Composable Constraint Models for Permutation Enumeration

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

Permutations, which describe how sets of objects can be ordered, are a fundamental concept in mathematics and computing with numerous theoretical and real-world applications. One key area of research into permutations studies the permutations with one or more properties known as patterns. Enumerating the permutations that contain a list of patterns is a computationally challenging task with a long history of theoretical and practical research. Existing approaches focus on handling a single pattern type at a time. This means finding permutations which contain multiple patterns, or contain a pattern while satisfying some other property, requires either a generate-and-test approach or the development of bespoke algorithms.

In this paper, we show how to declaratively define permutation properties, permutation pattern avoidance and containment constraints and solve the resulting problems using constraint programming. This approach enables the arbitrary composition of these conditions. We demonstrate the effectiveness of our techniques by modelling the containment and avoidance of six permutation patterns and seven additional permutation properties. This approach allows mathematicians to quickly and efficiently investigate permutation pattern problems.

2012 ACM Subject Classification Mathematics of computing → Permutations and combinations; Mathematics of computing → Combinatoric problems; Theory of computation → Constraint and logic programming

Keywords and phrases Constraint Modelling, Permutation Pattern, Enumeration

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

The concept of permutation pattern classes emerged from an exercise question in Knuth's Art of Computer Programming Volume 1 (section 2.2.1 exercise 5) [15], in which Knuth explored which permutations can be sorted in a limited stack-based system, called stack-sortable permutations. Simion and Schmidt [20] were among the pioneers in characterizing and enumerating sets of permutations based on the patterns that they avoid. Further enumeration results garnered interest because the number of the original set of permutations, identified in Knuth's exercise, proved to be the Catalan numbers.

The enumeration results and subsequent research were primarily motivated by a conjecture proposed by Herbert Wilf at the 1992 SIAM meeting. This conjecture stated that every permutation pattern class that avoids one permutation pattern exhibits an exponential growth rate. Marcus and Tardos later confirmed and proved this conjecture in [17].

There are several systems which use computational techniques to assist with the enumeration of permutation classes and sets, some examples include Permuta[4], PatternClass[3], Combinatorial Specification Searcher[21] and PermCode (previously PermLab) [2].

Permuta is a Python package which supports the enumeration of permutation classes using the BiSC [16] algorithm. PatternClass is a GAP[13] package that encodes the permutations

Conditions		Statistics
Pattern Avoidance/Containment	Properties	Number of
Classic	Simple	Inversions
Vincular	Plus-Decomposable	Ascents
Bivincular	Minus-Decomposable	Descents
Mesh	Blockwise Simple	Excedances
Boxed Mesh	Derangement	Major Index
Consecutive	Non-Derangement	
	Involution	

Table 1 Table of all permutation patterns, properties and statistics modelled

and classes into regular languages, enabling efficient investigation of permutation sets. Combinatorial Specification Searcher is another Python package that provides a more exploratory approach by asking the user to define strategies on how to build combinatorial sets from other sets. It combines these into a general enumeration algorithm. PermCode is a Java library with a GUI which allows for the exploration of the sets of permutations which avoid classical patterns.

In this paper, we apply constraint programming to the study of permutations and permutation patterns. A significant benefit of using constraints is that it allows us to build a general framework where new patterns and properties can be easily added, and patterns and properties can be combined in arbitrary ways. We demonstrate the effectiveness of our techniques by modelling the containment and avoidance of six permutation patterns and seven permutation properties. Our systems can also answer a range of different questions, from checking for existence or enumeration to more complicated statistical properties of permutations.

Section 2 defines permutations and permutation patterns as well as describes the different permutation properties and statistics we have implemented. These properties and statistics are summarised in Table 1. Section 3 shows how the properties and statistics are modelled in the Essence [12] constraint language. Section 4 shows an extended example of how using constraints allows us to study combinations of properties easily and efficiently and Section 5 summarises our contributions and discusses future work.

All of the code, models, raw data and outputs of our computational experiments can be found on an online appendix: `https://anonymous.4open.science/r/permutation-classes-cp-F054/` (anonymised for blind review).

2 Background

We will consider a *permutation* σ to be an arrangement of the set $\{1, 2, \dots, n\}$ for some $n \in \mathbb{N}$. The size of the set a permutation σ is defined on is called the length of the permutation and is denoted $|\sigma|$. S_n is used for the set of all permutations of length n. σ can be viewed as a bijective function, where $\sigma(i)$ denotes the ith member of the permutation and $\sigma^{-1}(j)$ gives the index where j occurs in the permutation. Two sequences $\pi = \pi(1), \dots, \pi(n)$ and $\sigma = \sigma(1), \dots, \sigma(n)$ of the same length are said to be *order isomorphic* [5] if $\forall i, j, \pi(i) \leq \pi(j)$ if and only if $\sigma(i) \leq \sigma(j)$.

We will represent permutations by giving the arrangement of the set (often called sequence notation) and as a permutation plot. A permutation plot represents a permutation as a grid

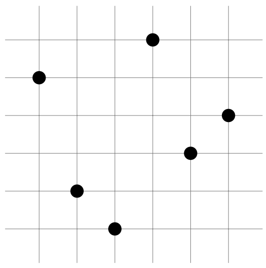


Figure 1 Plot representation of the permutation 521634. The x-axis represents the indexes of the permutation, and the y-axis represents the values. Reading left to right, the location of the dot in the ith column represents the ith value in the permutation.

where the x-axis represents the indexes of the permutation, the y-axis represents the values and the values of the permutation are denoted by nodes. Figure 1 gives an example of a permutation plot.

The rest of this Section defines the patterns, properties and statistics which we have currently implemented. These terms are summarised in Table 1.

2.1 Pattern types

A permutation pattern identifies a subsequence of a permutation which satisfies a list of properties. If a subsequence of a permutation satisfies the requirements of a given permutation pattern, it is said to *involve* or *contain* the pattern. If no part of a permutation matches a permutation pattern then it *avoids* the pattern. There are many types of permutation pattern, in this Section we introduce the ones considered in this paper.

We say that a permutation $\pi = \pi(1) \dots \pi(k)$ is *classically contained* in a permutation $\sigma = \sigma(1) \dots \sigma(m)$, where $k \leq m$, if there is a subsequence $\sigma(i_1) \dots \sigma(i_k)$ in σ that is order isomorphic to π. For example, the permutation $\pi = 123$ can be found in $\sigma = 521634$ as the order isomorphic subsequence 134 at indices $[3, 5, 6]$. Being *classically contained* is a partial order on the set of permutations [11]. In Figure 2a on the left is the pattern permutation, on the right we have highlighted an occurrence of the classical pattern by circling the elements. The other patterns we discuss all extend classical patterns by adding further conditions.

Vincular patterns (introduced by Babson and Steingrímsson [7]) specify adjacency conditions. Let $\pi = \pi(1) \dots \pi(k)$, $\sigma = \sigma(1) \dots \sigma(m)$ and let $A \subseteq \{1, \dots, k-1\}$. An occurrence of the vincular pattern (π, A) in σ is a subsequence $\sigma(i_1) \dots \sigma(i_k)$ of σ such that $\sigma(i_1) \dots \sigma(i_k)$ is an occurrence of π in the classical sense, and $\forall a \in A.\ i_{a+1} = i_a + 1$. We call A the set of *adjacencies*.

For example, the vincular pattern $(132, \{1\})$ can be found in $\sigma = 521634$ as the subsequence 164. Figure 2b shows an example of a vincular pattern by showing the pattern with the order isomorphic subsequence and highlights the adjacency requirement by shading the column between the indices both in the pattern and the permutation which contains the pattern.

Bivincular patterns (as introduced in [9]) are vincular patterns which additionally require some values to be adjacent. More formally, an occurrence of the bivincular pattern (π, A) in σ is a subsequence $\sigma(i_1) \dots \sigma(i_k)$ of σ such that the following all hold:

- $\sigma(i_1) \dots \sigma(i_k)$ is an occurrence of π in the classical sense,
- $\forall a \in A.\ i_{a+1} = i_a + 1$
- $\forall a \in A.\ \sigma(i_{a+1}) = \sigma(i_a) + 1$

As with vincular patterns, A is the set of adjacencies. For example, the bivincular pattern $(312, \{2\})$ can be found in $\sigma = 521634$ as the subsequence 534, Figure 2c illustrates this bivincular containment with the pattern permutation and the shading of the adjacency in values (rows) and indices (columns) highlighted.

All of the above patterns can be generalised as *mesh patterns*, which were introduced in [10]. A mesh pattern of length k is a pair (π, R) with $\pi \in S_k$ and $R \subseteq [0, k] \times [0, k]$, a set of pairs of integers. The elements of R identify the lower left corners of unit squares in the plot of π and specify forbidden regions. An occurrence of a mesh pattern (π, R) in a permutation σ is a subsequence $\sigma(i_1) \dots \sigma(i_k)$ such that the following holds

- $\sigma(i_1) \dots \sigma(i_k)$ is order isomorphic to π
- $(x, y) \in R \Rightarrow$ there does not exist $i \in \{1, \dots, n\} : i_x < i < i_{x+1} \wedge \sigma(i_{\pi^{-1}(y)}) < \sigma(i) < \sigma(i_{\pi^{-1}(y+1)})$.

For just this definition we extend the subsequence i_j, σ and π with $i_0 = 0$, $i_{k+1} = n + 1$, $\pi(0) = 0$, $\pi(k+1) = k+1$, $\sigma(0) = 0$ and $\sigma(n+1) = n+1$. The extra terms for i_j, σ and π in the second part of the definition above allow mesh patterns to constrain a permutation outside of the pattern. An example of a mesh pattern in $\sigma = 521634$ is $(132, (0, 0), (2, 1), (2, 2))$, which can be found as the subsequence 263. This example is illustrated in Figure 2d, where each of the forbidden regions/squares is shaded.

A *boxed mesh pattern* (or *boxed pattern*, as introduced in [6]), is a special case of a mesh pattern $P = (\pi, R)$ where π is a permutation of length k and $R = [1, k-1] \times [1, k-1]$. P is then denoted by $\boxed{\pi}$. For example the boxed pattern $\boxed{231}$ is contained in the permutation 236514 as the subsequence 351. Figure 2e shows this example, and illustrates that for a boxed mesh pattern the box inside the permutation is the forbidden region.

Consecutive patterns are a special case of vincular patterns, where it is necessary that *all* entries are adjacent. For example, the consecutive pattern $(312, \{1, 2\})$ can be found inside 152463 as the subsequence 524. This example is shown in Figure 2f.

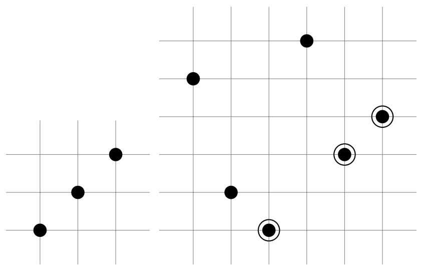
We say a permutation σ *avoids* any of the above pattern types, if the permutation π of the pattern is classically avoided in σ, or it is classically contained in σ but for every occurrence of the classical pattern the additional constraints on indices or values are not upheld.

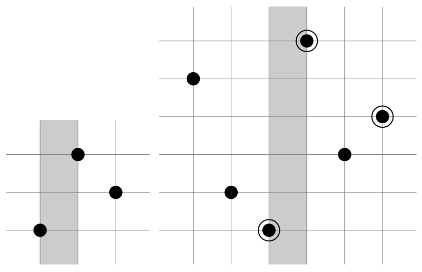
So for example the permutation 12345 avoids the classic pattern 21, as there are no occurrences of it. Figure 3a attempts to illustrate this avoidance. Similarly, the permutation 12345 avoids the mesh pattern $(132, (0, 0), (2, 0), (2, 1), (2, 2))$ as there is no classical occurrence of 132 in 12345, as can be seen from Figure 3b. Finally, $(132, (0, 0), (2, 0), (2, 1), (2, 2))$ is not contained in the permutation 652413 even though there is an occurrence of 132 in the permutation, namely as 243, as there is an element of the permutation that violates the forbidden region. In Figure 3c we can see that the element 1 is in the $(2, 0)$ shaded/forbidden region.

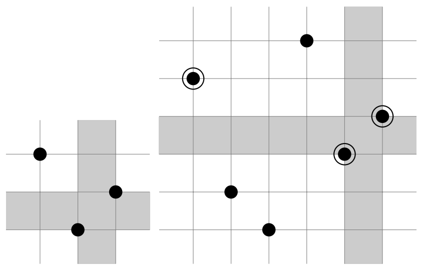
2.2 Permutation Properties

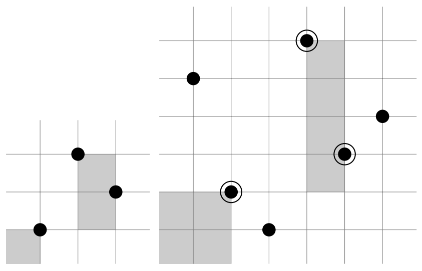
In addition to containing (or avoiding) patterns in permutations and enumerating these permutations, researchers are also interested in permutations with a range of different properties.

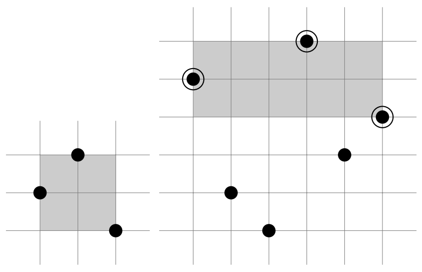
An *interval* of a permutation π corresponds to a set of contiguous indices $I = [a, b]$ such that the set of values $\pi(I) = \{\pi(i) : i \in I\}$ is also contiguous. Every permutation of length n has intervals of lengths 0, 1 and n. If a permutation π has no other intervals, then π is said to be *simple* [11]. For example, $\pi = 1632547$ is not simple as it contains the intervals $\pi(3)\pi(4)$, $\pi(5)\pi(6)$, $\pi(3)\pi(4)\pi(5)\pi(6)$, $\pi(2)\pi(3)\pi(4)\pi(5)\pi(6)$, $\pi(2)\pi(3)\pi(4)\pi(5)\pi(6)\pi(7)$ and


(a) Classic Pattern


(b) Vincular Pattern


(c) Bivincular Pattern


(d) Mesh Pattern


(e) Boxed Mesh Pattern

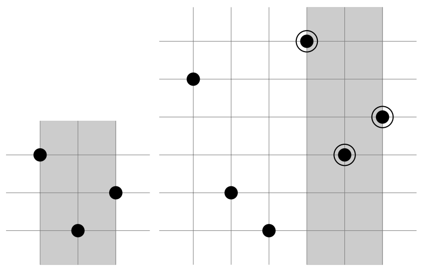
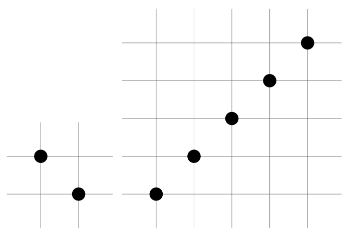
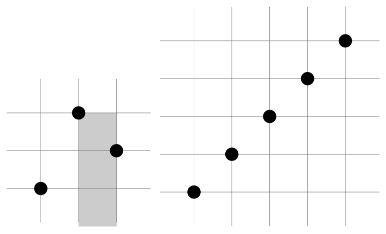

(f) Consecutive Pattern

Figure 2 Examples of all patterns and for each an example containment in a larger permutation. In each subfigure the pattern is on the left and an occurrence of the pattern is indicated in the right permutation. We highlight a classic occurrence (the elements that are order isomorphic to the pattern permutation) with circles around the nodes. All additional constraints (in terms of adjacency in index or values, or the forbidden cells/regions) are shaded. In the target permutation the shaded regions scale with respect to the elements which are representative of the classic pattern.


(a) 12345 avoids the classic pattern 21.


(b) 652413 avoids the mesh pattern $(132,(2,0),(2,1),(2,2))$.

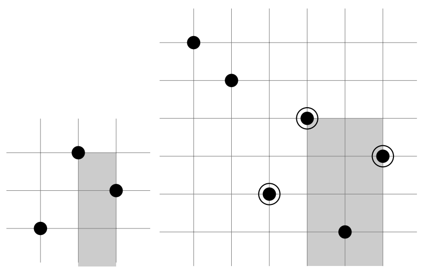

(c) 652413 avoids the mesh pattern $(132,(2,0),(2,1),(2,2))$.

Figure 3 Examples of pattern avoidance. In subfigures (a) and (b) there is no subsequence in the target permutation which is order isomorphic to the pattern permutation (i.e. there is no occurrence of the classic pattern in the target permutation). Subfigure (c) avoids the mesh pattern even though the pattern permutation is contained classically, in each occurrence (there is only one) there are (at least one) elements in the target permutation which lie in the shaded/forbidden region.

$\pi(1)\pi(2)\pi(3)\pi(4)\pi(5)\pi(6)$, as indicated in Figure 4a. The permutation 246135 in Figure 4e is simple, as the only intervals it contains are of length 0, 1 or the whole permutation.

Given a permutation $\sigma = \sigma(1)\ldots\sigma(m)$ of length m and non-empty permutations $\alpha_1,\ldots,\alpha_m$ the *inflation* of σ by $\alpha_1,\ldots,\alpha_m$, written as $\sigma[\alpha_1,\ldots,\alpha_m]$, is the unique permutation obtained by replacing each entry $\sigma(i)$ by an interval that is order isomorphic to α_i, where the relative ordering of the intervals corresponds to the ordering of the entries of σ. Conversely, a *block-decomposition* or *deflation* of a permutation π is any expression of π written as an inflation $\pi = \sigma[\alpha_1,\ldots,\alpha_m]$. For example 521634 can be decomposed as $3142[1,21,1,12]$. In Figure 4b we have highlighted the blocks of the permutation 521634, we can observe that the blocks are placed order isomorphic to 3142.

A permutation π is *plus-decomposable* if it has a block-decomposition of the form $\pi = 12[\alpha_1,\alpha_2]$ for non-empty permutations α_1 and α_2. For example 213654 is a plus-decomposable permutation, but 546123 or 521634 (which is also not simple) are not. Figure 4c shows the blocks of 213654, plus-decomposable permutations will always be in this layout (elements in the bottom left quadrant, and elements in the top right quadrant).

A permutation π is *minus-decomposable* if it has a block-decomposition of the form $\pi = 21[\alpha_1,\alpha_2]$ for non-empty permutations α_1 and α_2. For example 546123 is a minus-decomposable permutation. But 213654 (which is plus-decomposable) or 236145 (which is not simple) are not. Figure 4d contains the minus-decomposable permutation 546123. All minus-decomposable permutations will have a similar layout, with elements in the top left quadrant, followed by elements in the bottom right quadrant.

A permutation $\pi \in S_n$ is *block-wise simple* if and only if it has no interval which can be decomposed into $12[\alpha_1,\alpha_2]$ or $21[\alpha_1,\alpha_2]$. This property was introduced in [8] alongside an equivalent recursive recursive definition. For example $2413[3142,1,1,1] = 4253716$ is block-wise simple (Figure 4f), but 24513 is not as the interval 45 can be decomposed into

12[1, 1] (Figure 4g).

A fixed point of a permutation π is an integer i such that $\pi(i) = i$. A *derangement* is a permutation with no fixed points. 4312 is a derangement whereas 1234 is not. As shown in Figure 4i none of the elements of the permutation 4312 are on the red diagonal (which represents $\pi(i) = i$).

Similarly, a *nonderangement* is a permutation with at least one fixed point. 2431 is a non-derangement whereas 4321 is not. The plot of 2431 in Figure 4j shows $\pi(3) = 3$ lies on the diagonal.

A permutation $\pi \in S_n$ is called an *involution* if $\pi = \pi^{-1}$, or equivalently if $\forall i, j.\pi(i) = j \iff \pi(j) = i$. 1243 is an involution but 2431 is not. Figure 4h contains the plot of the involution 1243.

2.3 Permutation Statistics

Currently we support 5 different permutation statistics, where we count the occurrences of some property of the elements of the permutation. The properties we currently consider are ascents, descents, inversions and excedances.

An *ascent* in a permutation σ is an index i such that $\sigma(i) < \sigma(i+1)$. Similarly, a *descent* is an i such that $\sigma(i) > \sigma(i+1)$. A pair of indices (i, j) in a permutation σ such that $i < j$ and $\sigma(i) > \sigma(j)$ is called an *inversion*. An *excedance* is an index where $\sigma(i) > i$. So for a permutation σ the statistics are

- the number of *inversion* $\text{inv}(\sigma) = |\{(i, j) : i < j \text{ and } \sigma(i) > \sigma(j)\}|$
- the number of *descents* $\text{des}(\sigma) = |\{i : \sigma(i) > \sigma(i+1)\}|$
- the number of *ascents* $\text{asc}(\sigma) = |\{i : \sigma(i) < \sigma(i+1)\}|$
- the number of *excedances* $\text{exc}(\sigma) = |\{i : \sigma(i) > i\}|$.

Finally, we have implemented the *Major index* which is the sum of the positions of the descents $\text{maj}(\sigma) = \sum_{\sigma(i)>\sigma(i+1)} i$.

As an example, in the permutation $\sigma = 7164523$, $\text{inv}(\sigma) = 14$, $\text{des}(\sigma) = 3$, $\text{asc}(\sigma) = 3$, $\text{exc}(\sigma) = 2$ and $\text{maj}(\sigma) = 9$.

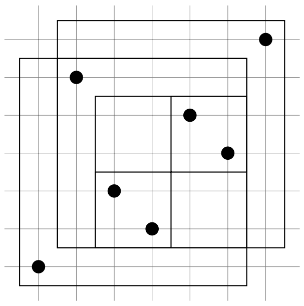
3 Constraint Programming models

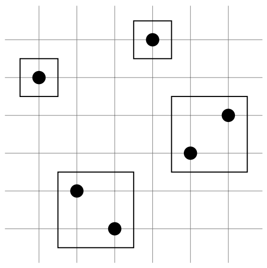
We use the Essence [12] problem specification language to define constraint programming models for each pattern, condition and statistic listed in Table 1.

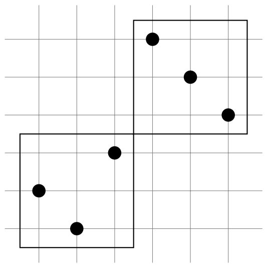
All of these problem specifications are converted to a solver independent constraint programming model using Conjure [1] and subsequently converted to input for the constraint solver Minion using Savile Row [18]. Savile Row supports several additional solver backends, including other CP solvers, SAT solvers, SMT solvers and MIP solvers. We use Minion [14] in the experiments in this paper, but changing to the other available solvers just requires changing a command line argument.

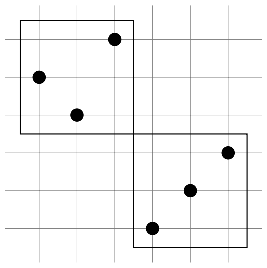
All the models share a common set of parameter and decision variable definitions. Through these definitions they can compose arbitrarily. Listed in Figure 5, the common set of definitions are the length of the permutation we are searching for and a single decision variable representing the permutation itself. The permutation variable has a sequence domain that has a fixed length and contains the integers from 1 to *length*. Using the injective attribute we limit assignments to the sequence to valid permutations.

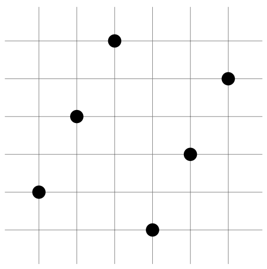
Next, we show a demonstrating example of the two types of conditions: one property (Simple) and one pattern avoidance (Mesh).

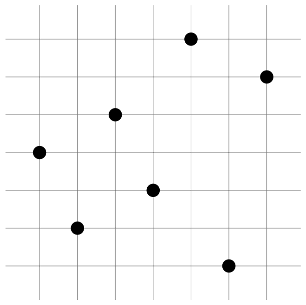

(a) Intervals in a permutation

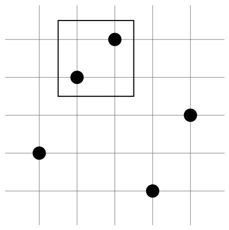

(b) Block-decomposition of a permutation

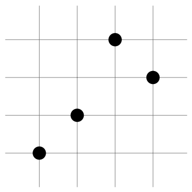

(c) Plus-decomposable permutation

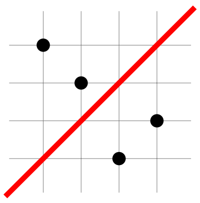

(d) Minus-decomposable permutation


(e) Simple permutation


(f) Block-wise simple permutation


(g) Not block-wise simple permutation


(h) Involution


(i) Derangement

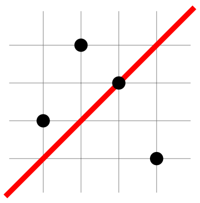

(j) Non-derangement

Figure 4 Examples permutation properties. Intervals (of length 1 and up to n-1, where n is the length of the permutation) and blocks are indicated using squares around the nodes that contain them. The red diagonal line for the derangement/non-derangement indicates the $\pi(i) = i$ property.

```
language Essence 1.3

$ The permutation we are searching for (1..length is the permutation)
given length : int
find perm : sequence (size length, injective) of int(1..length)
```

Figure 5 Common definitions shared across the different models

```
$ Simple permutations do not contain intervals.
$ An interval is a set of contiguous indices where the values are also contiguous.

such that
  [ max(subperm) - min(subperm) + 1 != |subperm| $ the values are not contiguous
  | i : int(1..length)                          $ start index of the sub perm
  , j : int(1..length)                          $ end index of the sub perm
  , i < j                                       $ start comes before end
  , (i,j) != (1,length)                         $ it's not the full permutation
  , letting subperm be [perm(k) | k : int(i..j)] $ give the sub perm a name
  ]
```

Figure 6 Permutation *perm* is simple

3.1 Permutation property: Simple

We use the *Simple* property defined in Section 2.2 as a representative of how permutation properties are handled. In Figure 6 we give the Essence problem specification in terms of *length* and *perm*.

This condition is written in Essence using a single constraint expression. For every ordered pair of indices (i, j), with $i < j$, in the original permutation (except the full permutation), we define a sub-permutation using list comprehension. We then post a constraint enforcing that the values in the sub-permutation are not contiguous.

3.2 Pattern type: Mesh avoidance

We use the *Mesh avoidance* property defined in Section 2.1 as a representative of how pattern types are handled. In Figure 7 we give the Essence problem specification in terms of *length* and *perm*.

In addition to the common definitions, this model takes *mesh_avoidance* as an additional parameter. This parameter contains data required for a set of mesh avoidance conditions. The model supports an arbitrary number of mesh avoidance conditions at once. Each mesh avoidance condition requires and underlying pattern, represented using an injective sequence and a binary relation that captures the set of cells that are shaded.

This model also defines a local matrix decision variable, *permPadded*, which is a functionally defined variable in terms of the primary decision variable *perm*, extending the variable with a value for `0` and `length+1`. These extra variables and constants will be removed before the model is executed and are only introduced to simplify modelling the mesh constraint.

The mesh constraint is quite complex, much of this complexity comes from the complication of the original definition. Users who only want to use mesh avoidance patterns are of course not required to understand this model. For efficency, these constraints make use of many language features of Essence, most notably quantified expressions over complex domains.

```
 1 $ This of mesh patterns to avoid
 2 given mesh_avoidance : set of (sequence(injective) of int, relation of (int * int))
 3
 4 $ creating a padded version of perm, where position 0 contains the value 0 and
       position length+1 contains the value length+1
 5 $ this is only used for mesh avoidance/containment
 6 find permPadded : matrix indexed by [int(0..length+1)] of int(0..length+1)
 7 such that permPadded[0] = 0, permPadded[length+1] = length+1
 8 such that forAll i : int(1..length) . perm(i) = permPadded[i]
 9
10 such that
11   forAll (pattern, mesh) in mesh_avoidance .
12   $ Build the inverse of 'pattern'. This is completely evaluated before solving.
13   exists patterninv: matrix indexed by [int(0..|pattern|+1)] of int(0..|pattern|+1)
14         , patterninv[0] = 0 /\ patterninv[|pattern|+1] = |pattern|+1 /\
15         (forAll i: int(1..|pattern|) . patterninv[pattern(i)] = i) .
16     $ Look for all places where the pattern can occur
17     forAll ix : matrix indexed by [int(0..|pattern|+1)] of int(0..length+1),
18       and([ ix[0]=0
19         , ix[|pattern|+1]=length+1
20         , forAll i : int(0..|pattern|) . ix[i] < ix[i+1]
21         , forAll n1, n2 : int(1..|pattern|) , n1 < n2 .
22           pattern(n1) < pattern(n2) <-> permPadded[ix[n1]] < permPadded[ix[n2]]
23         ]) .
24       (
25         $ If we find the pattern, make sure there is at least one value in some
               cell of the mesh
26         exists (i,j) in mesh.
27           exists z: int(ix[i]+1..ix[i+1]-1). (permPadded[ix[patterninv[j]]] <=
                 permPadded[z] /\ permPadded[z] <= permPadded[ix[patterninv[j+1]]])
28       )
```

Figure 7 Mesh pattern avoidance model

Lines 13–15 defines an existential quantification over a matrix domain, whose value is assigned to be the inverse of the pattern permutation. Many parts of this model, in particular the definition of *patterninv*, which represents the inverse of the pattern, are completely evaluated before solving.

Lines 16–23 use a universally quantified matrix of potential index values (*ix*), representing all locations this pattern might occur at. Finally, on lines 25–28, we post the mesh avoidance condition: there has to be at least one value in a shaded cell of the mesh.

Model fragments for all the patterns, properties and statistics we discuss in Section 2 are provided in an online appendix: `https://anonymous.4open.science/r/permutation-classes-cp-F054/`.

4 Composability in Action

We demonstrate the composability of the models using three illustrative (albeit hypothetical) example scenarios. Suppose we are a permutation pattern researcher seeking insights into the set of permutations that classically avoid the permutation 1324. It is worth noting that the growth function for this set of permutations has yet to be discovered; see OEIS [19, A061552]. A mathematician working on this problem will adopt some assumptions, enumerate permutations under these assumptions and refine these assumptions by studying the enumeration. Given the rapid growth of the numbers, we examine prevalent patterns and properties within our permutations, either contained or avoided, and aim to filter them out.

4.1 Scenario 1: no such permutations

We are aiming to enumerate the permutations that avoid the classical pattern 1324. Simultaneously we have decided to determine the number of permutations containing the classic pattern 21354. In order to solve this problem we instantiate our constraint programming model to classically avoid 1324 and contain 21354. We find no solutions to this problem.

Upon further examination, it becomes apparent that there are no permutations of length five or greater that satisfy both of these conditions. The reason for this impossibility is that the pattern 21354 inherently contains the pattern 1324 within it. As a result, it is not possible for a permutation to both avoid the presence of 1324 and include it at the same time.

4.2 Scenario 2: avoiding a large enumeration

Subsequently, we have reason to believe that a specific set of permutations exists, which not only manage to avoid the pattern 1324, but also include the permutation 6721345 classically. We test this hypothesis for length 9 permutations, of which there are 362,880. Out of these permutations, 94,776 avoid the 1324 pattern. Amongst these, a considerably smaller number, 824, contain 6721345 pattern as well. When we instantiate our constraint programming model to avoid 1324 classically and contain 6721345 we are able to enumerate the 824 permutations without enumerating 94,776 permutations first and then filtering them.

4.3 Scenario 3: combining six properties

In the context of exploring permutations that traditionally avoid the permutation 1324, we have finished exploring permutations that contain the following mesh pattern:
$(213, (0, 0), (0, 1), (1, 0), (1, 0))$, so from now on we only want to focus on permutations which avoid it. Consider that we want to also focus on permutations that contain the classic pattern (132) and the mesh pattern $(123, (1, 2), (2, 1), (1, 3), (3, 1))$.

We instantiate a constraint programming model that combines the four patterns above. We add two additional permutation properties as well: minus-decomposable and involution.

Existing approaches allow a limited form of compositionality: they support the enumeration of permutations that avoid a set of (mesh) patterns simultaneously, as every pattern type can be turned into a mesh pattern. So the traditional approach would be to enumerate the set of permutations which avoid the two patterns 1324 and $(213, (0, 0), (0, 1), (1, 0), (1, 0))$. We would then need to iterate over this enumeration and filter permutations to identify those containing the patterns 132 and $(123, (1, 2), (2, 1), (1, 3), (3, 1))$.

As an example, for length 9 there are $9! = 362880$ permutations of which, 4862 avoid the two patterns. 2841 of these permutations then contain the other two patterns. 1865 of these are minus-decomposable and finally only 19 are also involutions. Using a composed constraint programming model for this problem we are able to directly enumerate the 19 permutations of interest without going through several levels of generate-and-test. We conduct the same experiment on permutation lengths ranging from 5 to 16. Table 2 shows the number of permutations enumerated at each step as well as the number of search nodes Minion requires to perform this solution enumeration. In addition to allowing a mathematician to focus on a

Length	Number of permutations found				Number of search nodes required by Minion			
	Step 1	Step 2	Step 3	Step 4	Step 1	Step 2	Step 3	Step 4
5	42	8	2	0	128	65	5	0
6	132	41	19	1	487	318	159	1
7	429	180	102	2	1,916	1,434	774	3
8	1,430	730	455	9	7,749	6,365	3,515	53
9	4,862	2,841	1,865	19	32,058	27,986	15,593	130
10	16,796	10,815	7,321	53	135,137	122,910	68,596	321
11	58,786	40,700	28,096	106	578,677	541,297	301,301	672
12	208,012	152,325	106,555	255	2,511,199	2,395,126	1,326,807	1,430
13	742,900	568,883	401,729	493	11,022,455	10,657,385	5,868,862	2,906
14	-	-	-	1,118	-	-	-	6,057
15	-	-	-	2,120	-	-	-	12,119
16	-	-	-	4,664	-	-	-	25,005

Table 2 Running the 4 steps given in scenario 3 on various permutation lengths. Dash indicates timeout after 1-hour of Minion search

specific subset of permutations, using a combined constraint programming model significantly reduces the required number of search nodes. This further demonstrates that this technique is more efficient than generate-and-test based methods. For some permutation lengths (14–16), generate-and-test is not feasible because earlier steps cannot be enumerated (within the 1-hour timelimit given to Minion), but the last step can be.

In addition to only enumerating permutations of interest, we can ask for permutation statistics to be listed together with the enumeration as well. For example we can request the number of descents and the number of inversions for each the permutations in the final set within the same constraint programming model.

5 Conclusion

In this paper, we have presented a new approach to enumerating permutations under multiple conditions by leveraging constraint programming. This approach allows for a declarative definition of permutations properties and pattern avoidance/containment conditions. Conditions and properties can be arbitrarily composed. We have demonstrated the versatility of this approach by modelling the pattern avoidance/containment for six conditions and seven permutation properties.

This work contributes to the growing body of research on permutation enumeration. The constraint programming based approach complements existing computational tools, offering an alternative method that allows for greater flexibility when solving non-standard permutation enumeration problems.

The application of constraint programming to this important field in mathematics empowers mathematicians with a new flexible tool for investigating complex permutation enumeration problems. We expect our approach to inspire further research in this area and potentially lead to new mathematical discoveries.

References

1 Özgür Akgün, Alan M Frisch, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter

Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022.

2 Michael Albert. Permlab: Software for permutation patterns, 2012. URL: `https://github.com/mchllbrt/PermCode`.

3 Michael Albert, Steve Linton, and Ruth Hoffmann. Patternclass–permutation pattern classes, 2012.

4 Ragnar Pall Ardal, Tomas Ken Magnusson, Émile Nadeau, Bjarni Jens Kristinsson, Bjarki Agust Gudmundsson, Christian Bean, Henning Ulfarsson, Jon Steinn Eliasson, Murray Tannock, Alfur Birkir Bjarnason, Jay Pantone, and Arnar Bjarni Arnarson. Permuta, April 2021. `doi:10.5281/zenodo.4725759`.

5 M. D. Atkinson. Restricted permutations. *Discret. Math.*, 195(1-3):27–38, 1999. `doi:10.1016/S0012-365X(98)00162-9`.

6 Sergey Avgustinovich, Sergey Kitaev, and Alexander Valyuzhenich. Avoidance of boxed mesh patterns on permutations. *Discrete Applied Mathematics*, 161(1-2):43–51, January 2013. `doi:10.1016/j.dam.2012.08.015`.

7 Eric Babson and Einar Steingrímsson. Generalized permutation patterns and a classification of the mahonian statistics. *Séminaire Lotharingien de Combinatoire [electronic only]*, 44:B44b–18, 2000.

8 Eli Bagno, Estrella Eisenberg, Shulamit Reches, and Moriah Sigron. Blockwise simple permutations, 2023. `arXiv:2303.13115`.

9 Mireille Bousquet-Mélou, Anders Claesson, Mark Dukes, and Sergey Kitaev. (2+ 2)-free posets, ascent sequences and pattern avoiding permutations. *Journal of Combinatorial Theory, Series A*, 117(7):884–909, 2010.

10 Petter Brändén and Anders Claesson. Mesh patterns and the expansion of permutation statistics as sums of permutation patterns. *The Electronic Journal of Combinatorics*, pages P5–P5, 2011.

11 Robert Brignall. A survey of simple permutations. *Permutation patterns*, 376:41–65, 2010.

12 Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

13 The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.12.2*, 2022. URL: `https://www.gap-system.org`.

14 Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006.

15 Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison-Wesley, 1968.

16 Hjalti Magnusson and Henning Ulfarsson. Algorithms for discovering and proving theorems about permutation patterns. *arXiv preprint arXiv:1211.7110*, 2012.

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

18 Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artificial Intelligence*, 251:35–61, 2017.

19 OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2023. Published electronically at `http://oeis.org`.

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

21 Émile Nadeau, Christian Bean, Henning Ulfarsson, Jon Steinn Eliasson, and Jay Pantone. Permutatriangle/comb_spec_searcher: Version 4.0.0, June 2021. `doi:10.5281/zenodo.4946832`.