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

WILLIAM KUSZMAUL
Stanford University
kuszmaul@stanford.edu

Abstract. Given a set Π of permutation patterns of length at most k, we present an algorithm for building $S_{\leq n}(\Pi)$, the set of permutations of length at most n avoiding the patterns in Π, in time $O(|S_{\leq n-1}(\Pi)| \cdot k + |S_n(\Pi)|)$. Additionally, we present an $O(n!k)$-time algorithm for counting the number of copies of patterns from Π in each permutation in S_n. Surprisingly, when $|\Pi| = 1$, this runtime can be improved to $O(n!)$, spending only constant time per permutation. Whereas the previous best algorithms, based on generate-and-check, take exponential time per permutation analyzed, all of our algorithms take time at most polynomial per outputted permutation.

If we want to solve only the enumerative variant of each problem, computing $|S_{\leq n}(\Pi)|$ or tallying 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.

Our algorithms extend to considering permutations in any set closed under standardization 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 $\tau \in S_n$, a π-*hit* or *copy of* π in τ is a k-letter subsequence of τ order-isomorphic to π. For example, 857 is a 312-hit in 18365472 (Figure 1). If τ contains no π-hits, we say that τ *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 312, of which there are the Catalan number C_n [20]. 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 [19]. 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 former 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, 19].

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-hard [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 $\tau \in S_n(\pi)$, which we will refer to as the *PPM* problem.

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

In 1998, Bose, Buss, and Lubiw showed that PPM is NP-hard in general [6]. Since then, research on PPM algorithms has traveled down two paths, the first to find an exponential-time 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 $|\pi| = 4$ [3, 14]; and an algorithm whose runtime depends on a natural complexity-measure of π, running fast for π with small complexity-measure [1]. Additionally, results have been found for more general types of patterns such as vincular patterns [8].

For experimental research purposes, 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 copies of π in each $\tau \in S_n$.

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 copies of π 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 5.5), 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 $O(|S_{\leq n-1}(\pi)| \cdot k + |S_n(\pi)|)$ time, spending linear time per output permutation. Our algorithms are the first proven to spend sub-exponential time per output permutation.

In Section 6, for the enumerative versions of PPA and PPC, 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 runtimes depend only on $k = \max_{\pi \in \Pi} |\pi|$, building $S_n(\Pi)$ in time $O(|S_{\leq n-1}(\Pi)| \cdot k + |S_n(\Pi)|)$ (Theorem 4.6) and counting Π-patterns in each τ in S_n in time $O(n! \cdot k)$ (Theorem 5.4). Additionally, our algorithms easily adapt to finding avoiders and counting copies of patterns in Π in arbitrary downsets 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.

Our algorithms open new doors for data-driven research studying the structure of permutation classes. Previously daunting large-scale computations are now easily within reach. For example, our software can generate $|S_1(\Pi)|, \ldots, |S_{16}(\Pi)|$ for every $\Pi \subseteq S_4$ (regardless of $|\Pi|$) in just under twenty-five minutes on our Amazon C3.8xlarge machine[1]. A brief analysis of the resulting number sequences reveals that hundreds of OEIS sequences seemingly previously unaffiliated with pattern avoidance can be used to enumerate $|S_n(\Pi)|$ for some Π. Several of these seem quite interesting. For example, OEIS sequence A204746 [26]x counts the number of $n \times n$ binary arrays with every 3×3 subblock containing exactly three runs of three equal elements, where each run can be on a horizontal, vertical, diagonal, or antidiagonal. We conjecture that this sequence enumerates $|S_n(\Pi)|$ for 67 distinct $\Pi \subseteq S_4$ (after accounting for trivial symmetries), among the shortest of which is $\Pi = \{2341, 2314, 4213, 2413, 4132, 1432, 1234\}$. Our software, our data on $\Pi \subseteq S_4$, and our OEIS-matches analysis can be found at `github.com/williamkuszmaul/patternavoidance`. A full description of the analysis is available in [22].

The layout of this paper is as follows. In Section 2, we introduce (mostly standard) conventions. In Section 3, we introduce and analyze a simplified version of our PPA algorithm, which is then refined in Section 4, and extended to PPC in Section 5. In Section 6, we modify our algorithms to achieve good space utilization. In Section 7, we compare our algorithms (running in serial) experimentally to the best alternatives. Finally, Section 8 concludes with directions of future work and some results on vincular patterns.

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

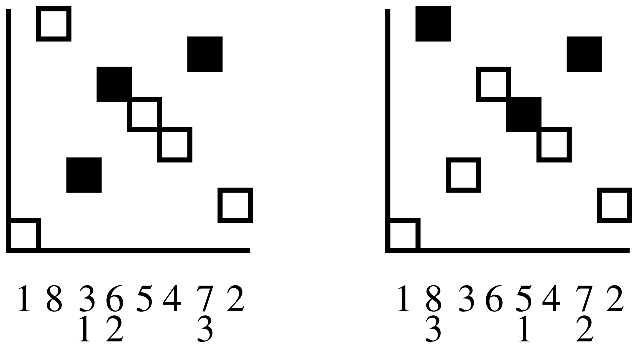

Figure 1. Example 123-hit and 312-hit in 18365472. In this figure, a permutation is represented graphically. A square is placed at position i, j when the ith element of the permutation is j. In the left figure the subword 367 is shown to form a 123-hit, and in the right figure the subword 857 is shown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

2. Definitional 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.*

Definition 2.2. *Given a word τ of n distinct letters, the* standardization *$st(\tau)$ is the permutation $\sigma \in S_n$ such that $\tau_i < \tau_j$ exactly when $\sigma_i < \sigma_j$.*

Example 2.3. The standardization of 5397 is $\text{st}(5397) = 2143$.

Definition 2.4. *Two words τ_1 and τ_2 are order-isomorphic if $st(\tau_1) = st(\tau_2)$.*

Definition 2.5. *Let $\pi \in S_k$ and $\tau \in S_n$. A* π-hit *is any subword of τ order-isomorphic to π. On the other hand,* τ avoids the pattern π *if τ has no π-hits*

Example 2.6. 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 $\tau \in S_n$ 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 τ *avoids the patterns* in Π.

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

Definition 2.7. *We use $S_{\leq n}$ to denote $S_1 \cup S_2 \cup \cdots \cup S_n$.*

Definition 2.8. *Let π (resp. Π) be a pattern (resp. set of patterns), and D be a set. Then $D(\pi)$ (resp. $D(\Pi)$) is the subset of D which avoids π (resp. Π).*

Example 2.9. Since S_n is the set of 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 *downsets* of permutations.

Definition 2.10. *A set of permutations D is a* downset *if for all $\tau \in D$, for all non-empty subwords τ' of τ, $st(\tau') \in D$.*

Examples of downsets include $S_{\leq n}$, the permutations with j or fewer inversions (for a constant j), the permutations with j or smaller major index, the permutations avoiding a given set of patterns, and the separable permutations. Additionally, the unions and the intersections of downsets are also downsets.

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

Definition 2.11. *Given $\tau \in S_n$ and $i \in \{1, \ldots, n+1\}$, we define $\tau \uparrow^i$ to be the permutation obtained by inserting $n+1$ to be in the i-th position of τ.*

Definition 2.12. *Given $\tau \in S_n$ and $i \in \{1, \ldots, n\}$, we define $\tau \downarrow_i$ to be the standardization of the word obtained by removing the letter $(n-i+1)$ from τ.*

Example 2.13. For example, $13524 \uparrow^2 = 163524$, while $13524 \downarrow_2 = \mathrm{st}(1352) = 1342$.

Note that $\uparrow^i$ and $\downarrow_i$ are not inverses. Whereas $\uparrow^i$ inserts a letter into the i-th position, $\downarrow_i$ removes the i-th largest-valued letter. Though subtle, these distinctions will play a critical role in the optimizations presented in Section 4.

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

3. PPA in time polynomial per avoider

In this section, we introduce the key ideas for obtaining an asymptotically fast algorithm to build $S_{\le n}(\Pi)$. Combined, these ideas yield a simple algorithm running in time $O(S_{\le n-1}(\Pi)n^2k)$, the first algorithm to spend only polynomial time per Π-avoiding permutation. This algorithm can additionally be adapted to build $D(\Pi)$ for a downset D (assuming constant-time membership queries for D.) In later sections, we will introduce techniques for reducing the polynomial term and for achieving good space bounds.

Our algorithm relies fundamentally on a simple observation which transforms pattern detection into a dynamic programming problem. Whereas detecting whether a permutation $\tau \in S_n$ contains a pattern $\pi \in S_k$ naively takes time $O\left(\binom{n}{k}k\right)$, Proposition 3 shows how to perform the same computation in polynomial time using information about smaller permutations.

Proposition 3.1. *Let Π be a set of patterns, each of length at most k, and let τ be a permutation length n. Pick X to be any set of at least $\min(k+1, n)$ distinct entries of τ. Then τ lies in $S_n(\Pi)$ if and only if the following two conditions hold.*

(1) *$\tau \notin \Pi$ and*
(2) *for each entry $x \in X$, the standardization of τ with the entry x removed lies in $S_{n-1}(\Pi)$.*

Proof. Suppose $\tau \in S_n(\Pi)$. Then Condition (1) holds trivially, and Condition (2) holds because $S_{\le n}(\Pi)$ is a downset.

On the other hand, suppose Conditions (1) and (2) hold. Observe that if the standardization of τ with the letter x removed lies in $S_{n-1}(\Pi)$, then any Π-hit in τ must use the letter x. Thus Condition (2) implies that any Π-hit in τ must use at least $\min(k+1, n)$ distinct letters of τ. If $k < n$, this is impossible, since the longest pattern in Π is length at most k. If $k \ge n$, then τ can only contain a Π-hit if that Π-hit comprises all of τ, a contradiction by Condition (1). $\square$

Example 3.2. In Figure 2, we apply Proposition 3 to 25143 and to 34215 in order to determine whether each avoids 123. For each permutation, we remove its first, second, third, and fourth letters, standardize the result, and record whether it avoids 123. Assuming that we have already computed which 4-letter permutations avoid 123, this entire process takes polynomial time for each permutation.

Because all four tests pass for 25143, we conclude that it avoids the pattern 123. On the other hand, 34215 fails two tests and contain a 123 pattern.

The decision to remove each the first four letters was arbitrary, since Proposition 3 allows us to use any four letters. In fact, our actual algorithms will always use the letters $n, n-1, \ldots, (n - \max(k+1, n) + 1)$ when testing for avoidance. Although unmotivated for the time being, this decision will make optimizations in Section 4 easier to discuss.

Armed with Proposition 3 we can now derive a fast algorithm. The simplest algorithm for building $S_{\le n}(\Pi)$ is to brute-force check whether each permutation τ in $S_{\le n}$ is Π-avoiding. If we do this by checking every $|\pi|$-subsequence of τ 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 time $O\left(n! \cdot \binom{n}{k} kl\right)$.

Letter removed	Permutation in S_4	Avoids 123?
First letter:	25143	
	4132	yes
Second letter:	25143	
	2 143	yes
Third letter:	25143	
	14 32	yes
Fourth letter:	25143	
	241 3	yes

Letter removed	Permutation in S_4	Avoids 123?
First letter:	34215	
	3214	yes
Second letter:	34215	
	3 214	yes
Third letter:	34215	
	23 14	no
Fourth letter:	34215	
	231 4	no

FIGURE 2. Applying Proposition 3 to determine whether 25143 and whether 34215 avoid the pattern 123.

Our first task is to shrink the $n!$ term. Observe that $S_{\leq n}(\Pi)$ is a downset. Consequently, every element in $S_n(\pi)$ can be obtained by inserting n into some position of a permutation in $S_{n-1}(\Pi)$. Thus we can build $S_n(\Pi)$ from $S_{n-1}(\Pi)$ by checking pattern-avoidance for each permutation τ obtained by inserting n into some position of an element in $S_{n-1}(\Pi)$. Since there are at most $|S_{n-1}(\Pi)| \cdot n$ such τ, this yields an algorithm which generates $S_{\leq n}(\Pi)$ in time

$$O\left(|S_{\leq n-1}(\Pi)| \cdot n \cdot \binom{n}{k} \cdot kl\right).$$

Our next task is to shrink the $\binom{n}{k}$ and eliminate the dependence on l. Recall that proposition 3 shows that if $S_{n-1}(\Pi)$ is already computed, then checking whether $\tau \in S_n(\Pi)$ for some $\tau \in S_n$ can be achieved in $O(kn)$ time, rather than in $O\left(\binom{n}{k} \cdot kl\right)$ time. In particular, to see that $\tau \in S_n(\Pi)$ we need only check that $\tau \notin \Pi$ and that $\tau \downarrow_i \in S_{n-1}(\Pi)$ for each $i \in [\min(k+1, n)]$ (Algorithm 1). This brings our total runtime down to $O(|(S_{\leq n-1}(\Pi)| \cdot n^2 k)$. Note that the number of patterns in Π does not increase the time needed to detect whether a permutation is Π-avoiding.

Theorem 3.3. *Let Π be a set of patterns and $k = \max_{\pi \in \Pi} |\pi|$. The set $S_{\leq n}(\Pi)$ can be constructed in $O(|(S_{\leq n-1}(\Pi)| \cdot n^2 k)$ time.*

Proof. By Proposition 3, this is accomplished through Algorithm 2. Note that one can easily obtain each $\tau \downarrow_i$ from τ in $O(n)$ time. □

Remark 3.4. Note that for single patterns π, we have $|S_n(\pi)| \leq |S_{n+1}(\pi)|$ for all n. In particular, depending on π, one of the maps $\tau \to \tau \uparrow^1$ or $\tau \to \tau \uparrow^{n+1}$ 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 runtime $O(|S_n(\pi)| \cdot n^3 k)$, which using results from the next section can be reduced to $O(|S_n(\pi)| \cdot nk)$.

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].

Algorithm 1: DetectAvoider

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

Algorithm 2: BuildAvoiders

```
Input: Hash table $\Pi$, $k := \max_{\pi\in\Pi} |\pi|$, $n$
Output: A hash table containing $S_{\leq n}(\Pi)$
UnorderedSet Avoiders;
Queue Unprocessed;
if $1 \notin \Pi$ then
    Unprocessed.enqueue(1);
    Avoiders.add(1);
while not Unprocessed.empty() do
    Perm := Unprocessed.dequeue();
    for $i \in \{1, \ldots, Perm.size() + 1\}$ do
        NewPerm := Perm$\uparrow^i$;
        if DetectAvoiders(Avoiders, $\Pi$, k, NewPerm) then
            Avoiders.insert(NewPerm);
            if NewPerm.size() < n then
                Unprocessed.enqueue(NewPerm);
return Avoiders;
```

Observe that Algorithm 2 can be easily modified to generate $S_{\leq n}(\Pi) \cap D$ for downsets D, assuming membership in D can be determined in constant time. In particular, prior to checking whether NewPerm is an avoider, we throw out NewPerm if it is not in D. In fact, using the optimized version of Algorithm 2 which will be presented in Section 4 (Theorem 4.6), we can build $D(\Pi)$ in time $O(|D(\Pi) \cap S_{\leq n-1}|n)$. An example candidate for D is the set of permutations in $S_{\leq n}$ with j or fewer inversions for a fixed j; 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.[2]

Other examples of downsets 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].

4. Optimizations for PPA

In the preceding section, we presented Algorithm 2 which builds $S_{\leq n}(\Pi)$ in time $O(|(S_{\leq n-1}(\Pi)| \cdot n^2 k)$. In this section, we introduce two optimizations, each of which reduces the runtime by a factor of n, bringing the total runtime down by a factor of n^2 to $O(|S_{\leq n-1}(\Pi)| \cdot k + |S_n(\Pi)|)$. The first optimization relies on encoding permutations as integers, allowing permutation operations to be performed using bit manipulations. The second optimization performs pattern detection on multiple permutations at once, leading to additional speedup.

Because S_n and $S_n(\pi)$ grow quickly, foreseeable applications of our algorithms are likely to use permutations that can 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). Using this assumption, we can shave off a factor of n from Algorithm 2's runtime.

Theorem 4.1. *By representing permutations as integers, Algorithm 2 can be implemented to run in time $O(|S_{\leq n-1}(\Pi)| \cdot nk)$.*

Proof. The analysis from Theorem 3.3 of Algorithm 2 assumes that each computation of $\tau \uparrow^i$ or $\tau \downarrow_i$ takes time $O(n)$. In this analysis, we will show that in the context of Algorithm 2, and with a bit of extra bookkeeping, these computations can each be reduced to constant time. In particular, each $\tau \uparrow^i$

[2] In this case, a clever implementation could further reduce the time to $O(|D(\Pi)| \cdot k)$ by only considering Perm $\uparrow^i$ for values of i large enough to keep the number of inversions below j.

can be accomplished in constant time using bit hacks, and each $\tau\downarrow_{i+1}$ can be obtained from $\tau\downarrow_i$ using bit hacks and information about τ^{-1}.

Note that the following operations are constant time for integers representing a word τ stored as a permutation with block-size j: $\tau(i)$, which returns the i-th letter of τ; $\mathbf{setpos}(\tau, i, j)$, which sets the i-th letter of τ to value j; $\mathbf{insertpos}(\tau, u, v)$, which slides the final $n-u+1$ letters of τ one position to the right, and inserts the value v in the u-th position; and $\mathbf{killpos}(\tau, i)$, which slides the final $n-i$ letters of τ 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 τ is an integer representing a word,

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

which can be implemented in C as

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

Using these basic operations, if τ represents a permutation in S_n, we can compute $\tau\uparrow^i = \mathbf{insertpos}(\tau, i, n+1)$ in constant time. We can compute $\tau\downarrow_{i+1}$ from $\tau\downarrow_i$ and τ^{-1} (i.e., the integer representation of the inverse permutation) in constant time by inserting $n-i$ into position $\tau^{-1}(n-i+1)$ of $\tau\downarrow_i$, and then killing the $\tau^{-1}(n-i)$-th letter of the result. Finally, we can also compute $(\tau\uparrow^{i+1})^{-1}$ from $(\tau\uparrow^i)^{-1}$ and τ in constant time, by incrementing the $\tau(i)$-th position of $\tau\uparrow^i$ and decrementing the $(n+1)$-th position. Consequently, for Algorithm 2, all computations of $\tau\uparrow^i$ and $\tau\downarrow_i$ can be performed in constant time, as long as one also computes and stores $(\tau\uparrow^i)^{-1}$ when computing $\tau\uparrow^i$.

This reduces the runtime for Algorithm 2 from $O(|S_{\leq n-1}(\Pi)| \cdot n^2 k)$, as derived in Theorem 3.3, to $O(|S_{\leq n-1}(\Pi)| \cdot nk)$, as desired. □

Surprisingly, we can further optimize the algorithm to shave off another linear factor. To do this, we must introduce the notion of an *extension map.*

Definition 4.2. *Let $\tau \in S_n(\Pi)$. Let I be the set of $i \in [n+1]$ such that $\tau\uparrow^i \in S_{n+1}(\Pi)$. Then the* extension map *$\Psi^\Pi(\tau)$ of τ is the $(n+1)$-letter bit map with i-th letter equal to 1 exactly when $i \in I$, and equal to 0 otherwise.*

Example 4.3. 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 but inserting 3 in the third position does not.

Definition 4.4. *Let $j \in [n]$ and $\tau \in S_n(\Pi)$. Let I be the set of $i \in [n+1]$ such that $\tau\uparrow^i\downarrow_{j+1} \in S_n(\Pi)$. Then the* $(n-j+1)$-ignoring extension map *$\Psi^\Pi_{n-j+1}(\tau)$ of τ is the $(n+1)$-letter bit map with i-th letter equal to one exactly when $i \in I$.*

Example 4.5. 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 permutation 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.

Theorem 4.6. *Let Π be a set of patterns, the longest of which is length k. The values $|S_1(\Pi)|, \dots, |S_n(\Pi)|$ can be computed in time $O(|(S_{\leq n-1}(\Pi)| \cdot k)$. Moreover, in time $O(|(S_{\leq n-1}(\Pi)| \cdot k + |S_n(\Pi)|)$, one can construct $S_{\leq n}(\Pi)$.*

Proof. Our computational model allows us to store $O(n)$ bits in an integer. As a result, we can store extension maps as unsigned integers, allowing us to perform integer operations on them in constant time.

Consider a Π-avoiding permutation $\tau \in S_m(\Pi)$ for some $m \geq k$. (We will handle smaller τ later.) By Proposition 3,

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

where $\wedge$ denotes the *and* operator. (Call this Observation (1).)

Moreover, given τ^{-1} and $\Psi^{\Pi}(\tau\downarrow_{m-j+1})$, we can compute $\Psi^{\Pi}_j(\tau)$ in constant time. (Call this Observation (2).) In particular, since $\Psi^{\Pi}(\tau\downarrow_{m-j+1})$ is the extension map of the standardization of τ with j removed, and since $\Psi^{\Pi}_j(\tau)$ is the j-ignoring extension map of τ, we get the following relationship. For $i \in [1, \tau^{-1}(j)]$, the i-th bit of $\Psi^{\Pi}_j(\tau)$ is the same as that of $\Psi^{\Pi}(\tau\downarrow_{m-j+1})$; and for $i \in [\tau^{-1}(j)+1, n+1]$ the i-th bit of $\Psi^{\Pi}_j(\tau)$ equals the $(i-1)$-th bit of $\Psi^{\Pi}(\tau\downarrow_{m-j+1})$. Thus $\Psi^{\Pi}_j(\tau)$ can be obtained from $\Psi^{\Pi}(\tau\downarrow_{m-j+1})$ by shifting bits in positions $\tau^{-1}(j)+1, \ldots, n+1$ to the right by one, and inserting a copy of the $\tau^{-1}(j)$-th bit in the $(\tau^{-1}(j)+1)$-th position.

Combining Observations (1) and (2), we can build $\{\Psi^{\Pi}(\tau) : \tau \in S_m(\Pi)\}$ in time $O(|S_m(\Pi)| \cdot k)$ out of $\{(\tau, \tau^{-1}) : \tau \in S_m(\Pi)\}$ and $\{\Psi^{\Pi}(\tau) : \tau \in S_{m-1}(\Pi)\}$. 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 $O(|S_{n-1}(\Pi)|)$ time and gives us a value for $|S_n(\Pi)|$. If $m < n-1$, then we want to build $\{(\tau, \tau^{-1}) : \tau \in S_{\le m+1}(\Pi)\}$ and then repeat the entire process for $m+1$.

From the extension maps of avoiders in S_m, we can obtain $S_{m+1}(\Pi)$ in time $O(|S_{m+1}(\Pi)|)$ by repeatedly taking advantage of the **ctz** operation in order to extract the 1-bit positions from each map. Constructing $\{\tau^{-1} : \tau \in S_{m+1}(\Pi)\}$ is not as easy however, and would take $O(|S_{m+1}(\Pi)| \cdot n)$ time to do naively. We are saved, however, by the fact that we only need each τ^{-1} to be correct in its largest k values. Thus if we choose to only update these values, then we can obtain the inverses in time $O(|S_{m+1}(\Pi)| \cdot k)$.

At this point we have an algorithm which only starts to work once we have already built the avoiders in S_k. In particular, Observation (1), which states that

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

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

5. Counting Pattern Occurrences in $S_{\le n}$

Building on the ideas in Sections 3 and 4, 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 downset $D \subseteq S_{\le n}$ and the inverses of each $\tau \in D$, our algorithm extends to run in $O(|S|k)$ time. The inverses for each $\tau \in D$ are required so that $\tau\downarrow_1, \ldots, \tau\downarrow_{k+1}$ may be computed in $O(k)$ time (using the same technique as in Theorem 4.1); recall, however, that they can be obtained at no additional asymptotic cost if we build D 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 τ, let $P(\tau)$ denote the number of Π-hits in τ.

Definition 5.1. *Let $P_i(\tau)$ be the number of Π-hits in τ containing the entire i-upfix of τ. (Recall that the i-upfix of τ refers to the i largest-valued letters in τ.)*

Example 5.2. Suppose $\tau = 1234$ and $\Pi = \{123\}$. Then $P_0(\tau) = 4$, $P_1(\tau) = 3$, $P_2(\tau) = 2$, $P_3(\tau) = 1$, and $P_4(\tau) = 0$.

Observe that $P_0(\tau) = P(\tau)$. Surprisingly, whereas $P(\tau)$ satisfies no straightforward recurrence relation, $P_i(\tau)$ does. The following proposition can be thought of as a natural extension of Proposition 3 from the context of pattern detection to the context of pattern counting.

Proposition 5.3. *Let $\tau \in S_n$. Then*

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

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

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

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

Given a permutation τ and its inverse τ^{-1}, and using the optimizations introduced in Theorem 4.1, Proposition 5.3 yields an $O(k)$ algorithm to compute each $P_i(\tau)$ for a permutation in terms of each $P_i(\tau')$ for smaller permutations τ' (Algorithm 3)[3]. Note that Algorithm 3 treats each P_i as a globally accessible hash table mapping permutations to integers, and that Algorithm 3 assumes access to Π and k.

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

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

Theorem 5.4. *Given a downset $D \subseteq S_{\leq n}$, and the inverse of each $d \in D$, one can construct $P(\tau)$ for each $\tau \in D$ in $O(|D| \cdot k)$ time.*

Proof. Given D, bucket-sort can be used to construct each of $D \cap S_i$ for $1 \leq i \leq n$ in $O(|D|)$ total time. One can then use Algorithm 3 to compute $P(\tau)$ for each $\tau \in (D \cap S_i)$ for i from 1 to n (as well as $P_i(\tau)$ for $O(k)$ different i). This takes $O(|D| \cdot k)$ time. Note that we are assuming each $\tau \downarrow_i$ in the algorithm takes constant time to compute; this is easily accomplished using the exact same technique as in Theorem 4.1, and is the reason we require the inverse of each $d \in D$. □

The algorithm in Theorem 5.4 can also be adapted for downsets $D \subseteq S_{\leq n}$ for which set membership is conditional on the number of Π-hits of a permutation.

For one important example of this, suppose D is the set of permutations in $S_{\leq n}$ with j or fewer Π-hits for some fixed j. Since D is a downset, every element in $S_n \cap D$ is of the form $\tau \uparrow^i$ for some $i \in \{1, \dots n\}$ and $\tau \in S_{n-1} \cap D$. Thus we can build $S_n \cap D$ out of $S_{n-1} \cap D$ while simultaneously using Proposition 5.3 to compute $P(\tau)$ for each $\tau \in D$. To accomplish this, we use Algorithm 4 to identify whether a permutation τ is in $S_n \cap D$ based on values of $P_i(\tau')$ for $\tau' \in S_{n-1} \cap D$. At the same time, if Algorithm 4 concludes that a permutation is in $S_n \cap D$, it computes $P_i(\tau)$ for each i. In turn, Algorithm 5 uses Algorithm 4 to compute each $P_i(\tau)$ for all $\tau \in D$. Observe that Algorithm 5 runs in $O(|D \cap S_{\leq n-1}| \cdot nk)$ time. In particular, for each permutation τ in $D \cap S_{\leq n-1}$, we run Algorithm 4 on each $\tau \uparrow^i$.

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

Theorem 5.5. *Let $\pi \in S_k$. Then the number of π-hits in each $\tau \in S_n$ can be computed in $\Theta(n!)$ time, regardless of k.*

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

Note that given that the $(i-1)$-upfix of τ is order-isomorphic to the $(i-1)$-upfix of π, and using information about τ^{-1}, one can check whether the i-upfix is as well in constant time. With this in mind, we can analyze our new algorithm.

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

[3]Recall τ^{-1} is needed for fast computation of $\tau \downarrow_i$ for $i \in \{1, \dots, k+1\}$.

Algorithm 4: CountHitsBounded Counts Π-hits in τ if τ has at most j Π-hits (and is thus said to be in D); returns false if τ has more than j Π-hits.

```
Input: HashTable $H$ such that $H \cap S_{n-1} = D \cap S_{n-1}$, Permutation $\tau \in S_n$, $j$
Output: Returns whether $\tau$ has $\leq j$ $\Pi$-hits. If true, assigns values to $P_i(\tau)$ for each
        $i \in \{0, \ldots, k+1\}$
$P_{k+1}(\tau) := 0$;
for $i \in \{k, \ldots, 0\}$ do
    $P_i(\tau) := 0$;
    if $i = n$ and $\tau \in \Pi$ then
        $P_i(\tau) := 1$;
    if $i < n$ then
        if $\tau \downarrow_{i+1} \notin H$ then
            for $r \in \{k+1, \ldots, i+1\}$ do
                $P_r.remove(\tau)$;
            return false;
        $P_i(\tau) := P_i(\tau \downarrow_{i+1}) + P_{i+1}(\tau)$
if $P_0(\tau) > j$ then
    for $i \in \{k+1, \ldots, 0\}$ do
        $P_i.remove(\tau)$;
    return false;
return true;
```

Algorithm 5: BuildPermsWithBoundedHits

```
Input: $n$, $j$, $\Pi$
Output: Returns set of permutations $\tau$ in $S_{\leq n}$ with $\leq j$ $\Pi$-hits; Also computes values of $P_i(\tau)$.
UnorderedSet D;
Queue Unprocessed;
if $1 \notin \Pi$ or $j \geq 1$ then
    Unprocessed.enqueue(1);
    D.add(1);
while not Unprocessed.isempty() do
    Perm := Unprocessed.dequeue();
    for $i \in \{1, \ldots, Perm.size + 1\}$ do
        NewPerm := Perm$\uparrow^i$;
        if CountHitsBounded(D, NewPerm, j) then
            D.insert(NewPerm);
            if NewPerm.size() $< n$ then
                Unprocessed.enqueue(NewPerm);
return D;
```

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. This would not necessarily reduce the runtime of the enumeration algorithm in Theorem 4.6 to $O(|S_{n-1}(\pi)|)$, however, since the algorithm would still be asymptotically bottle-necked by the updating of inverses. Regardless, the hack can be added to both algorithms to reduce cache misses.

To prove the conjecture, one would show that $\mathbb{E}(T_r(\tau))$ is small for τ from $S_n(\pi) \uparrow := \{\tau \uparrow^i | \ i \in \{1, \ldots, n\}, \tau \in S_{n-1}(\pi)\}$. For example, when $\pi = 123 \cdots k$, this can be done as follows. Suppose $\tau \in S_n(\pi) \uparrow$ has an increasing r-upfix $p_1 p_2 \cdots p_r$. Since τ's r-upfix is in increasing order and τ avoids $12 \cdots k$, it follows that any reordering of the letters in τ's r-upfix will also result in a permutation avoiding

$12 \cdots k$. If we reorder the letters in the r-upfix to be $p_2p_3p_4 \cdots p_i$ with p_1 inserted in some position other than the first, then we see that we can match each $\tau \in S_n(\pi) \uparrow$ having an increasing r-upfix with $r-1$ permutations, each in $S_n(\pi) \uparrow$ and each with an increasing $(r-1)$-upfix (but not an increasing r-upfix). It follows that $\mathbb{E}(T_r(\tau)) \leq \mathbb{E}(T_{r-1}(\tau))/r$ over $\tau \in S_n(\pi) \uparrow$, implying the conjecture for $\pi = 123 \cdots k$. In fact, proving $\mathbb{E}(T_r(\tau)) \leq \mathbb{E}(T_{r-1}(\tau))/c$ for any constant $c > 1$ would be sufficient, which is why the conjecture seems very likely to be true in general.

Remark 5.6. In practice, the technique introduced in Theorem 5.5 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-upfix of a permutation τ is an i-upfix of *any* permutation $\pi \in \Pi$. An efficient technique for this, taking constant time per i-upfix, is discussed in Appendix A.

6. Eliminating the memory bottleneck and making parallelism easy

So far, our algorithms have required space nearly proportional to their runtime. 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 $P(\tau)$ for each $\tau \in S_n$, then space efficiency is futile. However, in this section, we assume that the goal is *enumeration*, to either evaluate $|S_n(\Pi)|$ or to tally how many $\tau \in S_n$ have each value of $P(\tau)$.

For this entire section, define T to be the inclusion tree of all permutations, meaning that 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. In particular, these are the permutations in $S_{|v|+j}$ whose smallest $|v|$ letters are order-isomorphic to v.

The following lemma will play a key role in improving memory utilization. In particular, the recursions on which both our PPA and PPC algorithms are based compute information about a given $\tau \in S_n$ based only on information about $\tau \downarrow_i$ for each $i \in \min(n, k+1)$. Lemma 6.1 tells us that we can therefore compute information about the $(k+1)$-th level children of v based only on information about the k-th level children of v.

Lemma 6.1. *For a given a set of permutations A, define $A \downarrow_i$ as $\{s \downarrow_i : s \in A\}$. Then for any node $v \in T$ and for any positive integers i and j satisfying $i \leq j$,*

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

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

Our approach to PPC in Section 5 uses $O((n-1)!k)$ space. In particular, we perform a breadth-first traversal of $T \cap S_{\leq n}$, using Proposition 5.3 at each node v to compute each $P_i(v)$. However, Lemma 6.1 suggests an $O(n^{k+1}k)$-space approach.

Theorem 6.2. *Let Π be a set of patterns, the longest of which is length k. We can count permutations in S_n by Π-hits in $O(n!k)$ time using $O(n^{k+1}k)$ space.*

Proof. If $n < k$, then Algorithm 3 is already sufficient. Otherwise, we restructure the algorithm as follows.

First compute $P_i(\tau)$ for each $\tau \in S_{\leq k}$ using the Algorithm 3 (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 by Lemma 6.1 this computation depends only on elements of $C^k(v \downarrow_1)$, which we will have already computed due to the depth-first nature of our computation[4]. Having computed each P_i of each $c \in C^k(v)$, we update our tally of how many permutations have each number of Π-hits, and we 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 of T, we no longer need to store these $P_i(c)$ values and we throw them out.

[4]Note that as a base case we consider C^k of the empty permutation to be the permutations of size k.

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 needed to keep tally of permutations by Π-hits is bounded above by $\binom{n}{k} + \binom{n}{k-1} + \cdots + \binom{n}{1} \leq n^{k+1}$. □

Remark 6.3. When $|\Pi| = 1$, we can use the technique from Theorem 5.5 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. Additionally, the technique brings the runtime down to $O(n!)$.

We can apply a similar optimization to the PPA algorithm introduced in Theorem 4.6.

Theorem 6.4. *Let Π be a set of patterns, the longest of which is length k. The values $|S_1(\Pi)|, \ldots, |S_n(\Pi)|$ can be computed in time $O(|(S_{\leq n-1}(\Pi)| \cdot k)$ and space $O(n^k)$.*

Proof. If $n \leq k-1$, then Theorem 4.6 is already sufficient, requiring space no more than $n! \leq n^k$. Otherwise, we restructure the theorem's algorithm as follows.

As a base case, use the algorithm from Theorem 4.6 to count Π-avoiders in $S_{\leq k-1}(\Pi)$ and build the extension map for each avoider in $S_{k-1}(\Pi)$.

Computing the extension map for an avoider v uses only the extension maps of $v\downarrow_1, \ldots, v\downarrow_k$. Therefore, by Lemma 6.1, to build the extension maps for each permutation in $C^{k-1}(v) \cap S_{\leq n}(\Pi)$, it suffices to have stored the extension maps for each permutation in $C^{k-1}(v\downarrow_1) \cap S_{\leq n}(\Pi)$. Thus after we have built the extension maps for $S_{k-1}(\Pi)$ as a base case, we can restructure the algorithm from Theorem 4.6 as follows. We perform a depth-first traversal on $T \cap S_{n-k}$. When visiting a node v, build the extension map of each permutation in $C^{k-1}(v) \cap S_{\leq n}(\Pi)$. Store these extension maps to be accessed later in the depth-first traversal; upon returning to v's parents, however, we throw these extension maps away.

In order for this to not compromise the algorithm's runtime, there is a slight subtlety. We need to be able to build $C^{k-1}(v) \cap S_{\leq n}(\Pi)$ out of $C^{k-1}(v\downarrow_1) \cap S_{\leq n}(\Pi)$ in time $O(|C^{k-1}(v\downarrow_1) \cap S_{\leq n}(\Pi)| \cdot k + |C^{k-1}(v) \cap S_{\leq n}(\Pi)|)$. To accomplish this, when visiting $v\downarrow_1$ in the depth-first traversal, one partitions $C^{k-1}(v\downarrow_1) \cap S_{\leq n}(\Pi)$ according to the position of $|v|$ relative to $1, 2, \ldots, |v|-1$[5]. Then, when visiting v, one can build $C^{k-1}(v) \cap S_{\leq n}(\Pi)$ from the extension maps of elements of $C^{k-1}(v\downarrow_1) \cap S_{\leq n}(\Pi)$ having $|v|$ in the same position relative to $1, 2, \ldots, |v|-1$ as in v[6]. This resolves the issue, allowing us to retain our original runtime.

At any given moment in the algorithm, at each depth of the depth-first traversal, we store no more than n^{k-1} extension maps. Thus the algorithm uses $O(n^k)$ space. □

In practice, if $C^k(v) \cap S_{\leq n}(\Pi)$ is never very large for any v, then the space usage for PPA may be much smaller than $O(n^k)$. In particular, the expected memory consumption at a given instance in the algorithm is $O(\sum_{j=k}^{n-1} |S_j(\Pi)|/|S_{j-k+1}(\Pi)|)$, which by the former 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 many pattern-avoidance computations in parallel can treat space usage as growing linearly with n.

In addition to reducing space-usage asymptotically, the optimizations in this section make parallelizing our algorithm easy. Indeed, by visiting multiple of a node's children at a time, the depth-first traversals of T can be parallelized without risking write-conflicts for hash maps containing either extension maps or P_i-values. Although we test our algorithms in serial in Section 7, we have released a parallelized implementation at `github.com/williamkuszmaul/patternavoidance`.

7. Implementations and Performance Comparisons

In this section, we test our algorithms' performance against other algorithms[7]. 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.

[5]Since the algorithm in Theorem 4.6 remembers the positions of the final k letters of the inverses of the permutations in $C^{k-1}(v\downarrow_1) \cap S_{\leq n}(\Pi)$, we can build this partition in time $O(k|C^{k-1}(v\downarrow_1) \cap S_{\leq n}(\Pi)|)$.

[6]Recall that the **ctz** operator can be used to quickly determine for which positions an extension map takes value 1

[7]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.

In Section 7.1, 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 7.2, we test both of our algorithms for PPA and PPC 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`.

7.1. **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.

We implemented our $O(|S_{\le n-1}(\Pi)|\cdot k)$-time and $O(n^k)$-space algorithm for counting $|S_1(\Pi)|, \ldots, |S_n(\Pi)|$, 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 $S_{\le n}$ such that the children of $\tau \in S_k$ are each option for $\tau \uparrow^i$. 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 $T_n \cap S_{\le n}(\Pi)$, visiting a node's children only if the node itself avoids Π. In order to determine whether a permutation τ avoids Π, the algorithm applies a straightforward variant of the technique presented in Appendix A to the subsequences of at most k letters in τ, only considering a given subsequence of length i if the $(i-1)$-upfix of that sequence is order-isomorphic to the $(i-1)$-upfix of some pattern in Π.

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 $T_n \cap S_{\le n-1}(\Pi)$, 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 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 upfix 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 upfix.

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, just as we did for our generate-and-check implementation, we use a straightforward variant of the technique from Appendix A to check whether a subsequence is order-isomorphic to an upfix of *any* Π-hit in amortized constant time (using information about previous subsequences). This leads to significant speedup when there are many shared upfixes among the permutations in Π. On the other hand, if $|\Pi| = 1$, then the overhead of using the small hash table required for the technique from Appendix A 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 3 (the final subfigure of which is discussed later), we compare the performances of the algorithms handling single patterns[8], 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 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 upfix. Thus the only speedup comes from not using a small hash table to store pattern upfixes.

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

$n\backslash k$	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	0.000	0.001	0.001
9	0.000	0.004	0.009	0.010
10	0.001	0.019	0.045	0.050
11	0.003	0.062	0.217	0.351
12	0.008	0.339	1.779	3.590
13	0.029	2.183	16.293	39.665

(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 3. 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 $\{231, 2431, 24531, 246531\}$.

Whereas the naive generate-and-check algorithm's disadvantage grows with n, Permlab's algorithm appears to largely hide its asymptotic disadvantage for single patterns. Both algorithms perform many times worse than our algorithm.

In Figure 4, 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. Unlike our algorithm, V2 and the naive generate-and-check algorithm do not scale well to large sets of large patterns. 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 than ten times slower (11.7 seconds for V1 versus .97 seconds for V2). 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.

There are downsets of permutations where the difference in algorithm performances 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 downset of permutations, many of which contain numerous hit-upfixes, the contrast between the algorithmic performances would be highlighted.

$n\backslash k$	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	0.000	0.000	0.001	0.001
11	0.002	0.002	0.003	0.003
12	0.008	0.008	0.011	0.013
13	0.029	0.031	0.039	0.046
14	0.103	0.110	0.140	0.163
15	0.368	0.396	0.504	0.589
16	1.314	1.432	1.822	2.128

(C) Our Algorithm

Figure 4. Time in seconds to compute $|S_n(X_k(231))|$.

Alg \ Set-Size	1	2	3	4
Our Alg.	99.765	3.731	0.197	0.049
ΠDD-Based	30.648	28.377	18.328	32.820

Figure 5. Time in seconds to generate $S_{15}(1234)$, $S_{15}(1234, 2341)$, $S_{15}(1234, 2341, 3412)$, $S_{15}(1234, 2341, 3412, 4123)$ respectively.

7.2. **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 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 3, 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 stay roughly constant as $|\Pi|$ grows, instead of rapidly shrinking with $|S_n(\Pi)|$ as is the case for our algorithm. For an example of this, see Figure 5. 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 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 standardizing the result to a permutation. For example, $12345678 \uparrow_5^2 = \mathrm{st}(1(4.5)2345678) = 152346789$. In turn, given a set of permutations A, define $A \uparrow_j^i$ to be $\{s \uparrow_j^i | s \in A\}$. Then Proposition 3 yields the following proposition.

Proposition 7.1. *Let Π be a set of permutations, the largest of which is size k. Then for $n > k$,*

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

$n\backslash k$	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) ΠDD-based algorithm

FIGURE 6. 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) ΠDD-based algorithm

FIGURE 7. Time in seconds to count for each $\tau \in S_{\leq n}$ the number of k-letter sequences containing a 231 pattern.

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 runtime 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 $\tau \in S_n$ [16]. In particular, they build the ΠDD for the set of permutations with i Π-hits for each i. In Figure 6, we compare the runtime-performance of the ΠDD-based algorithm to our own for single patterns $\pi \in \{231, 2431, 24531\}$, as well as to an optimized generate-and-check implementation; this time, our algorithm tends 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 $|\Pi|$. This can be seen, for example, in Figure 7, in which our PPC algorithm, the ΠDD-based PPC algorithm, and the generate-and-check implementation are tested on the pattern set $\{\pi \in S_k \mid \pi \text{ contains a 231-hit}\}$. Both the implementation of our PPC algorithm and the generate-and-check algorithm are available at `github.com/williamkuszmaul/patternavoidance`.

There are many interesting questions still to be asked about the ΠDD-based algorithms. Can they be extended to apply to a downset of permutations rather than S_n? Can theoretical bounds be proven for their worst-case runtime performances?

8. Conclusion

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

Our investigation prompts several directions for future algorithmic work. Can Algorithm 2's runtime be improved to $\Theta(|S_{\leq n-1}(\Pi)| \cdot n)$ using the technique from Theorem 5.5? Can Algorithm 2 be efficiently implemented to take advantage of ΠDD's (Section 7.2)? Do the ΠDD-based algorithms of Inoue, Takashisa, and Minato [16, 17] have good worst-case or expected runtimes?

Additionally, it would be interesting to extend our results to vincular patterns, in which patterns may come with additional adjacency constraints. In the rest of this section, we will present our progress on this so far, as well as the challenges involved in making further progress.

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].

Vincular patterns come with position-adjacency constraints, meaning certain pairs of adjacent positions in the pattern are required to also be adjacent in the hit. In the context of our algorithms, it is more convenient to discuss covincular patterns, however, which are equivalent to vincular patterns and come with value-adjacency constraints. A covincular patterns is a pair (π, X) where $\pi \in S_k$ and $X \subseteq \{0, \dots, k\}$. An element $x \in X$ from 1 to $k-1$ indicates that the letters x and $x+1$ must be represented by adjacently-valued letters in any pattern occurrence. If $0 \in X$ (resp. $k \in X$), then the smallest (resp. largest) letter in any pattern-occurrence must also be the smallest (resp. largest) letter in the entire permutation.

Example 8.1. The covincular pattern $(123, \{0, 2\})$ appears $n-2$ times in the identity permutation $e_n \in S_n$. In particular, any three letters a_1, a_2, a_3 forming the pattern must satisfy $a_1 = 1$ and $a_3 = a_2 + 1$.

Given a covincular pattern (π, X) and a permutation τ, define $P_i(\tau)$ to be the number of (π, X)-hits in τ using the entire i-upfix of τ. The following proposition extends Proposition 5.3 to the case where Π comprises a single covincular pattern.

Proposition 8.2. *Let $\tau \in S_n$. Let (π, X) be a covincular pattern. Then*

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

Proof. Cases (2)–(4) follow just as in the proof of Proposition 5.3. Suppose $i < n$, $i \leq |\pi|$, and $i \in X$. If $i = |\pi|$, then since $i < n$ and $i \in X$ we see that $P_i(\tau) = 0$, which Case (4) tells us is also the value of $P_{i+1}(\tau)$, as desired. On the other hand, if $i < |\pi|$, then since $i \in X$, any copy of π in τ^{-1} using the i-upfix of τ must also use the $i+1$-upfix of τ. Thus $P_i(\tau) = P_{i+1}(\tau)$. □

Using this recurrence, analogues of Theorems 5.4 and 5.5 follow with only slightly modified proofs.

Theorem 8.3. *Let (π, X) be a covincular pattern. Given a downset $D \subseteq S_{\leq n}$ and d^{-1} for each $d \in D$, one can count (π, X)-hits in τ for each $\tau \in D$ in $O(|D| \cdot |\pi|)$ time.*

Proof. If one modifies Algorithm 3 to use the recurrence from Proposition 8.2 on τ rather than the recurrence from Proposition 5.3 on τ, then the proof follows just as for Theorem 5.4.

□

Theorem 8.4. *Let (π, X) be a covincular pattern. Then the number of (π, X)-hits in each $\tau \in S_n$ can be computed in $\Theta(n!)$ time, regardless of $|\pi|$.*

Proof. The result follows using the same technique as in the proof of Theorem 5.5. In particular, when applying the recursion from Proposition 8.2 to compute $P_i(\tau)$ for some $\tau \in S_n$, one checks whether the i-upfix of τ is order-isomorphic to the i-upfix of π. If the two are not order-isomorphic, $P_i(\tau)$ must be zero. $\square$

Theorem 8.4 shows that we can count (π, X)-hits for each covincular 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 covincular permutations in $O(n!|\Pi|)$ time.

It is still an open problem, however, to quickly build $S_{\leq n}(\Pi)$ if Π comprises covincular patterns. The difficulty in this comes from the fact that $S_{\leq n}(\Pi)$ needs not be a downset in this case. Indeed, removing a letter from an avoider τ may introduce a vincular pattern which was not previously present. For example, the permutation 1324 does not contain a $(123, 1)$ pattern, but removing 3 results in a permutation which does.

One special case of a covincular pattern is when $X = \{1, \ldots, k-1\}$, meaning that every pair of adjacently valued letters in the pattern must also be adjacently valued in any occurrence of the pattern. This is what's known as a *consecutive pattern.* For consecutive patterns π, Theorem 8.3 counts π-hits in a downset D in $O(|D| \cdot |\pi|)$ time (assuming d^{-1} is known for each $d \in D$). Interestingly, in this case, the PPM problem (detecting a π-pattern in a single permutation $\tau \in S_n$) already has a linear time solution due to Kubica, Kulczyński, Radoszewski, Rytter, and Waleń [21]. A similar result was found independently by Kim et. al. [18].

9. 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.

10. Appendix A

Given a set of pattern Π, a permutation $\tau \in S_n$, and the inverse τ^{-1}, a common computation is to compute for which i the i-upfix of τ is order-isomorphic to the i-upfix of any permutation $\pi \in \Pi$. It turns out that one can run this check for all i in the range $1, \ldots, r$ in time $O(r)$.

To do this, for each $i \in [r]$, we compute the standardization of the i-upfix of τ, and then check its membership in a hash table[9] containing the standardized i-upfixes of each $\pi \in \Pi$. In fact, it turns out we can compute the standardizations of each of the successive i-upfixes 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 τ's i-upfix appear to the right of $n-i+1$; this tells us in what position to insert 1 into the standardized $(i-1)$-upfix in order to obtain the standardized i-upfix. The insertion can then be performed using bit hacks in constant time.

[9]Note that there is a small preprocessing cost which must be paid at the beginning of the algorithm to build these hash tables.

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] 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.

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

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

[21] 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.

[22] William Kuszmaul. Fast algorithms for finding pattern avoiders and counting pattern occurrences in permutations. *arXiv preprint arXiv:1509.08216*, 2015.

[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.