Identifying Combinatorial Structures for Binary Strings and Set Partitions

Sigurður Helgason   James Robb

Final Report in Computer Science B.Sc.

2018

Name: Sigurður Helgason   James Robb
Kennitala:
Primary Advisor: Christian Bean

School of Computer Science
Tölvunarfræðideild

Heiti verkefnis:

Identifying Combinatorial Structures for Binary Strings and Set Partitions

Námsbraut:

Computer Science B.Sc.

Tegund verkefnis:

Final Report in Computer Science B.Sc.

Önn:

2018–1

Námskeið:

T LOKA 404

Ágrip:

The goal of enumerative combinatorics is to count the number of some well described objects. We extend the algorithm of Bean et al. that automates this process to work with binary strings and set partitions. This involves teaching the computer about binary strings, set partitions, and the ideas behind the proofs of results in this area.

Höfundur:

Sigurður Helgason
James Robb

Umsjónarkennari:

Christian Bean

Leiðbeinandi:

Henning Ulfarsson

Fyrirtæki/stofnun:

Reykjavik University
Menntavegi 1
101 Reykjavik

Dagsetning:

10.05.2018

Lykilorð íslensk:

fléttufræði, bitastrengir, mengisskiptingar

Lykilorð ensk:

combinatroics, binary strings, set partitions

Dreifing:

opin [✓ X] **lokuð** ☐ **til:**

Acknowledgements

We wish to extended our deepest thanks and gratitude to Christian Bean and Henning Ulfarsson for their unwavering support throughout the duration of this work and throughout our years of study. Without them we would not have been able to enjoy the level of success we've achieved thus far. We would also like to thank the entire computer science department at Reykjavik University for their continued support in our studies and development as computer scientists.

Contents

Chapter 1

Introduction

The field of automatic discovery of combinatorial specifications is rather young and no general method has been implemented. The ECO method [1] can in theory be applied to any subfield of combinatorics but has not been implemented. Our advisors along with a group of researchers have recently implemented a general purpose algorithm, `CombSpecSearcher` [2], for constructing combinatorial specifications. However it relies on the user to write domain-specific strategies to be useful in a particular field. Currently strategies for the avoidance of classical permutation patterns are the only existing strategies.

In this report we explore strategies for other combinatorial objects. In Section 2.1 we develop strategies to find combinatorial specifications for binary strings avoiding consecutive sequences. In Section 2.2 we develop strategies to find combinatorial specifications for pattern avoiding set partitions.

We expand on the work of past Reykjavik University students on the PermPal system [3], which can visualize automated proofs for the class of permutations avoiding some pattern. Our work allows for the visualization of the automated proofs produced by `CombSpecSearcher` for binary strings, set partitions, and permutation patterns. We call the system ComboPal [4], and it will be discussed in greater detail in Section 3.1.

1.1 CombSpecSearcher

`CombSpecSearcher` is a system created by Bean *et al.* [2] at Reykjavik University. The `CombSpecSearcher` system utilizes a series of strategies in order to find a combinatorial specification of a combinatorial class. The `CombSpecSearcher` produces these combinatorial specifications in the form of a *proof tree*. The combinatorial specification can be translated to a system of equations which can be solved to yield a generating function which enumerates the class of the combinatorial specification.

1.1.1 Strategies

There are four types of strategies used by `CombSpecSearcher`:

- Batch strategies which split a combinatorial class into disjoint sub-classes of that combinatorial class.
- Inferral strategies which infer information about a combinatorial class based on the context of the class.
- Verification Strategies which verify if a combinatorial class is directly countable. It is necessary for `CombSpecSearcher` to determine whether or not a combinatorial class

is equal to the empty set $\emptyset$, a strategy which achieves this is considered a verification strategy.

- Decomposition strategies which convert a combinatorial class into multiple combinatorial classes that together represent the original class.

These strategies all represent somehow breaking apart or gaining new information on a given combinatorial class. When searching for a combinatorial specification, these strategies represent different mathematical operations in the system of equations within the combinatorial specification.

1.1.2 CombSpecSearcher Results

`CombSpecSearcher` is initialized with some initial combinatorial class c and a set $C = \{c\}$. `CombSpecSearcher` applies a series of strategies to all combinatorial classes in C. These strategies return additional combinatorial classes, and those additional combinatorial classes are added to C. `CombSpecSearcher` repeats this process until a valid combinatorial specification has been found.

A valid combinatorial specification is a combinatorial specification where every node either appears once internally or is verified. Once a combinatorial specification is found for the initial combinatorial class, `CombSpecSearcher` returns it.

1.1.3 Combinatorial Specification Example

Figure 1.1 is an example of a `CombSpecSearcher` proof tree for a combinatorial class which we discuss in-depth in Section 2.1. The tree represents the combinatorial specification of all binary strings that don't contain 10. Notice that every leaf node is either verified or recurses to a node seen elsewhere in the tree.

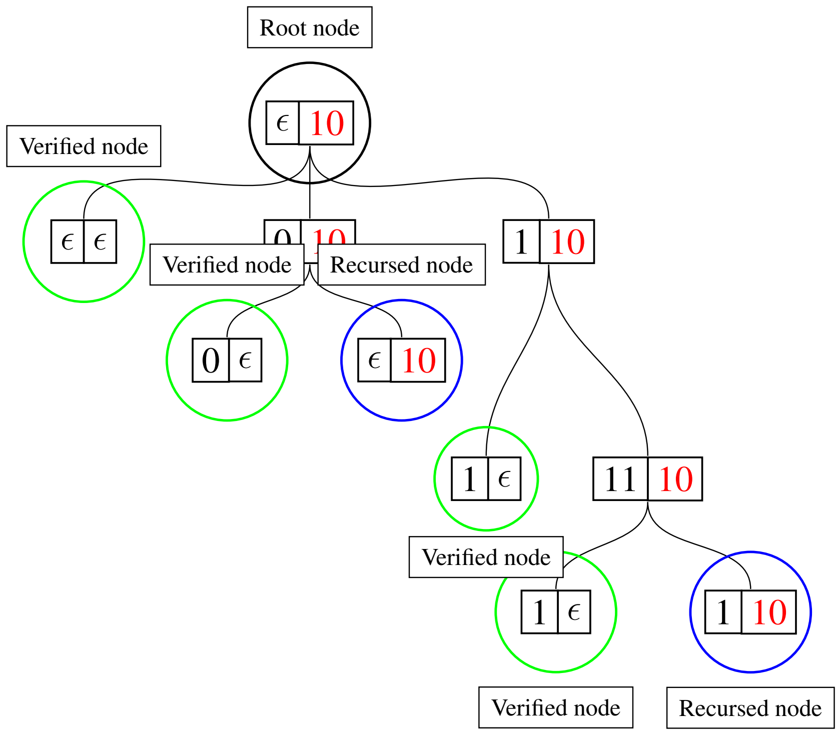

Figure 1.1: Example of a proof tree from `CombSpecSearcher`

Chapter 2

Methods

2.1 Binary Strings

2.1.1 Defintions and Binary Strings

In this section we introduce strategies used by CombSpecSearcher for classes of binary strings that avoid one or more patterns of consecutive sequences. To describe this we must first introduce a few definitions:

Definition 2.1.1. We say that $\mathcal{B}$ represents the set of all binary strings.

Definition 2.1.2. Let $\sigma, \tau \in \mathcal{B}$. We say σ *contains* τ if τ is a consecutive substring of σ. Conversely, we say that a binary string σ *avoids* a binary string τ if σ does not contain τ. If U is a set of binary strings, we say σ avoids U if σ *avoids* all binary strings in U.

For example the binary string 1001 avoids the binary string 101 but it contains the binary string 100.

Definition 2.1.3. Let $U \subseteq \mathcal{B}$. We define $Av(U) = \{s \mid s \in \mathcal{B}, s \text{ avoids } \forall u \in U\}$ as the *class* of binary strings that avoid the set U. If U is minimal with respect to containment we call U the *basis* of the class $Av(U)$.

Definition 2.1.4. Let $p \in \mathcal{B}, U \subseteq \mathcal{B}$. We define a *binary string descriptor* (BSD) as $BSD(p, U) = \{ps \mid s \in Av(U)\}$. That is, a BSD represents a class of binary strings where each string starts with the *prefix* p followed by a string from the class $Av(U)$.

As an example consider B = BSD(10, $U = \{11\}$). It can be seen that $1 \in Av(U)$, and as such the binary string $101 \in B$.

We introduce a visual representation for BSD B = BSD(p, U). The visual representation takes the form of two blocks. The first block depicts the prefix of the BSD and the second block depicts the strings that make up the basis of B. The basis block is drawn in red. It is often the case that the $Av(U) = \varnothing$, in that case instead of displaying the basis strings in the second block we draw an ϵ. Examples of this visual representation can be see in Figure 2.1.

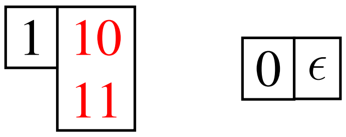

Figure 2.1: On the left a BSD(1,$\{10, 11\}$) is displayed to the right a BSD(0,$\{0, 1\}$)

2.1.2 Strategies

We now introduce strategies `CombSpecSearcher` can apply to BSDs to find a combinatorial specification for a given BSD. We employ four strategies:

Strategy (Expansion)**.** *Expansion* is a batch strategy that is applied to BinaryStringDescriptors. Given a BSD B_0 = BSD(p,U), three new BSDs $B_1, B_2,$ and B_3 are produced. They are defined as such:

- $B_1 = BSD(p, \{1, 0\})$. That is, B_1 is has the same prefix as B_0 and its basis represents the class that only contains the empty string.
- $B_2 = BSD(p0, U)$. That is B_2 has the same prefix as B_0 appended with a 0.
- $B_3 = BSD(p1, U)$. That is B_3 has the same prefix as B_0 appended with a 1.

For example applying the expansion strategy to BSD(ϵ, $\{101\}$) produce three new BSD as seen in Figure 2.2. If the expansion strategy is called on a BSD with prefix ϵ and then on the produced BSD, and so on n-times, it will construct all BSDs whose prefixes are the binary strings of length less than or equal to n.

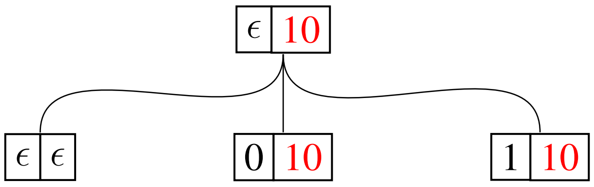

Figure 2.2: Expansion strategy being applied to BSD with prefix ϵ and basis 10

Strategy (Is literal)**.** *Is literal* is a verification strategy that is applied to BinaryStringDescriptors. Given a BSD(p, U), if $U = \{0, 1\}$ (represents the set containing only ϵ) we verify that we can directly enumerate this class of binary strings. Since this class represents $\{p\}$ the direct enumeration of this class is 1.

Strategy (Is empty)**.** *Is empty* is a verification strategy that is applied to BinaryStringDescriptors. Given a BSD B = BSD(p, U) if p does not avoid U, then we verify that this BSD doesn't contribute to the combinatorial specification of the class. We call BSD's which are verified by this strategy *empty* BSD's.

Consider BSD(110, $\{11, 001\}$) and BSD(101, $\{111\}$), the two BSD displayed in Figure 2.3. The prefix of the BSD on the left contains 11 which is a string in its basis, and is therefore empty. While the BSD on the right will not be verified by the is empty strategy.

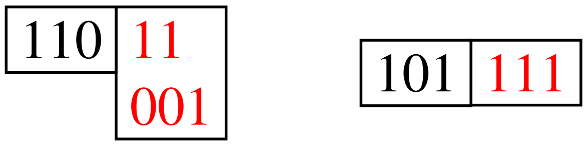

Figure 2.3: Is empty strategy being applied to BSD's

Definition 2.1.5. For a BSD(p, U), Let p_{suffix} be the longest suffix of p which is a prefix of some $u \in U$, and let p_{prefix} be the characters in p before p_{suffix}.

Theorem 2.1.1. Let B = BSD(p, U) be a non-empty BSD, then B = BSD(p_{prefix}, $\{0, 1\}$) $\times$ BSD(p_{suffix}, U).

Proof. Let B = BSD(p, U) be a non-empty BSD, i.e, the set of all binary strings which start with p and end with a binary string avoiding U. In B, we have that p_{prefix} is some amount of characters, possibly zero, which don't contribute to the creation of a pattern in U by definition. Therefore, p_{suffix} is the only portion of p which could possibly construct a pattern in U. Let B_1 = BSD(p_{prefix}, $\{0, 1\}$) = $\{p_{prefix}\}$, and B_2 = BSD(p_{suffix}, U). In order to construct B from B_1 and B_2 we can prepend p_{prefix} to all strings in B_2. This is done by cartesian product. □

For example B = BSD($p = 1$, $U = \{0\}$) is the set consisting of the binary strings $\{1, 11, 111, 1111, \ldots\}$. We see that $p_{prefix} = 1$ and $p_{suffix} = \epsilon$. Construct B_1 = BSD(p_{prefix}, $\{0, 1\}$) = $\{1\}$ and B_2 = BSD(p_{suffix}, U) = $\{\epsilon, 1, 11, 111, \ldots\}$, as in Theorem 2.1.1. If we prepend all elements in B_1 to all elements in B_2 we get $\{1, 11, 111, \ldots\}$ or B. So $B_1 \times B_2 \cong B$.

Strategy (Splitting). *Splitting* is a decomposition strategy that is applied to BinaryStringDescriptors. Given a B = BSD(p, U), the splitting strategy produces two BSD's. It splits B into the BSD's B_1 = BSD(p_{prefix}, $\{0, 1\}$) and B_2 = BSD(p_{suffix}, U). If $p_{prefix} = \epsilon$ the splitting strategy doesn't produce any results.

For example, BSD($p = 01$, $U = \{10\}$) has $p_{prefix} = 0$ and $p_{suffix} = 1$. As $p_{prefix} \neq \epsilon$ the splitting strategy produces two BSD's which can be seen in Figure 2.4.

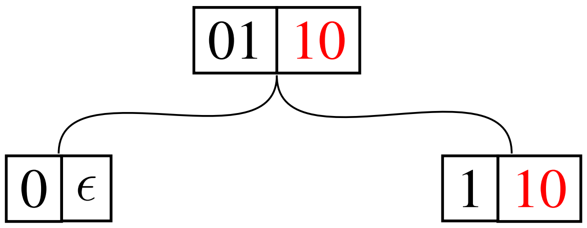

Figure 2.4: Strategy decomposition being applied to BSD(01, $\{10\}$)

2.1.3 Enumerating Binary Strings

To be able to enumerate an avoidance class of binary strings, we must solve the system of equations that result from the production of the combinatorial specification. To solve the system of equations one must consider each node as either directly countable (verified), or the node represents an equation where the operands of the equation are the equations of its children.

The mathematical operation applied to the operands is selected based on the strategy used to produce the children.

- Expansion strategy: The produced BSD's are a disjoint union of their parent BSD, we calculate the disjoint union by addition.
- Splitting strategy: The produced BSD's are representative of the same object and the operation is a multiplication.
- Is Literal: The BSD's verified by the is literal strategy return the generating function for counting a single item of a given length. For a binary string of length n, the resulting generating function is x^n.
- Is Empty: They represent an empty set so they return 0.

Proposition 2.1.1. Running `CombSpecSearcher` with these strategies on a BinaryStringDescriptor is guaranteed to halt, with a valid combinatorial specification.

Proof. Let $B =$ BSD(p, U) be a non-empty BSD, let l be the maximum length of the patterns in U and call the longest pattern σ. If the length of p is greater than l then the splitting strategy could split p, as we've asserted that σ is the longest pattern in U. Also, p_{suffix} has to be shorter than σ, otherwise p would contain σ. There are a finite amount of binary strings of length less than or equal to l. Using the expansion strategy all BSD's with prefix of length less than or equal to l are constructed. Constructing all BSD's of length less than or equal to l will result in a tree where each of the leaf nodes are either verified by Is Empty, Is Literal, or they have recursed. That represents a valid combinatorial specification. □

Proposition 2.1.2. The resulting combinatorial specification can be converted to be represented by a Non-deterministic finite automata (NFA).

Proof. We take a step by step approach to construct an NFA describing the combinatorial specification.

- Create an accepting state in an NFA for every non literal BSD in the combinatorial specification.
- The initial state in the NFA corresponds to the root of the combinatorial specification.
- If the expansion strategy is applied to a BSD, construct edges from the expanded object to the products labelled by the corresponding appended character.
- If the splitting strategy is applied to a BSD, construct an ϵ-transition to the non-literal BSD that was produced.
- If a BSD B_1 produces a BSD which recurses to a BSD B_2, construct an ϵ-transition from B_1 to B_2

□

Consider the combinatorial specification for the class of binary strings avoiding 10. We begin by looking at the resulting combinatorial specification as a tree, seen in Figure 2.5. Note that the BSD's which have been verified by the is empty strategy do not contribute to the enumeration of this class and are not present in the tree. For this reason they are not required for the NFA conversion.

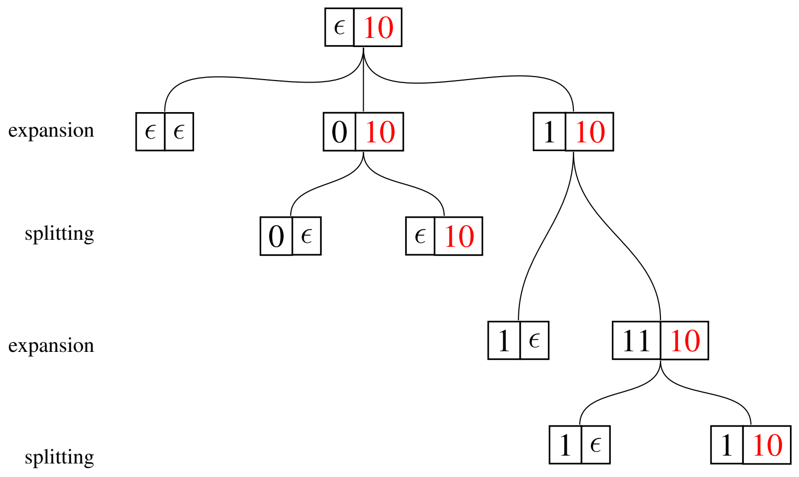

Figure 2.5: Visual representation of the combinatorial specification for binary strings that avoid 10.

The resulting NFA after converting the combinatorial specification in Figure 2.5 can be seen in Figure 2.6.

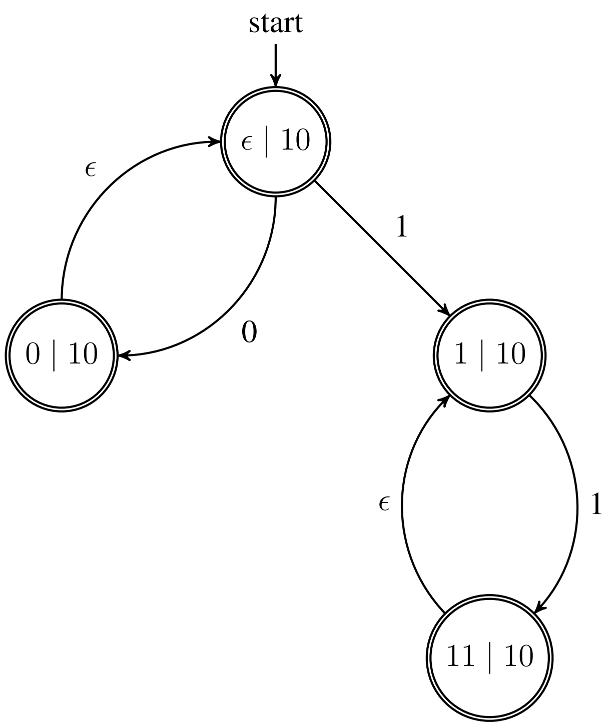

Figure 2.6: NFA representing the combinatorial specification shown in Figure 2.5

Proposition 2.1.3. The combinatorial specification represents a system of equations that can be solved to construct a rational generating function.

Proof. We know a combinatorial specification for an avoidance class of binary strings has a representation as an NFA, and as such it has a representation as a deterministic-finite automata (DFA). It is a well known property that the language of a DFA has a rational generating function [5]. Therefore each avoidance class of binary strings has a rational generating function that enumerates the class. □

2.2 Set Partitions

2.2.1 Definitions and Partition Tilings

To begin to examine set partition in the context of combinatorial specification we utilize the definitions of a set partition, standard form, canonical standard form, avoidance, and patterns from Vít Jelínek et al [6]. We repeat them here before beginning our analysis.

Definition 2.2.1. For a given integer $n \geqslant 1$ we say a *set partition* of $S_n = \{1, 2, \ldots, n\}$ is a set of non-empty subsets called *blocks* of S_n that are pairwise disjoint and their union is S_n. We allow $n = 0$ and in such case $S_n = \varnothing$. A partition X of S_n with k-blocks is called a k-partition. We denote the set of all k-partitions of S_n by $P_{n,k}$.

Definition 2.2.2. A set partition $X \in P_{n,k}$ is said to be in *standard form* if it is written $X = X_1/X_2/\ldots/X_k$ and arranged such that $min(X_1) < min(X_2) < \ldots < min(X_k)$ and each subset $X_i \in X, 1 \leqslant i \leqslant k$ is ordered in ascending order by its elements. The same set partition X can equivalently be written in *canonical standard form* as $\pi = \pi_1\pi_2 \cdots \pi_n$ where $\pi_i = j$ such that $i \in X_j, 1 \leqslant j \leqslant n$.

For example, consider the set $S_5 = \{1, 2, 3, 4, 5\}$ and the set partition $X = 1, 3/2, 5/4$ which is written in standard form. The canonical standard form equivalent of X is $\pi = 12132$. This is because we see 1 and 3 in the first block, 2 and 5 in the second block, and finally 4 in the third block.

Throughout this report we will consider set partitions in standard canonical form (when not represented visually) as they provide for simple pattern occurrence checking. This is of great interest to us as we are primarily concerned with describing classes of set partitions that avoid a given pattern.

Definition 2.2.3. Let $\sigma = \sigma_1\sigma_2 \cdots \sigma_n$ and $\tau = \tau_1\tau_2 \cdots \tau_m$ be set partitions in canonical standard form. We say an *occurrence* of τ exists in σ if there exists a subsequence in σ that is order-isomorphic to τ; that is, σ contains a subsequence $\sigma_{f(1)}, \sigma_{f(2)}, \ldots, \sigma_{f(m)}$ where $1 \leqslant f(1) \leqslant f(2) \leqslant \ldots \leqslant f(m) \leqslant n$ such that for each $i, j \in S_m$ we have $\sigma_{f(i)} < \sigma_{f(j)}$ if and only if $\tau_i < \tau_j$ and $\sigma_{f(i)} > \sigma_{f(j)}$ if and only if $\tau_i > \tau_j$. Otherwise, we say σ *avoids* τ. In this context we refer to τ as a *pattern*.

Consider the partition $\sigma = 1123$. It contains three subsequences which are order-isomorphic to the pattern 12. They are $\sigma_2\sigma_3 = 12$, $\sigma_3\sigma_4 = 23$, and $\sigma_1\sigma_4 = 13$. In this instance σ does not avoid the pattern 12.

We introduce a visual representation for set partitions that takes the form of a grid. For each block in a set partition a corresponding column exists in the grid. Elements in the blocks are then placed as points on the grid where their x-coordinate corresponds to the block they are in, and their y-value corresponds to their value in canonical form. For example the set partition $\sigma = 121$ (or $1, 3/2$ in standard form) is depicted in Figure 2.7.

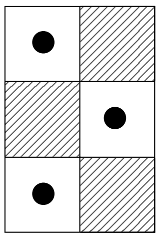

Figure 2.7: Visual representation of the set partition 121.

In examining set partitions we are concerned with describing combinatorial specifications for classes of set partitions that avoid one or more patterns.

Definition 2.2.4. Let P be the set of all set partitions and $B \subseteq P$. We define $Av(B) = \{p \mid p \in P \text{ such that } \forall b \in B \text{ we have that } p \text{ avoids } b\}$ as the class of set partitions that avoids B. We call B the *basis* of the class $Av(B)$ if B is minimal with respect to containment; that is, no pattern in B contains an occurrence of another.

We expand the idea of the visual representation introduced for set partitions to accommodate searching for a combinatorial specification using `CombSpecSearcher` by introducing the idea of a *partition tiling*. This acts both as an abstract notation of a class of set of partitions and as a data structure used in code.

Definition 2.2.5 (Partition Tiling)**.** Let M be a matrix whose entries are from $\{\bullet, \vdots, \mathcal{B}, s\}$ with a barrier separating the columns at index l. Given a set partition X represented visually, an M-gridding of X is a set of vertical and horizontal lines such that the grid produced by (and consisting only of) the drawing of these lines on X produces a grid with the same dimensions as M. The cells in M are all one of the following:

- Point cell represented by $\bullet$. After drawing gridding lines there must only be a single point in the cell.
- Unfinished cell represented by $\vdots$. Points placed in this cell must all be in the same block.
- Basis cell represented by $\mathcal{B}$. Points placed in this cell must avoid $\mathcal{B}$. When a column in the gridding has multiple basis cells the column as a whole must also avoid $\mathcal{B}$.
- Shaded cell represented by s. This cell must remain empty in the gridding.

Let $T(M)$ denote the set of set partitions with a gridding on M. We abuse notation and use T as a shorthand for $T(M)$ and call it a *partition tiling*. The columns to the left of index l in M are called *concrete blocks* of the partition tiling, while the columns to the right are called *abstract columns* of the partition tiling. Concrete blocks may only contain point cells, unfinished cells, and shaded cells. Abstract columns may only contain basis cells and shaded cells.

A partition tiling T splits the visual representation of a set partition into concrete blocks and abstract columns by its vertical barrier. The concrete blocks are blocks that will definitely appear in each set partition the partition tiling could possibly represent. Point cells represent a point that must appear in every set partition in T while unfinished cells represent zero or more points. Basis cells represent (and could be replaced by) any set partition in $Av(B)$. We use a thick black line to denote the vertical barrier between concrete blocks and abstract columns.

For example, let us consider the partition tiling in Figure 2.8 that depicts the partition tiling with the basis $B = \{121\}$. We assert that that the unfinished cell contains no points and we insert a set partition from $Av(B)$ into the basis cell. After drawing the blue lines on the resulting set partition we can see that it has a gridding on T and therefore is in T.

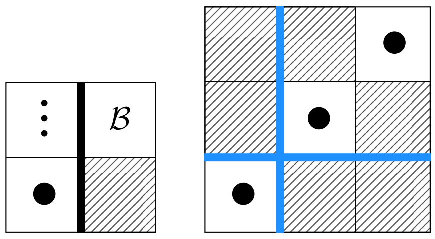

Figure 2.8: The partition tiling T with basis $B = \{121\}$ and a set partition with a gridding on T. The gridding lines are highlighted in blue.

Let us modify the partition tiling in Figure 2.8 by placing a point in the unfinished cell (and therefore claiming there are definitely two points in the first block), and asserting it is the next highest point in that block. The resulting partition tiling seen in Figure 2.9 has two basis cells. This is because when placing another point it can be inserted between any of the points in the basis cell. The resulting partition tiling reflects all possible interleavings of the newly placed point. To illustrate the idea of point interleaving over multiple basis cells, we insert the partition 122 into the partition tiling seen in Figure 2.10. All of the possible results from this insertion are seen the figure.

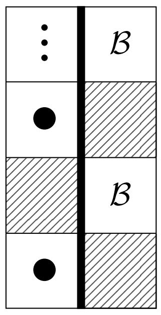

Figure 2.9: The partition tiling resulting from inserting a point into the unfinished cell in Figure 2.8.

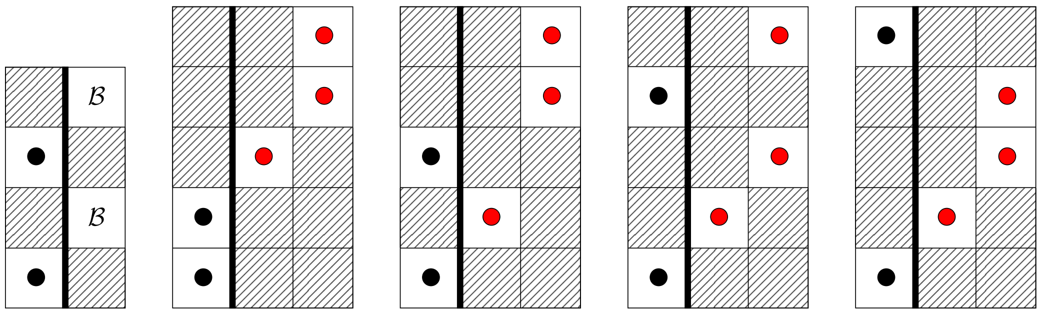

Figure 2.10: A partition tiling with all possible ways to insert the set partition 122 (highlighted in red) into its basis cells

2.2.2 Strategies

The partition tiling notation and data structure provides a sufficient means of describing set partitions in an abstract manner, but before combinatorial specifications can be found for avoidance classes of set partitions a set of strategies for `CombSpecSearcher` will need to be defined. We begin with column insertion.

Strategy (Column Insertion)**.** *Column insertion* is a batch strategy applied to a partition tiling. Given a partition tiling T, a set of new partition tilings are returned. Two different cases occur during column insertion:

- The partition tiling T is duplicated and the left-most abstract column is removed. This is asserting that there are no more points in that column that could be placed.
- In the other case, we assert that there is at least one point that is contained in the left-most abstract column T. A copy C_0 is taken of the left-most abstract column of T and the basis cells in it are converted into unfinished cells. Then for each unfinished cell u in C_0, a copy of T is made with C_0 inserted as the right-most concrete block and a point inserted into u. For each u in where a point is inserted, the unfinished cells below it are shaded as placing a point also asserts the new point is the minimum point in that block. Note that if placing a point in any u violates the ordering restrictions we place on set partitions, it is also shaded.

From the description of the column insertion strategy we know that it is possible that the strategy will yield multiple objects. In Figure 2.11, we see that we obtain three results. The first result demonstrates the case in where the abstract column is asserted to be empty, while the latter two results demonstrate the possible point placements. Note that we see only one basis cell in the last result. This is due to the fact that we assert the newly placed point in that block is the minimum point in that block, and therefore no points can exist lower than it later in the partition tiling without violation of the ordering we impose on set partitions.

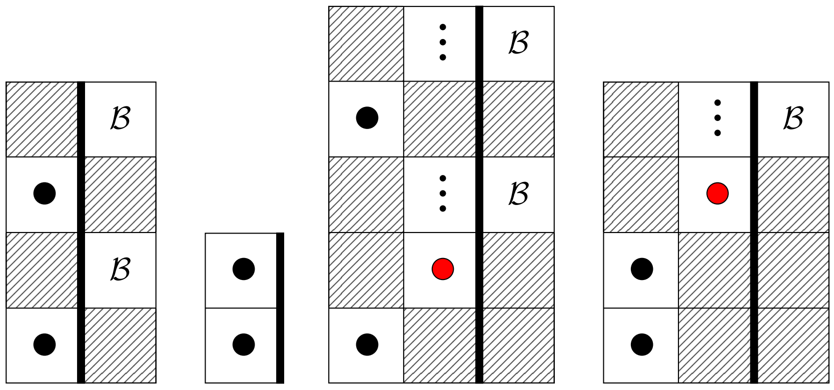

Figure 2.11: A partition tiling followed by the result of applying column insertion to it. New points are highlighted in red.

Strategy (Point Insertion)**.** *Point insertion* is a batch strategy applied to a partition tiling. Given a partition tiling T, a set of new partition tilings is returned. For each unfinished cell u inside the concrete blocks of T, either we shade that cell, or insert a point into it; that is, for each u two new partition tilings are returned. Two variants of this strategy exist; minimum point insertion and maximum point insertion. When placing a point inside an unfinished cell we assert that the newly placed point is either the minimum point that the cell could represent, or the maximum point the cell could represent.

This difference between minimum and maximum point insertion is illustrated in Figure 2.12. We can see that minimum point insertion and maximum point insertion yield two different results. When we use `CombSpecSearcher` to search for a combinatorial specification both variants of point insertion are used simultaneously. Given that we applied two strategies to the partition tiling in Figure 2.12 we expect to see four different results. However, as in both instances we would shade the unfinished cell, we only include that result once.

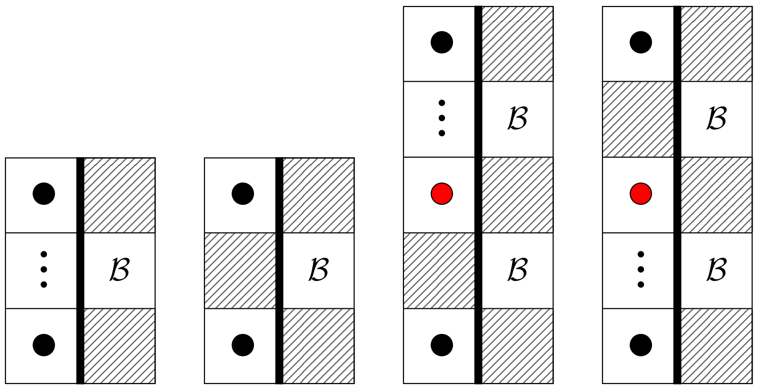

Figure 2.12: A partition tiling with minimum and maximum point insertion applied. First result shades the unfinished cell, the second result inserts a minimum point, and the third inserts a maximum point. New points are highlighted in red.

As described earlier all the leaf nodes in a proof tree need to be verified. Either we can verify a partition tiling by determining that it is not possible for that partition tiling to produce a set partition with an occurrence of a pattern in the basis, or through recursion. To do the former, we need to generate all the set partitions that the partition tiling could produce. As it is not possible generate and check set partitions of all possible lengths, we want an adequate upper bound on the length of the set partitions we will generate. We present an upper bound on this length in the following theorem.

Theorem 2.2.1. Let:

- T be a partition tiling with basis B.
- $L(T) = |\{x \mid x \text{ is a point cell in } T\}| + max(\{|p| \mid p \in B\})$.
- $Av(B)_{\leqslant L(T)}$ be the set partitions up to (and including) length $L(T)$ that avoid B.

If $T \subseteq Av(B)_{\leqslant L(T)}$ then $T \subseteq Av(B)$.

Proof. Assume $T \subseteq Av(B)_{\leqslant L(T)}$ and $T \nsubseteq Av(B)$. Let X be a set partition on T such that $X \notin Av(B)$. Then X contains an occurrence of a pattern in B. As X is a set partition on T, it contains all of the point cells on T. Remove all points from X that are not point cells on T nor a point involved in the occurrence of a basis pattern in X. We call this new set partition X', and then $X' \in T$ but $X' \notin Av(B)$. However, X' has a length of at most $L(T)$ so $T \nsubseteq Av(B)_{\leqslant L(T)}$. This is a contradiction. We have then that $X \in Av(B)$ and that $T \subseteq Av(B)$. □

Given this upper bound on the length of set partition we need to generate for verification, we can introduce the verification strategies *is empty* and *set avoidance*.

Strategy (Is Empty)**.** *Is empty* is a verification strategy applied to a partition tiling. This strategy determines if a partition tiling T represents the empty set by attempting to generate the set partitions of T of length up to $L(T)$. If no valid set partitions are generated then T represents the empty set and we say it *is empty*.

Strategy (Set Avoidance)**.** *Set avoidance* is a verification strategy applied to a partition tiling. A partition tiling T with basis B is verified if it is a subset of the class $Av(B)$. This can be checked using Theorem 2.2.1.

Many of the interesting avoidance classes of set partitions that we could build combinatorial specifications for rely on a recursion being found in the proof tree. If we use batch strategies alone, that is, just insert columns or points, we will not arrive in a situation where a new partition tiling we have created is one we have seen earlier in the tree. To achieve recursion we need to introduce a means of breaking a partition tiling into smaller pieces.

Definition 2.2.6. Let T be a partition tiling, c a cell in T, and X a set partition on T. We define *cell removal*, denoted $X - c$, as removing all points from X that exist in cell c in the gridding of X on T.

To make the concept of cell removal clear, we present an example in Figure 2.13. The shaded rows that normally would be removed are kept as a visual aid.

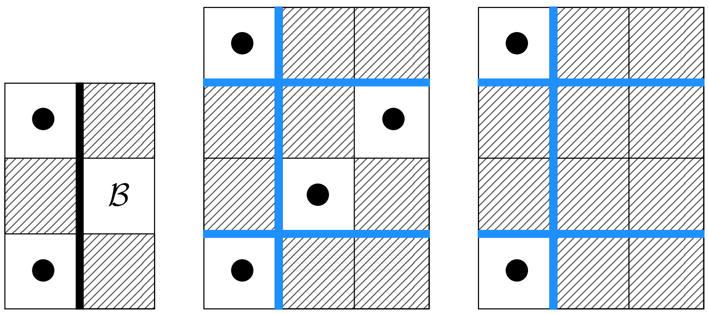

Figure 2.13: A partition tiling T, a set partition $X \in T$, and the resulting set partition from performing cell removal on X with the basis cell in T. The gridding lines are highlighted in blue.

Definition 2.2.7. Let T be a partition tiling with basis B and c a cell in T. We say c is *removable* if $\forall X \in T$ we have $X - c \in Av(B)$ implies $X \in Av(B)$.

Theorem 2.2.2. Let:

- T be a partition tiling with basis B.
- c be a cell in T.
- $L(T) = |\{x \mid x \text{ is a point cell in } T\}| + max(\{|p| \mid p \in B\})$.
- $T_{\leqslant L(T)}$ be the set partitions up to (and including) length $L(T)$ in T.

The cell c is removable if the if the removability condition holds for all $X \in T_{\leqslant L(T)}$.

Proof. If c is removable then the removability conditions holds for all $X \in T_{\leqslant L(T)}$. Assume c is not removable. Then there exists some $X \in T$ such that $X - c$ avoids B, but X does not avoid B. Let σ be an occurrence of a pattern in B, and let X' be the set partition created by removing all points that are not point cells on T nor points involved in the occurrence σ. We have then that $|X'| \leqslant L(T)$, and X' contains σ, but $X' - c$ avoids σ. Therefore, there is always some $X' \in T_{\leqslant L(T)}$ such that the removability conditions fails if c is not removable. □

Strategy (Cell Splitting)**.** *Cell splitting* is a decomposition strategy applied to a partition tiling. For a given partition tiling T, we check if each cell c in T is removable by utilizing Theorem 2.2.2. If c is removable we return two new partition tilings. The first consists only of the cell c, and the second is a copy of T with c shaded.

Let us consider the example seen in Figure 2.14. We can see that the resulting two partition tilings are a single point, and a simpler partition tiling. It is easy to reason that if the pattern being avoided is 122, the second point in the first block would not contribute to any occurrence of the pattern. The benefit now is that the partition tiling containing a single point can be easily verified using the set avoidance strategy, and the other is a simple enough partition tiling that an occurrence of it is likely elsewhere in the proof tree being built, which makes recursion likely.

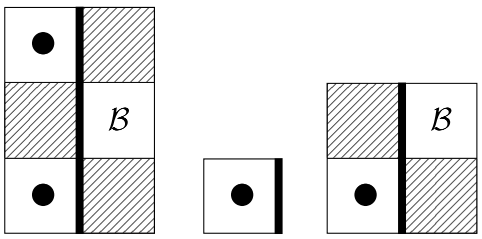

Figure 2.14: The partition tiling T with the basis $\{122\}$ on the left is decomposed via cell splitting into the two partition tilings on the right.

The last set of strategies we will cover are inferral strategies. These are strategies used to infer information about a partition tiling such as a condition within the tiling that cannot exist or a condition within the tiling that must be modified for the containing proof tree to ever be able to find a combinatorial specification for the class of set partitions being looked at.

Strategy (Empty Cell Inferral)**.** *Empty cell inferral* is an inferral strategy applied to a partition tiling. For a given partition tiling T, we temporarily place a point in each unfinished cell and basis cell one at a time. If adding a point to one of these cells causes an occurrence of a pattern in the basis of T to appear, we shade that cell.

Empty cell inferral on a partition can be seen in Figure 2.15 on the partition tiling that is avoiding the pattern 122. The unfinished cell is inferred to be empty as we can clearly see that placing a point in that cell would result in an occurrence of the pattern 122.

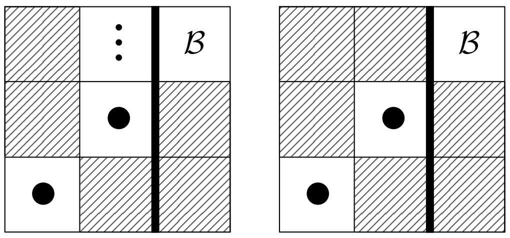

Figure 2.15: A partition tiling with $B = \{122\}$ and the result of applying empty cell inferral to it.

Strategy (Column Separation Inferral)**.** *Column separation inferral* is an inferral strategy applied to a partition tiling. For a given partition tiling T, each abstract column C is examined to see if the bottom most basis cell u_0 in C needs to exist in a different abstract column than the remaining basis cells in C. This is done by pair-wise placement of points in u_0 and each remaining basis cell while checking for an occurrence of a pattern in the basis of T. If for each pair-wise placement of points a pattern occurrence was found, we say the basis cells are *incompatible*, and then move all the basis cells in C except u_0 into a new abstract column directly to the right of C at their same vertical positions. If the cells are not incompatible, that is, at least two of them can exist in the same column up to single point placement, we say they are *compatible*.

Proposition 2.2.1. Repeated repetition of the column separation inferral strategy on a partition tiling will result in a partition tiling in where the basis cells in all of its abstract columns are compatible. Furthermore, the number of applications of the strategy needed to accomplish this is finite.

Proof. Consider the following cases for a partition tiling T:

- Case 1 - The basis cells in each abstract column of T are compatible. No further work needed.
- Case 2 - There are one or more abstract columns in T where the basis cells are incompatible. We examine the first such column and denote it with C_0. After applying the column separation inferral strategy, the number of basis cells moved to the new column C_1 are one fewer than the number of basis cells that were in C_0. We now have the following sub cases:
  - One basis cell is in C_1, and therefore the cells in C_1 are compatible.
  - The basis cells in C_1 are incompatible and so we apply the column separation strategy again.
  - The basis cells in C_1 are compatible.

We see that an abstract column with a finite number of basis cells can be converted into an ordered set of abstract columns that have compatible basis cells in a finite number of steps as on each application of the strategy there is one less basis cell to work with in the newly produced column. Given that T has a finite amount of abstract columns, it follows that we can convert T into a partition tiling in where none of its abstract column contains a set of incompatible basis cells using a finite number of operations. □

We demonstrate column separation in Figure 2.16. It is simple to reason through visual inspection of the partition tiling in the figure that placing a point in both of the basis cells would cause an occurrence of the pattern 1212. Movement of the top basis cell to a new column allows us to capture the structure of two basis cells with a single point interleaved while still being able to avoid 1212.

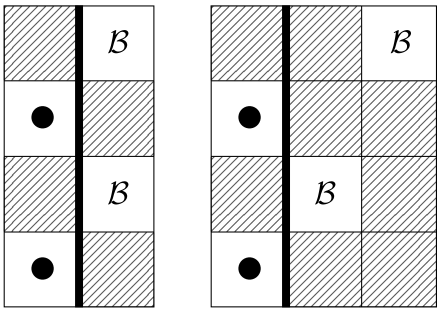

Figure 2.16: A partition tiling with $B = \{1212\}$ and the result of applying column separation to it.

2.2.3 Combinatorial Specification for Non-Crossing Partitions

We describe a step by step process for finding a combinatorial specification for set partitions avoiding the pattern 1212, often called the non-crossing partitions [7, Corollary 4.2]. The combinatorial specification can be seen in Figure 2.17. The figure also contains labels we will use to reference nodes in this section. We produce the tree in the following steps:

- We begin with the root partition tiling F_1. Verification strategies are not applied to this node as it is the class we wish to find a specification for.
- Column insertion is applied to F_1 to produce F_2 and F_3. The partition tiling F_3 represents the empty set partition denoted with E.
- Set avoidance is used to verify F_3.
- Point insertion is applied to F_2 which produces F_4 and F_5.
- Set avoidance is used to verify F_5.
- Column separation inferral is applied to F_4 to produce F_6.
- Cell splitting is applied to F_6 to produce F_1 and F_7. We see a recursion on F_1 and consider it verified.
- Cell splitting is applied to F_7 to produce F_2 and F_8. We see a recursion on F_2 and consider it verified.
- Set avoidance is used to verify F_8.
- The combinatorial specification has been found.

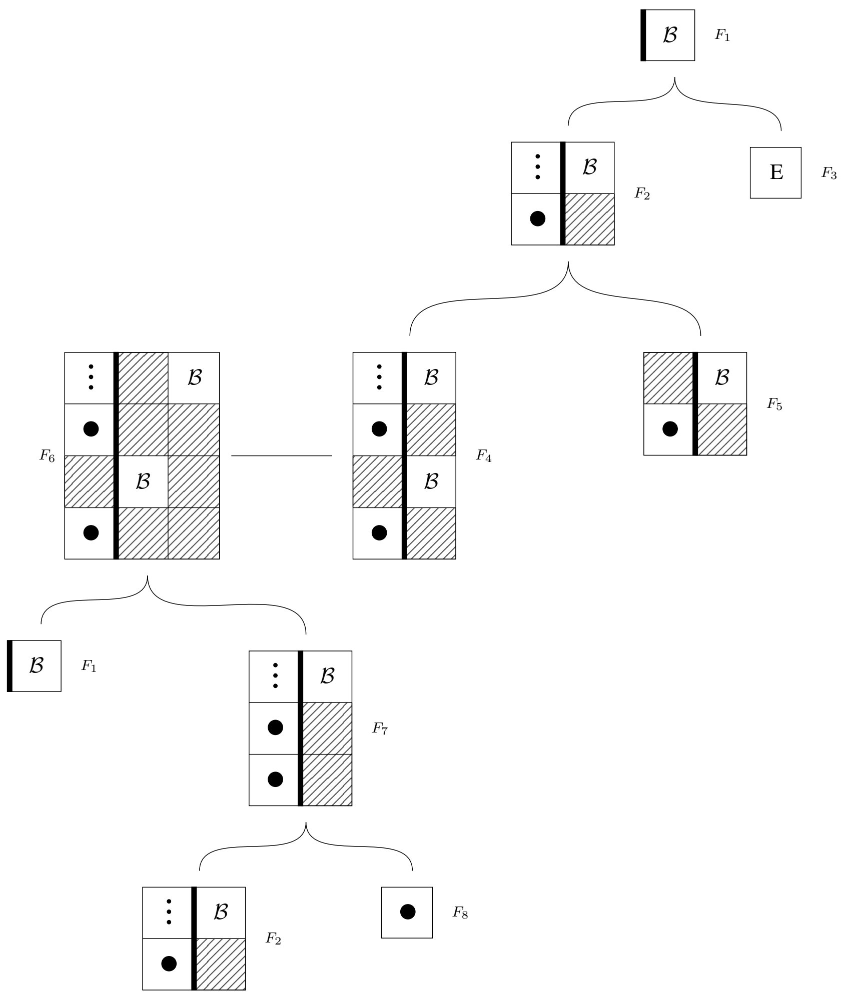

Figure 2.17: A visual representation of the combinatorial specification for non-crossing partitions.

The combinatorial specification in Figure 2.17 can be converted into a system of equations that can be solved to produce a generating function that enumerates the set partitions avoiding 1212. The process of solving this system of equations is as follows:

$$\begin{aligned}
F_1 &= F_2 + F_3\\
F_2 &= F_4 + F_5\\
F_3 &= 1\\
F_4 &= F_6\\
F_5 &= x \cdot F_1\\
F_6 &= F_1 \cdot F_7\\
F_7 &= F_2 \cdot F_8\\
F_8 &= x
\end{aligned}$$

With the systems of equations laid out we can now begin to solve for F_1 which will produce the generating function for the set partitions avoiding 1212.

$$\begin{aligned}
F_1 &= F_4 + F_5 + 1\\
F_1 &= F_6 + x \cdot F_1 + 1\\
F_1 &= x \cdot F_1 \cdot (F_1 - 1) + x \cdot F_1 + 1\\
F_1 &= x \cdot F_1^2 + 1\\
0 &= x \cdot F_1^2 - F_1 + 1
\end{aligned}$$

Finally, by application of the quadratic formula we arrive at the following generating function, which is the generating function for the Catalan numbers.

$$F_1 = \frac{1 - \sqrt{1 - 4x}}{2x}$$

Chapter 3

ComboPal

3.1 Description

ComboPal is the web application we developed for the visualization of combinatorial classes and long term storage of combinatorial specifications. The work done on ComboPal was initially intended to be an extension of the previous work done by students at Reykjavik University on PermPAL [3]. Upon review of PermPAL's code base, we decided to re-engineer the system from the ground up such that it could handle drawing trees of any combinatorial object. PermPAL focused exclusively on drawing proof trees for permutation patterns, and as such its code was written to do that task specifically.

We created ComboPal with an emphasis on it being combinatorial object agnostic; that is, its core focus was on drawing and storage functionality. ComboPal comes with a base template for others to use when describing the combinatorial objects they wish to render trees of with ComboPal. This design keeps ComboPal simple and well-defined in its responsibilities as it outsources the work of describing the drawing of a combinatorial object to those that have domain-specific knowledge.

ComboPal is able to visualize combinatorial specifications for avoidance classes of binary strings, set partitions as well as permutations patterns. Additionally, it currently stores 5415 combinatorial specifications. The storage of these specifications is particularly valuable as some combinatorial specifications can take days or weeks to produce on high-performance hardware.

3.2 ComboPal Technology

ComboPal frontend was created using HTML, CSS, and JavaScript. And utilized the following libraries:

- TreantJS [8], facilitates the process of displaying a combinatorial specification in a tree format.
- Bootstrap 4 [9], we use Bootstrap 4 for the user interface of ComboPal.
- JQuery [10], used as a utility in ComboPal.

The ComboPal backend was implemented in Python using the Flask web framework [11]. Long term storage of combinatorial specifications is accomplished through the use of MongoDB [12], a NoSQL database.

Chapter 4

Results

We built two systems on top of `CombSpecSearcher` titled `BinaryScope` and `PartiScope` which search for combinatorial specifications of avoidance classes of binary strings and avoidance classes of set partitions respectively. Utilizing the strategies outlined in the earlier chapters, these systems have had success enumerating and finding combinatorial specifications for many of the avoidance classes of their respective combinatorial object.

The enumeration of binary strings and set partitions has been a topic of interest to many researchers in the mathematics community. The systems we have designed has begun to be able to automate the work done by various researchers investigating these combinatorial objects.

The tests that were performed in this section were done on a desktop computer with an Intel i7 6700k overclocked to 4.2Ghz, and 16GB of RAM.

4.1 Binary Strings

The `BinaryScope` system we developed is able to find a combinatorial specification for any avoidance class of binary strings. This results from the fact that underlying `CombSpecSearcher` algorithm terminates for any avoidance class of binary strings as shown by proposition 2.1.1. Furthermore it is possible to produce a generating function for each avoidance class of binary strings as shown by Proposition 2.1.3.

4.2 Set Partitions

For the purposes of describing the search results we define P to be the set of all set partitions and P_n to be the set of all set partitions of $S_n = \{1, 2, \dots, n\}$.

`PartiScope` is able to find a combinatorial specification for a large number of avoidance classes of set partitions. We attempted to find combinatorial specifications for avoidance classes of set partitions whose basis were various subsets of the sets P_1, P_2, P_3, P_4 and their unions. In the results we sometimes specify how many patterns from P_n were used to construct the basis. When not specified all possible subsets of P_n are used.

For each combinatorial specification searched for we set a 60 second time limit. The result of these searches can be see in Table 4.1. In the searches where the possible basis patterns come from a union of P_i and P_j where $i \neq j$, we forced the basis to consist of at least one pattern from P_i and P_j.

Set for Basis	$\lvert\text{Successful}\rvert/\lvert\text{Set}\rvert$	Percentage
P_2	$3/3$	100%
P_3	$31/31$	100%
P_4, 1 pattern	$10/15$	66.6%
P_4, 2 patterns	$45/105$	42.8%
P_4, 3 patterns	$208/455$	45.7%
P_4, 4 patterns	$695/1365$	50.9%
$P_2 \cup P_3$	$2/2$	100%
$P_2 \cup P_4$	$2/2$	100%
$P_3 \cup P_4$	$249/956$	26%

Table 4.1: Proportion of avoidance classes of set partitions we are able to find a combinatorial specification for

It would be expected that as we increase the amount of patterns that are in the basis of an avoidance class that finding a combinatorial specification would become easier as the size of the class decreases with each additional basis pattern. However, as the amount of patterns in a basis grows, so does the time spent on doing case analysis. We are optimistic that if we dedicated more processing time to finding the combinatorial specifications for the avoidance classes that have multiple patterns in their basis, it would be possible to discover a combinatorial specification for more of them.

In the case of avoidance classes with a single pattern in their basis, we believe it to be unlikely that more processing time would produce a combinatorial specification for most of them where a specification was not found. We conjecture that more strategies will need to be developed in order to produce a specification for them. This stems from our experience when finding a specification for the set partitions that avoid 1212. In order to adequately explain its structure both the column separation strategy and cell splitting strategy needed to be present as only then did the strategies have enough descriptive power to produce a specification for it.

4.3 Conclusion

Implementing `BinaryScope` and `PartiScope` on top of `CombSpecSearcher` allowed us to automate a very exciting area of research. With `BinaryScope` we were able to automate the process of finding a combinatorial specification for any avoidance class of binary strings and producing a generating function for the class. The work on `PartiScope` has allowed us to find combinatorial specifications for many avoidance classes of set partitions, including those discussed in academic papers such as the class of non-crossing set partitions [6].

The success of ComboPal is demonstrated in its current daily use by the researchers of the Permuta Triangle group. The ability to easily visualize combinatorial specifications provides an intuitive means of understanding the output of software built on top of `CombSpecSearcher`. In our own work we have found quick access to visual representations of the combinatorial specifications we work with has afforded us the opportunity to develop strategies more effectively and intuitively. ComboPal is available on the public internet and can be accessed at http://combopal.ru.is.

4.4 Future Work

Much work remains to be done on `PartiScope`. There are many classes of set partitions that it does not adequate facilities to describe. Moving forward additional strategies will need to be created. The time complexity of the code required to generate the set partitions that have a valid gridding on a partition tiling is very high, and as a result the verification strategies take a long time to run when proof trees become more than 6 levels deep. Future work on `PartiScope` should include designing more efficient ways of generating the set partitions that have a gridding on a partition tiling.

Unlike with `BinaryScope`, we are currently unable to speak meaningfully about whether or not a search performed by `PartiScope` will ever terminate. Theoretical work remains to be done in producing a set of strategies that is adequate to discover a combinatorial specification for any avoidance class of set partitions, or conversely, to show that such a set of strategies cannot exist.

Both `BinaryScope` and `PartiScope` demonstrated that `CombSpecSearcher` works for combinatorial objects beyond that of permutation patterns. Interesting future work would include examining other combinatorial objects like patterns in lattice paths and classes of combinatorial objects that are defined in terms of something other than avoidance.

Bibliography

[1] E. Barcucci, A. Del Lungo, E. Pergola, and R. Pinzani, "ECO: a methodology for the enumeration of combinatorial objects.", *J. Differ. Equations Appl. 5*, pp. 435–490, 1999.

[2] C. Bean, "Finding Structures in Permutation Sets", Christian Bean's PhD thesis, not yet released.

[3] S. Viktorsson, A. Arnarsson, U. Erlendsson, and Á. Birkir, *Permpal*, 2017. [Online]. Available: `http://permpal.ru.is/`.

[4] S. Helgason and J. Robb, *ComboPal*, 2018. [Online]. Available: `http://combopal.ru.is/`.

[5] Flajolet and Sedgewick, *Analytic Combinatorics*. Cambridge University Press, 2009, pp. 57–58.

[6] V. Jelínek, T. Mansour, and M. Shattuck, "On Multiple Pattern Avoiding Set Partitions", *ArXiv e-prints*, Jan. 2013. arXiv: `1301.6509 [math.CO]`.

[7] G. Kreweras, "Sur les partitions non croisées d'un cycle", *Discrete Math.*, vol. 1, no. 4, pp. 333–350, 1972, ISSN: 0012-365X. DOI: `10.1016/0012-365X(72)90041-6`. [Online]. Available: `https://doi.org/10.1016/0012-365X(72)90041-6`.

[8] F. Peručić *et al.*, *TreantJS*, 2018. [Online]. Available: `http://fperucic.github.io/treant-js/`.

[9] *Bootstrap*. [Online]. Available: `https://getbootstrap.com/`.

[10] *JQuery*. [Online]. Available: `https://jquery.com/`.

[11] *Flask - A Python Microframework*. [Online]. Available: `http://flask.pocoo.org/`.

[12] *MongoDB*. [Online]. Available: `https://www.mongodb.com/`.