Searching for combinatorial covers using integer linear programming

by

Bjarni Jens Kristinsson

Thesis of 30 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

December 2019

Examining Committee:

Henning A. Ulfarsson, Supervisor
Assistant Professor, Reykjavik University, Iceland

Christian Bean, Supervisor
Postdoctoral Researcher, Reykjavik University, Iceland

Anders Claesson, Examiner
Professor, University of Iceland, Iceland

Bjarki Ágúst Guðmundsson, Examiner
Software Engineer, Google, Switzerland

Searching for combinatorial covers using integer linear programming

Bjarni Jens Kristinsson

December 2019

Abstract

We introduce the CombCov framework which is a generalization of the Struct algorithm introduced by Bean, Gudmundsson, and Ulfarsson in *"Automatic discovery of structural rules of permutation classes"*. We give a simple example of an application of the framework to *avoidance sets* of *words* and discuss in detail how to generate *rules* of lesser complexity and how a cover is verified up to a certain size using integer linear programming. We then apply the framework to various published results on *permutations* avoiding *mesh patterns* and try to find covers of similar problems with some success. We show that CombCov is a powerful tool in guiding humans by coming up with conjectures that would otherwise have required substantial effort to discover manually.

Þakningar fléttufræðilegra fyrirbrigða með aðstoð línulegrar heiltölubestunar

Bjarni Jens Kristinsson

desember 2019

Útdráttur

Við kynnum hugbúnaðinn CombCov sem er útvíkkun á Struct reikniritinu eftir Christian, Bjarka og Henning úr „*Automatic discovery of structural rules of permutation classes*“. Við skoðum einfalt dæmi um notkun þess með *forðunarmengjum* af *orðum* og skýrum hvernig *reglur* eru búnar til og hvernig þakning er sannreynd upp að ákveðinni lengd með hjálp línulegrar heiltölubestunar. Síðan beitum við hugbúnaðinum á ýmis þekkt vandamál á sviði *umraðana* sem forðast *möskvamynstur* og reynum einnig að finna lausnir á áður óleystum vandamálum með einhverjum árangri. Þannig sýnum við að CombCov er öflugt tól sem getur aðstoðað mannfólk við að fá hugmyndir að lausnum sem því hefði annars kannski ekki dottið í hug nema með mikilli fyrirhöfn.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last few years we have seen increased interest in *experimental mathematics* where computers are used to perform experiments and analyze data in greater quantities than humans are able to process by hand. Combinatorics is one field of mathematics that has greatly benefited from the advent of computers. Researchers routinely write computer programs that can quickly verify or refute hypotheses made by themselves to enable them to more efficiently allocate their time and energy with regards to the direction of the research.

Even more interesting are computer programs that can act as a source of inspiration by finding obscure and hidden patterns in large problem spaces. This can help the human researcher by giving them the correct idea to further pursue. One such tool is Struct by Bean, Gudmundsson, and Ulfarsson [1] which can automatically discover the structural rules of *permutation classes*[1] and thus conjecture the *enumeration* of the set as a function of the length of the *permutations* in the set. This is a problem whose solution can be hard to discover but easy to prove once found.

Other more complex computer programs exist but they often come with a drawback or compromise between usability and effectiveness. CombSpecSearcher by Bean [2] can find the enumeration of permutation classes, along with a proof of its validity, but it required significant work on behalf of the researcher in implementing *strategies* specific to the domain of permutations. Applying it to another domain would require similar effort.

Struct and our extended version CombCov have a lower barrier of entry for users in the hope that it will be more widely adopted by researchers. The lack of computer generated proofs are mitigated by the fact that the conjectures are often easily verified or disproved by the user.

1.1 Organization of thesis

We dedicate Chapter 2 to explaining how CombCov works from a high-level perspective. In Section 2.1 we define *words* as an intuitive example of *avoidance sets* and in Section 2.2 we explain how we implement them with CombCov. In Section 2.3 we present some results and analyze the effectiveness of CombCov as a tool.

Chapter 3 concerns itself with our main application of CombCov. In Section 3.1 we define *permutations* and *permutation classes*, the domain in which Struct operates,

[1]We will define this and other terminology in Section 3.1.

and *mesh patterns* and *mesh tilings*. Our implementation of mesh tilings with the framework is discussed in Section 3.2 and the results are presented in Section 3.3.

Finally, Chapter 4 concludes the thesis and discusses future work on this topic.

Chapter 2

Words

This chapter aims to explain the algorithm behind CombCov using avoidance sets of words as an intuitive example. Some results from this is discussed in Section 2.3 giving insight into the limitations of CombCov as a tool.

2.1 Background

Bean, Gudmundsson, and Ulfarsson [1] implemented the Struct algorithm in Python and published it on GitHub [3]. The core idea is to write a permutation class $\mathrm{Av}(\Pi)$ as a disjoint set of "simpler" sets of permutations S_i so that

$$\bigcup_i S_i = \mathrm{Av}(\Pi)$$

and $S_i \cap S_j = \varnothing$ for $i \neq j$. The challenge is to come up with these simpler subsets and verify that the two conditions hold (union and disjointedness). In this context we call $\mathrm{Av}(\Pi)$ the *root object* and S_i the *rules* or *subrules*. CombCov aims to be a general framework to solve these problems for any kind of combinatorial object. It is available as a Python module[1] (with source code hosted on GitHub [4]) and only requires the user to come up with, and implement in Python, a method to generate the rules and a method to return all elements of specific size. The rest is handled by the framework.

Definition 2.1.1. A *word of length n* is a sequence of *characters* $c_1 \cdots c_n$ over an *alphabet* Σ. If $n = 0$ then the word is the *empty word* and we denote it with ϵ.

In what follows of this thesis we only concern ourselves with words over the two letter alphabet $\Sigma = \{a, b\}$ and the reader should assume this choice of alphabet if it is not specifically stated. An example of a word of length 4 is *abba*.

Definition 2.1.2. We say that a word $u = u_1 \cdots u_n$ *contains* another word $v = v_1 \cdots v_k$ as a *subword* if there exists an i such that $u_{i+1} \cdots u_{i+k} = v_1 \cdots v_k$. If u does not contain v, we say that u *avoids* v and define $\mathrm{Av}_n(v)$ as the set of all words of length n avoiding v and write $\mathrm{Av}(v) = \bigcup_{n=0}^{\infty} \mathrm{Av}_n(v)$.

Example 2.1.3. The word *abba* contains the subword *bb* but avoids *aa*.

[1] Installable via `pip install CombCov`

Definition 2.1.4. The *concatenation* of two words $u = u_1 \cdots u_m$ and $v = v_1 \cdots v_n$ is the word $uv = u_1 \cdots u_m v_1 \cdots v_n$ of length $m + n$. The concatenation of a word u and a set $\mathcal{S}$ of words is written either $u\mathcal{S}$ or $u + \mathcal{S}$ and defined as the set $\{us \colon s \in \mathcal{S}\}$. In this context we say that u is the *prefix* of uv and $u\mathcal{S}$.

For a set of words $\mathcal{Z}$ over an alphabet Σ we define the *avoidance set of words* $\mathrm{Av}(\mathcal{Z})$ as the set of words that avoid all words in $\mathcal{Z}$. The avoidance set is *closed downwards*, meaning that if W is a word in the avoidance set then all subwords w of W are also in the avoidance set. The easiest way to see this is by contradiction: If for a word $W \in \mathrm{Av}(\mathcal{Z})$ there exists a subword w of W such that $w \notin \mathrm{Av}(\mathcal{Z})$ then there exist a $z \in \mathcal{Z}$ such that z is contained in w, and because w is a consecutive subsequence in W then z is contained in W meaning $W \notin \mathrm{Av}(\mathcal{Z})$, contradicting the initial assumption.

2.2 Implementation

We now discuss how we implement the abstract idea of words with CombCov and go step by step through the algorithm and how the framework searches for covers of these avoidance sets.

2.2.1 Computing with finite sets

Recall that for any set S of words we can write $\mathrm{Av}(S) = \bigcup_{n=0}^{\infty} \mathrm{Av}_n(S)$. CombCov takes in a parameter `max_elmnt_size`[2] and considers only

$$R = \bigcup_{n=0}^{\texttt{max\_elmnt\_size}} \mathrm{Av}_n(S)$$

for its computations. The hope is that the conclusions CombCov makes for R generalize to all element sizes, something that is often easy for a human to verify (or disprove).

Now assume $R = \{x_1, \ldots, x_m\}$. For a subset R' of R we define the *binary containment string* (or simply *containment string*) of R' to be the m-long binary string $B' = b_1 \cdots b_m$ where b_i is equal to 1 if $x_i \in R'$ and 0 otherwise. We assume a consistent order for the elements of R and in the case of words we use lexicographical ordering.

For example, if $R = \{\epsilon, a, b, ab, ba, bb\}$ then $B' = 111111$ denotes the whole set R, $B'' = 011001$ denotes the subset $\{a, b, bb\}$ and $B''' = 100000$ the subset $\{\epsilon\}$.

2.2.2 Disjoint subsets

We want to find a disjoint set of non-empty subsets R_i of R that covers R, i.e.,

$$\bigcup_i R_i = R$$

with $R_i \cap R_j = \varnothing$ for $i \neq j$. Just as R is a finite representation of the (possibly) infinite $\mathrm{Av}(S)$ (the root object), each R_i should be a finite version of a (possibly) infinite set S_i (the rules). The hope is that the finite cover generalizes to the (possibly infinite) cover

$$\bigcup_i S_i = \mathrm{Av}(S)$$

[2]In Section 2.3 we set `max_elmnt_size = 7`

with $S_i \cap S_j = \varnothing$ for $i \neq j$. We summarize this in Table 2.1.

	(Possibly) infinite set	Finite representation
Root object	$\mathrm{Av}(S)$	R
Rules	S_i	R_i

Table 2.1: CombCov uses finite representations of the root object and the rules

2.2.3 Generating the subsets

As previously mentioned, the way we generate these simpler subsets cannot be abstracted away in CombCov as we need to do it differently for different combinatorial objects. The rules are *descriptions* of the subsets S_i of $\mathrm{Av}(S)$ and the finite counterparts R_i of R.

This is the part where the user can apply their expertise to come up with a way of generating these rules and be smart in how they are doing it to get the best possible conjectures from CombCov. The framework can only check if the R_i's are proper subsets of R, but not if the S_i's are proper subsets of $\mathrm{Av}(S)$. The user should generate rules that are likely to generalize well but increasing the `max_elmnt_size` also increases the chance of that happening (at the cost of computing time). Wrong covers are often easily spotted when trying to prove (and then disprove) the conjectures.

Assuming $S = \{s_1, \dots, s_n\}$ is a set of n words, the longest of length k, we decided to create rules of the form $S_i = u\mathrm{Av}(S')$ where u is a word in $\mathrm{Av}(S)$ of length at most k or 1, whichever is higher, chosen from R and $S' = \Sigma$ (the whole alphabet) or S' is a set of words each of which is a subword of a word in S. In addition we sort S' by lexicographical order and check for every word $s' \in S'$ if it is contained in a longer word $s'' \in S'$, and if so, remove s'' from S'.

CombCov now generates the finite sets R_i of all elements in S_i of size up to `max_elmnt_size`. If the same element is generated in more than one way the rule is discarded as *invalid*. This does not happen with our subrules, but a rule like $\mathrm{Av}(a)+\mathrm{Av}(a)$ would be invalid because it generates bbb in multiple ways.

After this the framework checks each R_i is a proper subset of R and if so the corresponding rule S_i is said to be *valid*. CombCov discards all invalid rules. To optimize performance, the user should avoid creating too many invalid rules.

2.2.4 Finding a cover with the rules

For each of the valid rules CombCov constructs the corresponding binary containment string $B^{(i)}$, e.g., with `max_elmnt_size = 2` and R as in Section 2.2.1 the rule $a\mathrm{Av}(a)$ generates the set $R_1 = \{a, ab\} \subseteq R$ with the corresponding containment string 010100. The rule $a\mathrm{Av}(b)$ generates the set $R_2 = \{a, aa\} \nsubseteq R$ and is thus invalid.

Without loss of generality, assume that the k valid rules are the first k ones $S_1, \dots, S_k$. The sets $R_1, \dots, R_k$ with corresponding containment strings $B^{(1)}, \dots, B^{(k)}$ are not necessarily disjoint and some may even be equal to each other. CombCov constructs the following *integer linear programming model* on k binary variables $x_1, \dots, x_k$ with m equations:

$$\begin{array}{rl}
\text{Min} & z = x_1 + \cdots + x_k \\
\text{s.t.} & x_1 b_1^{(1)} + \cdots + x_k b_1^{(k)} = 1 \\
& \vdots \qquad \ddots \qquad \vdots \qquad \vdots \\
& x_1 b_m^{(1)} + \cdots + x_k b_m^{(k)} = 1 \\
\text{with} & x_i \in \{0,1\} \ \text{ for } i = 1, \ldots, k.
\end{array}$$

The variable x_i denotes whether rule i is part of the cover or not, and $b_i^{(j)}$ is a binary value representing if element j of R is in the finite set R_i or not. CombCov uses the linear programming solver Gurobi [5] (with fallback on COIN CLP/CBC LP [6]) to return a subset $I \subseteq [\![k]\!]$ with indices of the rules constituting the cover, if a cover is found. Note that the linear programming model uses minimization because we want solutions consisting of as few rules as possible.

2.2.5 An example with $\mathrm{Av}(aa)$

In Table 2.2 we list the rules, corresponding subsets R_i and binary containment strings for the avoidance set $\mathrm{Av}(aa)$ with `max_elmnt_size = 2`. Those are the subrules of the root object $\mathrm{Av}(aa)$.

Note that no two rules in the table are the same but because of the low value of `max_elmnt_size` some of the subsets R_i (the containment strings) are the same. This would not happen with higher values of `max_elmnt_size` as the rules are truly different and would start generating different words, as we see in Section 2.3.2. Running this exact problem in CombCov gives the 3-rule solution

$$\mathrm{Av}(aa) = \epsilon\mathrm{Av}(a,b) \cup a\mathrm{Av}(a) \cup b\mathrm{Av}(aa)$$

with corresponding bitstrings 100000, 010100 and 001011. This is certainly a correct cover of R, but *not* of $\mathrm{Av}(aa)$. It's easy to see that $abba \in \mathrm{Av}(aa)$ but none of the rules are able to generate this word. After seeing this, and running CombCov again with a sufficiently high value of `max_elmnt_size` we eventually get the correct solution

$$\mathrm{Av}(aa) = \epsilon\mathrm{Av}(a,b) \cup a\mathrm{Av}(a,b) \cup b\mathrm{Av}(aa) \cup ab\mathrm{Av}(aa)$$

as seen in Section 2.3 where `max_elmnt_size = 7`.

2.3 Results

All results presented in this section were obtained using version `v0.6.5` of CombCov, which is the latest version at the time of writing, and `max_elmnt_size = 7`. The software ran on a 2017 model MacBook Pro laptop with execution times ranging from seconds to minutes.

2.3.1 Overview of results

We applied CombCov on a select few avoidance sets of words over the alphabet $\Sigma = \{a,b\}$. The avoiding subword sets were selected with one or two words, each of length

Rule	R_i	Containment string
$\epsilon\text{Av}(a,b)$	$\{\epsilon\}$	100000
$a\text{Av}(a,b)$	$\{a\}$	010000
$b\text{Av}(a,b)$	$\{b\}$	001000
$ab\text{Av}(a,b)$	$\{ab\}$	000100
$ba\text{Av}(a,b)$	$\{ba\}$	000010
$bb\text{Av}(a,b)$	$\{bb\}$	000001
$a\text{Av}(a)$	$\{a, ab\}$	010100
$a\text{Av}(aa)$	$\{a, aa, ab\}$	*invalid*
$b\text{Av}(a)$	$\{b, bb\}$	001001
$b\text{Av}(aa)$	$\{b, ba, bb\}$	001011
$ab\text{Av}(a)$	$\{ab\}$	000100
$ab\text{Av}(aa)$	$\{ab\}$	000100
$ba\text{Av}(a)$	$\{ba\}$	000010
$ba\text{Av}(aa)$	$\{ba\}$	000010
$bb\text{Av}(a)$	$\{bb\}$	000001
$bb\text{Av}(aa)$	$\{bb\}$	000001

Table 2.2: Subrules of $\text{Av}(aa)$ over the alphabet $\Sigma = \{a, b\}$

at most three. CombCov prints out the enumerations of the sequences up to length `max_elmnt_size` which we include along with the corresponding OEIS [7] sequence number. The results are presented in Table 2.3.

The fact that CombCov manages to find covers for this many examples is encouraging. We now take a closer look at some of the results.

2.3.2 The avoidance set $\text{Av}(aa)$

The algorithm suggests that

$$\text{Av}(aa) = \epsilon\text{Av}(a,b) \cup a\text{Av}(a,b) \cup b\text{Av}(aa) \cup ab\text{Av}(aa)$$

where on the right-hand side there are four disjoint subsets, verified by the algorithm for all elements of size up to 7. The cover is indeed correct, as can be seen by thinking of a word $w \in \text{Av}(aa)$ in a series of *if-else* statements of the first few characters and using recursion. We will now show that the cover yields a (shifted) sequence of the Fibonacci numbers.

Recall that ϵ is the empty word and note that $\text{Av}(a,b)$ avoids all words that contains either a or b so $\epsilon\text{Av}(a,b) = \{\epsilon\}$ and $a\text{Av}(a,b) = \{a\}$. Now assume c_n is the number of words in $\text{Av}(aa)$ of length n and write $\sum_{n\geqslant 0} c_n x^n$ as the generating function for

Avoiding subwords	Cover Found	Enumeration	OEIS
$\varnothing$	$\epsilon\text{Av}(a,b) \cup a\text{Av}(\varnothing) \cup b\text{Av}(\varnothing)$	$1,2,4,8,16,32,64,128$	A000079
$\{a\}$	$\epsilon\text{Av}(a,b) \cup b\text{Av}(a)$	$1,1,1,1,1,1,1,1$	A000012
$\{a,b\}$	$\epsilon\text{Av}(a,b)$	$1,0,0,0,0,0,0,0$	A000007
$\{aa\}$	$\epsilon\text{Av}(a,b) \cup a\text{Av}(a,b)$ $\cup b\text{Av}(aa) \cup ab\text{Av}(aa)$	$1,2,3,5,8,13,21,34$	A000045 (shifted)
$\{aa,b\}$	$\epsilon\text{Av}(a,b) \cup a\text{Av}(a,b)$	$1,1,0,0,0,0,0,0$	A019590
$\{aa,bb\}$	*No*	$1,2,2,2,2,2,2,2$	A040000
$\{ab\}$	$\epsilon\text{Av}(a,b) \cup a\text{Av}(b) \cup b\text{Av}(ab)$	$1,2,3,4,5,6,7,8$	A000027
$\{ab,ba\}$	$\epsilon Av(a,b) \cup a\text{Av}(b) \cup b\text{Av}(a)$	$1,2,2,2,2,2,2,2$	A040000
$\{aaa\}$	$\epsilon\text{Av}(a,b) \cup a\text{Av}(a,b)$ $\cup aa\text{Av}(a,b) \cup b\text{Av}(aaa)$ $\cup ab\text{Av}(aaa) \cup aab\text{Av}(aaa)$	$1,2,4,7,13,24,44,81$	A000073 (shifted)
$\{aaa,b\}$	$\epsilon\text{Av}(a,b) \cup a\text{Av}(aa,b)$	$1,1,1,0,0,0,0,0$	A130716
$\{aaa,bb\}$	*No*	$1,2,3,4,5,7,9,12$	A164001
$\{aaa,bbb\}$	*No*	$1,2,4,6,10,16,26,42$	A128588
$\{aba\}$	*No*	$1,2,4,7,12,21,37,65$	A005251 (shifted)
$\{aba,aa\}$	$\epsilon\text{Av}(a,b) \cup a\text{Av}(a,b)$ $\cup b\text{Av}(aa,aba) \cup ab\text{Av}(a,b)$ $\cup abb\text{Av}(aa,aba)$	$1,2,3,4,6,9,13,19$	A000930 (shifted)
$\{aba,bb\}$	$\epsilon\text{Av}(a,b) \cup a\text{Av}(ba,bb)$ $\cup b\text{Av}(a,b) \cup ba\text{Av}(ba,bb)$	$1,2,3,4,4,4,4,4$	A158411 (shifted)
$\{aba,bab\}$	*No*	$1,2,4,6,10,16,26,42$	A128588

Table 2.3: Some avoidance sets of words over the alphabet $\Sigma = \{a,b\}$

$|\text{Av}_n(aa)|$. Then the cover gives us that

$$\sum_{n\geqslant 0} c_n x^n = 1 + x + \sum_{n\geqslant 0} c_n x^{n+1} + \sum_{n\geqslant 0} c_n x^{n+2}.$$

By looking at the coefficients at x^0 we see that $c_0 = 1$ and by comparing the coefficients at x^1 we get that $c_1 = 1 + c_0$ i.e., $c_1 = 2$. For $n \geqslant 2$ we get the recurrence relation

$$c_n = c_{n-1} + c_{n-2}$$

which proves the enumeration.

2.3.3 The fault lies with the user

It is clear that our implementation of subrule generation for avoidance sets of words, the format of $u\text{Av}(S')$, is not general enough to find a cover for all avoidance sets of words. It is not the fault of the tool, but of the one who wields it. With some tweaking [8] to the rule generation logic, CombCov should be able to find covers for all avoidance sets of words.

It is interesting to see that $\mathrm{Av}(aaa, bbb)$ and $\mathrm{Av}(aba, bab)$ both have the same enumeration (up to length 7) but CombCov is unable to find a cover for either of the avoidance sets of words. Meanwhile, $\mathrm{Av}(aa, bb)$ and $\mathrm{Av}(ab, ba)$ also have the same enumeration of which CombCov finds a cover for the latter but not the former. It is simple to confirm that

$$\mathrm{Av}(ab, ba) = \epsilon Av(a, b) \cup a\mathrm{Av}(b) \cup b\mathrm{Av}(a)$$

as the first rule is the set of the empty word, the second consists of all non-empty words of only a's and the third rule generates all non-empty words consisting of only b's. It is a different description of the set of words that contain neither ab nor ba.

Chapter 3

Mesh tilings

In this chapter we review the terminology on permutations and mesh patterns before defining *mesh tilings*. In Section 3.2 we explain how we apply CombCov to mesh tilings and discuss the results in Section 3.3. We consider this to be our main contribution apart from the framework itself.

3.1 Background

Definition 3.1.1. A *(classical) permutation of length n* is a bijection from the set of the first n integers, $[\![n]\!] = \{1, 2, \dots, n\}$ to itself. The set of all permutations of length n is denoted with $\mathfrak{S}_n$ and $\mathfrak{S} = \bigcup_{n=0}^{\infty} \mathfrak{S}_n$ is the set of all permutations.

A permutation π can also be viewed as a string $\pi(1) \cdots \pi(n)$ by listing the values that π takes at each index. An example of this is $\pi = 35142$, a permutation of length five with $\pi(1) = 3$, $\pi(2) = 5$, $\pi(3) = 1$, $\pi(4) = 4$ and $\pi(5) = 2$. The visual *grid representation* of π, denoted with $\mathrm{Gr}(\pi)$, is the plot of $\{(i, \pi(i)) \mid i \in [\![n]\!]\}$ in a Cartesian coordinate system, shown in Figure 3.1. The unique permutation of length 0 is called the *empty permutation* and it is denoted by ϵ.

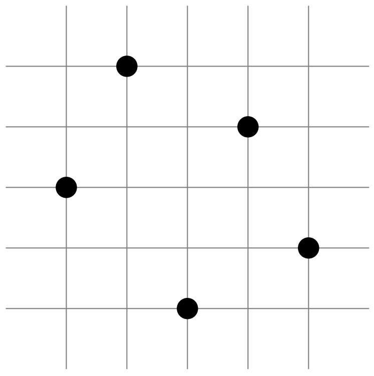


Figure 3.1: The grid representation of the permutation 35142

Definition 3.1.2. The *standardization* of a length n string of unique numbers $s_1 \cdots s_n$ is the permutation $\sigma \in \mathfrak{S}_n$ which has the same relative ordering as the string, meaning that for every i, j, $\sigma(i) < \sigma(j)$ if $s_i < s_j$. We write $\mathrm{st}(s_1 \cdots s_n) = \sigma$.

Every *index set* $\{i_1, \ldots, i_k\} \subseteq [\![n]\!]$ (with $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$) *induces* a substring $\pi(i_1) \cdots \pi(i_k)$ of length k of π and is called an *occurrence* of the *pattern* $p = \mathrm{st}(\pi(i_1) \cdots \pi(i_k))$ in π. We say that π *contains* p as a *subpermutation* or *(classical) pattern* and use the notation $p \preceq \pi$. If there exists no index set that induces an occurrence of p we say that π *avoids* p. The set of all patterns contained in π is denoted with $\Delta(\pi) = \{p \in \mathfrak{S} \mid p \preceq \pi\}$ and $\Delta(\Pi) = \bigcup_{\pi \in \Pi} \Delta(\pi)$ for a set of permutations Π.

Example 3.1.3. The permutation $\pi = 35142$ contains the pattern $q = 213$, highlighted with red circles in Figure 3.2, because $\{1, 3, 4\}$ induces the substring $\pi(1)\pi(3)\pi(4) = 314$ that has the same relative ordering as q (meaning $\mathrm{st}(314) = 213 = q$). Note that this is the only occurrence of q in π.

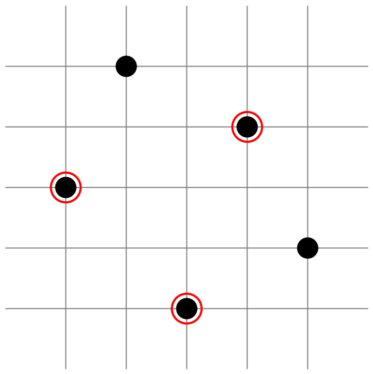


Figure 3.2: An occurrence of 213 in 35142 highlighted in red

The set of all permutations of length n that avoid π is denoted by $\mathrm{Av}_n(\pi)$ and we write $\mathrm{Av}(\pi) = \bigcup\limits_{n=0}^{\infty} \mathrm{Av}_n(\pi)$. If Π is a set of permutations we similarly write $\mathrm{Av}(\Pi) = \bigcap\limits_{\pi \in \Pi} \mathrm{Av}(\pi)$, i.e., avoiding a set of permutations means avoiding every permutation in the set. Note that $\mathrm{Av}(\varnothing) = \mathfrak{S}$.

In contrast to the set of avoiding permutations, we define the *containment set* as the complement, i.e., for a set of permutations Π we define $\mathrm{Co}_n(\Pi) = \mathfrak{S}_n \backslash \mathrm{Av}_n(\pi)$ and write $\mathrm{Co}(\Pi) = \bigcup\limits_{n=0}^{\infty} \mathrm{Co}_n(\Pi)$.

Example 3.1.4. The set of all permutations avoiding 21 is the set of *identity* permutations:

$$\mathrm{Av}(21) = \{\epsilon, 1, 12, 123, \ldots\}$$

A *permutation class* is a set $\mathcal{C}$ of permutations such that $\Delta(\mathcal{C}) \subseteq \mathcal{C}$. It it well known that $\mathrm{Av}(\Pi)$ is a permutation class for any set of permutations Π. A permutation class is *"closed downwards"* as every pattern contained in every permutation in $\mathcal{C}$ is also in $\mathcal{C}$.

3.1.1 Mesh patterns

The concept of *mesh patterns* in permutations was introduced by Brändén and Claesson [9] as a pair

$$p = (\pi, R) \text{ with } \pi \in \mathfrak{S}_k \text{ and } R \subseteq [\![0, k]\!] \times [\![0, k]\!]$$

where $[\![0,k]\!] = \{0,1,\ldots,k\}$ and R is a set of Cartesian coordinates $\ulcorner i,j \urcorner$ denoting the lower left corners of the squares in the grid representation of π which are *shaded*. An example with $p = (\pi, R)$ where $\pi = 213$ and $R = \{\ulcorner 1,2 \urcorner, \ulcorner 1,3 \urcorner, \ulcorner 2,2 \urcorner\}$ is shown in Figure 3.3.

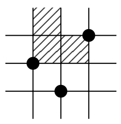


Figure 3.3: The mesh pattern $p = (213, \{\ulcorner 1,2 \urcorner, \ulcorner 1,3 \urcorner, \ulcorner 2,2 \urcorner\})$

Definition 3.1.5 (Brändén and Claesson)**.** An occurrence of p in $\sigma \in \mathfrak{S}_n$ is a subset ω of $\mathrm{Gr}(\sigma)$ such that there are order-preserving injections $\alpha, \beta \colon [\![k]\!] \mapsto [\![n]\!]$ satisfying the following two conditions. Firstly, ω is an occurrence of π in the classical sense, that is

i. $\omega = \{(\alpha(i), \beta(j)) \mid (i,j) \in \mathrm{Gr}(\pi)\}$

Define $R_{ij} = [(\alpha(i)+1, \alpha(i+1)-1)] \times [(\beta(j)+1, \beta(j+1)-1)]$ for $i,j \in [\![0,k]\!]$ where $\alpha(0) = \beta(0) = 0$ and $\alpha(k+1) = \beta(k+1) = n+1$. Then the second condition is

ii. if $\ulcorner i,j \urcorner \in R$ then $R_{ij} \cap \mathrm{Gr}(\sigma) = \varnothing$.

We call R_{ij} the *region corresponding to* $\ulcorner i,j \urcorner$ and π the *underlying classical pattern* of p.

Note that we defined an occurrence of a pattern in a classical permutation to be the subsequence of the permutation matching the pattern, but Brändén and Claesson defined an occurrence of a pattern in a mesh pattern to be the subset of the grid representation matching the pattern. For the sake of simplicity in what is to come we will keep with the original authors' definition of occurrences.

Example 3.1.6. Recall Example 3.1.3 where we saw that $\pi = 35142$ contains $q = 213$ as a pattern with only one occurrence. There is only one occurrence of q in $\sigma = 53241$ as well, namely 324. Figure 3.4 shows us two things. On the left side, we see that $p = (213, \{\ulcorner 1,2 \urcorner, \ulcorner 1,3 \urcorner, \ulcorner 2,2 \urcorner\})$ is *not* contained in π, because the point $(2,5) \in \mathrm{Gr}(\pi)$ lies inside R_{13}. On the right side we see that σ contains p, since for the only occurrence of q in σ the shaded area does not contain any of the points of $\mathrm{Gr}(\sigma)$.

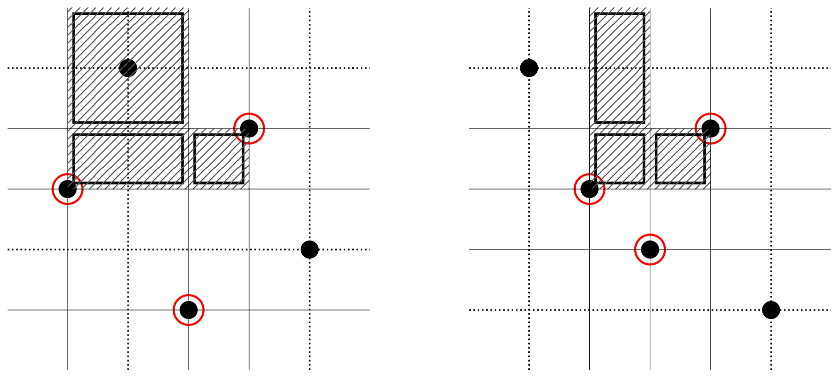


Figure 3.4: The mesh pattern p from Figure 3.3 is not contained in π (left), but is contained in σ (right)

Our definitions of $\mathrm{Av}(p)$ and $\mathrm{Co}(p)$ naturally extend to be the set of all permutations that avoid and contain p, respectively. Similarly $\mathrm{Av}(M) = \bigcap_{m \in M} \mathrm{Av}(m)$ and $\mathrm{Co}(M) = \mathfrak{S} \setminus \mathrm{Av}(M)$ for a set M of mesh patterns.

Vincular [10], *covincular* [11] and *bivincular* [12] permutation patterns are special cases of the more general mesh patterns where there are restrictions on the occurrence to have adjacent indices, values or both in the permutation. We write them as a triple (π, X, Y) where π is the underlying permutation of length n and $X, Y \subseteq [\![0, n]\!]$ denote which columns and rows are shaded in the mesh pattern. Using our notation, this is equal to the mesh pattern

$$(\pi, R) \text{ with } R = \bigcup_{x \in X} \{x\} \times [\![0, n]\!] \cup \bigcup_{y \in Y} [\![0, n]\!] \times \{y\}.$$

If $X \neq \varnothing$ and $Y = \varnothing$ it is a vincular pattern, if $X = \varnothing$ and $Y \neq \varnothing$ it is a covincular pattern and if neither X nor Y is the empty set it is a bivincular pattern. See Figure 3.5 for examples of a vincular, a covincular and a bivincular permutation pattern.

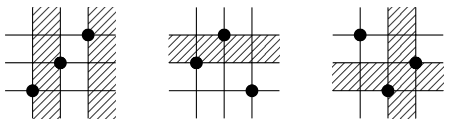


Figure 3.5: $(123, \{1, 3\}, \varnothing)$, $(231, \varnothing, \{2\})$ and $(312, \{2\}, \{1\})$

So far we have defined the concept of permutations containing permutations and mesh patterns. Tannock and Ulfarsson [13] defined mesh pattern containment of mesh patterns, recalled below.

Definition 3.1.7 (Tannock and Ulfarsson)**.** A mesh pattern $q = (\kappa, T)$ *contains* a mesh pattern $p = (\tau, R)$ as a *subpattern* if κ contains p and $\left(\bigcup_{(i,j) \in R} R_{ij} \right) \subseteq T$.

If p does not contain q we say that p *avoids* q.

3.1.2 Mesh tilings

Definition 3.1.8. A *block*[1] $\mathfrak{B}$ is a tuple $(\mathcal{O}, \mathcal{R})$ of two sets of mesh patterns, called *obstructions* and *requirements*, that *generates* the set of permutations

$$\mathrm{Av}(\mathcal{O}) \cap \mathrm{Co}(\mathcal{R}).$$

The block $\overleftarrow{\mathfrak{B}} = (\mathcal{R}, \mathcal{O})$ is called the *flipped* block (of $\mathfrak{B}$).

We use the term "block" interchangeably for the tuple and the set of permutations that it generates.

Example 3.1.9. Some notable blocks include the *point block* $(\{12, 21\}, \{1\})$, the *empty block* $(\{1\}, \varnothing)$ and the *free block* $(\varnothing, \varnothing)$ that generate the sets $\{1\}$, $\{\epsilon\}$ and $\mathfrak{S}$ respectively.

[1]In the Python source code this is called a *cell*

Before moving on to define mesh tilings we review generalized grid classes introduced by Vatter [14]. Given a permutation π of length n and sets $X, Y \subseteq [\![n]\!]$, we denote with $\pi(X \times Y)$ the standardization of the subsequence of π with indices from X and values in Y. For example $25134([1,4] \times [2,5]) = \mathrm{st}(253) = 132$.

Now suppose $\mathcal{M}$ is a $c \times r$ matrix (indexed from left to right and bottom to top) whose entries are sets of permutations. An *$\mathcal{M}$-gridding* of the permutation π of length n is a pair of sequences $1 = c_1 \leqslant \cdots \leqslant c_{t+1} = n+1$ (the column divisions) and $1 = r_1 \leqslant \cdots \leqslant r_{u+1} = n+1$ (the row divisions) such that $\pi([c_k, c_{k+1}) \times [r_l, r_{l+1}))$ is either empty or is a member of $\mathcal{M}_{k,l}$ for all $k \in [\![t]\!]$ and $l \in [\![u]\!]$. The *generalized grid class* of $\mathcal{M}$, $\mathrm{Grid}(\mathcal{M})$, is the set of all permutations with an $\mathcal{M}$-gridding.

Definition 3.1.10. A *mesh tiling* $\mathcal{M}$ is a $c \times r$ matrix of *blocks* (as in Definition 3.1.8), representing the set of permutations $\mathrm{Grid}(\mathcal{M})$.

A 1×1 mesh tiling generates the same set of permutations as the (only) block in the mesh tiling.

Example 3.1.11. The 2×2 mesh tiling (depicted in Figure 3.6) with empty blocks in coordinates $(0,1)$ and $(1,0)$, a point block at $(0,0)$ and a free block at $(1,1)$ generates the set of permutations starting with 1.

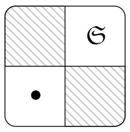


Figure 3.6: A mesh tiling generating all permutations starting with one

3.2 Implementation

With mesh tilings we are seeking to find covers for avoidance sets of permutations of the form $\mathrm{Av}(M)$ where M is a set of mesh patterns. Often we are interested in the complementary set $\mathrm{Co}(M)$ as well because a cover for it might help us understand the structure of the permutation class. In practice, therefore, we usually apply CombCov on a single 1×1 mesh tiling hoping to find a cover in the form of one or more mesh tilings of various sizes. Our implementation expects the root object to be a 1×1 mesh tiling but it works as well for mesh tilings of larger sizes.

The framework requires us to implement two main functions for our mesh tiling object in Python. Given a $c \times r$ mesh tiling $\mathcal{M}$ these methods are:

(i) `get_elmnts`: Takes in a parameter l and outputs all elements of length l generated by $\mathcal{M}$ in a consistent order. This is easy to describe mathematically but harder to implement programmatically. We refer to the original implementation by Bean, Gudmundsson, and Ulfarsson [3] for further details.

(i) `get_subrules`: Generates the subrules S_i of the root object $\mathcal{M}$. Recall that not all subrules need to be *valid*, but from a performance perspective it is preferable that not too many invalid rules are generated. Algorithm 1 shows in pseudo-code how we implemented the subrule generation that yielded the results in Section 3.3.

This is the product of several iterations where we edited the algorithm to find more covers while keeping the search space small enough for tolerable running times.

Algorithm 1 Mesh tiling subrule generation

```
Require: $1 \times 1$ mesh tiling $\mathcal{M}$ and positive integers max_columns, max_rows and
  max_active_blocks
  $\mathcal{C} \leftarrow \mathcal{M}_{1,1}$
  $P \leftarrow \varnothing$
  for all patterns $p$ in the union of the obstructions and requirements of $\mathcal{C}$ do
    if $p$ is a permutation then
      append all subpermutations of $p$ to $P$
    else if $p$ is a mesh pattern then
      append all subpatterns of $p$ to $P$
    end if
  end for
  $C \leftarrow$ the set of $\mathcal{C}$, $\overleftarrow{\mathcal{C}}$, the point block and the free block
  for all patterns $p$ in $P$ do
    append the block $(\{p\}, \varnothing)$ to $C$
  end for
  $M \leftarrow$ the set of the $1 \times 1$ mesh tiling with the empty block
  for all $i$ such that $1 \leqslant i \leqslant$ max_columns do
    for all $j$ such that $1 \leqslant j \leqslant$ max_rows do
      for all $a$ such that $i * j \leqslant a \leqslant$ max_active_blocks do
        for all combinations of $a$ blocks (with repetition) from $C$ do
          for all combinations of $a$ squares in an $i \times j$ grid of which there is no
          column or row with no selected square do
            append the $i \times j$ mesh tiling with the $a$ blocks placed in the chosen
            squares to $M$
          end for
        end for
      end for
    end for
  end for
  return $M$
```

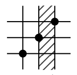
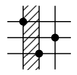
3.3 Results

Our aim here is twofold. One is to reproduce old results by finding covers for sets of permutations avoiding a set of mesh patterns (we will call this *avoidance sets* as they are not necessarily permutation classes) and compare them to the published results. The other is to try to find covers for avoidance sets that are not already known and thus contribute new knowledge to the field of enumerative combinatorics.

All results presented in this section were obtained using version `v0.6.3` of CombCov, and `max_elmnt_size = 7`, but it is identical to the version used in Section 2.3 apart from a few tweaks in the way that it generates subrules for avoidance sets of words. As these problems demanded more computing power we ran them on the computer cluster *Garpur*[2] managed by the University of Iceland. The execution times ranged from hours to days with a hard 2-week time limit after which the execution was stopped and deemed unsuccessful.

We inputted 620 different avoidance sets into CombCov along with their complementary containment sets for a total of 1240 unique jobs with the framework. In each job we tried multiple and increasing values for `max_columns`, `max_rows` and `max_active_blocks`, encoded as a triple (c, r, a). Their initial values were set to $(2, 2, 3)$, and if a cover was not found we next tried $(3, 3, 3)$, after that $(4, 4, 4)$, then $(4, 4, 5)$ and lastly $(5, 5, 5)$. We will not summarize all of the results here but instead highlight a few interesting covers and discuss their significance.

3.3.1 Vincular and covincular length 3 mesh patterns

Claesson [15] studied *generalized permutation patterns* (which later became called vincular patterns) of length 3 and enumerated the permutation sets avoiding one or two such patterns. He used a notation of letters and dashes to indicate shaded columns, e.g., *a-bc* =  and *ca-b* = .

In Figure 3.7 we show the cover which CombCov finds for an avoidance set that is enumerated by the *Bell numbers* (OEIS sequence number A000110), meaning that $|\text{Av}_n(\mathcal{A})|$ is the n-th Bell number, B_n.

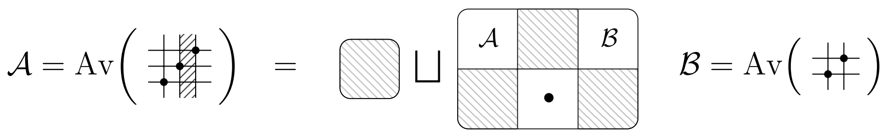
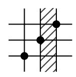


Figure 3.7: CombCov cover of Av(*a-bc*)

It is easy to see that the cover indeed generates the permutations that avoid . More interesting is deriving the *exponential generating function (EGF)* $F = \sum_{n\geqslant 0} \frac{a_n}{n!}x^n$ and showing that $a_n = B_n$, proving that the sequence is enumerated by the Bell numbers.

We start by noting that $|\text{Av}_n(12)| = 1$ for all n so $\mathcal{B}$ in Figure 3.7 is enumerated by $(1)_n$ and has EGF $\sum_{n\geqslant 0} \frac{1}{n!}x^n = e^x$. This gives us

$$
\begin{aligned}
& F = 1 + \int F e^x dx \\
\text{i.e.} \quad & \frac{d}{dx}F = Fe^x \\
\text{i.e.} \quad & \int \frac{1}{F} dF = \int e^x dx \\
\text{i.e.} \quad & \ln(F) = e^x + C \\
\text{i.e.} \quad & F = Ae^{e^x}
\end{aligned}
$$

We have shown that $\sum_{n\geqslant 0} \frac{a_n}{n!}x^n = Ae^{e^x}$ for some constant A. We know that there is only one permutation of length zero in $\mathcal{A}$ so putting $a_0 = 1$ into the equation at

[2]See `http://ihpc.is/garpur/` for details

$x = 0$ gives us $1 = Ae$ i.e. $A = e^{-1}$ so $F = e^{e^x - 1}$ which is indeed the EGF for the Bell numbers.

In Figure 3.8 we present the cover CombCov found for an avoidance set that is enumerated with I_n (A000085), the number of *involutions* in $\mathfrak{S}_n$.

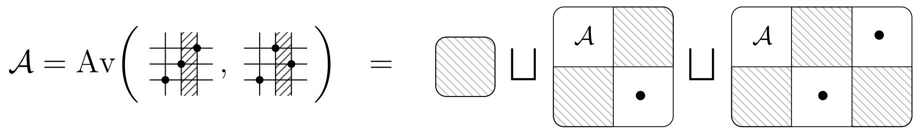
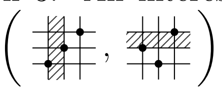
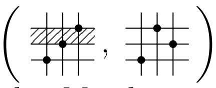


Figure 3.8: CombCov cover of Av(a-bc, a-cb)

Similarly we can easily see that the cover is indeed correct and derive the exponential generating function to prove the enumeration.

Bean, Ulfarsson, and Claesson [11] studied the set of permutations simultaneously avoiding a vincular and covincular mesh pattern of length 3. An interesting result is that CombCov finds exactly the same cover for both Av and Av which is shown in Figure 3.9. These avoidance sets are enumerated by the *Motzkin* numbers M_n (A001006) and it is easy to see from the cover that the generating function fulfills $F(x) = 1 + xF(x) + x^2F(x)^2$ which indeed gives us the coefficients M_n at x^n.

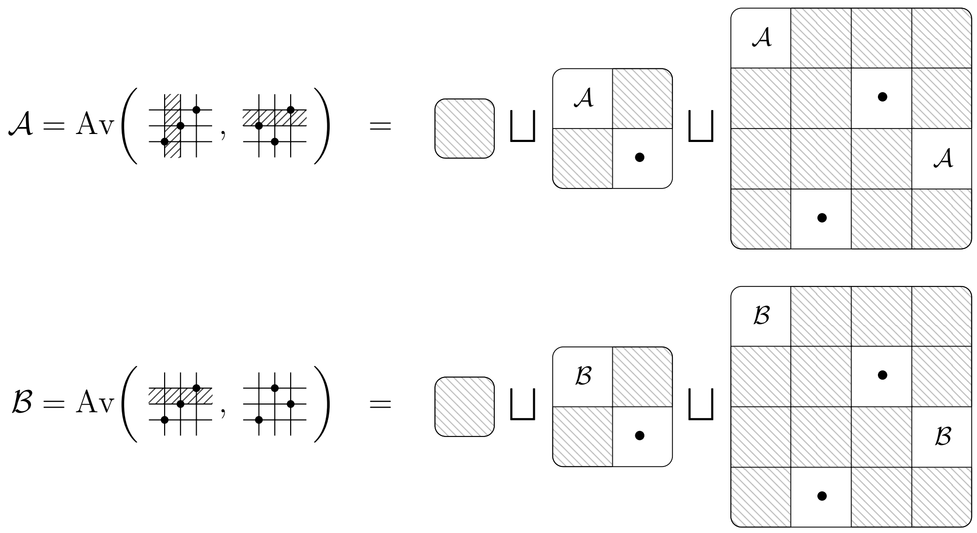
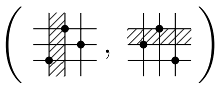
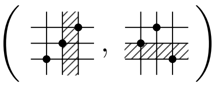


Figure 3.9: CombCov cover of avoidance sets that gives rise to the Motzkin numbers

Other successful covers in this paper include Av (Figure 3.10, A011782) and Av (Figure 3.11, A152947).

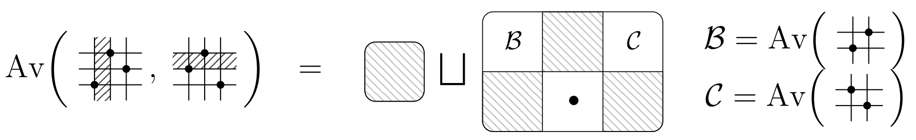


Figure 3.10: CombCov cover for a pair of vincular and covincular mesh patterns

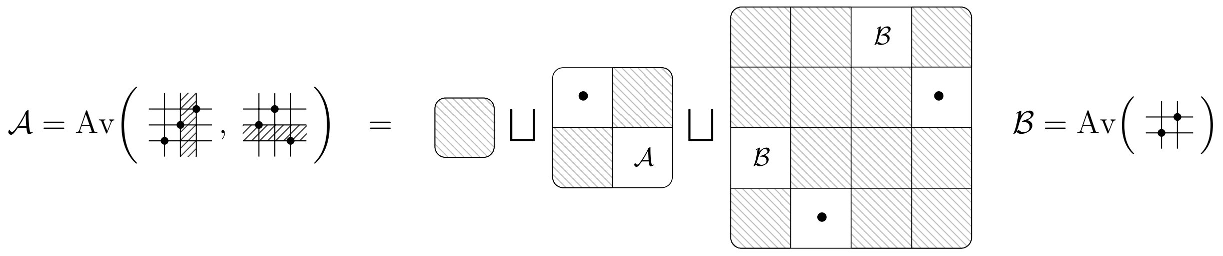
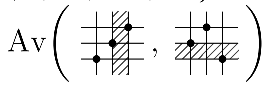


Figure 3.11: CombCov cover for another pair of vincular and covincular mesh patterns

We note that $\mathcal{B} = \{\epsilon, 1, 21, 321, \ldots\}$ is the set of all decreasing permutations so the generating function for  in Figure 3.11 fulfills

$$F(x) = 1 + xF(x) + \frac{x^2}{(1-x)^2}$$

which we can solve to find

$$F(x) = \frac{2x^2 - 2x + 1}{(1-x)^3}$$

which gives us the coefficients $1 + \binom{n}{2}$ at x^n.

3.3.2 Length 2 mesh patterns

Hilmarsson, Jónsdóttir, Sigurðardóttir, *et al.* [16] classified all mesh patterns of length 2 into *Wilf-equivalence* classes, i.e., classes having the same enumeration. Some notable covers are shown in Figures 3.12, 3.13 and 3.14.

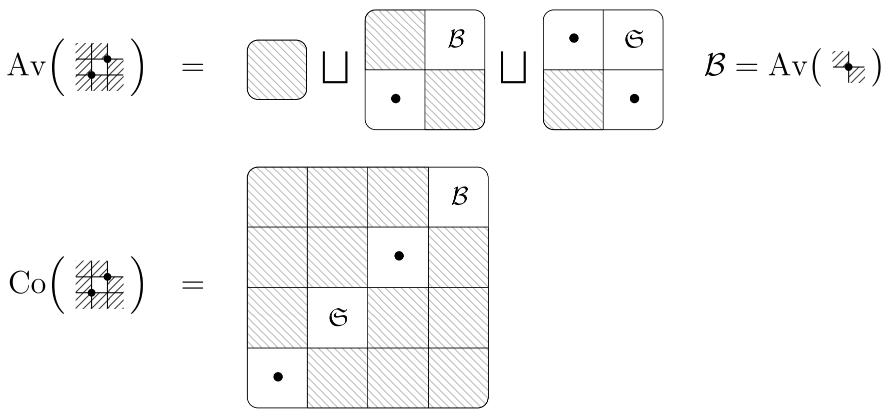


Figure 3.12: Avoidance and containment covers for a length 2 mesh pattern

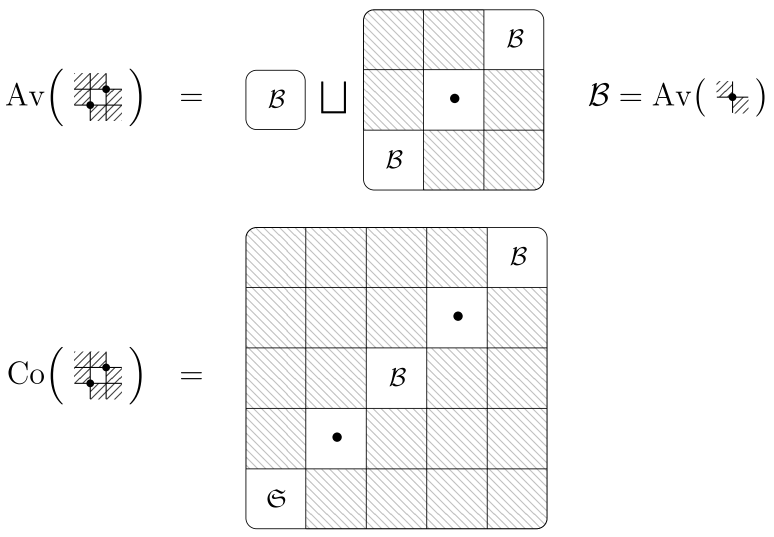
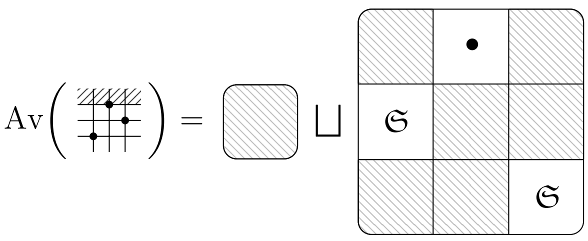


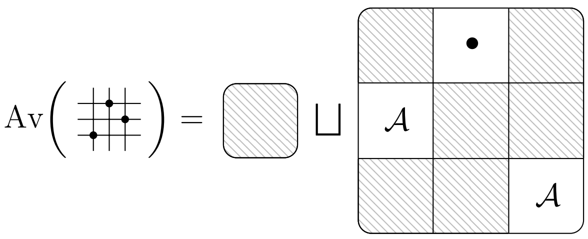
Figure 3.13: Avoidance and containment covers for another length 2 mesh pattern

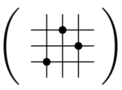
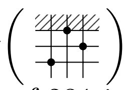
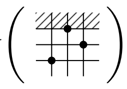
3.3.3 Bivincular permutation patterns

We now focus on some results of bivincular permutation patterns replicated from Parviainen [17]. Among them is the cover



(enumeration A003149) which is especially interesting when comparing it to the very well known



(enumerated by the *Catalan numbers*, A000108) with $\mathcal{A} = \mathrm{Av}$. The difference is that Av is the set of permutations π avoiding 231 in which the 3 in the occurrence of 231 is the *topmost point* of π. This is already secured by the point block in the second mesh tiling in the cover of Av and explains why it includes $\mathfrak{S}$ blocks instead of the root.

In addition to finding covers for the results presented in the paper, we searched for covers for all possible length 3 bivincular mesh patterns, discarding symmetries.

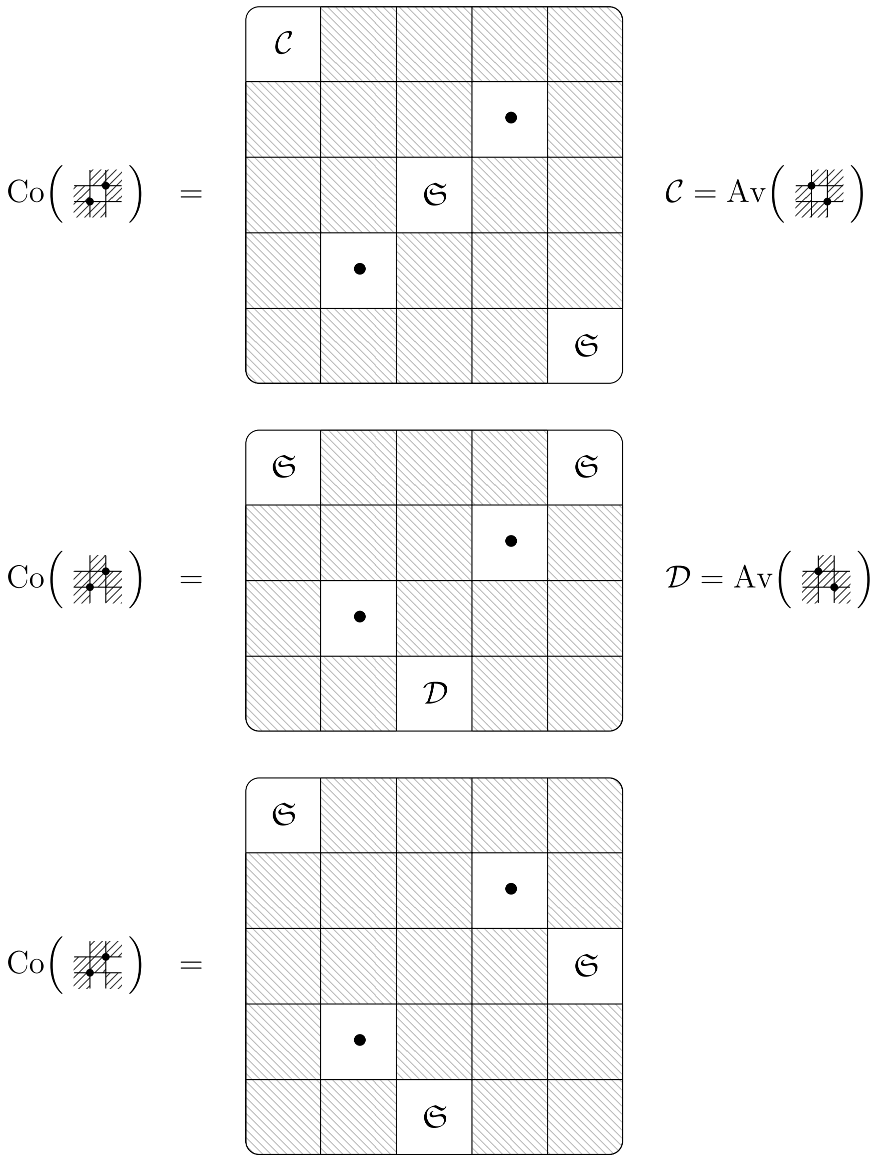


Figure 3.14: Containment covers for some length 2 mesh patterns

That made for 163 additional avoidance sets to find covers for, out of which 79 were successful. Some interesting covers are shown in Figure 3.15.

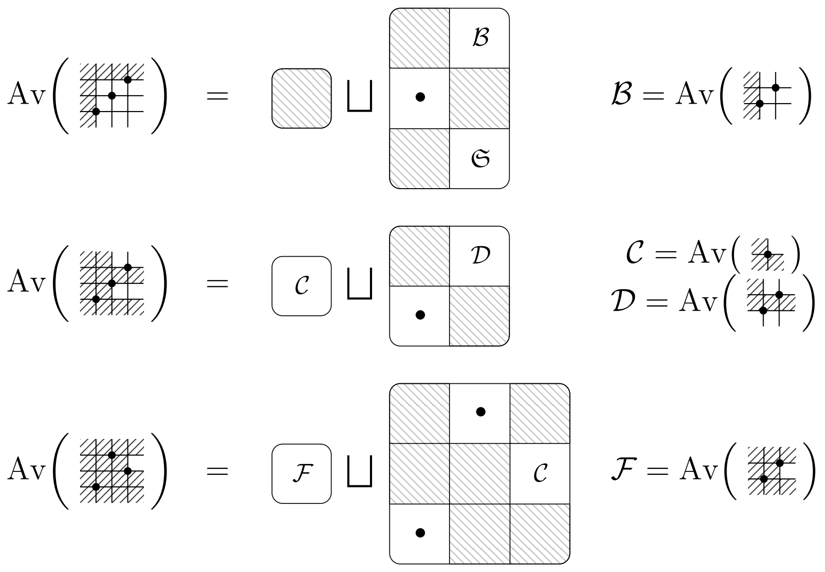
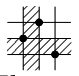
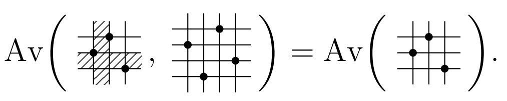

Av
=
⊔
$\mathcal{B}$
$\mathfrak{S}$
$\mathcal{B} = \text{Av}$
$\mathcal{C}$
$\mathcal{D}$
$\mathcal{C} = \text{Av}$
$\mathcal{D} = \text{Av}$
$\mathcal{F}$
$\mathcal{C}$
$\mathcal{F} = \text{Av}$

Figure 3.15: Avoidance covers for some bivincular mesh patterns of length 3

3.3.4 Pattern-avoiding Fishburn permutations

Gil and Weiner [18] defined *Fishburn permutations* as those that avoid the mesh pattern $p =$  because $|\text{Av}_n(p)| = \xi(n)$ are the *Fishburn numbers* (A022493).

They studied the set of Fishburn permutations avoiding a permutation π of length 3 or 4 (i.e., the set of permutations simultaneously avoiding p and π). For each of the six permutations π of length 3, they enumerated all sets $\text{Av}(p, \pi)$ and CombCov finds a cover for five of them. Then the authors enumerated some of the sets $\text{Av}(p, \pi)$ for $\pi \in \mathfrak{S}_4$ but focused mostly on proving that $|\text{Av}_n(p, \pi)| = C_n$ (the *Catalan numbers*, A000108) for $\pi \in \{1234, 1243, 1324, 1423, 2134, 2143, 3124, 3142\}$. We found a cover only for the last π, indicating that



The only other cover we found with a length 4 permutation π is shown in Figure 3.16.

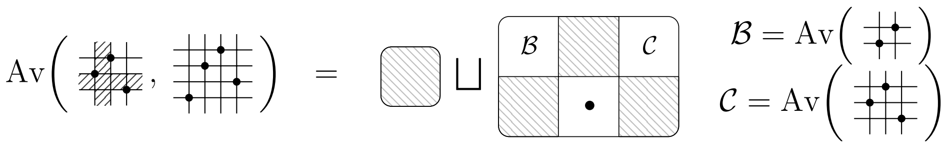


Figure 3.16: Fishburn permutations avoiding 1342

In addition to what the authors did we searched for covers of Fishburn permutations avoiding two length 4 permutations π_1 and π_2. Out of the 276 avoidance sets $\text{Av}(p, \pi_1, \pi_2)$ we found covers for 10 of them but none is interesting enough to show here.

3.3.5 Summary

Out of the 159 results from the papers that we tried to replicate with CombCov it found covers for 49 of them. This is summarized in Table 3.1, broken down by paper.

Paper	Section	Results in paper	Covers found
[15]	3.3.1	6	5
[11]	3.3.1	40	11
[16]	3.3.2	65	15
[17]	3.3.3	24	11
[18]	3.3.4	24	7

Table 3.1: Summary of replicating mesh pattern covers

Chapter 4

Discussion

Just as Struct is a useful tool for finding covers of permutation classes we have shown that the generalized version CombCov can find covers for sets of other types of combinatorial objects. We defined *words* and *mesh patterns* and applied CombCov on both known results and new problems with some success.

4.1 Conclusion

The framework did not find covers for all of the problems that we looked at. That is not the framework's fault but ours, as the user writing the algorithm to generate subrules. We suspect that the algorithm is theoretically capable of finding more covers than was presented in this paper but because of slow running times and concrete time limits we did not find more covers. CombCov has proved its helpfulness by being able to come up with ideas that previously required substantial effort by a human researcher.

4.2 Future work

Future work includes refining the library with ease-of-use in mind, writing documentation and a “Help get started” guide. It is our hope that other researchers pick it up and apply it to their combinatorial objects of interest such as polyominoes, set partitions, alternating sign matrices, ascent sequences or something completely different.

Bibliography

[1] C. Bean, B. Gudmundsson, and H. Ulfarsson, "Automatic discovery of structural rules of permutation classes", en, *Mathematics of Computation*, vol. 88, no. 318, pp. 1967–1990, 2019, ISSN: 0025-5718, 1088-6842. DOI: `10.1090/mcom/3386`. [Online]. Available: `https://www.ams.org/home/page/`.

[2] C. Bean, "Finding structure in permutation sets", en, Jun. 2018. [Online]. Available: `https://opinvisindi.is/handle/20.500.11815/1184`.

[3] C. Bean, B. Gudmundsson, and H. Ulfarsson, *PermStruct*, 2017. [Online]. Available: `https://github.com/PermutaTriangle/PermStruct`.

[4] B. J. Kristinsson, *CombCov: Searching for combinatorial covers.* 2019. [Online]. Available: `https://github.com/PermutaTriangle/CombCov`.

[5] LCC Gurobi Optimization, *Gurobi Optimizer Reference Manual*, 2019. [Online]. Available: `http://www.gurobi.com`.

[6] COIN-OR, *COIN CLP/CBC LP*, 2019. [Online]. Available: `https://www.coin-or.org/`.

[7] OEIS Foundation Inc., *The On-Line Encyclopedia of Integer Sequences*, 2019. [Online]. Available: `http://oeis.org/`.

[8] L. J. Guibas and A. M. Odlyzko, "String overlaps, pattern matching, and nontransitive games", en, *Journal of Combinatorial Theory, Series A*, vol. 30, no. 2, pp. 183–208, Mar. 1981, ISSN: 0097-3165. DOI: `10.1016/0097-3165(81)90005-4`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/0097316581900054`.

[9] P. Brändén and A. Claesson, "Mesh patterns and the expansion of permutation statistics as sums of permutation patterns", *The Electronic Journal of Combinatorics*, vol. 18, no. 2, p. 5, 2011.

[10] E. Babson and E. Steingrímsson, "Generalized Permutation Patterns and a Classification of the Mahonian Statistics", *Séminaire Lotharingien de Combinatoire*, vol. 44, no. B44b, 18 pp. 2000, ISSN: 1286-4889. [Online]. Available: `https://www.emis.de/journals/SLC/wpapers/s44stein.html`.

[11] C. Bean, H. Ulfarsson, and A. Claesson, "Enumerations of Permutations Simultaneously Avoiding a Vincular and a Covincular Pattern of Length 3", *Journal of Integer Sequences*, vol. 20, no. 2, Art. 17.7.6, 25, Jul. 2017.

[12] M. Bousquet-Mélou, A. Claesson, M. Dukes, and S. Kitaev, "(2+2)-free posets, ascent sequences and pattern avoiding permutations", en, *Journal of Combinatorial Theory, Series A*, vol. 117, no. 7, pp. 884–909, Oct. 2010, ISSN: 0097-3165. DOI: `10.1016/j.jcta.2009.12.007`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0097316509001885`.

[13] M. Tannock and H. Ulfarsson, “Equivalence classes of mesh patterns with a dominating pattern”, *Discrete Mathematics & Theoretical Computer Science ; Vol. 19 no. 2*, Permutation Patterns 2016, 13658050, 2018, ISSN: 1365-8050. DOI: `10.23638/dmtcs-19-2-6`. [Online]. Available: `https://dmtcs.episciences.org/4265`.

[14] V. Vatter, “Small permutation classes”, en, *Proceedings of the London Mathematical Society*, vol. 103, no. 5, pp. 879–921, Nov. 2011, ISSN: 00246115. DOI: `10.1112/plms/pdr017`. [Online]. Available: `http://doi.wiley.com/10.1112/plms/pdr017`.

[15] A. Claesson, “Generalized Pattern Avoidance”, *European Journal of Combinatorics*, vol. 22, no. 7, pp. 961–971, Oct. 2001, ISSN: 0195-6698. DOI: `10.1006/eujc.2001.0515`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0195669801905153`.

[16] Í. Hilmarsson, I. Jónsdóttir, S. Sigurðardóttir, L. Viðarsdóttir, and H. Ulfarsson, “Wilf-Classification of Mesh Patterns of Short Length”, *The Electronic Journal of Combinatorics*, vol. 22, no. 4 (2015), P4.13, Oct. 2015, ISSN: 1077-8926. [Online]. Available: `https://www.combinatorics.org/ojs/index.php/eljc/article/view/v22i4p13`.

[17] R. Parviainen, “Wilf classification of bi-vincular permutation patterns”, *arXiv:0910.5103 [math]*, Oct. 2009, arXiv: 0910.5103. [Online]. Available: `http://arxiv.org/abs/0910.5103`.

[18] J. B. Gil and M. D. Weiner, “On pattern-avoiding Fishburn permutations”, *arXiv:1812.01682 [math]*, Dec. 2018, arXiv: 1812.01682. [Online]. Available: `http://arxiv.org/abs/1812.01682`.