(T_{n-2} + e_3) & \text{if } e_1, e_2, e_4 \in T,\ e_3 \notin T \\ 2t_{n-1} + R_{n-2}(T_{n-2}) & \text{if } e_1, e_2 \in T,\ e_3, e_4 \notin T \\ 2t_{n-1} - R_{n-1}(T_{n-1}) + 1 & \text{if } e_2 \in T,\ e_1 \notin T \\ R_{n-1}(T_{n-1}) & \text{if } e_1 \in T,\ e_2 \notin T \end{cases}$$

where $e_1 = v_nv_{n-1}$, $e_2 = v_nv_\infty$, $e_3 = v_{n-2}v_\infty$, and $e_4 = v_{n-2}v_{n-1}$.

Determining the tree T at rank r in the listing $\mathcal{G}_n$ follows similar ideas by constructing T starting from a set of n isolated vertices one edge at a time. Let $U_n(T, r, e)$ return the tree T at rank r for the listing $\mathcal{G}_n$. Initially, T is the set of n isolated vertices, r is the specified rank, and $e = v_nv_\infty$. If $n = 2, 3, 4$, then T is easily derived from Fig. 2. For these cases, if the edge v_nv_∞ is present, then

it is replaced by the edge e that is passed in.

$$U_n(T,r,e) = \begin{cases} U_{n-1}(T{+}e_1, r, v_{n-1}v_\infty) & \text{if } 0 < r \leq t_{n-1}, \\ U_{n-1}(T{+}e, 2t_{n-1}{-}r{+}1, v_{n-1}v_\infty) & \text{if } t_{n-1} < r \leq 2t_{n-1}, \\ U_{n-2}(T{+}e_1{+}e, r{-}2t_{n-1}, e_4) & \text{if } 2t_{n-1} < r \leq 2t_{n-1}{+}t_{n-2}, \\ U_{n-2}(T{+}e_1{+}e, 2t_{n-1}{+}2t_{n-2}{-}r{+}1, e_3) & \text{otherwise.} \end{cases}$$

where $e_1 = v_n v_{n-1}$, $e_3 = v_{n-2}v_\infty$, and $e_4 = v_{n-2}v_{n-1}$.

Since the recursive formulae to perform the ranking and unranking operations each perform a constant number of operations and the recursion goes $O(n)$ levels deep, we arrive at the following result provided the first $2(n{-}2)$ Fibonacci numbers are precomputed. We note that the calculations are on numbers up to size t_{n-1}.

Theorem 3. *The listing $\mathcal{G}_n$ can be ranked and unranked in $O(n)$ time using $O(n)$ space under the unit cost RAM model.*

This answers **Research Question #2** for fan graphs.

4 Conclusion

We answer each of the three Research Questions outlined in Section 1 for the fan graph, F_n. First, we discovered a greedy algorithm that exhaustively listed all spanning trees of F_n experimentally for small n with an easy to define starting tree. We then studied this listings which led to a recursive construction producing the same listing that runs in $O(1)$-amortized time using $O(n)$ space. We also proved that the greedy algorithm does in fact exhaustively list all spanning trees of F_n for all $n \geq 2$, by demonstrating the listing is equivalent to the aforementioned recursive algorithm. It is the first greedy algorithm known to exhaustively list all spanning trees for a non-trivial class of graphs. Finally, we provided an $O(n)$ time ranking and unranking algorithms for our listings, assuming the unit cost RAM model. It remains an interesting open problem to answer the research questions for other classes of graphs including the wheel, n-cube, and complete graph.

References

1. Berger, I.: The enumeration of trees without duplication. IEEE Transactions on Circuit Theory **14**(4), 417–418 (1967). https://doi.org/10.1109/TCT.1967.1082758
2. Bogdanowicz, Z.R.: Formulas for the number of spanning trees in a fan. Applied Mathematical Sciences **2**(16), 781–786 (2008)
3. Chakraborty, M., Chowdhury, S., Chakraborty, J., Mehera, R., Pal, R.K.: Algorithms for generating all possible spanning trees of a simple undirected connected graph: an extensive review. Complex & Intelligent Systems **5**(3), 265–281 (2019)
4. Char, J.: Generation of trees, two-trees, and storage of master forests. IEEE Transactions on Circuit Theory **15**(3), 228–238 (1968)

5. Colbourn, C.J., Day, R.P., Nel, L.D.: Unranking and ranking spanning trees of a graph. Journal of Algorithms **10**(2), 271–286 (1989)
6. Cummins, R.: Hamilton circuits in tree graphs. IEEE Transactions on Circuit Theory **13**(1), 82–90 (1966). https://doi.org/10.1109/TCT.1966.1082546
7. Feussner, W.: Ueber stromverzweigung in netzförmigen leitern. Annalen der Physik **314**(13), 1304–1329 (1902)
8. Gabow, H.N., Myers, E.W.: Finding all spanning trees of directed and undirected graphs. SIAM Journal on Computing **7**(3), 280–287 (1978)
9. Hakimi, S.: On trees of a graph and their generation. Journal of the Franklin Institute **272**(5), 347–359 (1961)
10. Hartung, E., Hoang, H.P., Mütze, T., Williams, A.: Combinatorial generation via permutation languages. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1214–1225. SIAM (2020)
11. Hoang, H.P., Mütze, T.: Combinatorial generation via permutation languages. II. lattice congruences. arXiv preprint arXiv:1911.12078 (2019)
12. Holzmann, C.A., Harary, F.: On the tree graph of a matroid. SIAM Journal on Applied Mathematics **22**(2), 187–193 (1972). https://doi.org/10.1137/0122021
13. Kamae, T.: The existence of a Hamilton circuit in a tree graph. IEEE Transactions on Circuit Theory **14**(3), 279–283 (1967)
14. Kapoor, S., Ramesh, H.: Algorithms for enumerating all spanning trees of undirected and weighted graphs. SIAM Journal on Computing **24**(2), 247–265 (1995)
15. Kishi, G., Kajitani, Y.: On Hamilton circuits in tree graphs. IEEE Transactions on Circuit Theory **15**(1), 42–50 (1968). https://doi.org/10.1109/TCT.1968.1082762
16. Matsui, T.: A flexible algorithm for generating all the spanning trees in undirected graphs. Algorithmica **18**, 530–543 (1997)
17. Mayeda, W., Seshu, S.: Generation of trees without duplications. IEEE Transactions on Circuit Theory **12**(2), 181–185 (1965)
18. Merino, A., Mütze, T.: Efficient generation of rectangulations via permutation languages. In: 37th International Symposium on Computational Geometry (SoCG 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2021)
19. Minty, G.: A simple algorithm for listing all the trees of a graph. IEEE Transactions on Circuit Theory **12**(1), 120–120 (1965)
20. Shioura, A., Tamura, A.: Efficiently scanning all spanning trees of an undirected graph. Journal of the Operations Research Society of Japan **38**(3), 331–344 (1995)
21. Shioura, A., Tamura, A., Uno, T.: An optimal algorithm for scanning all spanning trees of undirected graphs. SIAM Journal on Computing **26**(3), 678–692 (1997)
22. Smith, M.J.: Generating spanning trees. Master's thesis, University of Victoria (1997)
23. Williams, A.: The greedy Gray code algorithm. In: Workshop on Algorithms and Data Structures. pp. 525–536. Springer (2013)
24. Winter, P.: An algorithm for the enumeration of spanning trees. BIT Numerical Mathematics **26**, 44–62 (1985)