Discrete Mathematics and Theoretical Computer Science ....
.... **vol. 26:1, Permutation Patterns 2023 #6 (2025)**

Composable Constraint Models for Permutation Enumeration

Ruth Hoffmann[1] Özgür Akgün[1] Christopher Jefferson[2,3]

[1] *School of Computer Science, University of St Andrews, St Andrews, Scotland*
[2] *School of Computer Science and Engineering, Central South University, Changsha, PR China*
[3] *Computing, School of Science & Engineering, University of Dundee, Dundee, Scotland*

revisions 30th Nov. 2023, 21st Oct. 2024, 9th Jan. 2025; accepted 9th Jan. 2025.

Constraint programming (CP) is a powerful tool for modeling mathematical concepts and objects and finding both solutions or counterexamples. One of the major strengths of CP is that problems can easily be combined or expanded. In this paper, we illustrate that this versatility makes CP an ideal tool for exploring problems in permutation patterns.

We declaratively define permutation properties, permutation pattern avoidance and containment constraints using CP and show how this allows us to solve a wide range of problems. We show how this approach enables the arbitrary composition of these conditions, and also allows the easy addition of extra conditions. We demonstrate the effectiveness of our techniques by modelling the containment and avoidance of six permutation patterns, eight permutation properties and measuring five statistics on the resulting permutations. In addition to calculating properties and statistics for the generated permutations, we show that arbitrary additional constraints can also be easily and efficiently added.

This approach enables mathematicians to investigate permutation pattern problems in a quick and efficient manner. We demonstrate the utility of constraint programming for permutation patterns by showing how we can easily and efficiently extend the known permutation counts for a conjecture involving the class of 1324 avoiding permutations. For this problem, we expand the enumeration of 1324-avoiding permutations with a fixed number of inversions to permutations of length 16 and show for the first time that in the enumeration there is a pattern occurring which follows a unique sequence on the Online Encyclopedia of Integer Sequences.

Keywords: Constraint Modelling, Permutation Pattern, Enumeration

1 Introduction

The concept of permutation pattern classes emerged from an exercise question in the Art of Computer Programming Volume 1 (Section 2.2.1, Exercise 5) by Knuth (1968), in which Knuth explored which permutations can be sorted in a limited stack-based system, called stack-sortable permutations. Simion and Schmidt (1985) 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 (2004) later confirmed and proved this conjecture.

There are several systems which use computational techniques to assist with the enumeration of permutation classes and sets, some examples include Permuta Ardal et al. (2021), PatternClass Albert et al. (2012), Combinatorial Specification Searcher Nadeau et al. (2021) and PermCode (previously PermLab) Albert (2012). Permuta is a Python package which supports the enumeration of permutation classes using the BiSC algorithm by Magnusson and Ulfarsson (2012). PatternClass is a GAP package that encodes the permutations 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 graphical user interface which allows for the exploration of the sets of permutations which avoid classical patterns.

Constraint programming (CP) is a generic and powerful paradigm that allows the expression of complex combinatorial problems in a declarative manner. In CP problems are expressed declaratively by giving a list of variables whose values must be found, and stating relationships between variables in the form of constraints. This declarative expression is called a model. CP solvers take this declarative version of the problem and use a range of different algorithms to solve the problem efficiently. The primary advantage of CP is that it allows the user to focus on the problem formulation, while the computer takes responsibility for the problem-solving methodology.

In the context of studying permutation patterns, constraint programming can offer significant benefits. Permutation patterns provide a prominent and widely researched topic within the domain of combinatorial mathematics. These patterns, which involve the arrangement of numbers in a certain order, exhibit intricate and complex structures. Constraint programming allows us to model these structures and constraints explicitly, enabling systematic and efficient exploration of the solution space.

In this paper we express problems in the high-level constraint language Essence by Frisch et al. (2008). Essence, as implemented by the system Conjure by Akgün et al. (2022), provides more abstract representation of combinatorial problems. Conjure converts high-level specifications given in Essence into a range of different formats, allowing many different constraint solvers (and related technologies, such as Mixed Integer Programming and Boolean Satisfiability) to be used when solving the written (also called *modelled*) constraint problem. This automation not only reduces the possibility of errors in translation but also frees the user to concentrate on the higher-level aspects of problem modelling. In this paper, we mainly use the constraint solver Minion created by Gent et al. (2006), which has been used to solve several large scale combinatorial problems in the past, including the number of semi-groups of order 10 Distler et al. (2012) and for the enumeration of set-theoretic solutions to the Yang-Baxter equation Akgün et al. (2022).

In this paper we show how CP can aid the research in permutation patterns through the following contributions:

1. A systematic and comprehensive treatment of 6 kinds of permutation pattern (as both avoidance and containment) as well as 13 properties and 5 statistics. Each one is implemented as a standalone constraint model. This is presented in Section 3.

2. The models of the patterns, properties and statistics are structured in a way that allows seamless composition of the individual models. Section 4 illustrates this compositionality.

3. An evaluation of the flexible compositional constraint programming approach in 2 settings, one extended illustrative example and one example showing how composable CP models can extend current results such as a conjecture in Claesson et al. (2012). In Section 4.1 and Section 4.2 we present the examples.

4. 2 new conjectures resulting from the model enumerating 1324-avoiding permutations with a fixed number of inversions which come from extending the computational results in Claesson et al. (2012).

We believe our library of models, which can be easily composed in a ”pick and mix” fashion, will further the research in permutation patterns by allowing for more efficient computational experimentation and exhaustive search as the constraint programming approach avoids the need for the traditional ”generate-and-test” approach, which generates intermediate assignments that only satisfy some of the properties for later filtering.

This ”generate-and-test” approach, which requires specialised algorithms to be created to find all permutations which satisfy some given property, can work very well for that property but this approach is difficult to extend and requires considerable expertise in programming to add new properties. On the other hand, combining CP models can immediately allow highly efficient searching for a combination of properties, and our experience is adding new patterns in CP is significantly easier than create new bespoke state-of-the-art programs for searching for new patterns.

We choose not to compare our models against tools such as PermLab Albert (2012), Combinatorial Specification Searcher Nadeau et al. (2021) or PatternClass Albert et al. (2012), as to our knowledge these tools do not support all features that we implement (for example number of inversions, parity, block-wise simplicity), or have a specialised focus (for example on regular classes) than the library of models which we will present.

2 Introduction to Constraint Programming

Constraint Programming (CP) is a powerful paradigm for solving complex combinatorial problems by focusing on the formulation of constraints rather than explicit solution algorithms. CP is particularly useful in situations where you need to find solutions that satisfy a set of conditions or optimise a certain objective under constraints.

The process of working in CP typically involves three main steps:

- Modelling the problem: This involves defining the problem in a high-level language such as Essence Frisch et al. (2008), where variables and constraints are described abstractly, without specifying how to solve the problem.
- Translation to a solver-friendly format: The high-level model is then automatically translated into a lower-level format that can be handled by a solver. Tools like Conjure Akgün et al. (2022) facilitate this step by converting the Essence model into formats compatible with solvers such as Minion Gent et al. (2006), Chuffed Chu et al. (2018), or MiniSAT Eén and Sörensson (2003).
- Solving the problem: The translated model is processed by a solver, which systematically searches for a solution by trial and error, backtracking when constraints are violated, until a valid solution is found.

Most users of CP interact primarily with the modelling step, where they define the problem abstractly in a language like Essence. The remaining steps—translation and solving—are typically handled by automated tools.

2.1 Constraint Modelling

Constraint modelling in CP consists of representing the problem using a set of variables, parameters, and constraints, all described declaratively. This allows the solver to explore the search space and identify

```
1  given <name> : <domain>    $ declaring a parameter
2  letting <name> be <value> $ setting the value of a parameter or a local alias
3  find <name> : <domain>     $ declaring a decision variable
4  such that <constraint>     $ posting a constraint
5
6  $ specific examples from models used in this paper
7  given length : int
8  given avoidances : set of sequence of int
9  find perm : sequence (size length, injective) of int(1..length)
10
11 $ constraint types
12 such that forAll i : int(1..length) . <constraint>
13 such that forAll m : matrix indexed by [int(1..pattern)] of int(1..length) . <constraint>
14 such that forAll q : <domain> , <condition> . <constraint>
15 such that forAll av in avoidances . <constraint>
16 such that perm(1) = 2, perm(2) = 3
```

Listing 1: A tour of the most relevant features of the problem specification language Essence

solutions that satisfy all of the specified constraints. In Listing 1 is a sample Essence code that demonstrates key constructs and patterns in constraint modelling:

We first start by an explanation of key constructs in Essence, using the example fragments in Listing 1.

Declaring parameters (line 1): The *given* keyword defines parameters that act as fixed inputs to the model. Parameters have a name and a domain. For instance, on line 6, length is declared as an integer parameter that will specify the length of the permutation. Parameters allow for flexibility, as they can be modified without changing the model's core structure.

Setting values (line 2): The *letting* keyword assigns a value to a parameter or local alias. This can be used for simplification or to define constants. In practice, this makes it easier to reference certain values throughout the model.

Declaring decision variables (line 3): The *find* keyword is used to define decision variables, which are the unknowns that the solver will try to assign values to. On line 8, for example, the variable perm is declared as a sequence, and it is required to be injective (meaning all elements in the sequence must be unique).

Adding constraints (Line 4): The *such that* keyword introduces constraints, which are conditions that must be satisfied by the decision variables. These constraints guide the solver to ensure valid solutions. On line 14, a constraint explicitly fixes certain values of the permutation, forcing the first element to be 2 and the second to be 3.

In the specific example of a fragment of permutation problem (lines 6–8), two parameters are declared: *length* defines the size of the permutation, *avoidances* is a set of sequences that must be avoided.

The decision variable perm represents a permutation, which is a sequence of integers from 1 to length. The constraint (line 8) ensures that the sequence is injective, meaning each element is unique.

2.1.1 Constraint Types

There are several types of constraints that can be applied, depending on the structure of the problem:

Iterating over elements (Line 10): The *forAll* keyword applies a constraint across all elements of the sequence. For example, this syntax might ensure that certain conditions hold for every element between 1 and length.

Matrix constraints (Line 11): In more complex problems, one can quantify over a matrix domain. This is useful for considering all permutations of a particular length, for example.

Conditional constraints (Line 12): Conditional constraints apply to a subset of the domain, based on certain conditions. For example, one can restrict constraints to cases where a specific condition is true.

Pattern avoidance (Line 13): One can iterate over a set of patterns (in this case, avoidances) and enforce constraints that prevent the sequence from containing any of these patterns.

Fixed values (Line 14): Lastly, constraints can assign fixed values to certain positions in the sequence. In this example, the first two positions of perm are assigned values of 2 and 3, respectively. This operation corresponds to permutation application when the sequence is viewed as a permutation.

In all of these examples the *forAll* quantifier can be replaced with *exists* as well, to achieve existential quantification as opposed to universal.

In Listing 2 we give a complete example constraint model, where the parameters values are defined within the model itself for simplicity. This problem is that of finding all permutations of length 4 that classically contain the pattern permutation 21. We will explain the model in more detail in Section 3.

```
letting length be 4
letting classic_containment be {sequence(2,1)}

find perm : sequence (size length, injective) of int(1..length)

such that
    forAll pattern in classic_containment .
         (exists ix : matrix indexed by [int(1..|pattern|)] of int(1..length) .
            (forAll i,j : int(1..|pattern|) . i < j -> ix[i] < ix[j]) /\
            (forAll n1, n2 : int(1..|pattern|) , n1 < n2 .
                pattern(n1) < pattern(n2) <-> perm(ix[n1]) < perm(ix[n2])))
```

Listing 2: Essence code which represents classic containment

This model (Listing 2) can be solved to obtain either a single solution, a specified number of solutions, or all possible solutions. In Listing 3 we present the result when asking for 10 solutions.

```
# A single, random permutation of length 4, containing 21
{"perm": [1, 2, 4, 3]}


# 10 random permutations of length 4, containing 21
[{"perm": [1, 2, 4, 3]}, {"perm": [1, 3, 2, 4]}, {"perm": [1, 3, 4, 2]}, {"perm": [1, 4, 2, 3]},
 {"perm": [1, 4, 3, 2]}, {"perm": [2, 1, 3, 4]}, {"perm": [2, 1, 4, 3]}, {"perm": [2, 3, 1, 4]},
 {"perm": [2, 3, 4, 1]}, {"perm": [2, 4, 1, 3]}]
```

Listing 3: Output from running a constraint solver over the 21 classical containment model.

The strength of constraint modelling lies in its approach to solving problems with multiple constraints. Unlike traditional methods, which typically generate a set of permutations and then filter them based on additional constraints, constraint solvers work by applying all constraints at once during the search process. This direct approach ensures efficiency and avoids unnecessary intermediate steps, making the search for valid solutions more streamlined. With some care in how parameters and variables are named, different constraints can be combined defined in a modular fashion and composed seamlessly, allowing the solver to find solutions that satisfy all conditions simultaneously.

The problem specification language Essence is particularly well-suited for modelling permutation pattern problems due to its expressive power and flexibility in defining complex combinatorial problems declaratively. It allows for a natural representation of sequences, set operations, and matrix indexing, which are fundamental in defining and manipulating permutations. The ability to declare injective sequences and easily handle universal or existential quantification over sets of constraints makes Essence an ideal choice for modelling pattern containment and avoidance problems. Furthermore, Essence's high-level abstraction allows users to focus on formulating the problem without needing to worry about implementation details, which are handled during the translation process into solver-compatible formats. This makes it an efficient tool for exploring and solving permutation pattern problems.

2.2 Constraint Translation

After modelling a problem in Essence, the next step is translating the high-level specification into a lower-level representation that can be handled by a solver. This translation process is automatically performed by Conjure and Savile Row Nightingale et al. (2017), which collectively convert Essence models into a form suitable for constraint solvers. The translation bridges the abstract, declarative model with the computational techniques necessary for solving it.

Typically, the solver chosen by default (e.g., Minion Gent et al. (2006) for constraint problems) is sufficient for most tasks, but users can opt for other solvers based on the problem's nature. For example, Chuffed Chu et al. (2018) is another constraint solver, while MiniSAT Eén and Sörensson (2003) is used for Boolean satisfiability (SAT) problems. Each solver offers different strategies and algorithms for navigating the solution space, and their suitability depends on the problem at hand.

What is key to note is that the user does not need to concern themselves with this translation process. They only need to focus on modelling the problem in a high-level language like Essence. The translation step transforms the model into formats that can be executed by various solvers, making it efficient and easy to experiment with different solvers without needing to rewrite the core model.

By automating this process, Essence and Conjure significantly reduce errors that may occur during manual translation, while also enabling the flexibility to leverage a range of solvers for a single problem. This ensures that the model remains solver-agnostic, allowing the user to concentrate on the conceptual aspects of the problem rather than the technicalities of the solving process.

2.3 Constraint Solving

Once the model has been translated into a solver-compatible format, the solver takes over to search for solutions using a combinatorial search algorithm, most commonly a form of backtrack search. This process systematically assigns values to variables while checking that all constraints are satisfied.

A solver processes the entire model simultaneously, as seen in Listing 2. Unlike traditional algorithms in permutation patterns, which may generate all permutations or subsets first and then apply filters based on constraints, constraint solvers apply the constraints immediately during the search. This approach avoids unnecessary computation and ensures that only valid solutions are considered.

The search procedure typically works as follows:

Variable-Value Assignment: The solver selects a variable and assigns a value from its domain (the range of possible values). The choice of which variable to assign next can be random, based on a heuristic, or follow a predefined ordering. This is only done after constraint propagation, to avoid evaluating values that are shown to be impossible.

Constraint Checking and Propagation: At the beginning of the search and after every time a value is assigned to a variable, the solver checks whether the assignment violates any constraints. If not, the solver propagates this assignment, which involves updating the domains of other variables to reflect the consequences of the assignment. For example, assigning a value to one variable might reduce the possible values for others.

Backtracking: If a constraint is violated at any point during the search, the solver backtracks. It undoes the most recent variable assignment and tries a different value, continuing this process until a valid solution is found or all possibilities are exhausted.

Solution Discovery and Continuation: When all variables have valid assignments that satisfy the constraints, the solver has found a solution. If multiple solutions are needed, the solver backtracks to

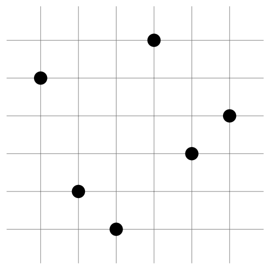


Fig. 1: Plot representation of the permutation 521634. The x-axis represents the indexes of the permutation, and the y-axis represents the values. Reading left to right, the location of the dot in the ith column represents the ith value in the permutation.

explore different combinations of variable assignments, continuing the search to find additional solutions.

The search process is highly efficient because the solver only explores variable assignments that are consistent with the constraints, avoiding unnecessary work. Propagation, in particular, helps to narrow down the search space by eliminating infeasible options early.

In summary, constraint solvers combine trial and error, propagation, and backtracking to efficiently navigate the search space. While this paper focuses on providing models in the Essence language, these models can be solved by a variety of solvers using similar search strategies to find valid solutions.

Having established the fundamental aspects of constraint programming, from modelling to solving, we now turn our attention to a more specialised domain: permutation pattern problems. Permutation patterns are a rich source of combinatorial problems, where one seeks to identify or avoid specific patterns within permutations. The flexibility and power of the Essence language, combined with constraint programming, allow for an elegant and expressive representation of these problems. In the following section, we present a comprehensive library of models specifically designed for permutation pattern problems, showcasing how various types of pattern containment, avoidance, and permutation properties can be encoded within the CP framework. These models form a foundation for exploring and solving a wide array of permutation-related challenges.

3 Library of Models

We will consider a *permutation* σ to be an arrangement of the set $\{1, 2, \ldots, n\}$ 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 to denote 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 permutations $\pi = \pi(1), \ldots, \pi(n)$ and $\sigma = \sigma(1), \ldots, \sigma(n)$ of the same length are said to be *order isomorphic* Atkinson (1999) if $\forall i, j, \pi(i) \leq \pi(j)$ if and only if $\sigma(i) \leq \sigma(j)$. The same property applies for sequences as well.

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

All CP models from the upcoming sections with definitions can be found in a supplementary repository Akgün et al. (2023). This repository contains executable CP models, sample parameter files, raw data

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	
	Parity	

Tab. 1: Table of all permutation patterns, properties and statistics modelled

for our computational experiments and scripts that can be used to fully rerun the experiments. In addition, the repository contains a Jupyter notebook that can be used to interactively run model fragments through Conjure. The notebook also contains the example scenarios from Section 4. The aim of the models is to be as close to the mathematical definitions as possible.

3.1 Pattern types

A permutation pattern identifies a subsequence of a permutation which satisfies a list of constraints. 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 patterns, in this section we introduce the ones considered in this paper. We will denote the set of permutations that contains a set of patterns P as Co(P) and the set of permutations which avoid a set of patterns P as Av(P).

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 $134 = \sigma(3)\sigma(5)\sigma(6)$. Being *classically contained* is a partial order on the set of permutations Brignall (2010). In Figure 2(a) on the left is the pattern permutation, on the right we have highlighted an occurrence of the classical pattern by circling the elements.

Listing 4 represents the model for classic containment, where in Lines 1 and 2 we limit the model to finding permutations of length 4 and limit the set of permutation patterns that we look for to the just the permutation 21.

```
1  letting length be 4
2  letting classic_containment be {sequence(2,1)}
3
4  find perm : sequence (size length, injective) of int(1..length)
5
6  such that
7  $ For each pattern in the classic_containment set we will look for at least one occurence of it
8      forAll pattern in classic_containment .
9  $ We use a matrix to represent the permutation in two dimensions, this is not really needed for classic patterns,
10 $ but to allow for the code of the different patterns to be composable we have added this here as well.
11              (exists ix : matrix indexed by [int(1..|pattern|)] of int(1..length) .
12 $ We now look constraint perm to contain an order isomorphic copy of pattern.
13                 (forAll i,j : int(1..|pattern|) . i < j -> ix[i] < ix[j]) /\
14                 (forAll n1, n2 : int(1..|pattern|) , n1 < n2 .
15                     pattern(n1) < pattern(n2) <-> perm(ix[n1]) < perm(ix[n2])))
```

Listing 4: Essence code which represents classic containment

Vincular patterns (introduced by Babson and Steingrímsson (2000)) specify adjacency conditions. Let $\pi = \pi(1)\dots\pi(k)$, $\sigma = \sigma(1)\dots\sigma(m)$ and let $A \subseteq \{0,\dots,k\}$. To simplify the pattern definition, we define $\pi(0) = 0$ and $\pi(k+1) = k+1$. 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*. The code in Listing 5 shows the code for any given length and looking for the vincular pattern $(132, \{1\})$.

For example, the vincular pattern $(132, \{1\})$ can be found in $\sigma = 521634$ as the subsequence $164 = \sigma(3)\sigma(4)\sigma(6)$. Figure 2(b) 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.

```
given length : int
letting vincular_containment be  { [[1,3,2], [1]] }

find perm : sequence (size length, injective) of int(1..length)

such that
$ We check each vincular pair which consists of the permutation (pattern) and adjacency set (bar).
    forAll (pattern, bars) in vincular_containment .
        exists ix : matrix indexed by [int(1..|pattern|)] of int(1..length) .
$ First we look for a classic pattern occurence
            (forAll i,j : int(1..|pattern|) . i < j -> ix[i] < ix[j]) /\
            (forAll n1, n2 : int(1..|pattern|) , n1 < n2 .
                (pattern(n1) < pattern(n2) <-> perm(ix[n1]) < perm(ix[n2]))) .
$ Then we look in the matrix representation if the adjacency for this pattern is not violated.
            (forAll bar in bars . ix[bar] + 1 = ix[bar+1])
```

Listing 5: Essence code which represents vincular containment

Bivincular patterns (as introduced in Bousquet-Mélou et al. (2010)) are vincular patterns which have an additional set defining which values have to be adjacent.

More formally, an occurrence of the bivincular pattern (π, A, B) in σ, with $A, B \subseteq \{0,\dots,k\}$ and $|\pi| = k$, 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,
- $\forall a \in A.\ i_{a+1} = i_a + 1$
- $\forall b \in B.\ j_{b+1} = j_b + 1$

where $\{\sigma(i_1),\dots,\sigma(i_k)\} = \{j_1,\dots,j_k\}$ and $j_1 < j_2 < \dots < j_k$. By convention $i_0 = j_0 = 0$ and $i_{k+1} = j_{k+1} = n+1$. As with vincular patterns, A is the set of adjacencies of indexes, but now the set B defines the adjacencies of values. In Listing 6 we show the model looking for permutations of a given length containing the set of bivincular patterns.

For example, the bivincular pattern $(312, \{2\}, \{2\})$ can be found in $\sigma = 521634$ as the subsequence $534 = \sigma(1)\sigma(5)\sigma(6)$. Figure 2(c) illustrates this bivincular containment with the pattern permutation and the shading of the adjacency in values (rows) and indices (columns) highlighted.

```
given length : int
given bivincular_containment : set of (sequence (injective) of int, set of int, set of int)

find perm : sequence (size length, injective) of int(1..length)

$ Creating a padded version of perm, where position 0 contains the value 0 and position length+1 contains the value length+1,
$ to be able to follow the convention.
find permPadded : matrix indexed by [int(0..length+1)] of int(0..length+1)
such that permPadded[0] = 0, permPadded[length+1] = length+1
such that forAll i : int(1..length) . permPadded[i] = perm(i)

such that
$ We look for each bivincular pattern consisting of the permutation, the index adjacencies and the value adjacencies.
    forAll (pattern, ind_bars, val_bars) in bivincular_containment .
$ Build the inverse of 'pattern'. This is completely evaluated before solving.
        exists patterninv: matrix indexed by [int(0..|pattern|+1)] of int(0..|pattern|+1),
            patterninv[0] = 0 /\ patterninv[|pattern|+1] = |pattern|+1 /\
            (forAll i: int(1..|pattern|) . patterninv[pattern(i)] = i).

$ Look for all places where the pattern can occur classically
        exists ix : matrix indexed by [int(0..|pattern|+1)] of int(0..length+1),
            and([ ix[0]=0, ix[|pattern|+1]=length+1 ,
                forAll i : int(0..|pattern|) . ix[i] < ix[i+1] ,
                forAll n1, n2 : int(1..|pattern|) , n1 < n2 .
                    pattern(n1) < pattern(n2) <-> permPadded[ix[n1]] < permPadded[ix[n2]]
                ]) .
$ Check if the index adjacency is not violated
            ((forAll bar in ind_bars . ix[bar] + 1 = ix[bar+1])
            /\
$ And check if the value adjacency is not violated
            (forAll bar in val_bars . permPadded[ix[patterninv[bar]]]+1 =
                    permPadded[ix[patterninv[bar+1]]]))
```

Listing 6: Essence code which represents bivincular containment

All of the above patterns can be generalised as *mesh patterns*, which were introduced in Brändén and Claesson (2011). 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 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. The model of mesh containment (together with the extended mesh) is in Listing 7. An example of a mesh pattern in $\sigma = 521634$ is $(132, \{(0,0),(2,1),(2,2)\})$, which can be found as the subsequence $263 = \sigma(2)\sigma(4)\sigma(5)$. This example is illustrated in Figure 2(d), where each of the forbidden regions/squares is shaded.

A *boxed mesh pattern* (or *boxed pattern*, as introduced in Avgustinovich et al. (2013)), 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 $351 = \sigma(2)\sigma(4)\sigma(5)$. Figure 2(e) shows this example, and illustrates that for a boxed mesh pattern the box inside the permutation is the forbidden region.

```
given length : int
given mesh_containment : set of (sequence(injective) of int, relation of (int * int))

find perm : sequence (size length, injective) of int(1..length)

$ Creating a padded version of perm, where position 0 contains the value 0 and position length+1 contains the value length+1,
$ to be able to follow the convention.
find permPadded : matrix indexed by [int(0..length+1)] of int(0..length+1)
such that permPadded[0] = 0, permPadded[length+1] = length+1
such that forAll i : int(1..length) . perm(i) = permPadded[i]

such that
$ pattern is the pattern, mesh is the mesh as a relation
    forAll (pattern, mesh) in mesh_containment .
$ Build the inverse of 'pattern'. This is completely evaluated before solving.
    exists patterninv: matrix indexed by [int(0..|pattern|+1)] of int(0..|pattern|+1),
                patterninv[0] = 0 /\ patterninv[|pattern|+1] = |pattern|+1 /\
                (forAll i: int(1..|pattern|) . patterninv[pattern(i)] = i).
$ Look for all places where the pattern can classically occur
        exists ix : matrix indexed by [int(0..|pattern|+1)] of int(0..length+1),
            and([ ix[0]=0, ix[|pattern|+1]=length+1 ,
                forAll i : int(0..|pattern|) . ix[i] < ix[i+1] ,
                forAll n1, n2 : int(1..|pattern|) , n1 < n2 .
                    pattern(n1) < pattern(n2) <-> permPadded[ix[n1]] < permPadded[ix[n2]]
                ]) .
$ If we find the pattern, make sure there is NOT at least one value in some cell of the mesh
            !( exists (i,j) in mesh.
               exists z: int(ix[i]+1..ix[i+1]-1). (permPadded[ix[patterninv[j]]] <= permPadded[z] /\
                        permPadded[z] <= permPadded[ix[patterninv[j+1]]]) )
```

Listing 7: Essence code which represents mesh containment

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 $524 = \sigma(2)\sigma(3)\sigma(4)$. This example is shown in Figure 2(f).

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.

Listing 8 shows the model for classic pattern avoidance. We can see that in comparison to classic containment (Listing 4) there is a negation sign (denoted by !) before the matrix which defines the pattern constraint.

Such simple negations apply through all the models, for example Listing 9 where if there is an occurrence of the pattern there are points that violate the mesh regions.

So for example the permutation $\sigma = 12345$ avoids the classic pattern $\pi = 21$, as there are no occurrences of it. Figure 3(a) attempts to illustrate this avoidance. Similarly, the permutation $\sigma = 12345$ avoids the mesh pattern $(132, \{(0,0),(2,0),(2,1),(2,2)\})$ as there is no classical occurrence of 132 in σ, as can be seen from Figure 3(b). Finally, $(132, \}(0,0),(2,0),(2,1),(2,2))\}$ is not contained in the permutation $\sigma = 652413$ even though there is an occurrence of 132 in the σ, namely as $243 = \sigma(3)\sigma(4)\sigma(6)$, but there is an element of the permutation that violates the forbidden region. In Figure 3(c) we can see that the element $1 = \sigma(5)$ is in the $(2,0)$ shaded/forbidden region.

3.2 Permutation Properties

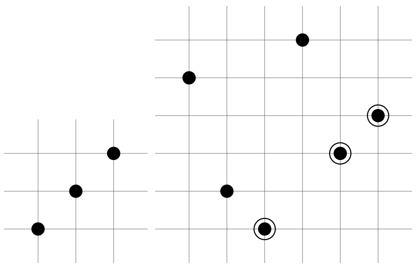
In addition to containing (or avoiding) patterns in permutations and enumerating these permutations, permutations can have certain structural properties. We give some of the most common properties here, all of which we will model with CP. Table 1 contains a list of the properties which we have modelled thus far. For

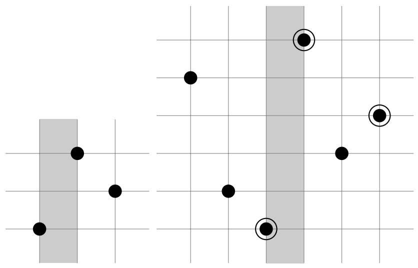
```
given length : int
given classic_avoidance : set of sequence of int

find perm : sequence (size length, injective) of int(1..length)

such that
$ For each pattern in the classic_avoidance set we will look for at least one occurence of it
    forAll pattern in classic_avoidance .
$ We use a matrix to represent the permutation in two dimensions, this is not really needed for classic patterns,
$ but to allow for the code of the different patterns to be composable we have added this here as well.
$ While the classic pattern constraint stays the same we are now negating the whole thing (see the ! before the exists statement)
$ So we are looking for a permutation in which it is not possible to find the pattern in.
             !(exists ix : matrix indexed by [int(1..|pattern|)] of int(1..length) .
$ We now look constraint perm to contain an order isomorphic copy of pattern.
                  (forAll i,j : int(1..|pattern|) . i < j -> ix[i] < ix[j]) /\
                  (forAll n1, n2 : int(1..|pattern|) , n1 < n2 .
                       pattern(n1) < pattern(n2) <-> perm(ix[n1]) < perm(ix[n2])))
```

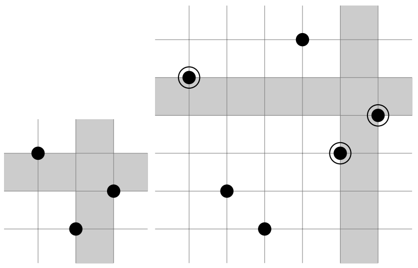
Listing 8: Essence code which represents classic avoidance

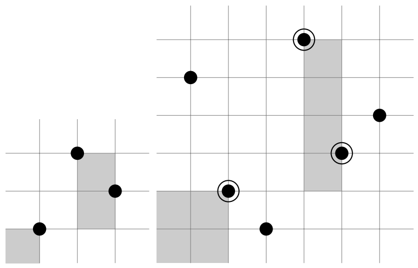
```
given length : int
given mesh_avoidance : set of (sequence(injective) of int, relation of (int * int))

find perm : sequence (size length, injective) of int(1..length)

$ Creating a padded version of perm, where position 0 contains the value 0 and position length+1 contains the value length+1,
$ to be able to follow the convention.
find permPadded : matrix indexed by [int(0..length+1)] of int(0..length+1)
such that permPadded[0] = 0, permPadded[length+1] = length+1
such that forAll i : int(1..length) . perm(i) = permPadded[i]

such that
$ pattern is the pattern, mesh is the mesh as a relation
     forAll (pattern, mesh) in mesh_avoidance .
$ Build the inverse of 'pattern'. This is completely evaluated before solving.
     exists patterninv: matrix indexed by [int(0..|pattern|+1)] of int(0..|pattern|+1),
                   patterninv[0] = 0 /\ patterninv[|pattern|+1] = |pattern|+1 /\
                   (forAll i: int(1..|pattern|) . patterninv[pattern(i)] = i).
$ Look for all places where the pattern can occur, in each we need to violate the mesh.
$ This changes from an 'exists' to a 'for all'
           forAll ix : matrix indexed by [int(0..|pattern|+1)] of int(0..length+1),
                 and([ ix[0]=0, ix[|pattern|+1]=length+1, forAll i : int(0..|pattern|) . ix[i] < ix[i+1]
                     , forAll n1, n2 : int(1..|pattern|) , n1 < n2 .
                         pattern(n1) < pattern(n2) <-> permPadded[ix[n1]] < permPadded[ix[n2]]
                     ]) .
$ If we find the pattern, make sure there is at least one value in some cell of the mesh
                 ( exists (i,j) in mesh.
                   exists z: int(ix[i]+1..ix[i+1]-1). (permPadded[ix[patterninv[j]]] <= permPadded[z] /\
                                  permPadded[z] <= permPadded[ix[patterninv[j+1]]]) )
```

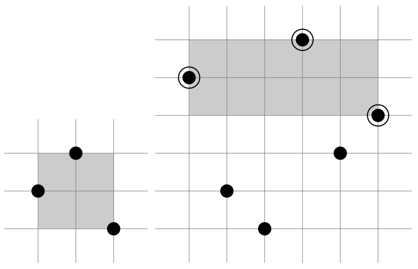
Listing 9: Essence code which represents mesh avoidance


(a) Classic Pattern


(b) Vincular Pattern


(c) Bivincular Pattern


(d) Mesh Pattern


(e) Boxed Mesh Pattern

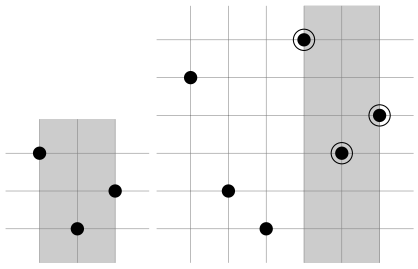
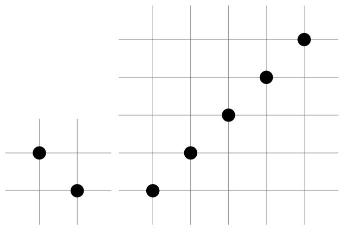
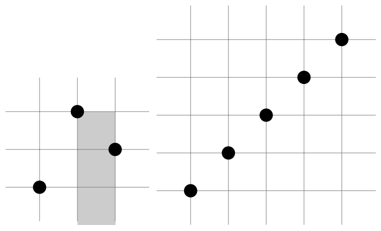

(f) Consecutive Pattern

Fig. 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 $(132, \{(2,0),(2,1),(2,2)\})$.

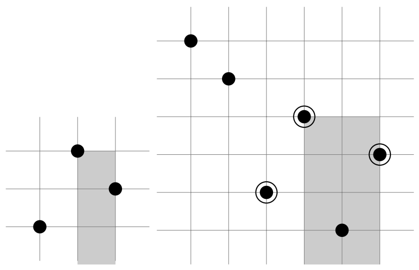

(c) 652413 avoids the mesh pattern $(132, \{(2,0),(2,1),(2,2)\})$.

Fig. 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.

each property, we can require solutions to either do, or do not, satisfy the property.

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* Brignall (2010). Listing 10 describes the constraints of a simple permutation (in terms of its intervals). 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 $\pi(1)\pi(2)\pi(3)\pi(4)\pi(5)\pi(6)$, as indicated in Figure 4(a) through bold boxes around the intervals. The permutation 246135 in Figure 4(d) is simple, as the only intervals it contains are of length 0, 1 or the whole permutation.

```
given length : int
find perm : sequence (size length, injective) of int(1..length)

$ Simple permutations do not contain intervals.
$ An interval is a set of contiguous indices such that the values are also contiguous.
$ perm is a simple permutation, subperm is a set of contiguous indices.
such that
   and ( [ 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)                             $ the subpermutation is not the full permutation
    , letting subperm be [perm(k) | k : int(i..j)]    $ give the sub perm a name
    ])
```

Listing 10: Essence code which represents simple permutations

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 4(e) 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 (Listing 11). For example 213654 is a plus-decomposable permutation, but 546123 or 521634 (which is also not simple) are not. Figure 4(b) 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).

```
given length : int
find perm : sequence (size length, injective) of int(1..length)

$ perm is plus−decomposable
such that
    exists sep : int(1..length-1) .
$ There is a split where there is an interval on the "left" where the maximal point is below the lowest point in the interval on the "right".
        max([ perm(i) | i : int(1..sep) ]) < min([ perm(i) | i : int(sep+1..length) ])
```

Listing 11: Essence code which represents plus-decomposable permutations

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 (Listing 12). For example 546123 is a minus-decomposable permutation. But 213654 (which is plus-decomposable) or 236145 (which is not simple) are not. Figure 4(c) 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.

```
given length : int
find perm : sequence (size length, injective) of int(1..length)

$ perm is minus−decomposable
such that
    exists sep : int(1..length-1) .
$ There is a split where there is an interval on the ”left” where the minimal point is above the highest point in the interval on the ”right”.
        min([ perm(i) | i : int(1..sep) ]) > max([ perm(i) | i : int(sep+1..length) ])
```

Listing 12: Essence code which represents minus-decomposable permutations

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 Bagno et al. (2023) alongside an equivalent recursive recursive definition. We represent this property as a constraint model in Listing 13. For example $2413[3142, 1, 1, 1] = 4253716$ is block-wise simple (Figure 4(f)), but 24513 is not as the interval 45 can be decomposed into $12[1, 1]$ (Figure 4(g)).

```
given length : int
find perm : sequence (size length, injective) of int(1..length)

such that
$ It does not decompose into 12 (i.e. there is a point in the interval that is above the smallest point on the right)
    [ and([ max([ perm(i) | i : int(start..middle) ]) > min([ perm(i) | i : int(middle+1..end) ])
$ It does not decompose into 21 (i.e. there is a point in the interval that is below the highest point on the right)
          , min([ perm(i) | i : int(start..middle) ]) < max([ perm(i) | i : int(middle+1..end) ])
          ])
$ The below is the setup of the intervals, and the indices
    | start, middle, end : int(1..length)
    , start <= middle
    , middle < end
    , letting minSE be min([ perm(i) | i : int(start..end) ])
    , letting maxSE be max([ perm(i) | i : int(start..end) ])
    , maxSE - minSE = end - start
    ]
```

Listing 13: Essence code which represents block-simple permutations

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 (Listing 14). 4312 is a derangement whereas 1234 is not. As shown in Figure 4(i) none of the elements of the permutation 4312 are on the red diagonal (which represents $\pi(i) = i$).

```
given length : int
find perm : sequence (size length, injective) of int(1..length)

such that
    forAll i : int(1..length) .
$ None of the indices are fixed points.
        perm(i) != i
```

Listing 14: Essence code which represents derangements

Similarly, a *nonderangement* is a permutation with at least one fixed point (Listing 15). 2431 is a nonderangement whereas 4321 is not. The plot of 2431 in Figure 4(j) shows $\pi(3) = 3$ lies on the diagonal.

```
given length : int
find perm : sequence (size length, injective) of int(1..length)

such that
$ At least one of the points is fixed.
    exists i : int(1..length) .
        perm(i) = i
```

Listing 15: Essence code which represents nonderangements

A permutation $\pi \in S_n$ is called an *involution* if $\pi = \pi^{-1}$, or equivalently if $\forall i, j.\, \pi(i) = j \iff \pi(j) = i$ (see Listing 16 for the model). 1243 is an involution but 2431 is not. Figure 4(h) contains the plot of the involution 1243.

```
given length : int
find perm : sequence (size length, injective) of int(1..length)

$ perm is an involution
such that
    forAll i : int(1..length) .
$ perm * perm = identity permutation (increasing permutation)
        perm(perm(i)) = i
```

Listing 16: Essence code which represents involutions

A permutation $\pi \in S_n$ is said to have *parity* if the values at odd indexes are odd and the values in even indexes are even, i.e. $\pi(i) = i \mod 2,\ \forall i \in \{1, \ldots, n\}$ (Listing 17). For example the permutation 3412 has parity, whereas 2413 as in the latter for example the value 2 (even) is at index 1 which is odd. Figure 4(k) and Figure 4(l) show the plots of the two example permutations.

```
given length : int
find perm : sequence (size length, injective) of int(1..length)

such that
    forAll i : int(1..length) .
$ Check for odd/even by doing modulo 2 arithmetic.
        (perm(i) % 2) = i % 2
```

Listing 17: Essence code which represents permutations with parity

3.3 Permutation Statistics

Currently we support 5 different permutation statistics, where we count the occurrences of some property of the elements of the permutation. The constraint solvers will output the statistics alongside the solution permutations. The statistics we currently consider are ascents, descents, inversions, excedances and the Major index (as listed in Table 1).

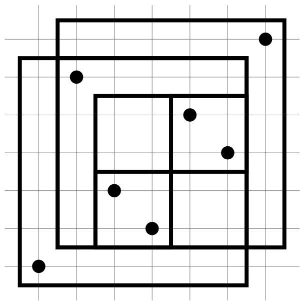
These statistics can be collected as part of finding the solutions, or constrained. These constraints on the statistics can be arbitrary arithmetic statements, for example permutations with 5 descents, or the number of descents and ascents added together is a multiple of three.

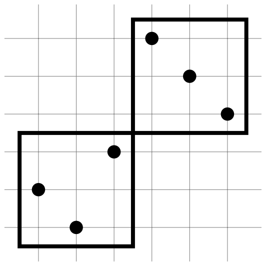
An *ascent* in a permutation σ is an index i such that $\sigma(i) < \sigma(i+1)$. Similarly, a *descent* is an i such that $\sigma(i) > \sigma(i+1)$. A pair of indices (i, j) in a permutation σ such that $i < j$ and $\sigma(i) > \sigma(j)$ is called an *inversion*. An *excedance* is an index where $\sigma(i)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

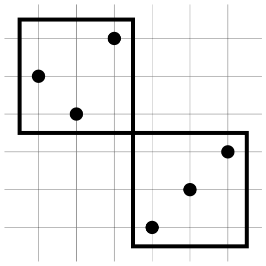
- the number of *inversions* $\text{inv}(\sigma) = |\{(i, j) : i < j \text{ and } \sigma(i) > \sigma(j)\}|$
- the number of *descents* $\text{des}(\sigma) = |\{i : \sigma(i) > \sigma(i+1)\}|$
- the number of *ascents* $\text{asc}(\sigma) = |\{i : \sigma(i) < \sigma(i+1)\}|$
- the number of *excedances* $\text{exc}(\sigma) = |\{i : \sigma(i) > i\}|$.
- the *Major index*, which is the sum of the positions of the descents $\text{maj}(\sigma) = \sum_{\sigma(i)>\sigma(i+1)} i$.

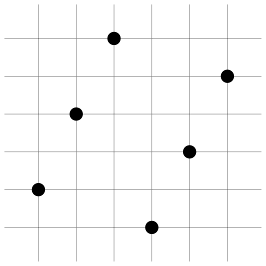
In Listing 18 we have the code for all five statistics in one.

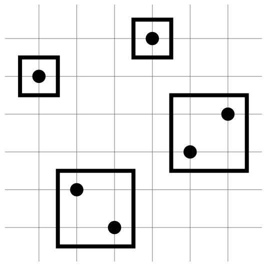
As an example, in the permutation $\sigma = 7164523$, $\text{inv}(\sigma) = 14$, $\text{des}(\sigma) = 3$, $\text{asc}(\sigma) = 3$, $\text{exc}(\sigma) = 2$ and $\text{maj}(\sigma) = 9$.

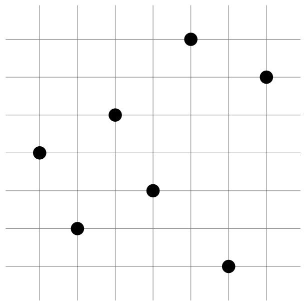

(a) Intervals in a permutation

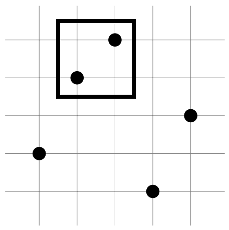

(b) Plus-decomposable permutation

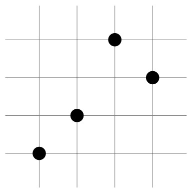

(c) Minus-decomposable permutation

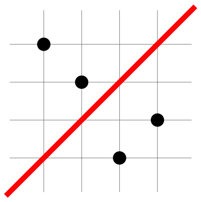

(d) Simple permutation

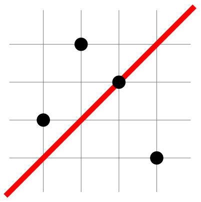

(e) Block-decomposition of a permutation

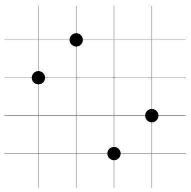

(f) Block-wise simple permutation


(g) Not block-wise simple permutation


(h) Involution


(i) Derangement


(j) Non-derangement


(k) Parity

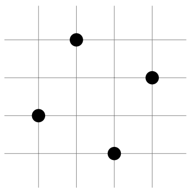

(l) Non-parity

Fig. 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 $\pi(i) = i$ fixed point property.

```
given length : int
find perm : sequence (size length, injective) of int(1..length)

$ the inversion count
find inversionCount : int(0..length**2)  $ We need to each statistic it a reasonable upper bound
such that
    inversionCount =
$ For each occurence of a larger value is found before a smaller one, we add a 1 to a list and we then sum them all up.
        sum([ 1 | i,j : int(1..length), i < j, perm(i) > perm(j)])

$ the descent count
find descentCount : int(0..length)
such that
    descentCount =
$ For each occurence of a descent, we add a 1 to a list and we then sum them all up.
        sum([ 1 | i : int(1..length-1), perm(i) > perm(i+1)])

$ the ascent count
find ascentCount : int(0..length)
such that
    ascentCount =
$ For each occurence of an ascent, we add a 1 to a list and we then sum them all up.
        sum([ 1 | i : int(1..length-1), perm(i) < perm(i+1)])

$ the excedance count
find excedanceCount : int(0..length)
such that
    excedanceCount =
$ For each occurence of an excedance, we add a 1 to a list and we then sum them all up.
        sum([ 1 | i : int(1..length), perm(i) > i])

$ the majorIndex
find majorIndex : int(0..length**2)
such that
    majorIndex =
$ We follow the sum by the definition of Major Index.
        sum([ i | i : int(1..length-1), perm(i) > perm(i+1) ])
```

Listing 18: Essence code which represents all statistics

As mentioned before, the statistics can be turned into properties of a permutation, e.g. rather than checking how many descents a permutation has, we ask for a permutation with a given number of descents. In Listing 19 we see the statistic of inversions of a permutation turned into a constraint model that looks for permutations with a given number of inversions.

```
given length : int
given nInversions : int
find perm : sequence (size length, injective) of int(1..length)

such that
$ The given number of inversions is equal to the number of inversions found in the permutation
    nInversions =
        sum([ 1 | i,j : int(1..length), i < j, perm(i) > perm(j)])
```

Listing 19: Essence code which represents permutations with a given number of inversions

4 Composability in Action

We demonstrate the composability of the models using an illustrative (albeit hypothetical) example scenarios and one example which extends the enumeration of solutions to a conjecture in Claesson et al. (2012). The latter model gives rise to 2 new conjectures.

4.1 Hypothetical Example

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 (OEIS Foundation Inc., 2023, 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. To illustrate how the CP approach easily cuts down the number of solutions without having to go through generating permutations and then filtering them we have split this example into 4 steps, in each step we add more constraints (patterns and/or properties). We then show how many solution permutations are found at each step in Table 2, with **step 4** containing all properties (which is the full model for our example problem). As the permutation pattern researcher we would only work with **step 4**, rather than illustrative which contain fewer constraints.

In the context of exploring permutations that traditionally avoid the permutation 1324, let us assume 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 want to focus on permutations which avoid this pattern (**step 1**, $\mathrm{Av}(\{1324,(213,\{(0,0),(0,1),(1,0),(1,0)\})\})$). Consider that we decide to look at permutations which contain both the classic pattern 132 and the mesh pattern $(123, \{(1,2),(2,1),(1,3),(3,1)\})$ (**step 2**, $\mathrm{Co}(\{132,(123,\{(1,2),(2,1),(1,3),(3,1)\})\})$).

We instantiate a constraint programming model that combines the four patterns above. We can also add properties, for example: minus-decomposable and involution (**step 3**, minus-decomposable; **step 4**, 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 (**step 1**). 2841 of these permutations then contain the other two patterns (**step 2**). 1865 of these are minus-decomposable (**step 3**) and finally only 19 are also involutions (**step 4**). 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. This table demonstrates that this technique is more efficient than generate-and-test based methods, as the numbers between the columns/steps decrease rapidly. For some permutation lengths (14–16), generate-and-test is not feasible because earlier steps cannot be fully enumerated within the 1-hour time limit we gave to Minion (a CP solver) Gent et al. (2006), but the combination of all properties can be enumerated.

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.

4.2 1324-avoiding permutations with a fixed number of inversions

Let us now look at the impact the constraint programming approach has on a combination of property type and property that was investigated by Claesson et al. (2012). We will focus on the conjecture 13 stated in Claesson et al. (2012). This involves combining the 1324 classical pattern avoidance with a fixed number of inversions.

Length	Number of permutations found			
	Step 1	Step 2	Step 3	Step 4
5	42	8	2	0
6	132	41	19	1
7	429	180	102	2
8	1,430	730	455	9
9	4,862	2,841	1,865	19
10	16,796	10,815	7,321	53
11	58,786	40,700	28,096	106
12	208,012	152,325	106,555	255
13	742,900	568,883	401,729	493
14	-	-	-	1,118
15	-	-	-	2,120
16	-	-	-	4,664

Tab. 2: Running the 4 steps on various permutation lengths. Dash indicates timeout after 1-hour of Minion search

Conjecture 1. *Claesson et al. (2012)*

For all nonnegative integers n and k, we have $S_n^k(1324) \leq S_{n+1}^k(1324)$ (where $S_n^k(1324)$ is the set of permutations of length n with k inversions and which avoid 1324).

For example, the set $S_5^9(1324) = \{45321, 53421, 54231, 54312\}$ is the set of permutations of length 5 with each with 9 inversions and avoiding 1324 classically.

In the paper Claesson et al. (2012) present results of an exhaustive enumeration algorithm, which found results of all inversions for permutations up to length 15. Unfortunately, the authors do not disclose how the algorithm worked, but we hypothesise that it might have been a generate and test approach. Which first generates all 1324 avoiding permutations and then filtered for the number of inversions.

As discussed before the constraint programming approach avoids this and thus we have been able to expand the enumeration further, and give additional insights into the conjecture. Further, there was no need for a specialised algorithm or coding to test this conjecture, we only had to combine our previously created model pieces for classical avoidance and counting inversions. While we are unable to fully prove the conjecture, our results demonstrate further support for the result.

Looking at Table 3 (this is a pattern that can be observed in Claesson et al. (2012) as well) there is a point at which the number of permutations per inversion count does not change. In other words, there is a point at which it does not matter what the length (n) of the permutation is the number of permutations which avoid 1324 and have a given number of inversion is the same as $n + 1$. To confirm this we have expanded the calculations to the permutations of length 16 and all possible number of inversions. Further, we are able to strongly suggest at which point this enumeration does not change, up to 20 inversions.

We enumerated the permutations for this problem for permutation lengths from 1 to 25 and for the number of inversions ranging from 0 to 20. We used a large compute server with 256 cores and 1TB of memory for this task. Each enumeration is allowed to use up to 250 cores on this machine. The longest running instance (length 23 and number of inversions 20) took just over 5 days in this parallel computing setting, which is the equivalent of running the same enumeration on a single core for 3.5 years. This parallel setting highlights a significant advantage of employing CP: the feasibility of leveraging high-performance, parallel CP solvers, such as Minion, which can take advantage of the widely accessible multi-core machine without requiring any special development.

Table 3 only contains some of these results, the full enumeration of up to length 16 (over all inversions) and up to length 23 (up to 20 inversions) can be found in our repository Akgün et al. (2023).

Conjecture 2. *For a fixed integer k, if $n > k + 2$, then $|S_n^k(1324)| = |S_{k+2}^k(1324)|$.*

Further we have evidence that there is an identifiable sequence which the stabilising points create.

Conjecture 3. *Let n be a given length of permutations, then $S_n^k(1324)$ for $k \in \{0, \dots, n-2\}$ will be the first $n-1$ entries of (OEIS Foundation Inc., 2023, A000712).*

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 by applying it to 2 examples, one of which demonstrates how the constraint programming approach avoids having to generate a large number of permutations before filtering for the ones based on the additional properties/constraints.

In the second example this paper demonstrated the utility of the library of constraint models by extending the computational results from Claesson et al. (2012), to find more evidence towards their conjecture as well as identifying 2 new conjectures. This allowed us to leverage the parallel support of modern CP solvers to solve in 5 days what would have taken 3.5 years on a single CPU core.

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.

Length	Number of permutations found with a fixed number of inversions																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1																				
2	1	1																			
3	1	2	2	1																	
4	1	2	5	6	5	3	1														
5	1	2	5	10	16	20	20	15	9	4	1										
6	1	2	5	10	20	32	51	67	79	80	68	49	29	14	5	1					
7	1	2	5	10	20	36	61	96	148	208	268	321	351	347	308	241	165	98	49	20	6
8	1	2	5	10	20	36	65	106	171	262	397	568	784	1019	1264	1478	1628	1681	1619	1441	1173
9	1	2	5	10	20	36	65	110	181	286	443	664	985	1416	1988	2715	3589	4579	5631	6654	7559
10	1	2	5	10	20	36	65	110	185	296	467	714	1077	1582	2305	3284	4617	6374	8665	11521	15012
11	1	2	5	10	20	36	65	110	185	300	477	738	1127	1682	2477	3584	5134	7240	10100	13915	18976
12	1	2	5	10	20	36	65	110	185	300	481	748	1151	1732	2577	3768	5450	7766	10976	15312	21171
13	1	2	5	10	20	36	65	110	185	300	481	752	1161	1756	2627	3868	5634	8098	11526	16216	22632
14	1	2	5	10	20	36	65	110	185	300	481	752	1165	1766	2651	3918	5734	8282	11858	16786	23568
15	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2661	3942	5784	8382	12042	17118	24138
16	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2665	3952	5808	8432	12142	17302	24470
17	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2665	3956	5818	8456	12192	17402	24654
18	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2665	3956	5822	8466	12216	17452	24754
19	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2665	3956	5822	8470	12226	17476	24804
20	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2665	3956	5822	8470	12230	17486	24828
21	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2665	3956	5822	8470	12230	17490	24838
22	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2665	3956	5822	8470	12230	17490	24842
23	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2665	3956	5822	8470	12230	17490	24842
A000712	1	2	5	10	20	36	65	110	185	300	481	752	1165	1770	2665	3956	5822	8470	12230	17490	24842

Tab. 3: The enumeration of permutations avoiding 1324 classically and containing a fixed number of inversions. The rows are the length of the permutation, the columns the number of inversions. The full results can be found in the supplementary repository Akgün et al. (2023). The last row shows the first 21 elements of the sequence A000712 (OEIS Foundation Inc., 2023, A000712).

References

Ö. Akgün, A. M. Frisch, I. P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022.

Ö. Akgün, M. Mereb, and L. Vendramin. Enumeration of set-theoretic solutions to the Yang-Baxter equation. *Math. Comput.*, 91(335):1469–1481, 2022. URL `https://doi.org/10.1090/mcom/3696`.

Ö. Akgün, R. Hoffmann, and C. Jefferson. stacs-cp/composable-permutation-patterns, Nov. 2023. URL `https://doi.org/10.5281/zenodo.10215929`.

M. Albert. PermLab: Software for permutation patterns, 2012. URL `https://github.com/mchllbrt/PermCode`.

M. Albert, S. Linton, and R. Hoffmann. PatternClass–Permutation Pattern Classes, 2012. URL `https://gap-packages.github.io/PatternClass/`.

R. P. Ardal, T. K. Magnusson, Émile Nadeau, B. J. Kristinsson, B. A. Gudmundsson, C. Bean, H. Ulfarsson, J. S. Eliasson, M. Tannock, A. B. Bjarnason, J. Pantone, and A. B. Arnarson. Permuta, Apr. 2021. URL `https://doi.org/10.5281/zenodo.4725759`.

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

S. Avgustinovich, S. Kitaev, and A. Valyuzhenich. Avoidance of boxed mesh patterns on permutations. *Discrete Applied Mathematics*, 161(1-2):43–51, Jan. 2013. ISSN 0166-218X. URL `https://doi.org/10.1016/j.dam.2012.08.015`.

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

E. Bagno, E. Eisenberg, S. Reches, and M. Sigron. Blockwise simple permutations, 2023.

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

P. Brändén and A. Claesson. Mesh Patterns and the Expansion of Permutation Statistics as Sums of Permutation Patterns. *The Electronic Journal of Combinatorics*, pages P5–P5, 2011.

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

G. Chu, P. J. Stuckey, A. Schutt, T. Ehlers, G. Gange, and K. Francis. Chuffed, a lazy clause generation solver, 2018. URL `https://github.com/chuffed/chuffed`.

A. Claesson, V. Jelínek, and E. 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. ISSN 0097-3165. URL `https://www.sciencedirect.com/science/article/pii/S0097316512000891`.

A. Distler, C. Jefferson, T. Kelsey, and L. Kotthoff. The Semigroups of Order 10. In M. Milano, editor, *Principles and Practice of Constraint Programming*, pages 883–899, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33558-7.

N. Eén and N. Sörensson. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

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

I. P. Gent, C. Jefferson, and I. Miguel. Minion: A Fast Scalable Constraint Solver. In *ECAI*, pages 98–102, 2006.

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

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

A. Marcus and G. Tardos. Excluded permutation matrices and the Stanley–Wilf conjecture. *Journal of Combinatorial Theory, Series A*, 107(1):153–160, 2004.

É. Nadeau, C. Bean, H. Ulfarsson, J. S. Eliasson, and J. Pantone. Combinatorial Specification Searcher (PermutaTriangle/comb_spec_searcher): Version 4.0.0, June 2021. URL `https://doi.org/10.5281/zenodo.4946832`.

P. Nightingale, Ö. Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017.

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

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