FAST ALGORITHMS FOR FINDING PATTERN AVOIDERS AND COUNTING PATTERN OCCURRENCES IN PERMUTATIONS

WILLIAM KUSZMAUL
Stanford University
kuszmaul@stanford.edu

Abstract. Given $\Pi \subseteq S_k$ and $w \in S_n$, define $P_\Pi(w)$ to be the number of k-letter subsequences of w that are order-isomorphic to some $\pi \in \Pi$. We introduce an $O(n!k)$-time algorithm computing $P_\Pi(w)$ for each $w \in S_n$, and an $O(|S_{\leq n-1}(\Pi)| \cdot nk)$-time algorithm computing $S_{\leq n}(\Pi) = \{w : w \in S_{\leq n}, P_\Pi(w) = 0\}$. Surprisingly, when $|\Pi| = 1$, we can improve the run-time to $\Theta(n!)$ for computing $P_\Pi(w)$ for each $w \in S_n$. In contrast, the best previous algorithms, based on generate-and-check, take exponential time per permutation analyzed.

If we want to only tally permutations according to Π-patterns, rather than to store information about every permutation, then all of our algorithms can be implemented in $O(n^{k+1}k)$ space.

Using our algorithms, we generated $|S_5(\Pi)|, \ldots, |S_{16}(\Pi)|$ for each $\Pi \subseteq S_4$ with $|\Pi| > 4$, and analyzed OEIS matches. We obtained thousands of novel pattern-avoidance conjectures, fourteen of which we present.

Our algorithms extend to considering permutations in any set closed under normalization of subsequences. Our algorithms also partially adapt to considering vincular patterns.

1. Introduction

Over the past thirty years, the study of permutation patterns has become one of the most active topics in enumerative combinatorics. Given a pattern $\pi \in S_k$ and a permutation $w \in S_n$, a *π-hit* in w is a k-letter subsequences of w order isomorphic to π. For example, 857 is a 312-hit in 18365472 (Figure 1). If w contains no π-hits, we say that w *avoids* π and is in $S_n(\pi)$. Moreover, for a set of patterns Π, $S_n(\Pi) = \cap_{\pi \in \Pi} S_n(\pi)$.

Permutation patterns were first introduced in 1968, when Donald Knuth characterized the stack-sortable n-permutations as exactly those avoiding 123, of which there are the Catalan number C_n [21]. In 1985, Simion and Schmidt began a systematic study of the combinatorial structures of $S_n(\Pi)$ for $\Pi \subseteq S_3$ [25]. Since then, permutation patterns have found applications throughout combinatorics, as well as in computer science, computational biology, and statistical mechanics [20]. In addition to the combinatorial structures of Π-hits being of interest for individual Π, researchers have worked to build a more general theory. The most famous result is the Stanley-Wilf Conjecture, posed in the 1980s independently by Richard Stanley and Herbert Wilf, and proven in 2004 by Marcus and Tardos, which prohibits $S_n(\Pi)$ growing at a more than exponential rate [23]. Other work has focused on characterizing when two sets Π_1 and Π_2 are *Wilf-equivalent*, meaning that $|S_n(\Pi_1)| = |S_n(\Pi_2)|$ for all n [5, 20].

Unfortunately, running large-scale experiments involving permutation patterns is generally regarded as quite difficult [3]. In particular, detecting whether a pattern π appears in a permutation w is NP-complete [6]. In this paper, however, we will circumvent this problem by detecting not whether π appears in a single permutation w, but instead finding the π-hits in large collections of permutations, allowing us to obtain algorithms which run in polynomial (and sometimes even constant) time per permutation. In contrast, the best previously known algorithms, based on generate-and-check, run in exponential time per permutation.

Significant research has already been conducted towards finding a fast algorithm for determining whether $w \in S_n(\pi)$, which we will refer to as the *PPM* problem.

Permutation Pattern Matching Problem (PPM): Given $w \in S_n$ and $\pi \in S_k$, determine whether $w \in S_n(\pi)$.

In 1998, Bose, Buss, and Lubiw showed that PPM is NP-complete in general [6]. Since then, research on PPM algorithms has traveled down two paths, the first to find an exponential algorithm with a small exponent, and the second to find fast PPM algorithms for special cases of π. Notable progress in the

first direction includes an $O(1.79^n \cdot nk)$ algorithm due to Bruner and Lackner [7], and a $2^{O(k^2 \log k)} \cdot n$ algorithm due to Guillemot and Marx [12]. Notable progress in the second direction includes polynomial-time algorithms when π is separable [3, 6, 13, 15, 28]; an easily parallelized linear-time algorithm when $|\pi| = 4$ [14]; and an algorithm whose run-time depends on a natural complexity-measure of π, running fast for π with small complexity-measure [1]. Additionally, progress has been made for more general types of patterns such as vincular patterns [8].

In practice, however, most permutation-pattern computations involve not just one permutation, but many. Indeed, the two most common computations are to build all of $S_{\leq n}(\pi)$, or to count π-occurrences in each $w \in S_n$.

Permutation Pattern Avoiders Problem (PPA): Given a permutation $\pi \in S_k$ and $n \in \mathbb{N}$, construct all permutations of size at most n that avoid the pattern π.

Permutation Pattern Counting Problem (PPC): Given a permutation $\pi \in S_k$ and $n \in \mathbb{N}$, find the number of π-occurrences in each permutation of size at most n.

One common approach to PPA and PPC, which we will refer to as *generate-and-check*, is to iterate through candidate permutations and apply PPM to each candidate [2, 3, 27]. However, recent algorithms introduced by Inoue, Takahisa, and Minato take a different approach, representing sets of permutations in highly compressed data structures called ΠDD's, and then using ΠDD-set-operations to solve PPA and PPC [16, 17]. Although the asymptotic nature of their algorithms is unknown due to the enigmatic compression performance of ΠDD's, their algorithms experimentally run much faster than the generate-and-check approach.

In this paper, we introduce the first provably fast algorithms for PPA and PPC. Surprisingly, PPC can be solved in $\Theta(n!)$ time (Theorem 3.14), spending only amortized constant time per permutation despite π appearing $n!\binom{n}{k}/k!$ times as a pattern in S_n. Similarly, PPA can be solved in $O(|S_{\leq n-1}(\pi)| \cdot nk)$ time, spending polynomial time per output permutation. Our algorithms are the first proven to spend sub-exponential time per output permutation.

In Section 4, we show how to implement both algorithms in $O(n^{k+1}k)$-space, making them practical even for very large computations on small machines.

Both algorithms extend to considering a set of patterns Π (of possibly varying lengths), rather than just a single pattern π. Interestingly, their run-times depend only on $k = \max_{\pi \in \Pi} |\pi|$, building $S_n(\Pi)$ in time $O(|S_{\leq n-1}(\Pi)| \cdot nk)$ (Theorem 3.3) and counting Π-patterns in each w in S_n in time $O(n! \cdot k)$ (Theorem 3.13). Additionally, our algorithms easily adapt to finding avoiders and counting Π-occurrences in arbitrary down-sets of permutations – for example, efficiently finding the separable permutations which are Π-avoiders. We also partially extend our results to when π is a vincular pattern.

Using a high-performance parallelized implementation of our PPA algorithm, we run a large-scale experiment, computing $|S_5(\Pi)|, \ldots, |S_{16}(\Pi)|$ for every $\Pi \subseteq S_4$ of size at least four. Matching these with OEIS entries, and filtering out the less interesting entries, we are able to obtain thousands of novel pattern-avoidance conjectures. We present fourteen of the most interesting conjectures, and have released the rest at `github.com/williamkuszmaul/patternavoidance`.

In Section 2, we introduce (mostly standard) conventions. In Section 3, we introduce and analyze our algorithms and useful extensions of them. In Section 5, we use our algorithms to run large-scale computations, automatically generating thousands of conjectures. In Section 6, we compare our algorithms (running in serial) experimentally to the best alternatives. Finally, Section 7 concludes with directions of future work.

2. Preliminaries

In this section, we set conventions for the paper. We begin by discussing pattern-avoidance.

Definition 2.1. *A* permutation in S_n *is a word containing each letter from* 1 *to* n *exactly once.*

Example 2.2. The word 25341 is in S_5.

Definition 2.3. *Given a word* w *of* n *distinct letters, the* normalization $N(w)$ *is the permutation* $u \in S_n$ *such that* $w_i < w_j$ *exactly when* $u_i < u_j$.

Example 2.4. The normalization of 5397 is $N(5396) = 2143$.

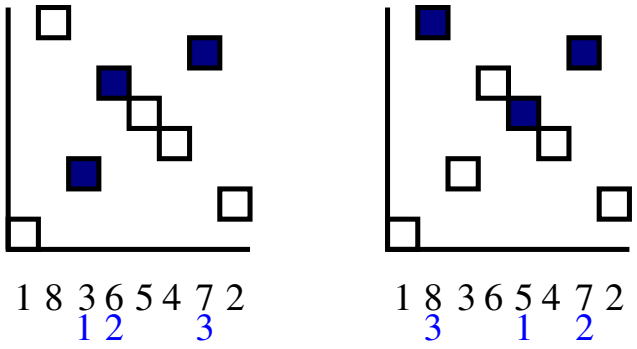

FIGURE 1. Example 123-hit and 312-hit in 18365472. In this figure, a permutation is represented graphically. A square is placed at position i, j when the ith element of the permutation is j. In the left figure the subword 367 is shown in normalized form in blue, whereas in the right figure the subword 857 is shown normalized in blue.

Definition 2.5. *Two words w_1 and w_2 are order-isomorphic if $N(w_1) = N(w_2)$.*

Definition 2.6. *Let $\pi \in S_k$ and $w \in S_n$. A* π-hit *in any subword of w order-isomorphic to π. On the other hand, w* avoids the pattern *π if w has no π-hits*

Example 2.7. An example 123-hit in 18365472 is the subword 367, while an example 312-hit is the subword 857. These hits are shown graphically in Figure 1. Observe however, that there is no 3124-hit in 18365472. Thus 18365472 avoids the pattern 3124.

Similarly, if Π is a set of permutations, then the Π-hits are just the π-hits for each $\pi \in \Pi$. And a permutation $w \in S_n$ avoids Π if it has no Π-hits. In this context, Π may be referred to as a *set of patterns*, and we say that w *avoids the patterns* in Π.

It will often be useful to refer to the word formed by the largest k-letters of a permutation as the *k-prefix* of the permutation. For example, the 3-prefix of 15234 is 534.

Next, we introduce common short-hands for sets which we will study.

Definition 2.8. *We use $S_{\leq n}$ to denote $S_1 \cup S_2 \cup \cdots \cup S_n$.*

Definition 2.9. *Let π (resp. Π) be a pattern (resp. set of patterns), and S be a set. Then $S(\pi)$ (resp. $S(\Pi)$) is the subset of S which avoids π (resp. Π).*

Example 2.10. Since S_n is the permutations of size n, the set $S_n(123)$ is the set of permutations of size n with no increasing subsequence of length three.

Our algorithms will build data about permutations up from data about smaller permutations. Consequently, they are designed to work on *down-closed* sets of permutations.

Definition 2.11. *A set of permutations S is* down-closed *if for all $w \in S$, for all non-empty subwords w' of w, $N(w') \in S$.*

Examples of down-closed sets include $S_{\leq n}$, permutations with j or fewer inversions (for a constant j), permutations with j or smaller major index, and separable permutations. Additionally, down-closed sets are closed under union and intersection.

Next, we introduce notation for obtaining from a permutation w a new permutation that is either one smaller or one larger in size.

Definition 2.12. *Given $w \in S_n$ and $i \in \{1, \ldots, n+1\}$, we define $w \uparrow^i$ to be the permutation obtained by inserting $n+1$ to be in the i-th position of w.*

Definition 2.13. *Given $w \in S_n$ and $i \in \{1, \ldots, n\}$, we define $w \downarrow_i$ to be the normalization of the word obtained by removing the letter $(n-i+1)$ from w.*

Example 2.14. For example, $13524 \uparrow^2 = 163524$, while $13524 \downarrow_2 = N(1352) = 1342$.

We use the RAM model of computation, assuming integer operations are constant-time.

Because S_n and $S_n(\pi)$ grow quickly, foreseeable applications of our algorithms are unlikely to use permutations that cannot be easily stored in a few machine words. Consequently, we assume that words can be stored as integers, with the i-th j-bit block representing the i-th letter for some fixed j (which we call the *block-size*; words may not contain a letter larger than 2^j). If we dump this assumption, all of our algorithms run a factor of n slower; the alternative would be to employ an array of length n to represent a permutation in S_n.

Note that the following operations are constant time for integers representing a word w stored as a permutation with block-size j: $w(i)$, which returns the i-th letter of w; $\mathbf{setpos}(w, i, j)$, which sets the i-th letter of w to value j; $\mathbf{insertpos}(w, u, v)$, which slides the final $n - u + 1$ letters of w one position to the right, and inserts the value v in the u-th position; and $\mathbf{killpos}(w, i)$, which slides the final $n - i$ letters of w one to position the left, erasing the i-th position. These are each easily implemented using standard integer operations, including bit-shifting which allows for multiplication and division by powers of two in constant time. For example, if w is an integer representing a word,

$$\mathbf{killpos}(w, i) = w \bmod 2^{j(i-1)} + \lfloor w/2^{ij} \rfloor 2^{j(i-1)},$$

which can be implemented in C as

$$w\&((1 << (j * i - j)) - 1) + (w >> (i * j)) << (j * i - j).$$

Using these, if w represents a permutation in S_n, we can compute $w \uparrow^i = \mathbf{insertpos}(w, i, n + 1)$ in constant time. We can compute $w \downarrow_{i+1}$ from $w \downarrow_i$ and w^{-1} (i.e., the integer representation of the inverse permutation) in constant time by inserting $n - i$ into position $w^{-1}(n - i + 1)$ of $w \downarrow_i$, and then killing the $w^{-1}(n - i)$-th letter of the result. Finally, we can also compute $(w \uparrow^{i+1})^{-1}$ from $(w \uparrow^i)^{-1}$ and w in constant time, by incrementing the $w(i)$-th position of $w \uparrow^i$ and decrementing the $(n + 1)$-th position. Consequently, for all of the algorithms in Section 3, all computations of $w \uparrow^i$ and $w \downarrow_i$ can be performed in constant time, as long as one also computes and stores $(w \uparrow^i)^{-1}$ when computing $w \uparrow^i$.

3. The Algorithms

This section introduces algorithms for efficiently constructing $S(\Pi)$ and counting Π-hits in each $s \in S$, given a down-closed set S.

Section 3.1 introduces an $O(|(S_{\leq n-1}(\Pi)| \cdot nk)$ algorithm for building $S_{\leq n}(\Pi)$, and extends this algorithm to building $S(\Pi)$ for a given finite down-closed set S.

Remark 3.1. Note that for single patterns π, we have $|S_n(\pi)| \leq |S_{n+1}(\pi)|$ for all n. In particular, depending on whether $\pi_1 = |\pi|$, one of the maps $w \to w \uparrow^1$ or $w \to w \uparrow^{n+1}$ is an injection from $S_n(\pi)$ to $S_{n+1}(\pi)$. Thus for a single pattern, our algorithm is efficient even if we only want to compute $S_n(\pi)$, with run-time $O(|S_n(\pi)| \cdot n^2 k)$.

However, $|S_n(\Pi)| \leq |S_{n+1}(\Pi)|$ need not be true when $|\Pi| > 1$. For example, if Π contains the increasing pattern of length a and the decreasing pattern of length b, then by the Erdös-Szekeres Theorem, no permutation of length greater than $(a + 1)(b + 1) + 1$ is Π-avoiding [11].

In Section 3.2, we introduce an $O(|S| \cdot nk)$ algorithm for counting the number of Π-hits in each element of a down-closed set of permutations S (Section 3.2). In particular, when $S = S_{\leq n}$, this becomes an $O(n!k)$ algorithm. More interestingly, if $|\Pi| = 1$, this can be transformed into an $O(n!)$ algorithm, spending only average constant time per permutation.

Finally, in Section 3.3, we partially extend our algorithms to consider vincular patterns, a generalized form of patterns which come with position-adjacency constraints. We pose several future directions of work for improving upon our results in this case.

3.1. Generating $S_{\leq n}(\Pi)$.

In this section, we present a dynamic algorithm for building the set $S_{\leq n}(\Pi)$ in time $O(|S_{\leq n-1}| \cdot nk)$.

The simplest algorithm for building $S_n(\Pi)$ is to brute-force check whether each permutation in $S_{\leq n}$ is Π-avoiding. If we do this by checking every $|\pi|$-subsequence of w for each $\pi \in \Pi$, this takes time

$$O\left(\sum_{\pi \in \Pi} n! \binom{n}{|\pi|} |\pi|\right).$$

This formula becomes simpler if Π comprises l permutations of size k. In this case, the algorithm runs in $O(n! \cdot \binom{n}{k} kl)$ time.

Our first task is to shrink the $n!$ term. Observe that $S_{\leq n}$ is a down-closed set. Consequently, every element in $S_n(\pi)$ is of the form $w \uparrow^i$ for some $i \in \{1, \dots n\}$ and $w \in S_{n-1}(\pi)$. Thus we can build each $S_i(\Pi)$ by checking pattern-avoidance for each permutation in $\{w \uparrow^i | i \in \{1, \dots, n\}, w \in S_{i-1}(\Pi)\}$. This yields an algorithm which generates $S_{\leq n}(\Pi)$ in $O\left(|S_{\leq n-1}(\Pi)| \cdot \binom{n}{k} \cdot nkl\right)$ time.

Our next task is to shrink the $\binom{n}{k}$ term and get rid of the l term. To do this, we observe that if $S_{i-1}(\Pi)$ is already computed, then checking whether $w \in S_n(\Pi)$ for some $w \in S_n$ can be achieved in $O(k)$ time, rather than in $O\left(\binom{n}{k} \cdot kl\right)$ time[1].

Proposition 3.2. *Let $w \in S_n$, Π be a set of patterns, and $k = \max_{\pi \in \Pi} |\pi|$. Then $w \in S_n(\Pi)$ exactly if the following conditions hold.*

(1) *$w \notin \Pi$;*
(2) *For each $i \in \{1, \dots, \min(k+1, n)\}$, $w \downarrow_i \in S_{n-1}(\Pi)$.*

Proof. Suppose $w \in S_n(\Pi)$. Then Condition (1) holds trivially, and Condition (2) holds because $S_{\leq n}(\Pi)$ is a down-closed set.

On the other hand, suppose Conditions (1) and (2) hold. Observe that if $w \downarrow_i \in S_{n-1}(\Pi)$, then any Π-hit in w must use the letter $(n - i + 1)$. Thus Condition (2) implies that any Π-hit in w must use the entire $\min(k+1, n)$-prefix of w. If $k < n$, this is impossible, since the longest pattern in Π is only length k. If $k \geq n$, then w can only contain a Π-hit if that Π-hit comprises all of w, a contradiction by Condition (1). □

Applying Proposition 3.2, we obtain our final algorithm for generating $S_{\leq n}(\pi)$. Note that the number of patterns in Π does not affect the time needed to detect whether a permutation is Π-avoiding.

Theorem 3.3. *Let k be the length of the longest pattern in Π. The set $S_{\leq n}(\Pi)$ can be constructed in $O(|(S_{\leq n-1}(\Pi)| \cdot nk)$ time.*

Proof. By Proposition 3.2, this is accomplished through Algorithm 2. □

The material below has not yet been integrated with the rest of the paper. Idea: By adding this as a second algorithm, we don't have to change our discussion of conjectures about the algorithm above. Except that we should note it doesn't apply to this one. It looks like the discussion in the computational section is not changed by this added material either. For the most part, we would only have to go through the paper and change the asymptotic claims everywhere. Lemma 4.2 would benefit from a very short discussion of how to do the partitioning efficiently.

It turns out that with a bit of work, we can optimize the algorithm to shave a factor of n off its runtime. To do this, we introduce the notion of an *extension map*.

Definition 3.4. *Let $w \in S_n(\Pi)$. Let I be the set of $i \in [n+1]$ such that $w \uparrow^i \in S_{n+1}(\Pi)$. Then the* extension map $\Psi^{\Pi}(w)$ *of w is the $(n+1)$-letter bit map with i-th letter equal to one exactly when $i \in I$.*

Example 3.5. Consider $12 \in S_2(123)$. Observe that $\Psi^{123}(12)$ is 110 because inserting 3 in either of the first two positions of 12 results in another 123-avoider.

Definition 3.6. *Let $j \in [n]$ and $w \in S_n(\Pi)$. Let I be the set of $i \in [n+1]$ such that $w \uparrow^i \downarrow_{j+1} \in S_n(\Pi)$. Then the $(n-j+1)$-*ignoring extension map $\Psi^{\Pi}_{n-j+1}(w)$ *of w is the $(n+1)$-letter bit map with i-th letter equal to one exactly when $i \in I$.*

Example 3.7. Consider $53412 \in S_n(123)$. Then the 4-ignoring extension map of 53412 tells us for which i we can insert 6 in position i to get a permuation whose only 123-patterns involve the letter 4. Consequently, $\Psi^{123}_4(53412) = 111110$.

The next theorem shows how to count Π-avoiders in only $O(k)$ time per avoider. In addition to the integer operations traditionally used in the RAM model, the algorithm uses two operations which most modern machines implement in a single instruction. The first is **popcount**, which returns the number of 1s in an integer's binary representation. The second is **ctz**, which returns the number of trailing 0-bits of an integer, starting at the least-significant bit position.

[1]This assumes we also have w^{-1}, so that each $w \downarrow_i$ for i from 1 to $k+1$ may be computed in $O(k)$ time.

Theorem 3.8. *Let k be the length of the longest pattern in Π. The values $|S_1(\Pi)|, \ldots, |S_n(\Pi)|$ can be computed in time $O(|(S_{\leq n-1}(\Pi)| \cdot k)$.*

Proof. Because n is not much bigger than the number of bits in a machine word, we can store extension maps as unsigned integers, allowing us to perform integer-operations on them in constant time.

Consider a Π-avoiding permutation $w \in S_m(\Pi)$ for some $m \geq k$. (We will handle smaller w later.) By Proposition 3.2,

$$\Psi^\Pi(w) = \wedge_{j \in [n-k,n]} \Psi_j^\Pi(w),$$

where $\wedge$ denotes the *and* operator.

Moreover, given w^{-1} and $\Psi^\Pi(w \downarrow_{m-j+1})$, we can compute $\Psi_j^\Pi(w)$ in constant time. In particular, for $i \in [1, w^{-1}(j)]$, the i-th bit of $\Psi_j^\Pi(w)$ is the same as that of $\Psi^\Pi(w \downarrow_{m-j+1})$; and for $i \in [w^{-1}(j)+1, n+1]$ the i-th bit of $\Psi_j^\Pi(w)$ equals the $(i-1)$-th bit of $\Psi^\Pi(w \downarrow_{m-j+1})$. Thus $\Psi_j^\Pi(w)$ can be obtained from $\Psi^\Pi(w \downarrow_{m-j+1})$ by shifting bits in positions $w^{-1}(j)+1, \ldots, n+1$ to the right by one, and inserting a copy of the $w^{-1}(j)$-th bit in the $(w^{-1}(j)+1)$-th position.

So far we have shown how to build $\{\Psi^\Pi(w) : w \in S_m(\Pi)\}$ out of $\{(w, w^{-1}) : w \in S_{\leq m}(\Pi)\}$ and $\{\Psi^\Pi(w) : w \in S_{m-1}(\Pi)$ in time $O(|S_m(\Pi)|k)$. If $m = n-1$, then at this point we can use the **popcount** instruction to to count the number of on-bits appearing in extension maps of permutations in $S_m(\Pi)$. This takes $|S_{n-1}(\Pi)|$ time and gives us a value for $|S_n(\Pi)|$. If $m < n-1$, then we want to build $\{(w, w^{-1}) : w \in S_{\leq m+1}(\Pi)\}$ and then repeat the process for $m+1$. From the extension maps of avoiders in S_m, we obtain $S_{m+1}(\Pi)$ in time $|S_{m+1}(\Pi)|$ by repeatedly taking advantage of the **ctz** operation in order to extract the 1-bit positions from each map.

Constructing $\{w^{-1} : w \in S_{m+1}(\Pi)\}$ is not as easy however, and would take $O(|S_{m+1}(\Pi)|n)$ time to do in the manner done everywhere else in this paper. We are saved, however, by the fact that we only need each w^{-1} to be correct in its final $k+1$ values. Thus if we choose to only update these values, then we can obtain the inverses in time $O(|S_{m+1}(\Pi)|k)$.

At this point we have an algorithm which only starts to work once we have already built the avoiders in S_k. In particular, the statement

$$\Psi^\Pi(w) = \wedge_{j \in [n-k,n]} \Psi_j^\Pi(w)$$

may not hold if $|w| < k$; if $w \uparrow^i \in \Pi$, then the formula may falsely identify $w \uparrow^i$ as an avoider. This is easily fixed, however, by simply checking Π-membership for each detected avoider. □

Remark 3.9. Theorem 3.8 easily extends to generate $S_{\leq n}(\Pi)$ in time $O(|S_{\leq n}(\Pi)| + k|S_{\leq n-1}(\Pi)|)$.

The material above has not yet been integrated with the rest of the paper.

Algorithm 1: DetectAvoider

Input: Hash table H such that $H \cap S_{n-1} = S_{n-1}(\Pi)$, Hash table Π, $k := \max_{\pi \in \Pi} |\pi|$, Permutation $w \in S_n$
Output: Whether $w \in S_n(\Pi)$
if $w \in \Pi$ **then**
  return *false*;
for $i \in \{1, \ldots, \min(k+1, n)\}$ **do**
  if $w \downarrow_i \notin H$ **then**
    return *false*;
return *true*;

Observe that Algorithm 2 can be easily modified to generate $S_{\leq n} \cap S$ for down-closed sets S. In particular, prior to checking whether NewPerm is an avoider, we throw out NewPerm if it is not in S.

For example, if S is the set of permutations in $S_{\leq n}$ with j or fewer inversions for a fixed j, then our algorithm can be modified to generate $S(\Pi)$ in $O(|S| \cdot nk)$ time. In particular, by keeping track of the inversion statistic for permutations in UnprocessedQueue, one can detect when NewPerm has inversion statistic greater than j in constant time.

Algorithm 2: BuildAvoiders

```
Input: Hash table Π, k := max_{π∈Π} |π|, n
Output: A hash table containing S_{≤n}(Π)
UnorderedSet Avoiders;
Queue Unprocessed;
if 1 ∉ Π then
    Unprocessed.enqueue(1);
    Avoiders.add(1);
while not Unprocessed.empty() do
    Perm := Unprocessed.dequeue();
    for i ∈ {1, ..., Perm.size() + 1} do
        NewPerm := Perm↑^i;
        if DetectAvoiders(Avoiders, Π, k, NewPerm) then
            Avoiders.insert(NewPerm);
            if NewPerm.size() < n then
                Unprocessed.enqueue(NewPerm);
return Avoiders;
```

Other examples of down-closed S include the separable permutations, and the permutations with major index at most a fixed constant. Recently, the study of permutation avoidance with respect to permutation statistics such as major index and inversion number have become of particular interest [10, 24].

3.2. **Counting Pattern-Occurrences in $S_{\leq n}$.** Building on the ideas in Section 3.1, in this section we present a dynamic algorithm for counting Π-hits in each permutation of S_n in $O(n!k)$ time. Interestingly, when $|\Pi| = 1$, this can be improved to an $O(n!)$ time algorithm. Additionally, given a preconstructed down-closed set $S \subseteq S_{\leq n}$ and the inverses of each $s \in S$, our algorithm extends to run in $O(|S|k)$ time. The inverses for each $s \in S$ are required so that $s\downarrow_1, \ldots, s\downarrow_{k+1}$ may be computed in $O(k)$ time; recall, however, that they can be obtained at no additional asymptotic cost if we build S through repeated applications of the $\uparrow^i$ operation.

For this section, fix Π to be a set of patterns, $k = \max_{\pi\in\Pi} |\pi|$, and $n \in \mathbb{N}$. For permutations w, let $P(w)$ denote the number of Π-hits in w.

Definition 3.10. *Let $P_i(w)$ be the number of Π-hits in w containing the entire i-prefix of w.*

Example 3.11. Suppose $w = 1234$ and $\Pi = \{123\}$. Then $P_0(w) = 4$, $P_1(w) = 3$, $P_2(w) = 2$, $P_3(w) = 1$, and $P_4(w) = 0$.

Observe that $P_0(w) = P(w)$. Although it appears that $P(w)$ satisfies no nice recurrence relation allowing for an efficient dynamic program, $P_i(w)$ does.

Proposition 3.12. *Let $w \in S_n$. Then*

$$P_i(w) = \left\{ \begin{array}{ll} P_{i+1}(w) + P_i(w\downarrow_{i+1}) & \text{if } i < n \text{ and } i \leq k, \\ 1 & \text{if } i = n \text{ and } w \in \Pi, \text{ and} \\ 0 & \text{otherwise.} \end{array} \right\}$$

Proof. Suppose $i < n$ and $i \leq k$. Then the Π-hits in w using w's entire $(i+1)$-prefix are counted by $P_{i+1}(w)$. And the Π-hits in w using w's entire i-prefix but not w's entire $(i+1)$-prefix are counted by $P_i(w\downarrow_{i+1})$.

Suppose $i = n$. Then the i-prefix of w forms a pattern in Π if and only if $w \in \Pi$.

Finally, if $i > k$ or $i > n$ then $P_i(w) = 0$. In particular, if $i > k$, then no pattern in Π can use all of the first i letters of w, since $k = \max_{\pi\in\Pi} |\pi|$. □

Given a permutation w and its inverse w^{-1}, Proposition 3.12 yields an $O(k)$ algorithm to compute each $P_i(w)$ for a permutation in terms of each $P_i(w')$ for smaller permutations w' (Algorithm 3)[2]. Note that Algorithm 3 treats each P_i as a globally accessible hash table mapping permutations to integers. Additionally, Algorithm 3 assumes access to Π and k.

Algorithm 3: Count(w): Counting Π-hits in w.

Input: Permutation $w \in S_n$
Output: Assigns values to $P_i(w)$ for each $i \in \{0, \ldots, k+1\}$
$P_{k+1}(w) := 0$;
for $i \in \{k, \ldots, 0\}$ **do**
  $P_i(w) := 0$;
  if $i = n$ *and* $w \in \Pi$ **then**
    $P_i(w) := 1$
  if $i < n$ **then**
    $P_i(w) := P_i(w \downarrow_{i+1}) + P_{i+1}(w)$;

Theorem 3.13. *Given a down-closed set $S \subseteq S_{\leq n}$, and the inverse of each $s \in S$, one can construct $P(w)$ for each $w \in S$ in $O(|S| \cdot k)$ time.*

Proof. Given S, bucket-sort can be used to construct each of $S \cap S_i$ for $1 \leq i \leq n$ in $O(|S|)$ total time. One can then use Algorithm 3 to compute $P(w)$ for each $w \in (S \cap S_i)$ for i from 1 to n. This takes $O(|S| \cdot k)$ time. □

The algorithm in Theorem 3.13 can also be adapted for down-closed sets $S \subseteq S_{\leq n}$ for which set membership is conditional on the number of Π-hits of a permutation.

For one important example of this, suppose S is the set of permutations in $S_{\leq n}$ with j or fewer Π-hits for some fixed j. Our first task is to shrink the $n!$ term. Since S is a down-closed set, every element in $S_n \cap S$ is of the form $w \uparrow^i$ for some $i \in \{1, \ldots n\}$ and $w \in S_{n-1} \cap S$. Thus we can build $S_n \cap S$ out of $S_{n-1} \cap S$ while simultaneously using Proposition 3.12 to compute $P(w)$ for each $w \in S$. To accomplish this, we use Algorithm 4 to identify whether a permutation w is in $S_n \cap S$ based on values of $P_i(w')$ for $w' \in S_{n-1} \cap S$. At the same time, if Algorithm 4 concludes that a permutation is in $S_n \cap S$, it computes $P_i(w)$ for each i. In turn, Algorithm 5 uses Algorithm 4 to compute each $P_i(w)$ for all $w \in S$. Observe that Algorithm 5 runs in $O(|S \cap S_{\leq n-1}| \cdot nk)$ time. In particular, for each permutation w in $S \cap S_{\leq n-1}$, we run Algorithm 4 on each $w \uparrow^i$.

When $j = 0$, Algorithm 5 simply builds $S_{\leq n}(\Pi)$. In fact, in this case the algorithm can be cleaned up to become Algorithm 2.

Theorem 3.13 allows us to count Π-hits in each $w \in S_n$ in $O(n!k)$ time. Surprisingly, this can be improved even further when $|\Pi| = 1$.

Theorem 3.14. *Let $\pi \in S_k$. Then the number of π-hits in each $w \in S_n$ can be computed in $\Theta(n!)$ time, regardless of k.*

Proof. For a permutation w, let i be the smallest i such that the i-prefix of w is not order isomorphic to the i-prefix of π. Then $P_i(w) = 0$. Thus we can modify Algorithm 3 to not bother computing $P_j(w)$ for $j > i$. In particular, $P_j(w)$ for $j > i$ will never be requested later in the algorithm; any w' such that $w' \downarrow_k = w$ for some $k > i$ will also have its i-prefix not order-isomorphic to π's.

Given that the $(i-1)$-prefix of w is order-isomorphic to the $(i-1)$-prefix of π, it is easy to check whether the i-prefixes are as well in constant time.

Let T_r be the indicator function taking value 1 when the r-prefix of a permutation is order-isomorphic to π's r-prefix. Then the new algorithm spends time proportional to $O(1)+\sum_r T_r(w)$ on each permutation w. However, $\mathrm{E}(T_r(w)) = 1/i!$ over all $w \in S_{\leq n}$. Thus the algorithm runs in $O(n!)$ time. □

[2]Recall w^{-1} is needed for fast computation of $w \downarrow_i$ for $i \in \{1, \ldots, k+1\}$.

Algorithm 4: CountHitsBounded Counts Π-hits in w if w has at most j Π-hits; returns false if w has more than j Π-hits.

Input: HashTable H such that $H \cap S_{n-1} = S \cap S_{n-1}$, Permutation $w \in S_n$, j
Output: Returns whether w has $\leq j$ Π-hits. If true, assigns values to $P_i(w)$ for each $i \in \{0, \ldots, k+1\}$.

```
P_{k+1}(w) := 0;
for i ∈ {k, ..., 0} do
    P_i(w) := 0;
    if i = n and w ∈ Π then
        P_i(w) := 1;
    if i < n then
        if w ↓_{i+1} ∉ H then
            for r ∈ {k+1, ..., i+1} do
                P_r.remove(w);
            return false;
        P_i(w) := P_i(w ↓_{i+1}) + P_{i+1}(w)
if P_0(w) > j then
    for i ∈ {k+1, ..., 0} do
        P_i.remove(w);
    return false;
return true;
```

Algorithm 5: BuildPermsWithBoundedHits

Input: n, j
Output: Returns set of permutations w in $S_{\leq n}$ with $\leq j$ Π-hits; Also computes values of $P_i(w)$.

```
UnorderedSet S;
Queue Unprocessed;
if 1 ∉ Π or j ≥ 1 then
    Unprocessed.enqueue(1);
    S.add(1);
while not Unprocessed.isempty() do
    Perm := Unprocessed.dequeue();
    for i ∈ {1, ..., Perm.size + 1} do
        NewPerm := Perm↑^i;
        if CountHitsBounded(S, NewPerm, j) then
            S.insert(NewPerm);
            if NewPerm.size() < n then
                Unprocessed.enqueue(NewPerm);
return S;
```

In fact, we conjecture that the same trick reduces Algorithm 1 to an $O(|S_{n-1}(\pi)|n)$ time algorithm for any pattern π. To prove this, one would show that $\mathrm{E}(T_r(w))$ is small for w from $S_n(\pi)\uparrow := \{w\uparrow^i \mid i \in \{1, \ldots, n\}, w \in S_{n-1}(\pi)\}$.

For example, when $\pi = 123\cdots k$, this can be done as follows. Observe that swapping two letters in an avoider can only introduce a π-hit if the first letter was originally greater than the second (The Swapping Observation). Suppose $w \in S_n(\pi)\uparrow$ has an increasing i-prefix $p_1p_2\cdots p_i$. By repeated applications of The Swapping Observation, rearranging the order of the letters in the i-prefix of w to be $p_2p_3p_5\cdots p_i$ with p_1 inserted in any position but the first or final will result in another permutation in $S_n(\pi)\uparrow$.

Thus we can match each $w \in S_n(\pi) \uparrow$ having an increasing i-prefix with $i-2$ permutations, each in $S_n(\pi) \uparrow$ and each with an increasing $(i-1)$-prefix (but not an increasing i-prefix). It follows that $\mathrm{E}(T_i(w)) \leq \mathrm{E}(T_{i-1}(w))/(i-1)$ over $w \in S_n(\pi) \uparrow$, implying the conjecture for $\pi = 123\cdots k$. In fact, proving $\mathrm{E}(T_i(w)) \leq \mathrm{E}(T_{i-1}(w))/c$ for any constant $c > 1$ would be sufficient, which is why the conjecture seems very likely to be true in general.

Remark 3.15. In practice, the technique introduced in Theorem 3.14 is worth implementing even for large sets of patterns Π (for both PPA and PPC). In order for this to be efficient, however, one needs to quickly identify whether the i-prefix of a permutation w is an i-prefix of *any* permutation $\pi \in \Pi$. In our implementations, we compute the normalization of the i-prefix of w, and check its membership in a hash table containing the normalized i-prefixes of each $\pi \in \Pi$.

Surprisingly, given w and w^{-1}, one can build the successive normalized prefixes of w one after another, each in constant time. This takes advantage of the **popcount** instruction, which on most modern machines obtains the number of 1s in an integer's binary representation through a single instruction. In particular, we maintain a bitmap b (in the form of an integer) where $b[j] = 1$ if some $k \in [n-i+1, \ldots, n]$ is in position j. We can then use **popcount** to query how many letters in w's i-prefix appear to the right of $n-i+1$; this tells us in what position to insert 1 into the normalized $(i-1)$-prefix in order to obtain the normalized i-prefix. The insertion can then be performed using bit-hacks in constant time.

3.3. **Algorithms for Vincular Patterns.** In this section, we discuss a generalization of permutation patterns known as *vincular patterns* in which patterns may come with additional adjacency constraints. Vincular patterns came into the spotlight in 2000 when Babson and Steingrímsson observed that essentially all Mahonian permutation statistics can be written as a linear combination of the vincular patterns appearing in a permutation [4]. Just as for traditional pattern-avoidance, relations to natural structures such as Dyck paths and set partitions arise in the study of vincular pattern-avoidance [9].

A vincular pattern is written as a permutation with dashes, either following a letter in the permutation or preceding the first letter. A dash between two letters indicates that those letters must be adjacent in any pattern-occurrence. A dash at the beginning (resp. end) of the pattern indicates that the pattern must start (resp. end) a permutation's first (resp. final) letter.

Example 3.16. The permutation pattern $-12-3$ appears $n-2$ times in the identity permutation $e_n \in S_n$. In particular, the first letter of the pattern must be the first letter of e_n, while the second and third letters of the pattern can be any two adjacent letters following the first letter (for which there are $n-2$ choices).

Given a vincular pattern π, define $P'_i(w)$ for w a permutation to be the number of π-hits in w^{-1} using all of the final i letters of w^{-1}. In turn, the inverse of each such π-hit in w^{-1} uses all of $(n-i+1), \ldots, n$ in w. Because position-adjacency constraints for π-hits in w^{-1} translate to value-adjacency constraints for the same hit in w, we will be able to find a recurrence for $P'_i(w)$ in terms of $P'_{i+1}(w)$ and $P'_i(w \downarrow_{i+1})$.

For a vincular pattern $\pi \in S_k$, define $T(\pi)$ to be the set of i such that there is a dash preceding π_{k-i+1}; additionally $T(\pi)$ includes 0 if and only if there is a dash following π. For example, $T(-14-23) = \{2, 4\}$.

The following Proposition extends Proposition 3.12 to the case where Π comprises a single vincular pattern π.

Proposition 3.17. *Let $w \in S_n$. Let π be a vincular pattern and T be $T(\pi)$. Then*

$$P'_i(w) = \begin{cases} P'_{i+1} & \text{if } i < n,\ i \leq |\pi|, \text{ and } i \in T, \\ P'_{i+1}(w) + P'_i(w \downarrow_{i+1}) & \text{if } i < n,\ i \leq |\pi|,\ i \notin T, \\ 1 & \text{if } i = n \text{ and } w \in \Pi, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Cases (2)–(4) follow just as in the proof of Proposition 3.12. Suppose $i < n$, $i \leq k$, and $i \in T$. If $i = |\pi|$, then since $i < n$ we see that $P_i(w) = 0$, which in turn is $P_{i+1}(w)$ by Case (4). On the other hand, if $i < |\pi|$, then any occurrence of π in w^{-1} using the final i letters of w^{-1} must also use the final $i+1$ letters of w^{-1} (which translate to $(n-i), \ldots n$ in w). Thus $P'_i(w) = P'_{i+1}(w)$. □

Using this recurrence, analogues of Theorems 3.13 and 3.14 follow with only slightly modified proofs.

Theorem 3.18. *Let π be a vincular pattern. Given a down-closed set $S \subseteq S_{\leq n}$ and s^{-1} for each $s \in S$, one can count π-hits in w for each $w \in S$ in $O(|S| \cdot |\pi|)$ time.*

Proof. If one modifies Algorithm 3 to use the recurrence from Proposition 3.17 on w^{-1} rather than the recurrence from Proposition 3.12 on w, then the proof follows just as for Theorem 3.13. □

Theorem 3.19. *Let $\pi \in S_k$ be a vincular pattern. Then the number of π-hits in each $w \in S_n$ can be computed in $\Theta(n!)$ time, regardless of $|\pi|$.*

Proof. The result follows using the same technique as in the proof of Theorem 3.14. In particular, when applying the recursion from Proposition 3.17 to compute $P'_i(w^{-1})$ for some $w \in S_n$, one checks whether the i-prefix of w^{-1} is order-isomorphic to the i-prefix of π^{-1} (ignoring π's adjacency constraints when computing its inverse). If the two are not order-isomorphic, $P'_i(w^{-1})$ must be zero. □

Theorem 3.19 shows that we can count π-hits for each permutation in $S_{\leq n}$ in $Theta(n!)$ time. By considering each pattern in Π separately, this extends to an algorithm for counting Π-hits for any set Π of vincular permutations in $O(n!|\Pi|)$ time.

It is still an open problem, however, to quickly compute $S_n(\Pi)$ if Π comprises vincular patterns. The difficulty in this comes from the fact that $S_{\leq n}(\Pi)$ need not be down-closed in this case. Indeed, removing a letter from an avoider w may introduce a vincular pattern which was not previously present. For example, the permutation 1324 does not contain a $1-23$ pattern, but removing 3 yields a permutation which does.

One special case of a vincular pattern is when there are dashes between every pair of letters but not at the beginning or end of the pattern. This is what's known as a *consecutive pattern*. For consecutive patters π, Theorem 3.18 counts π-hits in a down-closed set S in $O(|S|)$ time (assuming s^{-1} is known for each $s \in S$). Interestingly, in this case, the PPM problem (detecting a π-pattern in a single permutation $w \in S_n$) already has a linear time solution due to Kubica, Kulczyński, Radoszewski, Rytter, and Waleń [22]. A similar result was found independently by Kim et. al. [19].

4. Eliminating the memory bottleneck and making parallelism easy

So far, our algorithms have required space nearly proportional to their run time. In this section we restructure our algorithms so that, without changing their runtimes, we asymptotically reduce space usage to at most $O(n^{k+1}k)$. Consequently, our algorithms are practical for even very large computations on small computers. At the same time, these changes make our algorithms easily implemented in parallel.

Of course, if one wants to actually store $S_n(\Pi)$ or $P(w)$ for each $w \in S_n$, then space efficiency is futile. However, in this section, we assume that the goal is *enumeration*, to evaluate either $|S_n(\Pi)$ or to tally how many $w \in S_n$ have each value of $P(w)$.

Define T to be the tree of permutations where a node v has children $v\uparrow^i$ for each $i \in [1, |v|+1]$. For a node v, define v's *j-th level children* $C^j(v)$ to be the set of nodes in the $(j+1)$-th level of the subtree of which v is the root.

For a given a set of permutations S, define $S\downarrow_i$ as $\{s\downarrow_i : s \in S\}$.

The following observation will play a key role in reducing improving memory utilization.

Lemma 4.1. *Let $v \in T$. Let i, j be positive integers satisfying $i \leq j$. Then*

$$C^j(v)\downarrow_i \subseteq C^{j-1}(v).$$

Proof. The elements of $C^{j-1}(v)$ are precisely the permutations in $S_{|v|+j-1}$ with v as their $|v|$-postfix (i.e., the word formed by the letters $1, \ldots, |v_1|$). Every element of $C^j(v)$ also has $|v|$-postfix v. Moreover, $|v|+j-i+1 > |v|$ since $i \leq j$, implying that every element of $C^j(v)\downarrow_i$ has $|v|$-postfix v as well, completing the proof. □

Our approach to PPC in Section 3.2 uses $O((n-1)!k)$ memory. In particular, we perform a breadth-first-search on $T \cap S_{\leq n}$, using Proposition 3.12 at each node v to compute each $P_i(v)$. However, Lemma 4.1 suggests an $O(n^{k+1}k)$-memory approach.

First compute $P_i(w)$ for each $w \in S_{\leq k}$ using the breadth-first-search approach (in $O(k!)$ space). Then traverse $T \cap S_{\leq n-k}$ depth-first. When visiting a node v, compute each $P_i(c)$ for each $c \in C^k(v)$. Observe that this recursively depends only on elements of $C^k(v\uparrow^1)$ by Lemma 4.1[3]. Having computed each P_i of

[3]Here, we consider C^k of the empty permutation to be the permutations of size k.

each $c \in C^k(v)$, we update our tally of how many permutations have each number of Π-hits, and then store each $P_i(c)$ (to be accessed while visiting v's children in T). However, when we return to v's parents during our depth first traversal, we throw out the P_i values.

At a given point in the traversal of $T \cap S_{n-k}$, we may store as many $(P_0, P_1, \ldots, P_k)$-tuples as $O(\sum_{i=0}^{n-k}(i+1)(i+2)\cdots(i+k))$, bounding our memory-usage at $O(n^{k+1}k)$; note that the space taken to tally permutations by Π-hits is bounded above by $\binom{n}{k} + \binom{n}{k-1} + \cdots + \binom{n}{1} \leq n^{k+1}$.

When $|\Pi| = 1$, we can use the technique from Theorem 3.14 to obtain $O(n^{k+1})$-space usage, since instead of storing entire $(k+1)$-tuples of P_i's, we store on average a constant number per permutation.

Because PPC concerns all $w \in S_{\leq n}$, it is practical to obtain C^k of a node v simply by traveling to each node k levels below v; this takes time only $|C^k(v)|$ because nodes have many children ($(i+1)$ children for nodes in the i-th level). However, if we wish to count Π-hits over a down-closed set S, or to extend our memory-reduction results to PPA (in which case $S = S_{\leq n}(\Pi)$), then we need a more efficient method of obtaining $C^k(v) \cap S$ from $C^k(v \downarrow_1) \cap S$.

Lemma 4.2. *Let S be a down-closed set of permutations, and suppose we are given $C^k(v) \cap S$ for some $v \in S_j$. Moreover, suppose that for each $c \in C^{k+1}(v)$, we can detect whether $c \in S$ in T time. Then for a given $v' = v \uparrow^i$ one can obtain $C^k(v') \cap S$ in $O(nT|C^{k-1}(v')|)$ time, if one allows for time $O(n|C^k(v) \cap S|)$ to preprocess $C^k(v) \cap S$. Morever, preprocessing takes space $O(|C^k(v) \cap S|)$ and building $C^k(v') \cap S$ takes space $O(|C^k(v') \cap S|)$.*

Proof. To preprocess $C^k(v) \cap S$, we partition its elements into sets $C_1, C_2, \ldots, C_{|V|+1}$ based on the position of $|v|+1$ relative to $1, 2, \ldots, |v|$. Observe that the children of the nodes in C_i are the candidates for $C^k(v \uparrow^i) \cap S$ (since S is down-closed). Thus in $O(Tn|C_i|) = O(Tn|C^{k-1}(v \uparrow^i)|)$ time, we can construct a given $C^k(v \uparrow^i)$. □

Using Lemma 4.2, our space-reduction technique extends to our algorithms for PPA and to PPC for arbitrary down-closed sets. In $O(n^{k+1})$ space and $O(|S_{\leq n-1}(\Pi)|nk)$ time we can compute $|S_j(\Pi)|$ for each $j \leq n$. More generally, given a pre-constructed down-closed set of permutations $S \subseteq S_{\leq n}$, we can compute the number of $w \in S$ with each number of Π-hits in $O(n^{k+1}k)$ space and $O(|S \cap S_{\leq n-1}|nk)$ time.

Depending on S, the space usage may be much smaller, for example, if $C^k(v) \cap S$ is never very large for any v. In particular, for PPA, the expected memory consumption at a given instance in the algorithm is $O(\sum_{j=k}^{n} |S_j(\Pi)|/|S_{j-k}(\Pi)|)$, which by the Stanley-Wilf Conjecture (proven in [23]), grows at most linearly with n (with a potentially large constant depending on Π). Thus, a large machine running pattern-avoidance computations in parallel can treat space usage as growing linearly with n.

In addition to reducing memory-usage asymptotically, the optimizations in this section make parallelizing our algorithm easy. Indeed, the depth-first traverals of $T \cap S$ can be parallelized without risking write-conflicts for hash tables containing avoiders or maps containing P_i-values. Although we test our algorithms in serial in Section 6, we have released a parallelized implementation at `github.com/williamkuszmaul/patternavoidance`, which we used for our computations in Section 5.

5. 30,000 conjectures on pattern avoidance

Past research enumerating $|S_n(\Pi)|$ has tended to focus on small $|\Pi|$. Do larger sets of patterns also yield interesting number sequences? In this section, we address this question by mass-computing $|S_5(\Pi)|, \ldots, |S_{16}(\Pi)|$ for every choice of $\Pi \subseteq S_4$ satisfying $|\Pi| > 4$, and then searching for the resulting sequences in OEIS [26]. Filtering out sequences with cubic or smaller growth, we then filter these to 32,019 OEIS matches, enumerated by 446 OEIS sequences, most of which seem to have never appeared before in the context of pattern-avoidance, and many of which have interesting combinatorial interpretations. Finally, we select the 14 OEIS matches which seem most surprising, and pose their relation to pattern-avoidance as conjectures. We believe that many of the others are also interesting, and have released all of our code and data at `github.com/williamkuszmaul/patternavoidance`.

Because making millions of requests to OEIS is not practical, we downloaded a local copy of OEIS and built a rudimentary lookup program. In particular, we define the *OEIS match* of a number sequence s as the smallest-indexed OEIS sequence which can be shifted no more than 14 positions to the left in order to obtain s.

We computed $|S_5(\Pi)|, \ldots, |S_{16}(\Pi)|$ for each $\Pi \subseteq S_4$ with $|\Pi| > 4$, starting at $n = 5$ because $|S_4(\Pi)|$ depends only on $|\Pi|$. There are 2,137,358 such distinct Π up to trivial Wilf equivalence; in particular, a given choice of Π is equivalent to $\{f(\pi)|\pi \in \Pi\}$ where f is either the inverse, complement, or reverse function. In fact, many more Π still are likely subtle equivalent, with only 64,211 distinct sequences appearing among the 2,137,358 sequences computed (i.e., there appear to be 64,211 Wilf-classes).

Next, we checked which sequences have OEIS matches. Surprisingly, 1,464,023 of the 2,137,358 Π tested have OEIS matches, distributed among 940 distinct OEIS sequences. Aiming to filter out the less interesting sequences, we went on to throw away matches for sequences growing at constant rate (826,003 matches attributed to 290 OEIS sequences), linear rate (391,047 matches attributed to 291 OEIS sequences), quadratic rate (147,684 matches attributed to 264 OEIS sequences), or cubic rate (15,249 matches attributed to 95 OEIS sequences).[4] It would be interesting to study what sorts of sets of patterns result in polynomial sequences; progress in this direction has already been made by Kaiser and Klazar [18].

We are left over with 32,019 OEIS matches, attributed to 446 distinct OEIS sequences. These OEIS sequences seem quite interesting, many with combinatorial interpretations or linear recurrences, and warrant further individual attention.

It is natural to wonder how many of our selected 32,019 OEIS matches are false alarms. In order to test this, we examined whether running the same computations for n up to thirteen, fourteen, or fifteen, instead of sixteen, introduces false OEIS matches which are not revealed until n gets to sixteen. Surprisingly, we have to go all the way back to $n = 13$ before any false matches are introduced, at which point the OEIS sequence A246878 is incorrectly paired with two pattern-sets[5].

Thus we consider the 446 distinct OEIS matches, corresponding with 32,019 $\Pi \subseteq S_4$ to be reliable. Moreover, only 24 of these sequences (corresponding with 251 pattern sets) are obviously already connected to pattern avoidance. We believe that nearly all of the remaining 422 distinct OEIS matches, corresponding with 31,868 pattern-sets, represent novel and interesting pattern-avoidance conjectures. From these, we selected fourteen of the most interesting looking to present. In particular, we are bias towards those with no simple linear recurrence or with a formula likely to be explained by an interesting combinatorial interpretation; for lack of space, we also omit similar, equally interesting sequences. We now list these OEIS sequences, along with a sample Π yielding each of them. For each sequence, we provide: (1) OEIS number and (slightly edited) OEIS brief entry; (2) Number of pattern-sets matching to sequence; (3) Example matching pattern-set.

(1) A035929 Number of Dyck n-paths starting $U^m D^m$ (an m-pyramid), followed by a pyramid-free Dyck path.
Appears 14 times.
Example match: 2143 3142 1432 1342 1324

(2) A071742 Expansion of $(1 + x^4 \cdot C) \cdot C$, where C is the g.f. for Catalan numbers.
Appears 3 times.
Example match: 2431 2143 3142 4132 1432 1342 1324 1423 1243

(3) A025715 Index of 6^n within sequence of numbers of form $5^i \cdot 6^j$.
Appears 7 times.
Example match: 2341 2314 2413 2143 4312 1432 1324 1243

(4) A081662 Partial sums of $n + F(n + 1)$.
Appears 1 time.
Example match: 2431 2341 4213 2143 3142 3124 4132 1432 1342 4123

(5) A129847 Number of set partitions of $[1, n]$ whose blocks consist only of elements that differ by two or less.

[4]In particular, we considered $|S_5(\Pi)|, \ldots, S_{13}(\Pi)$ to be degree $\leq k$ if its k-th difference was zero by $n = 10$.

[5]At first glance, matches with sequence A133641 also seem to appear at $n = 13$ but then disappear at $n = 14$. However, this is simply because the OEIS sequence does not have enough terms in its entry. We artificially added more to our local copy of OEIS.

Appears 86 times.
Example match: 3214 2431 2413 2143 2134 3142 3124 4132 1432 1342 1234

(6) A164398 Number of binary strings of length n with no substrings equal to 0001 or 1000
Appears 90 times.
Example match: 3214 2431 2341 2413 2134 3124 4132 1432 1342 4123 1243

(7) A228180 The number of single edges on the boundary of ordered trees with n edges. G.f.: $(x \cdot C + 2 * x^3 \cdot C^4)/(1-x)$ where C is the g.f. for the Catalan numbers.
Appears 11 times.
Example match: 2413 4132 1432 1342 1324

(8) A024831 n-th term is least m such that if $r, s \in \{F(h)/F(2h) : h \in [1, n]\}$ satisfy r ¡ s, then $r < k/m < s$ for some integer k.
Appears 53 times.
Example match: 3214 2431 4213 2143 2134 3124 1432 1342 4123

(9) A081662 Partial sums of n+F(n+1).
Appears 1 time.
Example match: 2431 2341 4213 2143 3142 3124 4132 1432 1342 4123

(10) A178523 The path length of the Fibonacci tree of order n.
Appears 9 times.
Example match: 2314 4213 2413 3124 4132 1432 1342 4123

(11) A129715 Number of runs in all Fibonacci binary words of length n. A Fibonacci binary word is a binary word having no 00 subword. A run is a maximal sequence of consecutive identical letters. $a(n) = a(n-1) + a(n-2) + 2F(n)$ for $n \geq 3$.
Appears 8 times.
Example match: 2431 4213 2413 2143 2134 3142 1432 1342 1324 4123 1234

(12) A204746 Number of $(n+2) \times (n+2)$ binary arrays with every 3×3 subblock having three equal elements in a row horizontally, vertically, diagonally or antidiagonally exactly three ways.
Appears 67 times.
Example match: 2431 2341 2314 4213 2413 2143 2134 3124 1432 4123

(13) A239844 Number of $n \times 2$ trinary arrays with no element equal to one plus the sum of elements to its left or one plus the sum of elements above it or two plus the sum of the elements diagonally to its northwest, modulo 4.
Appears 4 times.
Example match: 2431 2143 2134 3412 4132 1432 1324

(14) A252814 Number of $n \times 2$ nonnegative integer arrays with upper left 0 and every value within 2 of its city block distance from the upper left, and every value increasing by 0 or 1 with every step right or down.
Appears 342 times.
Example match: 2314 4213 2413 2143 3412 3142 1432

6. Implementations and Testing

In this section, we test our algorithms' performance against other algorithms[6]. Our implementations represent the i-th letter of a permutation in the i-th nibble of a 64-bit integer, allowing for permutations of size up to 16. However, in our released code (`github.com/williamkuszmaul/patternavoidance`), one can choose the settings-option of allowing for larger permutations.

In Section 6.1, we test our algorithm for counting Π-hits in S_n against the generate-and-check algorithm.

In Section 6.2, we test our algorithm for finding $|S_n(\Pi)|$ against the naive generate-and-check algorithm and PermLab's more sophisticated generate-and-check algorithm. Along the way, we re-implement PermLab's algorithm, introducing optimizations resulting from our 64-bit representation of a permutation, and increasing efficiency for large sets of patterns. The difference in performance between PermLab's and our algorithm is most clear for large sets of large patterns; this is important because for large patterns, $S_n(\Pi)$ likely often only becomes combinatorially interesting when there are sufficiently many patterns to incur natural structure.

In Section 6.3, we test both of our algorithms against algorithms introduced by Inoue, Takashisa, and Minato [16, 17]. Their algorithms use a compression technique to get extremely good performance in certain cases. We suggest directions of future work for integrating those techniques into our algorithm for generating $S_n(\Pi)$.

Our implementations and tests are available at `github.com/williamkuszmaul/patternavoidance`.

6.1. Implementations counting Π-hits in each permutation in S_n.

In this section, we compare our algorithm for counting Π-hits in each n-permutation to the generate-and-check algorithm.

In addition to implementing our own algorithm, we tried to implement an efficient generate-and-check implementation for comparison. Suppose, for simplicity, that $\Pi = \{\pi\}$ with $\pi \in S_k$. Given $w \in S_n$, $\pi \in S_k$, and $j \in \{1, \ldots, k\}$, define $\beta_j(w, \pi)$ to be the set of j-letter subsequences of w's $n-k+j$-prefix that are order-isomorphic π's j-prefix. Observe that $\beta_k(w, \pi)$ is simply the set of Π-hits in w. Our generate-and-check algorithm constructs $\beta_{j+1}(w, \pi)$ out of $\beta_j(w, \pi)$ by looping through the options for which letter to add as the smallest letter in the new $(j+1)$-letter subsequence. The algorithm runs in time

$$\Theta\left(n! \sum_{i=1}^{k} \frac{\binom{n-k+i}{i}}{(i-1)!}\right), \tag{1}$$

since we spend $\Theta(1)$ time on an i-letter subsequence s in w exactly when s does not use any of the letters $\{1, \ldots, k-i\}$ and the largest $i-1$ letters of s are order isomorphic to the $(i-1)$-prefix of π.

Our generate-and-check implementation easily extends to any $\Pi \subseteq S_k$; in particular, we use the technique from Remark 3.15 to check subsequence membership in $\beta_{j+1}(w, \Pi)$ in $O(1)$ time (using information about previous subsequences), even when $|\Pi| \notin O(1)$. The resulting algorithm runs in time

$$\Theta\left(n! \sum_{i=1}^{k} k_i \frac{\binom{n-k+i}{i}}{(i-1)!}\right), \tag{2}$$

where k_i is the number of distinct elements of S_i order isomorphic to the i-prefix of some pattern in Π.

For single patterns π, our algorithm runs in $\Theta(n!)$ time, regardless of π; in fact, its run-time is almost exactly proportional to $n!$ (Figure 2). In contrast, the generate-and-check algorithm suffers as n grows, as suggested in Equation (1) (Figure 2). Additionally, our algorithm scales better to large sets $\Pi \subseteq S_k$, running in time $O(n!k)$ time (in comparison to Equation 2). As an example, we compute the number of k-letter subsequences containing 231-pattern in each permutation S_n (Figure 3).

6.2. Implementations computing $|S_n(\Pi)|$.

In this section we compare our pattern-avoidance algorithm to the naive generate-and-check algorithm and the more sophisticated algorithm of PermLab. While our code runs significantly faster for large sets of patterns, we find that PermLab's algorithm,

[6]All of our experiments are run in serial on an Amazon C4.8xlarge machine with two Intel E5-2666 v3 chips running at 2.90GHz; we are running Fedora 22 with kernel 4.0.4-301; we compile using g++ 5.1.1.

$n\backslash k$	3	4	5	6
8	0.021	0.020	0.013	0.007
9	0.258	0.265	0.187	0.120
10	3.361	3.763	2.791	1.940
11	46.973	57.216	44.352	32.621
12	705.082	930.591	752.467	581.081

(A) Generate-and-check algorithm

$n\backslash k$	3	4	5	6
8	0.003	0.002	0.003	0.002
9	0.027	0.026	0.026	0.027
10	0.285	0.302	0.302	0.309
11	3.520	3.657	3.666	3.766
12	42.741	44.752	44.791	45.716

(B) Our algorithm

$n\backslash k$	3	4	5	6
8	0.098	0.078	0.031	0.030
9	0.263	0.249	0.087	0.040
10	1.918	3.329	1.062	0.191
11	15.638	40.400	17.453	3.671
12	105.241	532.328	249.606	58.236

(C) IIDD-based algorithm

FIGURE 2. Time in seconds to find each Π-hit in each permutation in $S_{\leq n}$ with $n \in [8, 16]$ and for Π containing a single pattern of length k from the set $\{231, 2431, 24531, 246531\}$.

$n\backslash k$	3	4	5
8	0.021	0.046	0.054
9	0.258	0.650	0.911
10	3.363	9.818	16.149
11	46.960	156.46	297.638
12	704.189	2646.83	5746.63

(A) Generate-and-check algorithm

$n\backslash k$	3	4	5
8	0.003	0.004	0.006
9	0.027	0.041	0.055
10	0.286	0.453	0.637
11	3.554	5.735	8.359
12	42.842	74.717	110.991

(B) Our algorithm

$n\backslash k$	3	4
8	0.095	0.137
9	0.269	1.612
10	1.824	20.139
11	15.336	236.632

(C) IIDD-based algorithm

FIGURE 3. Time in seconds to count for each $w \in S_{\leq n}$ the number of k-letter sequences containing a 231 pattern.

when optimized, hides its asymptotic disadvantage well for avoiding a single pattern, usually running only slightly slower than our algorithm.

We implemented Algorithm 2 for building $S_n(\Pi)$, as well as a naive generate-and-check algorithm implementation, optimizing both for performance.

The naive generate-and-check algorithm runs as follows. Let T_n be the tree of permutations in $S_{\leq n}$ such that the children of $w \in S_k$ are each option for $w \uparrow^i$. Define $C(v)$ to be the set of children of a node v and $F(v)$ to be the parent. The generate-and-check algorithm performs a depth-first search on $T_n \cap S_{\leq n}(\Pi)$, visiting a node's children only if the node itself avoids Π. In order to determine whether a permutation avoids Π, the algorithm uses the same logic as in the previous section.

The best publicly available code for computing $S_n(\Pi)$, however, is PermLab, which makes several clever changes to the naive generate-and-check algorithm in order to hide its asymptotics for small n [2]. PermLab performs a depth-first search of $T_n \cap S_{\leq n-1}(\Pi)$, computing at a given node v whether each $c \in C(v)$ avoids Π. However, since PermLab only visits nodes avoiding Π, it only needs to check the children of a node in S_j for Π-hits involving $j+1$. At the same time, when visiting a node $v \in S_j$, PermLab remembers for each $x \in C(F(v))$ whether x avoids Π. Using this information, PermLab can

$n\backslash k$	3	4	5	6
8	0.001	0.013	0.016	0.012
9	0.004	0.063	0.094	0.072
10	0.009	0.299	0.832	0.958
11	0.037	2.377	9.530	13.518
12	0.151	19.068	112.187	198.764
13	0.615	153.8	1348.32	3032.45

(A) Naive generate-and-check algorithm

$n\backslash k$	3	4	5	6
8	0.000	0.006	0.010	0.008
9	0.001	0.030	0.059	0.051
10	0.003	0.100	0.337	0.452
11	0.009	0.654	3.388	5.559
12	0.035	4.573	35.202	72.320
13	0.131	32.533	378.392	985.548

(B) V2 algorithm

$n\backslash k$	3	4	5	6
8	0.000	0.004	0.007	0.007
9	0.001	0.020	0.042	0.040
10	0.003	0.070	0.212	0.289
11	0.007	0.396	2.024	3.326
12	0.028	2.665	20.160	41.060
13	0.102	18.101	201.086	519.022

(C) V1 algorithm

$n\backslash k$	3	4	5	6
8	0.018	0.019	0.029	0.034
9	0.016	0.047	0.122	0.151
10	0.024	0.147	0.581	0.757
11	0.051	0.915	3.980	6.795
12	0.123	5.020	35.127	74.387
13	0.286	30.549	333.422	911.032

(D) PermLab

$n\backslash k$	3	4	5	6
8	0.000	0.002	0.004	0.004
9	0.001	0.014	0.029	0.029
10	0.002	0.053	0.131	0.162
11	0.005	0.256	1.247	1.988
12	0.021	1.956	12.704	23.910
13	0.082	14.158	125.383	291.946

(E) Our Algorithm

$n\backslash k$	3	4	5	6
8	0.011	0.007	0.011	0.009
9	0.016	0.013	0.028	0.009
10	0.027	0.034	0.067	0.039
11	0.046	0.099	0.239	0.151
12	0.087	0.361	0.952	0.965
13	0.149	1.640	5.310	6.423
14	0.219	6.434	24.810	34.897
15	0.561	24.339	115.127	199.916
16	1.672	91.030	567.907	1254.01

(F) ΠDD-based algorithm

FIGURE 4. Time in seconds to compute $|S_n(\Pi)|$ with $n \in [8, 16]$ and for Π containing a single pattern of length k from the set $\{231, 2431, 24531, 246531\}$.

quickly determine for each $c \in C(v)$ whether c has a Π-hit not involving the letter j. Thus PermLab needs only search through brute-force for Π-hits in c involving both $j+1$ and j.

We re-implemented Permlab's algorithm, making optimizations specific to our representation of permutations as 64-bit integers. We also eliminated some wasted work by carefully examining only permutation subsequences which could potentially form the prefix of a Π-hit; in particular, we filter out subsequences which include a letter too small to allow for the rest of the Π-hit to appear after the prefix.

To search for a Π-hit in a permutation, PermLab searches independently for each $\pi \in \Pi$ until it succeeds or concludes the permutation avoids Π. However, this scales poorly to handling large sets of patterns, and allows for performance to be affected by the order patterns appear in Π. Instead, we use the technique from Remark 3.15 to efficiently check whether a subsequence is order-isomorphic to a prefix of *any* Π-hit in amortized constant time. This leads to significant speedup when there are many shared prefixes among the permutations in Π. On the other hand, if $|\Pi| = 1$, then the overhead of using the small hash table from Remark 3.15 leads to a slight slowdown. In order to demonstrate the difference, we implement both variants, calling the small-set-optimized version (using PermLab's scheme) V1 and the large-set-optimized version V2.

In Figure 4 (the final subfigure of which is discussed later), we compare the performances of the algorithms handling single patterns[7], including V1, V2, and the original PermLab. While V1 performs slightly faster than V2 in this experiment, it should never perform more than a constant factor faster. Indeed, to confirm that a permutation is an avoider, V1 must examine every permutation subsequence

[7]We choose not to use identity patterns, since are likely to yield abnormal performance for particular algorithms.

$n\backslash k$	3	4	5	6
10	0.009	0.026	0.049	0.062
11	0.037	0.124	0.258	0.362
12	0.151	0.572	1.345	2.120
13	0.623	2.607	6.838	11.945
14	2.490	11.801	34.316	66.623
15	10.155	53.014	169.297	359.042
16	41.299	236.709	822.06	1906.53

(A) Naive generate-and-check algorithm

$n\backslash k$	3	4	5	6
10	0.002	0.008	0.021	0.034
11	0.009	0.035	0.102	0.181
12	0.035	0.145	0.480	0.968
13	0.131	0.598	2.237	5.043
14	0.489	2.476	10.306	25.922
15	1.825	10.193	47.311	130.265
16	6.841	42.052	212.918	643.981

(B) V2 algorithm

$n\backslash k$	3	4	5	6
10	0.001	0.002	0.002	0.003
11	0.005	0.007	0.009	0.011
12	0.021	0.028	0.035	0.042
13	0.079	0.104	0.130	0.152
14	0.299	0.418	0.526	0.618
15	1.374	1.922	2.303	2.631
16	5.818	7.871	9.243	10.573

(C) Our Algorithm

FIGURE 5. Time in seconds to compute $|S_n(T_k(231))|$.

which V2 does; and to discover that a permutation is not an avoider, V1 is expected to look at at least as many sequences as V2, sometimes re-examining sequences because patterns share a prefix. Thus the only speedup comes from not using a small hash table to store pattern prefixes.

In Figure 4, the naive generate-and-check algorithm's asymptotic disadvantage to ours is clear, while PermLab's algorithm mostly hides its disadvantage because n is small. However, as we shall see briefly, the asymptotic difference stands out far more when Π is a large set of patterns. This occurs for two reasons. First, since $S_n(\Pi)$ grows slower, we can access larger n. Second, for a fixed-size of patterns, having more patterns leads to a larger constant behind the cost of detecting pattern-avoidance, so that its asymptotic nature is no longer masked for small n. Indeed, suppose we were to compute $S_n(\pi)$ for some $\pi \in S_k$. Then the average number of subsequences checked by PermLab to detect that a permutation in S_n avoids π might be something like

$$\sum_{i=0}^{k-2} \frac{\binom{n-k+i}{i}}{(i+1)!},$$

assuming each permutation in S_i is equally likely to appear as a given i-subsequence. But (using V2) if Π is a large set of patterns with k_i distinct i-prefixes, this becomes

$$\sum_{i=0}^{k-2} k_{i+2} \frac{\binom{n-k+i}{i}}{(i+1)!}. \tag{3}$$

If k_{i+2} is within an order of magnitude of $(i+2)!$, then the $1/(i+2)!$ term no longer hides the asymptotics for small n. In contrast, our algorithm runs in time $O(|S_{n-1}|nk)$ regardless of $|\Pi|$.

In Figure 5, we show algorithm performance for large sets of patterns. Let $X_k(231)$ be the set of permutations in S_k containing a 123-hit. Then $S_n(X_k(231))$ contains the permutations in S_n with no k-letter subsequences containing any 231-patterns; of course for $n \geq k$ this is simply $S_n(231)$, which has size the n-th Catalan number C_n. By computing $S_n(X_k(231))$ for a fixed k, we can see how each algorithm performs for varying n and a fixed large set of patterns in S_k. At the same time, by computing $S_n(X_k(231))$ for a fixed n, we can see how each algorithm's performance changes when we use a larger set of larger patterns to solve the exact same pattern-avoidance problem. For this experiment, we use our V2-variant of PermLab, since it is far more suitable for a large set of patterns. Indeed, while the V1 variant may compute $S_{12}(123456)$ more than twice as fast as V2, it computes $S_{12}(X_6(231))$ more

$n\backslash k$	3	4	5	6
8	0.777	0.932	0.953	0.945
9	0.787	0.946	0.969	0.970
10	0.797	0.956	0.978	0.983
11	0.806	0.963	0.984	0.989
12	0.814	0.969	0.988	0.993
13	0.822	0.973	0.991	0.995
14	0.829	0.977	0.993	0.996
15	0.836	0.980	0.994	0.997
16	0.842	0.982	0.995	0.998

FIGURE 6. Fraction of permutation-subsequences viewed by V2 that lie in permutations in $S_n(X_k(123))$, rather than in non-avoiders.

than ten times slower (11.7 seconds for V1 versus .97 seconds for V2 and 2.1 seconds even using naive generate-and-check algorithm). In fact, while V1 is ever at most some constant times faster than V2, V2 can be arbitrarily faster than V1 for large sets of patterns.

Unlike our algorithm, V2 and the naive generate-and-check algorithm do not scale well to large sets of large patterns. Let us take a moment to better understand V2's performance in Figure 5. The algorithm's run-time is essentially proportional to the total number of permutation subsequences it examines to determine whether permutations are avoiding, taking between 12 to 13 nanoseconds on average per subsequence (based on time per subsequence when $k = 6$). Although there are several times more non-avoiding permutations tested than avoiding ones, most of the time is devoted to the the avoiding ones, and the ratio of work for avoiding versus non-avoiding checks stays relatively constant (Figure 6). Thus for a given k, the run-time scales according to the number of avoiders times the average number of subsequences checked per avoider. The latter is estimated by Equation 3.

However, Equation (3) assumes that the i-letter subsequences of permutations in $S_n(\Pi)$ are equi-distributed among S_i. A good example where this is not the case is when computing $S_n(X_3(231))$. Here 12 is the only valid pattern 2-prefix, but 12 and 21 are *not* equally likely to be formed by $n-1$ and n in a 231-avoider. In particular, if $n-1$ precedes n, then n must be in the final position. As a result, Equation (3) overestimates the average work done per avoider by a bit less than two thirds. However, Equation (3) is more accurate for larger k, with percent error 15.9%, 4.9%, and 1.7% for $k = 4, 5, 6$ respectively and $n = 16$. More importantly, Equation (3)'s accuracy is relatively static, with the ratio of the actual work done to the predicted work done dropping from $n = 10$ to $n = 16$ by .06, .015, .0005, and $-.0002$ for $k = 3, 4, 5, 6$ respectively. As a result, the work to generate $S_n(X_k(231))$ from $S_{n-1}(S_k(231))$ is proportional to

$$C_n \sum_{i=1}^{k-2} k_{i+2} \frac{\binom{n-k+i}{i}}{(i+1)!}.$$

Equation (3)'s accuracy was also relatively static for the single pattern-case tested in Figure 4. The largest drop in the ratio of work done to work anticipated by Equation (3) from $n = 8$ to 13 is .88 dropping to .69 for $k = 5$. As a direction for future work, we suggest studying PermLab's (and the naive generate-and-check algorithm's) performance further. For example, is Equation (3) ever an underestimate for some $\Pi \subseteq S_k$? For a given $\Pi \in S_k$, is Equation (3)'s error bounded by some constant c, possibly depending on Π?

There are down-closed sets of permutations where the difference in algorithm performance might be much more extreme, even for single patterns. For an example, one could consider permutations with inversion number bounded above by some constant, and use a pattern with few inversions. Indeed, in any case where we are interested in a down-closed set of permutations, many of which contain numerous hit-prefixes, the contrast between the algorithmic performances would be highlighted.

6.3. **In Comparison with ΠDDbased Algorithms.** In 2013, Inoue, Takahisa, and Minato, introduced an algorithm for generating $S_n(\Pi)$ which, although asymptotically mysterious, runs very fast in certain cases [17]. Their algorithm represents sets of permutations in a data structure called a ΠDD, which in

Alg \ Set-Size	1	2	3	4
Our Alg.	11.559	0.8310	0.0652	0.0237
ΠDD-Based	2.31	2.39	2.00	3.34

FIGURE 7. Time in seconds to generate $S_{13}(1234)$, $S_{13}(1234, 2341)$, $S_{13}(1234, 2341, 3412)$, $S_{13}(1234, 2341, 3412, 4123)$ respectively.

practice compresses sets of related permutations well. They then use set operations, in addition to other select operations easily performed on a ΠDD, in order to construct the ΠDD for $S_n(\Pi)$. If the ΠDD's compression algorithm works sufficiently well, the algorithm can potentially run in less than $|S_n(\Pi)|$ time. On the other hand, with poor compression, the algorithm could perform far worse than the naive generate-and-check algorithm.

In Figure 4, we compare the ΠDD-based algorithm with our algorithm for computing $S_k(\pi)$ for $\pi \in \{231, 2431, 24531\}$ and n varying. While the ΠDD-based algorithm runs extremely fast for $|\Pi| = 1$, it performs far worse for sets of multiple patterns. In particular, as $|\Pi|$ increases, the time to compute $S_n(\Pi)$ tends to gradually increase as $|\Pi|$ grows, instead of rapidly shrinking with $|S_n(\Pi)|$ as does our algorithm. (For example, see Figure 7) This is possibly because the ΠDD-based algorithm works by generating the non-avoiders and subtracting those from S_n, rather than directly generating the avoiders.

Observe that Proposition 3.2 can be rewritten in terms of set-operations. Given a permutation s, define $S \uparrow_j^i$ to be the permutation obtained by inserting $(j - 0.5)$ in position i of s, and then normalizing the result to a permutation. For example, $12345678 \uparrow_5^2 = N(1(4.5)2345678) = 152346789$. In turn, given a set of permutation S, define $S \uparrow_j^i$ to be $\{s \uparrow_j^i \mid s \in S\}$. Then Proposition 3.2 can be restructured as follows.

Proposition 6.1. *Let Π be a set of permutations, the largest of which is size k. Then for $n > k$,*

$$S_n(\Pi) = \cap_{j=1}^{k+1} \cup_{i=1}^{n} S_{n-1}(\Pi) \uparrow_j^i .$$

Thus it would be an interesting direction of future work to efficiently implement the $\uparrow_j^i$ operation for sets represented using ΠDD. Using this, our PPA algorithm could potentially be re-implemented using ΠDD's with run time in practice less than $\Theta(|S_n(\Pi)|)$, even for large Π.

In 2014, Inoue, Takahisa, and Minato extended their algorithm to count Π-hits in each $w \in S_n$ [16]. In particular, they build the ΠDDfor the set of permutations with i Π-hits for each i. In Figure, 4, we compare the run-time-performance of the ΠDD-based algorithm to our own for single patterns $\pi \in \{231, 2431, 24531\}$; this time, our algorithm tend to have the edge. Additionally, unlike our algorithm, which runs in time $O(n!k)$ regardless of $|\Pi|$, the ΠDD-based algorithm tends to scale approximately linearly in terms of $|\Pi|$; for example, see Figure 3.

There are many interesting questions still to be asked about the ΠDD-based algorithms. Can they be extended to apply to a down-closed set of permutations, rather than S_n? Can theoretical bounds be proven for their worst-case run-time performance?

7. Conclusion

In this paper, we provided the first provably fast algorithms for constructing $S_n(\Pi)$ and for counting Π-hits in each $w \in S_n$. Surprisingly, even though detecting *whether* a permutation contains a pattern is NP-complete [6], detecting *which* permutations contain that pattern is polynomial time per permutation.

Our computationally-generated conjectures in Section 5 seem particularly ripe for future work. It would also be interesting to run additional large-scale computations to learn about trends in pattern-avoidance. Our software can generate $|S_1(\Pi)|, \ldots, |S_n(\Pi)|$ for every $\Pi \subseteq S_4$ (regardless of $|\Pi|$) in just over an hour on our Amazon C4.8xlarge machine[8]; since PermLab's disadvantage to our pattern-avoidance algorithm is not too severe for patterns of size four, we suspect the same computation could easily be orchestrated to run within ten hours using (a parallized version of) PermLab. However, our algorithm opens new doors for computing $S_n(\Pi)$ in cases where Π contains a large number of large patterns. This makes

[8]Run in parallel with hyperthreading enabled for a total of 36 hardware threads. Our code is parallelized using Cilk.

large-scale computations finally feasible for families of pattern-avoidance relations involving patterns of size six, seven, and eight. In addition, our $O(n!k)$ algorithm for counting Π-hits in each $w \in S_n$ also brings previously unobtainable computations within reach.

Our investigation prompts several directions for future algorithmic work. Can Algorithm 1's run-time be improved to $\Theta(|S_{\leq n-1}(\Pi)| \cdot n)$ using the technique from Theorem 3.14? How can our results for vincular patterns be further developed? Can Algorithm 1 be efficiently implemented to take advantage of ΠDD's (Section 6.3)? Do the ΠDD-based algorithms of Inoue, Takashisa, and Minato [16, 17] have good worst-case or expected run-times? What can be said about the accuracy of Equation (3)'s error for a given Π?

8. Acknowledgments

This research was conducted at the University of Minnesota Duluth REU and was supported by NSF grant 1358695 and NSA grant H98230-13-1-0273. Machine time was provided by an AWS in Education Research grant. The author thanks Joe Gallian for suggesting the problem; Samuel D. Judge and David Moulton for offering advice on directions of research; Yuma Inoue for discussing and providing implementations of ΠDD-based algorithms; and Michael Albert for his PermLab implementation.

References

[1] Shlomo Ahal and Yuri Rabinovich. On complexity of the subpattern problem. *SIAM Journal on Discrete Mathematics*, 22(2):629–649, 2008.

[2] Michael Albert. Permlab: Software for permutation patterns, 2012. `http://www.cs.otago.ac.nz/PermLab`, 2012.

[3] Michael H Albert, Robert EL Aldred, Mike D Atkinson, and Derek A Holton. Algorithms for pattern involvement in permutations. In *Algorithms and Computation*, pages 355–367. Springer, 2001.

[4] Eric Babson and Einar Steingrímsson. Generalized permutation patterns and a classification of the mahonian statistics. *Sém. Lothar. Combin*, 44(B44b):547–548, 2000.

[5] Jörgen Backelin, Julian West, and Guoce Xin. Wilf-equivalence for singleton classes. *Advances in Applied Mathematics*, 38(2):133–148, 2007.

[6] Prosenjit Bose, Jonathan F Buss, and Anna Lubiw. Pattern matching for permutations. *Information Processing Letters*, 65(5):277–283, 1998.

[7] Marie-Louise Bruner and Martin Lackner. A fast algorithm for permutation pattern matching based on alternating runs. In *Algorithm Theory–SWAT 2012*, pages 261–270. Springer, 2012.

[8] Marie-Louise Bruner and Martin Lackner. The computational landscape of permutation patterns. *Pure Mathematics and Applications*, 24(2):83–101, 2013.

[9] Anders Claesson. Generalized pattern avoidance. *European Journal of Combinatorics*, 22(7):961–971, 2001.

[10] Anders Claesson, Vít Jelínek, and Einar Steingrímsson. Upper bounds for the stanley–wilf limit of 1324 and other layered patterns. *Journal of Combinatorial Theory, Series A*, 119(8):1680–1691, 2012.

[11] Paul Erdös and George Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

[12] Sylvain Guillemot and Dániel Marx. Finding small patterns in permutations in linear time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 82–101. SIAM, 2014.

[13] Yijie Han, Sanjeev Saxena, and Xiaojun Shen. An efficient parallel algorithm for building the separating tree. *Journal of Parallel and Distributed Computing*, 70(6):625–629, 2010.

[14] Yijie Han and Shanky Saxena. Parallel algorithms for testing length four permutations. In *Parallel Architectures, Algorithms and Programming (PAAP), 2014 Sixth International Symposium on*, pages 81–86. IEEE, 2014.

[15] Louis Ibarra. Finding pattern matchings for permutations. *Information Processing Letters*, 61(6):293–295, 1997.

[16] Yuma Inoue, TODA Takahisa, and Shin-Ichi Minato. Generating sets of permutations with pattern occurrence counts using permutation decision diagrams. *TCS Technical Report, Series A*, 14(78), 2014.

[17] Yuma Inoue, TODA Takahisa, and Shin-Ichi Minato. Implicit generation of pattern-avoiding permutations by using permutation decision diagrams. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 97(6):1171–1179, 2014.

[18] Tomáš Kaiser and Martin Klazar. On growth rates of closed permutation classes. *Electron. J. Combin*, 9(2):03, 2002.

[19] Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S Iliopoulos, Kunsoo Park, Simon J Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014.

[20] Sergey Kitaev. *Patterns in permutations and words*. Springer Science & Business Media, 2011.

[21] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[22] Marcin Kubica, T Kulczyński, Jakub Radoszewski, Wojciech Rytter, and T Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.

[23] Adam Marcus and Gábor Tardos. Excluded permutation matrices and the stanley–wilf conjecture. *Journal of Combinatorial Theory, Series A*, 107(1):153–160, 2004.

[24] Michal Opler. Major index distribution over permutation classes. *arXiv preprint arXiv:1505.07135*, 2015.

[25] Rodica Simion and Frank W Schmidt. Restricted permutations. *European Journal of Combinatorics*, 6(4):383–406, 1985.

[26] Neil JA Sloane et al. The on-line encyclopedia of integer sequences, 2003.

[27] William A Stein, T Abbott, M Abshoff, et al. Sage mathematics software, 2011.

[28] V Yugandhar and Sanjeev Saxena. Parallel algorithms for separable permutations. *Discrete Applied Mathematics*, 146(3):343–364, 2005.