COMBINATORIAL GENERATION VIA PERMUTATION LANGUAGES

ELIZABETH HARTUNG, HUNG P. HOANG, TORSTEN MÜTZE, AND AARON WILLIAMS

ABSTRACT. In this work we present a general and versatile algorithmic framework for exhaustively generating a large variety of different combinatorial objects, based on encoding them as permutations. This approach provides a unified view on many known results and allows us to prove many new ones. In particular, we obtain the following four classical Gray codes as special cases: the Steinhaus-Johnson-Trotter algorithm to generate all permutations of an \( n \)-element set by adjacent transpositions; the binary reflected Gray code to generate all \( n \)-bit strings by flipping a single bit in each step; the Gray code for generating all \( n \)-vertex binary trees by rotations due to Lucas, van Baronaigien, and Ruskey; the Gray code for generating all partitions of an \( n \)-element ground set by element exchanges due to Kaye.

We present two distinct applications for our new framework: The first main application is the generation of pattern-avoiding permutations, yielding new Gray codes for different families of permutations that are characterized by the avoidance of certain classical patterns, (bi)vincular patterns, barred patterns, Bruhat-restricted patterns, mesh patterns, monotone and geometric grid classes, and many others. We thus also obtain new Gray code algorithms for the combinatorial objects that are in bijection to these permutations, in particular for five different types of geometric rectangulations, also known as floorplans, which are divisions of a square into \( n \) rectangles subject to certain restrictions.

The second main application of our framework are lattice congruences of the weak order on the symmetric group \( S_n \). Recently, Pilaud and Santos realized all those lattice congruences as \((n-1)\)-dimensional polytopes, called quotientopes, which generalize hypercubes, associahedra, permutahedra etc. Our algorithm generates the equivalence classes of each of those lattice congruences, by producing a Hamilton path on the skeleton of the corresponding quotientope, yielding a constructive proof that each of these highly symmetric graphs is Hamiltonian. We thus also obtain a provable notion of optimality for the Gray codes obtained from our framework: They translate into walks along the edges of a polytope.

(Elizabeth Hartung) MASSACHUSETTS COLLEGE OF LIBERAL ARTS, UNITED STATES
(Hung P. Hoang) DEPARTMENT OF COMPUTER SCIENCE, ETH ZÜRICH, SWITZERLAND
(Torsten Mütze) DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF WARWICK, UNITED KINGDOM
(Aaron Williams) BARD COLLEGE AT SIMON’S ROCK, UNITED STATES

E-mail addresses: e.hartung@mcla.edu, hung.hoang@inf.ethz.ch, torsten.mutze@warwick.ac.uk, awilliams@simons-rock.edu.

Key words and phrases. Exhaustive generation algorithm, Gray code, pattern-avoiding permutation, weak order, lattice congruence, quotientope, Hamilton path, rectangulation.

Torsten Mütze is also affiliated with Charles University, Faculty of Mathematics and Physics, and was supported by GACR grant GA 19-08554S, and by DFG grant 413902284.
1. Introduction

In computer science we frequently encounter different kinds of combinatorial objects, such as permutations, binary strings, binary trees, set partitions, spanning trees of a graph, and so forth. There are three recurring fundamental algorithmic tasks that we want to perform with such objects: counting, random generation, and exhaustive generation. For the first two tasks, there are powerful general methods available, such as generating functions \[^{FS09}\] and Markov chains \[^{Jer03}\], solving both problems for a large variety of different objects. For the third task, namely exhaustive generation, however, we are lacking such a powerful and unifying theory, even though some first steps in this direction have been made (see Section 1.2 below). Nonetheless, the literature contains a vast number of algorithms that solve the exhaustive generation problem for specific classes of objects, and many of these algorithms are covered in depth in the most recent volume of Knuth’s seminal series ‘The Art of Computer Programming’ \[^{Knu11}\].

1.1. Overview of our results. The main contribution of this paper is a general and versatile algorithmic framework for exhaustively generating a large variety of different combinatorial objects, which provides a unified view on many known results and allows us to prove many new ones. The basic idea is to encode a particular set of objects as a set of permutations \(L_n \subseteq S_n\), where \(S_n\) denotes the set of all permutations of \([n] := \{1, 2, \ldots, n\}\), and to use a simple greedy algorithm to generate those permutations by cyclic rotations of substrings, an operation we call a ‘jump’. This works under very mild assumptions on the set \(L_n\), and allows us to generate more than double-exponentially \((\text{in } n)\) many distinct sets \(L_n\). Moreover, the jump orderings obtained from our algorithm translate into listings of combinatorial objects where consecutive objects differ by small changes, i.e., we obtain Gray code algorithms \[^{Sav97}\], and those changes are smallest possible in a provable sense. The main tools of our framework are Algorithm J and Theorem 1 in Section 2. In particular, we obtain the following four classical Gray codes as special cases: (1) the Steinhaus-Johnson-Trotter algorithm to generate all permutations of \([n]\) by adjacent transpositions, also known as plain change order \[^{Tro62}, {Joh63}\]; (2) the binary reflected Gray code (BRGC) to generate all binary strings of length \(n\) by flipping a single bit in each step \[^{Gra53}\]; (3) the Gray code for generating all \(n\)-vertex binary trees by rotations due to Lucas, van Baronaigien, and Ruskey \[^{LvBR93}\]; (4) the Gray code for generating all set partitions of \([n]\) by exchanging an element in each step due to Kaye \[^{Kay76}\].

We present two distinct applications for our new framework: The first main application is the generation of pattern-avoiding permutations, yielding new Gray codes for different families of permutations characterized by the avoidance of certain classical patterns, (bi)vincular patterns \[^{BS00}, {BMCDK10}\], barred patterns \[^{Wes90}\], Bruhat-restricted patterns \[^{WY06}\], mesh patterns \[^{BC11}\], monotone and geometric grid classes \[^{HV06}, {AAB^+13}\], and many others. We thus also obtain new Gray code algorithms for the combinatorial objects that are in bijection to these permutations, in particular for five different types of geometric rectangulations/floorplans \[^{ABP06}, {Rea12b}, {ABBM^+13}, {CSS18}\], which are divisions of a square into \(n\) rectangles subject to different restrictions; see Figure 2. Our results in this area are summarized in Theorem 4, Remark 5, and Table 1 in Section 3.

The second main application of our framework are lattice congruences of the weak order on the symmetric group \(S_n\). This area has beautiful ramifications into groups, posets, polytopes, geometry, and combinatorics \[^{Rea12a}, {Rea16a}, {Rea16b}\]. There are double-exponentially many distinct such
lattice congruences, and they generalize many known lattices such as the Boolean lattice, the Tamari lattice [Tam62], and certain Cambrian lattices [Rea06, CP17]. Recently, Pilaud and Santos [PS19] realized all those lattice congruences as \((n - 1)\)-dimensional polytopes, called quotientopes, which generalize hypercubes, associahedra, permutahedra etc. Our algorithm generates the equivalence classes of each of those lattice congruences, by producing a Hamilton path on the skeleton of the corresponding quotientope, yielding a constructive proof that each of these highly symmetric graphs is Hamiltonian; see Figure 6. We thus also obtain a provable notion of optimality for the Gray codes obtained from Algorithm J: Jump operations translate into walks along the edges of a polytope. Our results in this area are summarized in Theorem 6 and Corollary 7 in Section 4. In this extended abstract, many formal proofs are omitted due to length constraints; for details see [HHMW19] and [HM19].

1.2. Related work. Avis and Fukuda’s [AF96] reverse-search is a general technique for exhaustive generation. They consider the objects to be generated as nodes of a graph, and connect them by edges that model local modifications. This flip graph is equipped with an objective function and traversed backwards by a local search algorithm. Reverse-search is complementary to our permutation based approach, as it uses a fundamentally different encoding of objects. The permutation encoding seems to allow for more fine-grained control (optimal Gray codes) and even faster algorithms.

Another method for combinatorial counting and exhaustive generation is the ECO framework by Barcucci, Del Lungo, Pergola, and Pinzani [BDLPP99]. It uses an infinite tree with integer node labels, and a set of production rules for creating the children of a node based on its label. Bacchelli, Barcucci, Grazzini, and Pergola [BBGP04] also used ECO for exhaustive generation. Dukes, Flanagan, Mansour, and Vajnovszki [DFMV08], Baril [Bar09], and Do, Tran and Vajnovszki [DTV19] used ECO for deriving Gray codes for different classes of pattern-avoiding permutations, which works under certain regularity assumptions on the production rules. The main difference between ECO and our framework is that the change operations on the label sequences of the ECO tree do not necessarily correspond to Gray-code like changes on the corresponding combinatorial objects. Minimal jumps in a permutation, on the other hand, correspond to minimal changes on the combinatorial objects in a provable sense, even though they may involve several entries of the permutation.

Li and Sawada [LS09] considered another tree-based approach for generating so-called reflectable languages, yielding Gray codes for \(k\)-ary strings and trees, restricted growth strings, and open meandric systems (see also [XCU10]). Ruskey, Sawada, and Williams [RSW12, SW12] proposed a generation framework based on binary strings with a fixed numbers of 1s, called bubble languages, which allows to generate e.g. combinations, necklaces, Dyck words, and Lyndon words. In the resulting cool-lex Gray codes, any two consecutive words differ by a cyclic prefix rotation.

Pattern avoidance is central in combinatorics, as illustrated by the books [Kit11, Bón12], and by the conference ‘Permutation Patterns’, held annually since 2003. Many fundamental classes of combinatorial objects are in bijection with pattern-avoiding permutations (see Table 1 and [Ten18]). For instance, Knuth [Knu98] proved that all 123-avoiding or 132-avoiding permutations are counted by the Catalan numbers. Concerning counting and generation, a few tree-based algorithms for pattern-avoiding permutations have been proposed [Eli07, DFMV08, Bar08, Bar09]. The problem has also been studied extensively from an algorithmic point of view. E.g., testing whether a permutation \(\pi\) contains a pattern \(\tau\) is NP-complete in general [BBL98]. Jelínek and Kynčl [JK17] proved that the problem remains hard even if \(\pi\) and \(\tau\) have no decreasing subsequence of length 4
and 3, respectively, which is best possible. On the other hand, Guillemot and Marx \cite{GM14} showed that the problem can be solved in time $2^{O(k^2 \log k) n} \log n$, where $n$ is the length of $\pi$ and $k$ is the length of $\tau$, a considerable improvement over the obvious $O(n^k)$ algorithm (see also \cite{Koz19}).

2. Generating permutations by jumps

In this section we present a simple greedy algorithm, Algorithm J, for exhaustively generating a given set $L_n \subseteq S_n$ of permutations, and we show that the algorithm works successfully under very mild assumptions on the set $L_n$ (Theorem 1).

2.1. Preliminaries. We use $S_n$ to denote the set of all permutations of $[n] := \{1, \ldots, n\}$, and we write $\pi \in S_n$ in one-line notation as $\pi = \pi(1)\pi(2)\ldots\pi(n) = a_1a_2\ldots a_n$. We use $\text{id}_n = 12\ldots n$ to denote the identity permutation, and $\varepsilon \in S_0$ to denote the empty permutation. For any $\pi \in S_{n-1}$ and any $1 \leq i \leq n$, we write $c_i(\pi) \in S_n$ for the permutation obtained from $\pi$ by inserting the new largest value $n$ at position $i$ of $\pi$, i.e., if $\pi = a_1\ldots a_{n-1}$ then $c_i(\pi) = a_1\ldots a_{i-1}n a_i\ldots a_{n-1}$. Moreover, for $\pi \in S_n$, we write $p(\pi) \in S_{n-1}$ for the permutation obtained from $\pi$ by removing the largest entry $n$. Here, $c_i$ and $p$ stand for the child and parent of a node in the tree of permutations discussed shortly. Given a permutation $\pi = a_1\ldots a_n$ with a substring $a_i\ldots a_j$ with $a_i > a_{i+1}, \ldots, a_j$, a right jump of $a_i$ by $j-i$ steps is a cyclic left rotation of this substring by one position to $a_{i+1}\ldots a_j a_i$. Similarly, given a substring $a_i\ldots a_j$ with $a_j > a_i, \ldots, a_{j-1}$, a left jump of $a_j$ by $j-i$ steps is a cyclic right rotation of this substring to $a_j a_i \ldots a_{j-1}$. For example, a right jump of 5 in the permutation $261354$ by 2 steps yields $261534$.

2.2. The basic algorithm. Our approach starts with the following simple greedy algorithm to generate a set of permutations $L_n \subseteq S_n$. We say that a jump is minimal (w.r.t. $L_n$), if a jump of the same value in the same direction by fewer steps creates a permutation that is not in $L_n$.

\begin{algorithm}
\textbf{Algorithm J} (Greedy minimal jumps). This algorithm attempts to greedily generate a set of permutations $L_n \subseteq S_n$ using minimal jumps starting from an initial permutation $\pi_0 \in L_n$.

\begin{enumerate}
\item \textbf{Initialize} Visit the initial permutation $\pi_0$.
\item \textbf{Jump} Generate an unvisited permutation from $L_n$ by performing a minimal jump of the largest possible value in the most recently visited permutation. If no such jump exists, or the jump direction is ambiguous, then terminate. Otherwise visit this permutation and repeat J2.
\end{enumerate}
\end{algorithm}

Put differently, in step J2 we consider the entries $n, n-1, \ldots, 2$ of the current permutation in decreasing order, and for each of them we check whether it allows a minimal left or right jump that creates a previously unvisited permutation, and we perform the first such jump we find, unless the same entry also allows a jump in the opposite direction, in which case we terminate. If no minimal jump creates an unvisited permutation, we also terminate the algorithm prematurely. For example, consider $L_4 = \{1243, 1423, 4123, 4213, 2134\}$. Starting with $\pi_0 = 1243$, the algorithm generates $\pi_1 = 1423$ (obtained from $\pi_0$ by a left jump of 4 by 1 step), then $\pi_2 = 4123$, then $\pi_3 = 4213$ (in $\pi_2$, 4 cannot jump, as $\pi_0$ and $\pi_1$ have been visited before; 3 cannot jump either to create any permutation from $L_4$, so 2 jumps left by 1 step), then $\pi_4 = 2134$, successfully generating $L_4$. If instead we initialize with $\pi_0 = 4213$, then the algorithm generates $\pi_1 = 2134$, and then stops, as no further jump is possible. If we choose $\pi_0 = 1423$, then we may jump 4 to the left or right (by 1 step), but as the direction is ambiguous, the algorithm stops immediately. As mentioned before, the
algorithm may stop prematurely only either because no minimal jump leading to a new permutation from \( L_n \) is possible, or because the direction of jump is ambiguous in some step. By the definition of step J2, the algorithm will never visit any permutation twice.

The following main result of our paper provides a sufficient condition on the set \( L_n \) to guarantee that Algorithm J is successful. This condition is captured by the following closure property of the set \( L_n \). A set of permutations \( L_n \subseteq S_n \) is called a zigzag language, if either \( n = 0 \) and \( L_0 = \{ \varepsilon \} \), or if \( n \geq 1 \) and \( L_{n-1} := \{ p(\pi) \mid \pi \in L_n \} \) is a zigzag language satisfying the following condition:

(z1) For every \( \pi \in L_{n-1} \) we have \( c_1(\pi) \in L_n \) and \( c_n(\pi) \in L_n \).

**Theorem 1.** Given any zigzag language of permutations \( L_n \) and initial permutation \( \pi_0 = \text{id}_n \), Algorithm J visits every permutation from \( L_n \) exactly once.

**Remark 2.** The number of zigzag languages is at least \( 2(\log 2)^! (n-2) = 2^\Theta(n \log n) \), i.e., more than double-exponential in \( n \). We will see that many of these languages do in fact encode interesting combinatorial objects. Moreover, minimal jumps as performed by Algorithm J always translate to small changes on those objects in a provable sense, i.e., our algorithm defines Gray codes for a large variety of combinatorial objects, and Hamilton paths/cycles on the corresponding flip graphs and polytopes.

2.3. **The tree of permutations.** There is an intuitive characterization of zigzag languages via the tree of permutations. This is an infinite (unordered) rooted tree which has as nodes all permutations from \( S_n \) at distance \( n \) from the root; see the figure on the right. Specifically, the empty permutation \( \varepsilon \) is at the root, and the children of any node \( \pi \in S_{n-1} \) are exactly the permutations \( c_i(\pi), 1 \leq i \leq n \), i.e., the permutations obtained by inserting the new largest value \( n \) in all possible positions. Consequently, the parent of any node \( \pi' \in S_n \) is exactly the permutation \( p(\pi') \) obtained by removing the largest value \( n \).

In the figure, for any node \( \pi \in S_{n-1} \), the nodes representing the children \( c_1(\pi) \) and \( c_n(\pi) \) are drawn black, whereas the other children are drawn white. Any zigzag language of permutations can be obtained from this full tree by pruning subtrees, where by condition (z1) a subtree may be pruned only if its root \( \pi' \in S_n \) is neither the child \( c_1(\pi) \) nor the child \( c_n(\pi) \) of its parent \( \pi = p(\pi') \in S_{n-1} \), i.e., only subtrees rooted at white nodes may be pruned. For any subtree obtained by pruning according to this rule and for any \( n \geq 1 \), the remaining permutations of length \( n \) form a zigzag language \( L_n \); see Figure 1.2

Consider all nodes in the tree for which the entire path to the root consists only of black nodes. Those nodes never get pruned and are therefore contained in any zigzag language. These are exactly all permutations without peaks. A peak in a permutation \( a_1 \ldots a_n \) is a triple \( a_{i-1}a_ia_{i+1} \) with \( a_{i-1} < a_i > a_{i+1} \), and the language of permutations without peaks is generated by the recurrence

\[
P_0 := \{ \varepsilon \} \quad \text{and} \quad P_n := \{ c_1(\pi), c_n(\pi) \mid \pi \in P_{n-1} \} \quad \text{for} \quad n \geq 1.
\]

It follows that we have \( |P_n| = 2^{n-1} \) and \( P_n \subseteq L_n \subseteq S_n \) for any zigzag language \( L_n \), i.e., \( L_n \) is sandwiched between the language of permutations without peaks and between the language of all permutations.

2.4. **Proof idea of Theorem 1.** Given a zigzag language \( L_n \), we define a sequence \( J(L_n) \) of all permutations from \( L_n \) and prove that Algorithm J generates the permutations of \( L_n \) exactly in
The reason is that it requires storing the list of all previously visited permutations in order to
under which conditions the algorithm generates a
subsequences of the Steinhaus-Johnson-Trotter order
is quite different from the one presented by Vajnovszki and Vernay \[19\].

Further properties of Algorithm J.

Algorithm J thus defines a left-to-right ordering of the nodes at distance \(n\) of the root in the tree representation of the zigzag language \(L_n\) described before, and this ordering is captured by the sequence \(J(L_n)\); see Figure 1. Clearly, the same is true for all the zigzag languages \(L_0, L_1, \ldots, L_{n-1}\) that are induced by \(L_n\) through the rule \(L_{k-1} := \{p(\pi) \mid \pi \in L_k\}\) for \(k = n, n-1, \ldots, 1\). The unordered tree of permutations is thus turned into an ordered tree, and it is important to realize that pruning operations change the ordering. Specifically, given two zigzag languages \(L_n\) and \(M_n\) with \(M_n \subseteq L_n\), the tree for \(M_n\) is obtained from the tree for \(L_n\) by pruning, but in general \(J(M_n)\) is not a subsequence of \(J(L_n)\), as shown by the example in Figure 1. This shows that our approach is quite different from the one presented by Vajnovszki and Vernay \[19\], which considers only subsequences of the Steinhaus-Johnson-Trotter order \(J(S_n)\).

2.5. Further properties of Algorithm J. From our proof of Theorem 1 we can easily deduce under which conditions the algorithm generates a cyclic listing of permutations:

Lemma 3. In the ordering of permutations \(J(L_n)\) generated by Algorithm J, the first and last permutation are related by a minimal jump if and only if \(|L_k|\) is even for all \(2 \leq k \leq n - 1\).

It also follows from the proof of Theorem 1 that instead of initializing the algorithm with the identity permutation \(\pi_0 = \text{id}_n\), we may use any permutation without peaks as a seed \(\pi_0\).

2.6. Efficiency considerations. Let us make it very clear that in its stated form, Algorithm J is not an efficient algorithm to actually generate a particular zigzag language of permutations. The reason is that it requires storing the list of all previously visited permutations in order to
decide which one to generate next. However, by introducing a few additional arrays, the algorithm can be made memoryless, so that such lookup operations are not needed anymore, and hence no permutations need to be stored at all. The efficiency of the resulting algorithm is then only determined by the efficiency with which we are able to compute minimal jumps with respect to the input zigzag language $L_n$ for a given entry of the permutation. This leads to an algorithm that computes the next permutation to be visited in polynomial time. In many cases, this can be improved to a loopless algorithm that generates each new permutation in constant worst-case time. The key insight here is that any jump changes the inversion table of a permutation only in a single entry. By maintaining only the inversion table, jumps can thus be performed efficiently, even if the number of steps is big. This extensive and important discussion, however, is not the focus of this paper, and is hence deferred to future parts of the series [HHMW19, HM19].

2.7. A general recipe. Here is a step-by-step approach to apply our framework to the generation of a given family $X_n$ of combinatorial objects. The first step is to establish a bijection $f$ that encodes the objects from $X_n$ as permutations $L_n \subseteq S_n$. If $L_n$ is a zigzag language, which can be checked by verifying the closure property, then we may run Algorithm J with input $L_n$, and interpret the resulting ordering $J(L_n)$ in terms of the combinatorial objects, by applying $f^{-1}$ to each permutation in $J(L_n)$, yielding an ordering on $X_n$. We may also apply $f^{-1}$ to Algorithm J directly, which will yield a simple greedy algorithm for generating $X_n$. The final step is to make these algorithms efficient, by introducing additional data structures that allow the change operations on $X_n$ (which are the preimages of minimal jumps under $f$) as efficiently as possible.

Let us illustrate these steps for the set $X_n$ of binary strings of length $n - 1$. We map any binary string $x = x_2 \ldots x_n$ to a permutation $f(x) \in S_n$ by setting $f(\varepsilon) := 1$ and $f(x_2 \ldots x_n) := c_n(f(x_2 \ldots x_{n-1}))$ if $x_n = 0$, and $f(x_2 \ldots x_n) := c_1(f(x_2 \ldots x_{n-1}))$ if $x_n = 1$, i.e., we build the permutation $f(x)$ by inserting the values $i = 2, \ldots, n$ one by one, either at the leftmost or rightmost position, depending on the bit $x_i$. Observe that $f(X_n)$ is exactly the set of permutations without peaks $P_n \subseteq S_n$ discussed in Section 2.3 before, and a jump of the entry $i$ in the permutation translates to flipping the bit $x_i$. Moreover, $f^{-1}(J(P_n))$ is exactly the well-known reflected Gray code BRGC for binary strings of length $n - 1$ [Gra53], which can be implemented efficiently [BER76]. Applying $f^{-1}$ to Algorithm J yields the following simple greedy algorithm for generating the BRGC (see [Wil13]): J1. Visit the initial all-zero string. J2. Repeatedly flip the rightmost bit that yields a previously unvisited string.

3. PATTERN-AVOIDING PERMUTATIONS

The first main application of our framework is the generation of pattern-avoiding permutations. As mentioned before, many combinatorial objects are in bijection to pattern-avoiding permutations, and we will see that generating the permutations with Algorithm J yields Gray codes for those objects. Our main results in this section are summarized in Theorem 4, Remark 5, and Table 1.

3.1. Preliminaries. Given two permutations $\pi \in S_n$ and $\tau \in S_k$, we say that $\pi$ contains the pattern $\tau$, if and only if $\pi = a_1 \ldots a_n$ contains a subpermutation $a_{i_1} \ldots a_{i_k}$, $i_1 < \cdots < i_k$, whose elements appear in the same relative order as in $\tau$. If $\pi$ does not contain the pattern $\tau$, then we say that $\pi$ avoids $\tau$. For example, $\pi = 635412$ contains the pattern $\tau = 231$ at the highlighted positions. On the other hand, $\pi = 654123$ avoids $\tau = 231$. We refer to this notion of pattern-containment as
<table>
<thead>
<tr>
<th>Tame patterns</th>
<th>Combinatorial objects and ordering</th>
<th>References/OEIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>permutations by adjacent</td>
<td>[Joh63, Tro62], A000142</td>
</tr>
<tr>
<td></td>
<td>transpositions → plain change</td>
<td>order</td>
</tr>
<tr>
<td>231 = 231</td>
<td>Catalan families</td>
<td>A000108</td>
</tr>
<tr>
<td></td>
<td>• binary trees by rotations →</td>
<td>[LvBR93]</td>
</tr>
<tr>
<td></td>
<td>Lucas-van Baronaigien-Ruskey</td>
<td>order</td>
</tr>
<tr>
<td></td>
<td>• triangulations by edge flips</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Dyck paths by hill flips</td>
<td></td>
</tr>
<tr>
<td>231</td>
<td>Bell families</td>
<td>A000110</td>
</tr>
<tr>
<td></td>
<td>• set partitions by element</td>
<td>[Kay76, Wil13]</td>
</tr>
<tr>
<td></td>
<td>exchanges → Kaye’s order</td>
<td></td>
</tr>
<tr>
<td>132 ∧ 231 = 132 ∧ 231:</td>
<td>binary strings by bitflips</td>
<td>[Gra53], A011782</td>
</tr>
<tr>
<td>permutations without peaks</td>
<td>→ reflected Gray code order (BRGC)</td>
<td></td>
</tr>
<tr>
<td>2143:</td>
<td>vexillary permutations</td>
<td>[LS85], A005802</td>
</tr>
<tr>
<td></td>
<td>conjunction of v_k tame patterns</td>
<td>B14, A224318,</td>
</tr>
<tr>
<td></td>
<td>with v_2 = 35, v_3 = 91, v_4 = 2346</td>
<td>A223034, A223905</td>
</tr>
<tr>
<td>(see [Bil13]): k-vexillary permutations (k ≥ 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2143 ∧ 3412:</td>
<td>skew-merged permutations</td>
<td>Sta94, Atk98, A029759</td>
</tr>
<tr>
<td></td>
<td>separator permutations</td>
<td></td>
</tr>
<tr>
<td>2143 ∧ 3142:</td>
<td>Schnied families</td>
<td>A006318</td>
</tr>
<tr>
<td></td>
<td>• slicing floorplans</td>
<td>[AN81, BBL98, ABP06]</td>
</tr>
<tr>
<td></td>
<td>• topological drawings of K_{2,n}</td>
<td></td>
</tr>
<tr>
<td>2143 ∧ 3142:</td>
<td>Baxter</td>
<td>A091181</td>
</tr>
<tr>
<td></td>
<td>mosaic floorplans (=diagonal</td>
<td>YCCG03 [ABP06]</td>
</tr>
<tr>
<td></td>
<td>rectangulations=R-equivalent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rectangulations)</td>
<td>LR12 [CSS18]</td>
</tr>
<tr>
<td>2143 ∧ 3412:</td>
<td>S-equivalent rectangulations</td>
<td>A001181</td>
</tr>
<tr>
<td></td>
<td>separable permutations</td>
<td>[Rea12b]</td>
</tr>
<tr>
<td></td>
<td>S-equivalent guillotine</td>
<td>ABBM^{+}13, A214358</td>
</tr>
<tr>
<td></td>
<td>rectangulations</td>
<td></td>
</tr>
<tr>
<td>35124 ∧ 35142 ∧ 24513 ∧ 42513:</td>
<td>generic rectangulations (=rectangular drawings)</td>
<td>ABBM^{+}13, A078482</td>
</tr>
<tr>
<td>2-clumped permutations</td>
<td>conjunction of c_k tame patterns</td>
<td>[Rea12b]</td>
</tr>
<tr>
<td></td>
<td>with c_k = 2(k/2)!/(k/2 + 1)! for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>k even and c_k = 2((k + 1)/2)!²</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for k odd: k-clumped permutations</td>
<td></td>
</tr>
<tr>
<td>12 tame patterns: perm. with 0-1 Schubert polynomial</td>
<td>conjunction of 12 tame patterns: perm. with 0-1 Schubert polynomial</td>
<td>[FMSD19]</td>
</tr>
<tr>
<td>2143 ∧ 3412 ∧ 3142 ∧ 3412 ∧ 3142:</td>
<td>X-shaped permutations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Widdershins permut.</td>
<td>[Wat07, Eli11], A006012</td>
</tr>
</tbody>
</table>

...a classical pattern. This notion can be generalized in many ways, allowing us to encode even more families of combinatorial objects. Some of these more general types of patterns are listed in Remark 5 below. In this extended abstract we formally define only one additional type, namely vincular patterns, introduced by Babson and Steingrímsson [BS00]. In a vincular pattern τ, there is exactly one underlined pair of consecutive entries, with the interpretation that an occurrence of τ in π requires that the underlined entries appear at adjacent positions in π. For instance, the permutation π = 3 1 4 2 contains the pattern τ = 231, but it avoids the vincular pattern τ = 231.

For any permutation π, we let S_n(τ) denote all permutations from S_n avoiding the (classical or vincular) pattern τ. For propositional formulas F and G consisting of logical ANDs ∧, ORs ∨, and patterns as variables, we define S_n(F ∧ G) := S_n(F) ∩ S_n(G) and S_n(F ∨ G) := S_n(F) ∪ S_n(G). For
instance, $S_n(\tau_1 \land \cdots \land \tau_\ell)$ is the set of permutations avoiding each of the patterns $\tau_1, \ldots, \tau_\ell$, and $S_n(\tau_1 \lor \cdots \lor \tau_\ell)$ is the set of permutations avoiding at least one of the patterns $\tau_1, \ldots, \tau_\ell$.

3.2. **Tame patterns.** The following theorem describes some mild conditions on the patterns in the formula $F$ which ensure that we can generate $S_n(F)$ with Algorithm J.

**Theorem 4.** Let $F$ be a propositional formula consisting of logical ANDs $\land$, ORs $\lor$, and patterns that satisfy one of the following conditions:

(i) classical patterns that do not have the largest value at the leftmost or rightmost position,

(ii) vincular patterns that do not have the largest value at the leftmost of rightmost position, and the largest value is part of the vincular pair.

Then $S_n(F)$, $n \geq 0$, is a zigzag language of permutations, and hence can be generated by Algorithm J.

We say that a pattern satisfying (i) or (ii) is *tame*. Table 1 lists several tame classical and vincular patterns and the combinatorial objects encoded by the corresponding zigzag languages. The bijections between those permutations and the combinatorial objects are well-known and are described in the listed papers (recall also Section 2.7). Some patterns are interesting in their own right and have no ‘natural’ associated combinatorial objects (empty second column). The resulting ordering for twisted Baxter permutations of length $n = 4$, and the resulting Gray code for diagonal rectangulations, is shown in Figure 2. See also Figures 3 and 4 for the resulting Gray codes for three different Catalan objects (231-avoiding permutations) and for set partitions (231-avoiding permutations), respectively.

**Remark 5.** In addition to classical and vincular patterns, the literature knows a large number of additional types of permutation patterns. Algorithm J works successfully for almost all of them, i.e.,...
Figure 3. 231-avoiding permutations of length $n = 4$ generated by Algorithm J and resulting Gray codes for Catalan families (binary trees, triangulations, Dyck paths).

Figure 4. 231-avoiding permutations of length $n = 4$ generated by Algorithm J and resulting Gray code for set partitions.

the list of pattern types started in Theorem 4 can be extended considerably, by formulating suitable (still very mild) conditions for each of the following pattern types: (iii) bivincular patterns, introduced by Bousquet-Mélou, Claesson, Dukes, and Kitaev [BMCDK10]; (iv) barred patterns, introduced by West [Wes90]; (v) patterns with Bruhat restrictions, introduced by Woo and Yong [WY06]; (vi) mesh patterns, introduced by Brändén and Claesson [BC11]; (vii) partially ordered patterns, introduced by Kitaev [Kit05]; (viii) monotone grid classes, introduced by Huczynska and Vatter [HV06];

3.3. Patterns with multiplicities. All the aforementioned results generalize to bounding the number of appearances of any tame pattern. Formally, a counted pattern is a pair $\sigma = (\tau, c)$, where $\tau$ is a pattern of any of the types discussed before, and $c$ is a non-negative integer. Moreover, $S_n(\sigma)$ denotes the set of all permutations from $S_n$ that contain at most $c$ occurrences of the pattern $\tau$, where the special case $c = 0$ is pattern-avoidance. We can now form propositional formulas $F$ consisting of logical ANDs $\land$, ORs $\lor$, and counted patterns $(\tau_i, c_i)$ with tame $\tau_i$ as variables, yielding a zigzag language $S_n(F)$ that can be generated by Algorithm J. A somewhat contrived example for such a language would be $F = ((231,3) \land (2143,5)) \lor (3142,2)$, the language of permutations that contain at most $3$ occurrences of the pattern $231$ AND at most $5$ occurrences of the pattern $2143$, OR at most $2$ occurrences of the vincular pattern $3142$.

4. Lattice congruences of the weak order

The second main application of our framework are lattice congruences of the weak order on the symmetric group $S_n$. With lattice congruences, we obtain a provable notion of optimality for the Gray codes obtained from Algorithm J: Jump operations translate into walks along the edges of a polytope; see Figure 6. The main results in this section are summarized in Theorems 6 and Corollary 7.

4.1. Preliminaries. We begin recalling a few basic notions from poset theory. The weak order on the symmetric group $S_n$ is the poset obtained by considering the inversion set of a permutation, defined as $\text{inv}(\pi) := \{ (i,j) \mid 1 \leq i < j \leq n \text{ and } \pi(i) > \pi(j) \}$, and by defining $\pi < \rho$ if and only if $\text{inv}(\pi) \subseteq \text{inv}(\rho)$; see the left hand side of Figure 5. The cover relations in this poset, drawn as edges in the cover graph shown in the figure, are exactly adjacent transpositions between the two involved permutations. Moreover, the weak order is a lattice, i.e., for any two $\pi, \rho \in S_n$ there is a unique smallest element $\pi$, called the join $\pi \lor \rho$ of $\pi$ and $\rho$, such that $\pi > \pi$ and $\pi > \rho$, and there is unique largest element $\rho$, called the meet $\pi \land \rho$ of $\pi$ and $\rho$, satisfying $\pi < \pi$ and $\pi < \rho$. A lattice congruence of the weak order is an equivalence relation $\equiv$ on $S_n$ that is compatible with taking joins and meets, i.e., $\pi \equiv \pi' \land \rho \equiv \rho'$ implies that $\pi \lor \rho \equiv \pi' \lor \rho'$ and $\pi \land \rho \equiv \pi' \land \rho'$. Given any lattice congruence $\equiv$, we obtain the lattice quotient $S_n/\equiv$ by taking the equivalence classes as elements, and ordering them by $X < Y$ if and only if there is a $\pi \in X$ and a $\rho \in Y$ such that $\pi < \rho$ in $S_n$. See Figure 5 for an example of a lattice congruence (left) and the resulting lattice quotient (right).

It turns out that there are double-exponentially many distinct lattice congruences of the weak order on $S_n$, and they generalize many known lattices, such as the Boolean lattice, the Tamari lattice [Tam02] (shown on the right hand side of Figure 5), and certain Cambrian lattices [Rea06, CP17]. This area of study has beautiful ramifications into groups, posets, polytopes, geometry, and combinatorics, and has been developed considerably in recent years, in particular thanks to Nathan Reading’s works, summarized in [Rea12a, Rea16a, Rea16b].

4.2. Jumping through lattice congruences. For any lattice congruence $\equiv$ of the weak order on $S_n$, a set of representatives for the equivalence classes $S_n/\equiv$ is a subset $R_n \subseteq S_n$ such that for every equivalence class $X \subseteq S_n/\equiv$, exactly one permutation is contained in $R_n$, i.e., $|X \cap R_n| = 1$. We let $X(\pi), \pi \in S_n$, denote the equivalence class from $S_n/\equiv$ containing $\pi$. A meaningful definition of ‘generating the lattice congruence’ is to generate a set of representatives for its equivalence
classes. We also require that any two successive representatives form a cover relation in the lattice quotient $S_n/\equiv$. This is what we achieve with the help of Algorithm J.

**Theorem 6.** For every lattice congruence $\equiv$ of the weak order on $S_n$, there is a set of representatives $R_n \subseteq S_n$, such that Algorithm J generates a sequence $J(R_n) = \pi_1, \pi_2, \ldots$ of all permutations from $R_n$ for which the equivalence classes $X(\pi_1), X(\pi_2), \ldots$ form a Hamilton path in the cover graph of the lattice quotient $S_n/\equiv$.

For every lattice congruence $\equiv$, Pilaud and Santos [PS19] defined a polytope, called the *quotientope* for $\equiv$, whose skeleton is exactly the cover graph of the lattice quotient $S_n/\equiv$. These polytopes generalize many known polytopes, such as hypercubes, associahedra, permutahedra etc. The following result is an immediate corollary of Theorem 6 and it is illustrated in Figure 6.

**Corollary 7.** For every lattice congruence $\equiv$ of the weak order on $S_n$, Algorithm J generates a Hamilton path on the skeleton of the corresponding quotientope.

5. **Acknowledgments**

We thank Michael Albert, Mathilde Bouvel, Sergi Elizalde, Vít Jelínek, Sergey Kitaev, Vincent Vajnovszki, and Vincent Vatter for very insightful feedback on this work, for pointing out relevant references, and for sharing their knowledge about pattern-avoiding permutations. We also thank Jean Cardinal and Vincent Pilaud for several stimulating discussions about lattice congruences of the weak order on the symmetric group. Figure 6 in this paper was obtained by modifying and augmenting Figure 9 from [PS19], and the original source code for this figure was provided to us by Vincent Pilaud.
Figure 6. Lattice congruences of the weak order on $S_4$, ordered by refinement and realized as polytopes (only full-dimensional polytopes shown). Hamilton paths generated by Algorithm J are highlighted, with start vertex (triangle) and end vertex (diamond). Permutahedron, associahedron (one of four isomorphic variants) and hypercube are framed.
REFERENCES


