# Computational Aspects *of* Orders, Posets and Lattices

Delyan Kirov

Mres Computer Science

University of St Andrews, School of Computer Science

October 2023

Abstract. The project focuses on the study of combinatorial structures known as posets with objective: to address enumeration questions surrounding lattices, improve existing permutation pattern algorithms, and generate hypercubes endowed with specific structural attributes. This research represents a confluence of innovative techniques drawn from the realms of SAT solving and computer algebra. Furthermore, this research extends its purview to the realm of SMT (Satisfiability Modulo Theories), in particular exploring the challenging domain of SMS (Satisfiability Modulo Symmetries). While SMS solvers remain a challenge for general problems, this project stands as an active contribution towards the realization of a versatile solver capable of addressing a broader class of combinatorial issues, thus expanding the horizons of SAT solving to lattice enumeration. Improved SAT models for permutation patterns are also presented.

This thesis is structured into four distinct parts. The initial chapters serve as a foundation, offering background information and the rationale behind the study of posets through the lens of SAT solving. The later sections delve into three separate problems, each accompanied by its individual literature review, methodology, findings, and discussion. A summary of the entire thesis is given in the final chapter.

I dedicate this master's thesis to two exceptional individuals who have been instrumental in its completion, not only through their guidance and support but also through their unwavering belief in my abilities.

To my dedicated supervisor, Dr. Ruth Hoffmann, your guidance, patience, and expertise have been the cornerstone of this research. Your commitment to excellence and your invaluable insights have been a constant source of inspiration.

To my supportive co-supervisor, Professor Christopher Jefferson, your depth of knowledge, enthusiasm, and dedication to advancing the field have been a driving force in this journey. Your mentorship and guidance have been instrumental in shaping the outcome of this work.

I am truly grateful for the opportunity to learn from both of you, and this thesis is a testament to your tireless dedication to the pursuit of knowledge. Thank you for believing in me and for your unwavering support.

# Contents

CHAPTER 1

# Basic Definitions

In this section, standard definitions used throughout the thesis are introduced. The reader is encouraged to go through this section even if they are familiar with very basic combinatorics, graph theory, and group theory, as some less-known definitions and notations are introduced.

Definition 1.1. *An $n$-**set** is the finite set $[n] := \{1, 2 \dots n-1\}$, where $n \in \mathbb{N}$. In this thesis, the natural numbers do not include* $0$.

Definition 1.2. *A **relation** $R$ between two sets $S$ and $T$ is a subset of $S \times T$. Instead of writing $R \subseteq S \times T$, we often write: $R : S \to T$. We also write $s\ R\ t$ or $R(s) = t$, instead of $(s, t) \in R$.*

*If the sets $S$ and $T$ are the same set, we say that $R$ is a relation over $S$.*

Theorem 1.1. *Let $R : A \to B$, $P : B \to C$ and $Q : C \to D$, be relations. Then the sets $P \circ R := \{(a, c) \mid \text{ exists } b \in B \text{ such that } (a, b) \in R \text{ and } (b, c) \in P\}$ and $Q \circ P := \{(b, d) \mid \text{ exists } c \in C \text{ such that } (b, c) \in P \text{ and } (c, d) \in Q\}$ are relations. Moreover the following holds: $(Q \circ P) \circ R = Q \circ (P \circ R)$. This property is known as associativity.*

Proof. See : [42] $\square$

Definition 1.3. *A **function** $f : S \to T$ is a relation, such that:*

*(1) For all $s \in S$, there exists $t \in T$, such that $(s, t) \in f$*
*(2) If $f(s) = t_1$ and $f(s) = t_2$, then $t_1 = t_2$*

Definition 1.4. *A function $f : A \to B$ is called **injective**, if for all $a, b \in A$, we have $f(a) = f(b) \iff a = b$. The function $f$ is called **surjective**, if for all $b \in B$, there exists an $a \in A$, such that $f(a) = b$. If $f$ is both injective and surjective, we say that $f$ is **bijective**. We write $|A| = |B|$.*

Definition 1.5. *Let $S$ be a set. The **identity function** $id_S : S \to S$ is defined by $id_S(s) = s$ for all $S \in S$.*

Definition 1.6. *Let $f : A \to B$ be a bijection. The function $f^{-1} : B \to A$ is called the **inverse** of $f$ when $f \circ f^{-1} = id_B$ and $f^{-1} \circ f = id_A$.*

Theorem 1.2. *Every function which has an inverse is bijection and every bijection has an inverse.*

PROOF. See [42] $\square$

THEOREM 1.3. *Let $A$ be a finite set. If $f : A \to A$ is injective then it is bijective. If $f : A \to A$ is surjective, then it is again bijective.*

PROOF. See [33] $\square$

DEFINITION 1.7. *Let $\pi : [n] \to [n]$ be a bijection, then $\pi$ is called a* ***permutation****. Let $a \in [n]$, then $a$ is called fixed if $\pi(a) = a$.*

DEFINITION 1.8. *Let $\pi$ be a permutation. The cyclic representation of $\pi = (a, b \dots f, g) \dots (i, j \dots k)$ is a way to write $\pi(a) = b \dots \pi(f) = g$, $\pi(g) = a \dots \pi(i) = j \dots \pi(k) = i$. Fixed points are either written $(a)$ or are left out.*

DEFINITION 1.9. *Let $\pi$ be a permutation of the set $[n]$. Let $s = a\ b\ c \dots\ d\ e$, be a sequence where $s(i) := \pi(i)$ for all $i \in [n]$.*

*So for example $\pi = 3\ 1\ 2$ is the bijection $\pi : [3] \to [3]$, where $\pi(1) = 3$, $\pi(2) = 1$ and $\pi(3) = 2$.*

DEFINITION 1.10. *A function $* : A \times A \to A$ is called a* ***binary operator*** *over $A$. Instead of $*(a, b)$, we write $a * b$ (infix notation). We say that $*$ is:*

*commutative* *If $a * b = b * a$*

*associative* *If $a * (b * c) = (a * b) * c$. We write $a * b * c$, as the parentheses are not relevant*

DEFINITION 1.11. *A* ***sequence*** *$T$ is function $T : [n] \to \mathbb{N}$ for some $n \in \mathbb{N}$ or a function $T : \mathbb{N} \to \mathbb{N}$.*

DEFINITION 1.12. *A* ***graph*** *$G$ is a set of finite sets $\{V, E\}$, where $E$ is a relation over $V$, that is: $E \in V \times V$. We call $V$ the set of vertices and $E$ the set of edges. You can assume through out this thesis that $V = \{1, 2 ... n\}$ for some $n \in \mathbb{N}$. We often write $E[G]$, to signify the edge set of $G$. The vertex set of $G$ is written $G$, instead of $V[G]$.*

DEFINITION 1.13. *A* ***path*** *in a graph $G$ is a sequence of vertices $(v_1, v_2 \dots v_n)$, such that all vertices are pair wise edges. That is: $(v_1, v_2), (v_2, v_3) \dots (v_i, v_{i+1}) \dots (v_{n-1}, v_n) \in E[G]$.*

DEFINITION 1.14. *A graph $G$ is called* ***connected*** *if there exists a path between any two edges.*

DEFINITION 1.15. *A graph $G$ is called* ***simple*** *if:*

*(1) For all edges $(v, u) \in G$, is equivalent to $(u, v) \in G$*
*(2) For all vertices $v \in G$, the loop $(v, v) \in G$*
*(3) $G$ is connected*

*In the literature, one can also find the negation of* (2).

Definition 1.16. *A graph $G$, with a vertex set and $n$-set is called* ***directed*** *if:*

*(1) $G$ is connected*
*(2) For all edges $(v, u) \in G$ and $v \neq u$ implies that $(u, v) \notin E[G]$*
*(3) For all $v \in G$ the loop $(v, v) \in G$*
*(4) For all $(v, u) \in G$, we have $v \geq u$*

*The last constraint (3) is not standard in the literature, usually loops are excluded. Similarly, constraint (4) is not required by definition.*

Definition 1.17. *A graph $G$ is called* ***acyclic*** *if there is no path from $v$ back to itself, for any $v \in G$, except for the loop $(v, v)$.*

Definition 1.18. *A graph which is acyclic and directed is called a* ***DAG*** *(acyclic directed graph).*

Definition 1.19. *A* ***group*** $(G, S)$ *is a tuple, where $S$ is some set and $G$ is a set of bijections over $S$, such that:*

*invertible For all $f \in G$, we have $f^{-1} \in G$*
*closed If $f, g \in G$, then $f \circ g \in G$ and $g \circ f \in G$*

*Note that in the literature, this is called a group action of $G$ over $S$, the definition of a group is more abstract.*

Definition 1.20. *The symmetric group $S_n$ is the group of all permutations over* $[n]$*, for $n \in \mathbb{N}$.*

Definition 1.21. *Two graphs $G$ and $H$ are* ***isomorphic*** $G \cong H$ *if there exists a bijection $\pi : G \to H$, such that if $(v, u) \in E[G]$ is equivalent to $(\pi(v), \pi(u)) \in H$.*

*This defines a bijection between the edges of $G$ and $H$, by $\pi_E : E[G] \to E[H]$, where $\pi_E((v, u)) = (\pi(v), \pi(u))$. We write $\pi(v, u)$ for $\pi_E((v, u))$, which is called the action of $\pi$ on the edge $(v, u)$.*

Definition 1.22. *Let $G$ be a graph and $\pi \in S_n$. If $\pi(G) \cong G$, we call $\pi$ a graph isomorphism. If $\pi(G) = G$, we call $\pi$ a graph* ***automorphism*** *of $G$.*

Theorem 1.4. *The set of all isomorphism of a graph form a group. If the graph is simple, this group is $S_n$.*

Proof. Indeed an isomorphism is just a relabeling of the vertices and this relabeling is reversible. For a formal proof see [67] □

Definition 1.23. *Let $Aut(G)$ be the set of automorphism of a graph $G$. Let $Sym(G)$ be the set of isomorphisms of $G$. The set of non trivial isomorphisms $R(G)$ is the set of all isomorphisms of $G$ that are not automorphisms.*

Note that the larger $Aut(G)$ is, the smaller $R(G)$ is. This will become important in later sections.

THEOREM 1.5. *$Aut(G)$ and $Sym(G)$ are groups.*

PROOF. See [67] $\square$

THEOREM 1.6. *Isomorphism of graphs is an equivalence relation over the set of all finite graphs with $n$ vertices $\mathcal{G}_n$, that is:*

*(1)* $G \cong G$
*(2)* $G \cong H \iff H \cong G$
*(3)* $G \cong H$ *and* $H \cong K$ *then* $G \cong K$

PROOF. See [67] $\square$

DEFINITION 1.24. *A **matrix** $M_n[R]$ over a set $R$ is a function $M_n : [n]\times[n] \to R$. We define $M_n[i, j] := M_n(i, j)$.*

DEFINITION 1.25. *The matrix representation $M(G)$ of a graph $G$ with $n$ vertices, is a matrix $M_n[F_2]$ over the Boolean Ring $F_2 := \{ 0, 1 \}$. Where $M[i, j] = 1 \iff (i, j) \in G$ and zero otherwise.*

DEFINITION 1.26. *Let $G$ be a connected graph. The **simplification** of $G$ is the graph $\overline{G}$, with the same vertex set as $G$. The edges of $\overline{G}$ are $E[\overline{G}] := \{ (v, u) \mid (u, v) \in E[G] \} \cup E[G] \cup \{ (v, v) \mid v \in G \}$.*

THEOREM 1.7. *$\overline{G} \ncong \overline{H} \implies G \ncong H$ for all graphs $G$ and $H$.*

PROOF. Consider the negation: $G \cong H \implies \overline{G} \cong \overline{H}$. Let the bijection be $\pi : G \to H$. Then the action $\pi(v, u) = (\pi(v), \pi(u))$ is well defined for $\overline{G}$ ,where $(v, u) \in E[\overline{G}]$. Therefore $\overline{\pi} : \overline{G} \to \overline{H}$, defined by $\overline{\pi}(v) = \pi(v)$ is a well defined graph isomorphism. $\square$

This is useful in that given a set of isomorphic copies of $G$, we can count the number of non isomorphic graphs by counting the simplified graphs instead.

DEFINITION 1.27. *The **boolean algebra** $\{ F_2, \vee, \wedge, \neg \}$, is a set where $F_2 := \{ 0, 1 \}$, together with binary operators $\vee : F_2 \times F_2 \to F_2$, $\wedge : F_2 \times F_2 \to F_2$ and $\neg : F_2 \to F_2$, corresponding to the usual disjunction, conjunction, and negation operators. A propositional formula can be built using the following rules:*

*(1) Let $x$ be an indeterminate, then $x$ and $\neg x$ are a propositional formulas, also called literals.*
*(2) If $x_1$ and $x_2$ are indeterminate values, then, $x_1 \vee x_2$ is a propositional formula, called a clause.*

*(3) If $x_1$ and $x_2$ are indeterminate values, then $x_1 \wedge x_2$ and $\neg x_1$ are propositional formulas.*
*(4) If $f$ and $g$ are propositional formulas, so are: $f \vee g$, $f \wedge g$ and $\neg f$.*

DEFINITION 1.28. *A propositional formula is in **conjunctive normal form**, if it is a conjunction of clauses, or only a single clause.*

The following are in Conjunctive Normal Form. This reordering is always possible[38].

(1) $(A \vee B) \wedge (C \vee D)$
(2) $A \wedge (B \vee C)$
(3) $(A \vee B)$
(4) A

CHAPTER 2

# Computer Specifications

Results run through out this thesis have been run on my personal computer, with specifications:

**System Information:**

- Kernel: 6.5.6-200.fc38.x86_64
- Architecture: x86_64
- Distribution: Fedora release 38 (Thirty Eight)

**Machine Information:**

- Type: Laptop
- Model: ThinkPad E15 Gen 4
- Motherboard Model: 21E6006VBM

**CPU Information:**

- Model: 12th Gen Intel Core i5-1235U
- Cores: 10 (2 multi-threaded, 8 single-threaded)
- Bits: 64
- Type: MST AMCP
- Cache: L2: 6.5 MiB

**CPU Speeds:**

- Average Speed: 824 MHz
- Min/Max Speed: 400/4400 MHz (turbo: 3300 MHz)
- Core Speeds: 685 MHz (average)

Some experiments have been run on the university machines, with specifications:

**Compute Nodes:**

- Number of Nodes: 110
- Cores per Node: 32
- Processor Type: Intel Broadwell (Xeon E5-2683 or Gold 6130)
- Processor Speed: 2.1GHz
- Total Cores in Cluster: 3,520
- Memory per Node: Between 128GB and 1.5TB
- GPUs: Two nodes equipped with two NVidia Tesla V100 GPU processors each
- Local Scratch Disk: Available on each node

**Storage:**

- Shared Disk Space: 400TB
- Scratch Space: Available via the parallel GPFS file system

CHAPTER 3

# SAT models and solvers

## 3.1. What is SAT solving

When solving a problem with a computer, the programmer typically formulates algorithms that provide instructions to the computer, which are executed sequentially to achieve the desired outcome. This conventional approach, although seemingly straightforward, has several inherent limitations. One of these limitations is the frequent absence of a well-defined algorithm to tackle the problem of interest. In such cases, programmers may resort to a brute-force search method to locate the desired solution, but even this approach can be challenging and often lacks efficiency when compared to alternative methods[45]. When the sought-after solution possesses finite and easily expressible constraints, another viable approach is the utilization of a satisfiability solver, often abbreviated as SAT, to determine the solutions to our problem.

SAT solvers[18] are algorithms designed to solve formulas of boolean variables. If we take a propositional formula $f$, that takes variables $x_1, x_2 \dots x_n$, and require $f(x_1, x_2 \dots x_n) = 1$, then we can find one or all solutions using a SAT solver.

A propositional formula can capture many more complicated constraints, including **for all** statements, **exists** statements, implications and more[19]. Currently, most solvers cannot handle arbitrary nesting of `exist` and `for all`, so for instance: `exists` $t_0$ `in` $T_0$ `such that (exists` $t_1$ `in` $T_1$ `(... (exists` $t_n$ `in` $T_n$`)))` is not allowed for general $n$, although this limitation is in general not a big problem in practice.

In the past 30 years, SAT solvers have been used to solve many classical problems in combinatorics, such as the famous Sudoku game[59], the eight queens puzzle[63], etc. They have also been used to solve problems in industry, for example in software source and version control[66], circut design[81] and more.

Several algorithms exist for solving Boolean formulas, with the most renowned ones being DPLL (Davis–Putnam–Logemann–Loveland) [64] and CDCL (Conflict-driven clause learning)[5]. The fundamental concept behind these algorithms is as follows: Initially, the Boolean formula is transformed into Conjunctive Normal Form (CNF). Once the formula is in CNF, a random value is assigned to one of the Boolean variables, and a node is created. Subsequently, the formula is simplified, and if it remains satisfied, the algorithm assigns a value to another variable, generating a new node and establishing a search tree structure. If, at any point, the formula becomes unsatisfiable, the search process reverts to the earliest possible assignment, effectively backtracking. This process continues until either a solution is found or the search tree is fully explored. CDCL and DPLL are the most widely employed algorithms, with CDCL approaches being more prevalent among solvers[48]. One key distinction between the two lies in their strategies for backtracking when an assignment proves infeasible to yield a result[69]. It is worth pointing out that more advanced solvers enhance these algorithms by employing sophisticated heuristics instead of random value assignment[32]. . Additionally, there are parallel approaches that build upon the foundations of CDCL and DPLL, often employing divide and conquer algorithms[60].

SAT solving can be used to solve a related problem SMT (satisfyability, module theories)[27]. In particular, the problem of SMS(satisfiability modulo symmetries) is quite relevant to this project. While there have been SMS solvers created[53][52], none of them can be used for general problems at the time of writing, but attempts to make a solver for a more general class of problems has been proposed and tested[51].

## 3.2. Savile Row

Savile Row[65] is a combinatorial optimization tool primarily employed for constructing models in the Essence language. Essence serves as a widely recognized standard, embraced by most constraint solvers, allowing models written in Essence to be solved by various backend solvers. This interoperability proves to be advantageous, as SAT solving, in general, poses a formidable computational challenge. In fact, SAT solving was among the first problems classified as NP-complete, a seminal discovery known as the Cook-Levin theorem [65]. Therefore, different solving techniques, encompassing a range of randomization algorithms and statistical approaches, can yield diverse results contingent upon the specific model.

Savile Row offers valuable abstractions, such as implications, tables, relations, functions, matrices, and loop-like syntax, featuring 'for all' and 'there exists,' among others. All these abstractions are systematically transformed into Conjunctive Normal Form (CNF) before the initiation of the solving process. This preprocessing step renders models authored in Savile Row notably comprehensible and amenable to reasoning. While it is possible to introduce constraints manually or via Python scripting, dealing with intricate existential quantifiers ('exists,' 'for all,' 'does not exist,' 'exists unique') over extensive domain sets is often infeasible by manual means and inefficient when using rudimentary algorithms [65].

Another important advantage of Savile Row lies in its incorporation of numerous hidden optimizations, substantially diminishing solving time in most cases. Nonetheless, it is essential to acknowledge that these advantages do come with certain drawbacks, which will be expounded upon in the concluding section, specifically regarding Permutation Patterns.

### 3.3. Conjure

Conjure[1] is a parcing language written as an interface to Savile Row. It provides a simpler environment for writing models and using Savile Row features, it is also convenient for preventing Savile Row errors. Here is a cheat-sheet for the reader, to help understand the models in later sections.

(1) The characters `/\` are used for the binary operator **and**.
(2) The characters `\/` are used for the binary operator **or**.
(3) The character `!` is used for logical **negation**.
(4) The characters `->` are used for **implies**.
(5) Similarly `<->` are used for **equivalent to**.
(6) Intervals are written `start..end`.
(7) There is support for `forAll` and `exists` quantifiers, that take a variable and quantify over some domain set, like this: `forAll i in (int 0..10)`.
(8) Many useful data structures are available, in particular matrices, pairs, sets, multi-sets and more.
(9) Relations are supported, written: `relation of` $S_1 * S_2 * S_3$, for some specified domain set `S`. Similarly, there is support for functions: `letting f be: function (total)` $S_1$ `->` $S_2$. We may also require the function to be injective, surjective, bijective. If these constraints are not specified, the function is assumed to be partial.

### 3.4. Computer Algebra Systems

A symbolic computer system, more commonly reffered to as computer algebra system(CAS for short) is software designed to manipulate mathematical expressions, that follow user defined inference rules[15]. CASes have been used to factor polynomials[6], check the validity of formulas[80], finding solutions to equations[80], theorem proving[6] code validation and cloud computing[10], etc.

The advantage of a CAS is that it can calculate a closed form for an expression[80], unlike other software that can only produce bounds for a solution. This is especially useful in combinatorics, where the formulation of a problem requires an exact solution. (Much work has been done to restate graph theory problems to use techniques from analysi[12]).

In this project, I have used different CASes to generate SAT models, symmetry constraints and to check the validity of my solutions. Here is a quick intoduction to them.

### 3.4.1. GAP

GAP[35] (Groups, Algorithms and Programming) is an open source CAS, originally specializing in Group theory algorithms, is now a general tool for symbolic computation. In this thesis, I have used the built in tools for generating an Orbit Stabilizer Chain[46] for the group $S_n$ over an $n$-set. I have also used the package Yags[17] as an interface for Nauty[61] and for its many convenient functions to generate the points of a specified graph.

### 3.4.2. Nauty

Nauty[61] is a tool for solving 3 related graph theory problems. The first is finding the Automorphism Group of a Graph[82]. The second is finding if two graphs are isomorphic[82] and lastly, generating a "Canonical Form"[61]. Since there is currently no known polynomial time algorithm, different strategies tend to work differently depending on the sample of given graphs. In some cases (trees or graphs with boundedness constraints) some approaches work significantly better then others. Nauty (and its companion Traces) have proven to be the best in many different subcases[61], while not requiring the user to choose which approach the algorithm to use, which is why I chose it for this project.

There is an interface to call different algorithms called Dreadnaut[61]. I find this tool hard to use, which is why I recommend using the functions inside GAP or by importing the Python module pynauty[30]. On the other hand, calling Dreadnaut inside other programs as a subprocess is much more efficient then other methods.

### 3.4.3. SageMath

SageMath[77] is similar to GAP. It is open source an contains many different packages, including GAP, Nauty and more. In particular, I have used their package for Posets and Lattices[77]. The biggest advantage of SageMath is that it contains many different libraries that can be easily used together, without needing to write parsing algorithms. It also supports many visualization tools and since it is written in python and uses a similar language to call its different functions, SageMath is excellent tool for complicated calculations with not much effort. The downside is that, while the libraries in SageMath are fast and efficient, the environment itself is not[84]. When working with large number of big objects, it is often best to use GAP instead if possible.

CHAPTER 4

# Orders and Partial Orders

## 4.1. Introduction to Order Theory

DEFINITION 4.1. *Let $(P, R)$ be a set $P$ and $R$ a relation over $P$. We say $(P, R)$ of just $P$ is a* ***poset*** *if:*

*(1)* $(a, b) \in R$ *and* $(b, a) \in R \iff a = b$

*(2)* $(a, b) \in R$ *and* $(b, c) \in R \implies (a, c) \in R$

*From now on we shall write $(P, \leq)$ or even just $P$ if the relation is clear, where $(a, b) \in R$ is denoted simply $a \leq_P b$ or even $a \leq b$.*

Consider the following pictures as examples: Figure 1, Figure 2. The natural numbers $(\mathbb{N}, \leq)$, form a poset, with their usual ordering. Any subset of the natural numbers is also ordered.

On the other hand, the natural numbers also form a poset by the relation "divides", where if $a$ divides $b$, we write $a \leq_{n\mathbb{Z}} b$. We assume $1 \leq_{n\mathbb{Z}} n \leq_{n\mathbb{Z}} 0$ for all $n \in \mathbb{Z}$. This is also called the lattice of ideals of $\mathbb{Z}$.

Another related example is the power set of a set $S$, usually written $\mathcal{P}[S]$, this set is naturally ordered by set inclusions, as shown in Figure 2 for the set $\{1, 2, 3\}$.

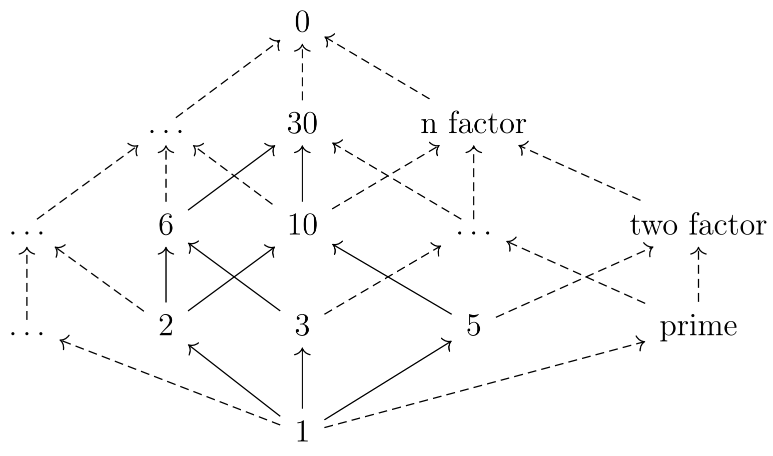

FIGURE 1. Partial drawing of the lattice of ideals of $\mathbb{Z}$. The places labeled prime, two factor and n-factor represent numbers with one, two and n-factors

$$\begin{tikzcd}
 & \{1,2,3\} & \\
\{1,2\} \arrow[ur] & \{2,3\} \arrow[u] & \{1,3\} \arrow[ul] \\
\{1\} \arrow[u] \arrow[urr] & \{2\} \arrow[ul] \arrow[u] & \{3\} \arrow[ul] \arrow[u] \\
 & \emptyset \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

FIGURE 2. Lattice of subsets of $\mathcal{P}(\{1,2,3\})$.

$$\begin{tikzcd}
 & 1 \arrow[r] & 2 \arrow[dr] & \\
\text{source} \arrow[ur] \arrow[dr] & & & \text{target} \\
 & 3 \arrow[r] & 4 \arrow[ur] &
\end{tikzcd}$$

FIGURE 3. Directed Graph

DAGs have a natural poset relation given by $a \leq b$ if there exists a path from $a$ to be $b$. See Figure 3.

It is often important to have a notion of when two posets are "essentially the same" and also when a poset can be found in another poset. The following definitions makes this precise.

DEFINITION 4.2. *Let $(P, \leq_P)$ and $(Q, \leq_Q)$ be two posets. It is said that $P$ and $Q$ are isomorphic if there exist two function $\sigma : P \to Q$ and $\sigma^{-1} : Q \to P$ such that:*

*(1) The functions $\sigma$ and $\sigma^{-1}$ are **bijective** and inverses of each other. So $\sigma \circ \sigma^{-1} = id_Q$ and $\sigma^{-1} \circ \sigma = id_P$.*
*(2) Both $\sigma$ and $\sigma^{-1}$ are **monotonic**, that is: $a \leq_P b$ implies $\sigma(a) \leq_Q \sigma(b)$, similarly for $\sigma^{-1}$.*

The reader may assume that requiring the existence of a monotonic inverse of $\sigma$ is superfluous, but this is not the case. Consider the poset $Q := \{1, 2, 3, 4\}$ ordered by "bigger or equal" and the lattice $P$ Figure 4 . There is a bijection from $P$ to $Q$, which is also monotonic, however, we definitely do not want to regard these two posets as equivalent, as they have different properties (one is a total order, while the other is not). The inverse of this function on the other hand, is not monotonic, which solves the problem. Note that poset isomorphism is an equivalence relation[74].

$$\begin{tikzcd}
4 & & & & \\
3 \arrow[u] & & & 4 & \\
2 \arrow[u] & & 2 \arrow[ur] & & 3 \arrow[ul] \\
1 \arrow[u] & & & 1 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

FIGURE 4. The lattices $P$ and $Q$

DEFINITION 4.3. *Let $(P, \leq_P)$ and $(Q, \leq_Q)$ be two posets. It is said that $P$ is a subposet of $Q$ if there exists a an injection $\sigma : P \to Q$, which is monotonic.*

Note that this definition does not require $P$ to be a subset of $Q$, thereby making the notion of substructure "label invariant"[8]. The importance of this will become clear in latter sections.

DEFINITION 4.4. *Let $(P, \leq_P)$ be a poset. If for all $x, y \in P$, either $x \leq_P y$ or $y \leq x$, then $P$ is called a total order.*

DEFINITION 4.5. *Let $(P, \leq_P)$ be a poset. Let $Q$ be a poset with points the same as $P$ and order reversed from that of $Q$. So $a \leq_Q b \iff b \leq_P a$. The poset $Q$ is called the dual of $P$.*

## 4.2. Orders and SAT solving

There are three motivating reasons to study ordering relations and SAT solving together. The first is that many problems can be modeled easily as ordering constraints, which are easily translated into boolean constraints. Indeed, directed acylic graphs (DAG for short) can be used to model circuit designs, scheduling tasks and many other problems. In this dissertation, we use DAGs as a convenient model for DNA patterns.

The second is that SAT solvers prove to be useful tools to solve many interesting problems in order theory. We illustrate this by enumerating a particular kind of order object, called a geometric lattice.

Finally, ordering relations can help in the solving process, by reducing symmetry. This is already done to some extent by the solver but by introducing ordering constraints, we can reduce the search even further. Since we shall encounter this problem twice in later sections, let us introduce the problem of breaking symmetry.

To illustrate the previous point, consider the simple formula $x_1 \vee x_2$. Applying the transformation $\pi = (1\ 2)$ to the indexing of the variables, we get $x_2 \vee x_1 = x_1 \vee x_2$. This seems innocent at first, but notice that we do not have check all 4 possibilities

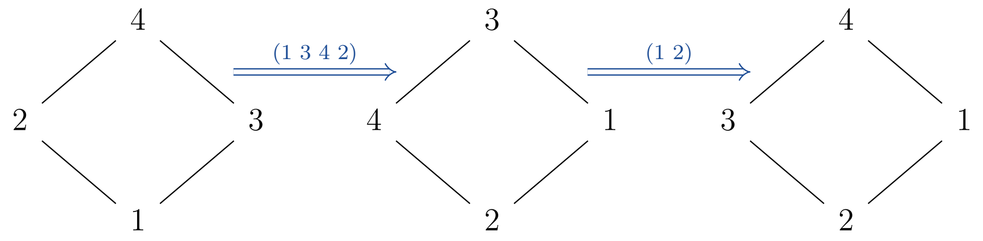

FIGURE 5. Isomorphic graphs with different relabeling.

for $x_1$ and $x_2$, we only need to check 3. At a larger scale, we can reduce the search tree by a significant amount if we have data about the symmetries of the model. SAT solvers tend to be very efficient at exploiting these kinds of shortcuts.

I will wrap this section with an important result used implicitly later on.

DEFINITION 4.6. *For any finite poset $P$, we say that a point $x$ covers $y$ and write $x :> y$, if $y \leq_P x$ and for any $z \in P$, where $z \leq_P y$ implies $x \leq_P z$ . The cover graph of $P$ is the graph with the same points as $P$ and $(x, y)$ is an edge $\iff$ $x :> y$.*

*Informally we may say that the cover graph of $P$ is its underlying graph.*

THEOREM 4.1. *The cover graphs of two non-isomorphic posets is non-isomorphic. A cover graph is a DAG.*

PROOF. See [79] □

## 4.3. Symmetry

Let us go back to symmetry in graphs and how this relates to SAT solving. Note that when considering problems in graph theory, such as degrees of vertices, connectivity constraints on vertices and edges and others, there is no difference what labeling the graph is given. We call these graph properties **label invariant**. When generating graphs with these properties, it is pointless to consider all possible isomorphic copies of a graph. When we are indeed interested in properties which depend on the labeling, it is generally easier to simply find the automorphism group of the graph in Nauty and relabel the edges by considering only non trivial isomorphisms. When modeling problems in SAT, the presentation of the problem makes a big difference, since some graph presentations have less isomorphisms then others. Here is a picture to illustrate the concepts compare Figure 5 and Figure 6

All graphs presented are isomorphic. The bijections between them are marked by the simple permutation notation.

For our intents and purposes, it is good if a graph has more automorphisms, because then the solver will not generate as many isomorphic solutions. Indeed, solvers have very good heuristics to deal with this problem[38]. On the other hand, the solver itself does not have a way to tell isomorphic solutions which are not the

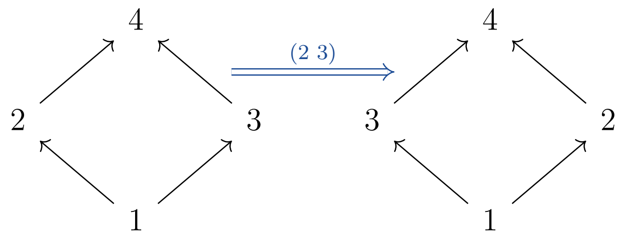

FIGURE 6. Directed graphs, with identical graph presentation. The permutation is a graph automorphism.

same, for the solver, these solutions are valid and unique. Since any isomorphic solution can be trivially obtained by applying a bijection, generating one copy out of each isomorphic class gives us all solutions. Therefore, generating isomorphic copies is more work that slows down the search for no benefit. Much effort has been put to reduce or outright eliminate isomorphic solutions.

One way to do this is by restating the problem. Different representations of the same object may have vastly different symmetry structures. To demonstrate, consider Figure 6.

We see the same graphs as before, except this time, we directed the edges and required $(a, b) \in G \implies a \leq b$. There is only one graph with this property, but we have shown two. What is going on here, is that the drawing itself (so called plane embedding) introduces more symmetry. We see that flipping the vertices 2 and 3 is actually a graph automorphism.

This is quite convenient, especially if the underlying graph has no cycles, that is, it us a DAG, which means that it can be labled using the ordering of the poset it defines. Unfortunately, not all symmetries can be broken this way.

### 4.3.1. Order of isomorphic copies

In this chapter, I present how to reduce isomorphic solutions in SAT models.

DEFINITION 4.7. *Let $A, B$ be two sequences with length $n$ and coefficients from a totally ordered set. We say that $A$ is* ***lexicographically*** *less then $B$ and write $A \leq_{lex} B$ if there exists $i \leq n$, such that for all $j < i$ we have $A(j) = B(j)$ and $A(i) \leq B(i)$.*

DEFINITION 4.8. *An $n$ by $n$ matrix $A$ is* ***lexicographically*** *smaller then an $n$ by $n$ matrix $B$ if the sequence $[A[1], A[2]...A[n]]$ of rows of $A$ is lexicographically smaller then $[B[1], B[2] \dots B[n]]$, the sequence of rows of $B$. Spelled out, there exists $i \leq n$ such that for all $j < i$ we have $A[i] = B[i]$ and $A[j] \leq_{lex} B[j]$.*

It is well known that this produces a total order on matrices[29]. If the entries are booleans, then the order also has a maximum and minimum element, that is, the order is total.

A lexicographic constraint is a labeling constrain of the vertices of a graph which reduces the lexicographic order of the solutions. That is to say, let $f$ be a constraint for the labeling of the graphs and assume out a SAT model $\mathcal{M}$ produces 3 graphs as solutions, $G_1, G_2, G_3$. Then adding the constraint $f$ to $\mathcal{M}$ will require the lexicographic order of the new graphs to be smaller or equal to the graphs $G_i, i \in \{1, 2, 3\}$. This will either produce the same solutions, or reduce the number of solutions. If the constraint $f$ reduces the solutions, we say that $f$ is effective, otherwise we say that $f$ is redundant. These types of constraints are referred to as **static symmetry breaking constraints**.

Using this lexicographic ordering, we can encode constraints which look for the lexicographically smallest or largest solution[29]. Breaking all symmetries is difficult using only this approach. The reason is that in the worst case, we need $n!$ constraints to ensure the matrix is the lexicographically biggest or smallest one[29]. Moreover, most of these constraints are usually redundant. We thus require a subset which is small and not redundant. Two methods have been tries in this paper, one first introduced by Codish[21], referred to as Codish constraints in this thesis. I have also tried using the constraints introduced by Jefferson[49], which are referred to as SGS in this thesis.

CHAPTER 5

# Lattices

DEFINITION 5.1. *A* lattice $(L, \leq, 0, 1)$ *is a* ***poset****, such that:*

*Meet* *For all $a, b \in L$, there exists $c \in L$, such that $a \leq c$ and $b \leq c$. Moreover, if $d \in L$, with $a \leq d$ and $b \leq d$, then $c \leq d$. We call $c$ the meet of $a$ and $b$.*

*Join* *Dually, reversing the order we have a join for all $a, b \in L$. That is, there exists $c \in L$, such that $c \leq a$ and $c \leq b$ and if $d \in L$ with $d \leq a$ and $d \leq b$ then $d \leq c$.*

*Bounded* *We require $0, 1 \in L$ with $0 \leq a \leq 1$ for all $a \in L$. We call $0$ the infimum and $1$ the supremum.*

Let is look at a concrete example. Consider the natural numbers again, but this time attach a *"point at infinity"* and say that every natural number is less then the *"the point at infinity"*. The meet of two numbers is the bigger one, the join is the smaller one, in other words the *min* and *max* functions. Similarly, any fully ordered set is a lattice in this way, including $\mathbb{Q}$ and $\mathbb{R}$, if we add two points of positive and negative infinity.

Another example is the powerset $\mathcal{P}[S]$ for some set $S$. The supremum is $S$, the infumum is the empty set $\emptyset$. For two sets $S, T \in \mathcal{P}[S]$ the meet is given by the set intersection $S \cap T$, while the join by the set union $S \cup T$. This example extends to include the lattice of normal subgroups and the lattice of groups and in general, the lattice of some equivalence relations over a set.

There is a more algebraic definition, which tends to be useful when working with lattices.

DEFINITION 5.2. *A lattice $(L, \vee, \wedge, 0, 1)$ is a set with two binary operations: $\vee : L \times L \to L$ and $\wedge : L \times L \to L$ and two distinguished points $0, 1 \in L$ with the following properties, for all $a, b \in L$ we have:*

*Idenpotent* $a \vee a = a = a \wedge a$

*Commutative* $a \vee b = b \vee a$ *and* $a \wedge b = b \wedge a$

*Absorptive* $a \vee (a \wedge b) = a = a \wedge (a \vee b)$

*Zero* $a \wedge 0 = 0$ *and* $a \vee 0 = a$

*Unit* $a \wedge 1 = a$ *and* $a \vee 1 = 1$

The connection between these two definitions is as follows: the meet and join operators are the same in both and the following holds $a \wedge b = a \iff a \leq b$.

Note that some definitions of a lattice require aslo that both operations **meet** and **join** be associative, that is $(a \wedge b) \wedge c = a \wedge (b \wedge c)$, dually for **join**. This constraint is redundant[11]. Moreover, for finite lattices specifically, the existence and interaction of a meet operator is implied by the existence of a joint operator and vice versa[11]. I have chosen to put all constraints in the SAT models, because from initial testing, the solver was able to calculate all lattices up to size 7 faster with the extra constraints. It could be interesting to see if this is also the case for the other models. For a similar reason, I have chosen to use the two definitions together, because the poset definition give a good symmetry breaking constraint, which makes the solving process faster.

Let us introduce some special cases of lattices

Definition 5.3. *Let $(L, \vee, \wedge, 0, 1)$ be a lattice. We say that $L$ is* ***distributive*** *if for all $a, b, c \in L$ we have that $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ holds.*

*It is also useful to note that the dual identity $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$, also holds.*

Definition 5.4. *Let $(L, \vee, \wedge, 0, 1)$ be a lattice. We say that $L$ is* ***modular*** *if for all $a, b, c \in L$ we have that $(a \wedge b) \vee (c \wedge b) = ((a \wedge b) \vee c) \wedge b$ holds.*

Definition 5.5. *Let $(L, \vee, \wedge, 0, 1)$ be a lattice. We say that $L$ is* ***Geometric*** *if:*

*Atomistic* *An atom $a \in L$ is an element bigger then the zero element $0$ and smaller then any other element. A lattice is* ***atomistic*** *if every element is can be expressed as $\bigvee_{a \in A} a$, where $A$ is a set of atoms, that is, every element is the meet of atoms.*

*Graded* *A function $r : L \to \mathbb{N}$ is called a* ***grading*** *if $r(a) \leq r(b)$ when $a \leq_L b$ and $r(a) = r(b) + 1$ when $b$ covers $a$ (there is no element between $a$ and $b$ and $a \leq b$). It is also required that for all $a, b, c \in L$ the expression $r(x) + r(y) \geq r(x \wedge y) + r(x \vee y)$ holds.*

Definition 5.6. *Let $(L, \vee, \wedge, 0, 1, r)$ be a graded lattice. Given two numbers $a, b \in L$, let $[a, b] := \{\, c \in L \mid a \leq c \leq b \,\}$ be the interval between $a$ and $b$. If for every non trivial interval, the set $[a, b]$ contains as many elements of even and odd rank, that is: $|\{\, c \in [a, b] \mid r(c) \text{ is even} \,\}| = |\{\, c \in [a, b] \mid r(c) \text{ is odd} \,\}|$, then we call $L$* ***Eulerian****.*

From a computational perspective, there are a few interesting directions for the study of lattices. One is to note that they form an **equational theory** [22], this just means that they can be described as sets, satisfying a collection of equations, as we have seen before. They have an associated monad, the power-set monad, sending a set to the lattice of subsets. Many theorems about lattices can be checked using software like Agda [41]. Another direction is to consider some combinatorial questions about lattice, such as their numerosity. An algorithm that counts finite lattices up to symmetry has been created [22], more recent developments have been able to improve the results up to size 20 [37]. We also have results for Modular and Distributive lattices [26]. We attempt to count the number of modular lattices using a different approach, namely SAT-solving.

## 5.1. SAT model for lattices

The following essence code can be used to generate any lattice. Let is break it down. We give the solver the number of points of the lattice, here denoted $n$. The meet and join operations are rendered as total functions over the $n$-set $N$. The graph cover is presented as a function $C$.

To start we first give the number of points we want the lattice to have, which is represented as **n**. Then the model asks the solver to find the meet, join, cover and rank relations, represented as **M**, **J**, **C** and **R**.

```
language ESSENCE 2.4.0

given n : int
letting N be domain int(0..n-1)

find M : function (total) (N,N) --> N
find J : function (total) (N,N) --> N
find R : function (total) N --> N
find C : function (total) (N,N) --> int(0..1)
```

The the model is given the constraints which these relations need to satisfy. Note that the binary relations are represented as functions and use function notation. With these constraints we can model any lattice.

```


such that

$Associative
forAll i,j,k: N. M((i,M((j,k)))) = M((M((i,j)),k)),
forAll i,j,k: N. J((i,J((j,k)))) = J((J((i,j)),k)),

$Commutative
forAll i,j: N. M((i,j)) = M((j,i)),
forAll i,j: N. J((i,j)) = J((j,i)),

$Absorption
forAll i,j: N. M((i,J((i,j)))) = i,
forAll i,j: N. J((i,M((i,j)))) = i,

$Idempotent
forAll i: N. M((i,i)) = i,
forAll i: N. J((i,i)) = i,
$Bounded
forAll i : N. M((n-1,i)) = i,
forAll i : N. J((0,i)) = i,
```

Additionally, the model is given constraints to model the geometric lattice constraints and a simple symmetry breaking constraint, that ensures point labels follow the lattice ordering.

```


$Symmetry breaking
forAll i,j : N. (R(i) > R(j)) -> (i > j),

$Cover relation
forAll i,j:N. (C((i,j)) = 1) <->
((J((i,j)) = i) /\ !(i = j) /\
!(exists k : N. !(i = k) /\ !(j = k) /\
(J((i, k)) = i) /\ (J((k,j)) = k))),

$Graded
forAll i,j: N. ((J((i,j)) = i) /\ !(i = j)) -> (R(i) > R(j)),
forAll i,j: N. (C((i,j)) = 1) -> (R(i) = R(j) + 1),

$Semi-modular
forAll i,j: N. R(i) + R(j) >= R(M((i,j))) + R(J((i,j))),

$Atomistic
forAll i: N. (R(i) = 1) \/ (R(i) = 0) \/
(exists j,k : N. !(i = k) /\ !(i = j) /\ (J((j,k)) = i)),
```

It is easy to use this as a base and define additional constraints that the lattice needs to have. In particular I have chosen to generate Geometric and Eulerian lattices.

## 5.2. Previous work

The enumeration strategy for lattices has followed three paths. One is to compute lower and upper bounds, which has proved difficult. One of the first papers to go with this approach [55] managed to provide upper and lower bounds which are not particularly close to known results. More then 20 years after, these bounds were slightly improved [34]. Since then, little progress has been made to tighten these bounds using non computer aided methods, except for some specific cases, such as modular and semi-modular lattice [25]

To produce example and to generate random samples of lattices, but not necessarily to enumerate all lattices exist. One approach taken by [3] is to generate a lattice using a random poset. The approach is interesting, so I will describe it here. Every lattice (indeed this also extends to posets) $L$ has subsets $F$ known as **filters**, which are sets where for all $a, b \in L$, there exists $z \in L$ such that $a \leq z$ and $b \leq z$. Moreover, for all $x \in F$, if $p \geq x$ then $p \in F$. A known result is that filters of lattices form posets [40] and these posets can be used to "reconstruct" the lattice [72]. The problem with this approach is that, while unique posets produce unique lattices (up to isomorphism), there is no known way to check what the size of the resulting lattice would be. There are some known lower bounds [83]. On the other hand, if we take a random sample of posets, the resulting lattices are essentially random [85]. This approach is suitable for generating random samples, which has found application in other parts of mathematics see [76]

A computational approach has been to generate general graphs or posets and then prune the results to search for matching lattices. This method first generated known results of all lattices up to size 10 [56]. The algorithm used in that paper is essentially a brute-force search that generates a poset and does some smart checks in order to avoid generating unnecessary posets. Unfortunately, some of the other results presented in that paper were wrong for sizes 12 and 13.

Another method is generating the structures using a recursive algorithm, first introduced by Heitzig and Reinhold [43], which produced results up to 14 points. The idea requires three results.

The first is that atoms (that is, elements hovering the zero point) can be removed from any lattice with $n$ points to produce a lattice with $n-1$ points [36]. The second is slightly more involved in its formulation, but essentially it says that any $n+1$ lattice can be built from some $n$ lattice by attaching an atom.[22] In the paper [44] gives a canonical form that is convenient to break symmetries with this formulation.

With this symmtry breaking condition, the algorithm ran for about a week with some specialized parallelism to produce results of all finite lattices up to 18 points. More specialized lattices can be counted to a larger size and faster by exploiting properties of their symmetry groups using Nauty. This has been done by Jipsen [50] and again quite recently with a more powerful symmetry breaking algorithms by Gebhardt [37]. The current record is 21 points.

Some more specific types of lattices, such as modular, semi-modular, geometric [50], vertically-decomposable [9], etc. have been counted with more efficient methods, using the the extra structure [58]. More recently, Kohonen [57] was able to generate many of these lattices and provide some corrections to previous results.

Known results:

## 5.3. Experiments and Results

To find all Eulerian and Geometric lattices of a given size, I ran the SAT model with two different set of Symmetry breaking constraints. Unfortunately, the SGS constraints were all redundant, except for the case of Geometric lattices of size 17, for which the constraints managed to prune some isomorphic copies. On the other hand, the Codish constraints seem to have worked quite well at removing a very large chunk of isomorphic solutions [compare the ratio]. The rest of the process is feeding the graphs to GAP, where the program Nauty[61] is used to remove the other isomorphic copies.

The algorithm was run on ... I was able to confirm results about Geometric lattices up to $n = 25$ and construct new results about Eulerian Lattices for $n < 20$, seeTable 1. The experiment ran for about a week and a half. The Figure 2 shows the growth of SAT clases and variavles used in the solving process. In Figure 3 we see how this affects the solving times of the models. Note that the Savile Row pass and Solver pass of the solving process are show separated, on a logarithmic scale.

Similarly for Eulerian lattices. Since results about them have not yet been known, solutions were tested using SAGE package Finite Lattices.

Results seem promising and it is likely that we can get further results if the algorithms were given slightly more time to run. The problem I ran into was that I had to run a lot of experiments at the same time and I ran out of space to store the solution files. Both the Eulerian and Geometric lattice experiment got stuck at the Nauty pruning phase. This is not the fault of Nauty, the program has been used to remove isomorphic copies of much larger pool of much larger graphs.

## 5.4. Discussion and Summary

Overall the two programs show one of the biggest advantages and disadvantage of SAT solving. One I was able to translate a correct model for lattices, it was generally not too difficult to generate lattices with more constraints. In fact, because more

| | Geometric | Graded | Lattices |
|---|---|---|---|
| **2** | 1 | 1 | 1 |
| **3** | 0 | 1 | 1 |
| **4** | 1 | 2 | 1 |
| **5** | 1 | 4 | 2 |
| **6** | 1 | 9 | 5 |
| **7** | 1 | 22 | 15 |
| **8** | 2 | 60 | 53 |
| **9** | 1 | 176 | 222 |
| **10** | 2 | 565 | 1078 |
| **11** | 1 | 1980 | 5994 |
| **12** | 3 | 7528 | 37622 |
| **13** | 2 | 30843 | 262776 |
| **14** | 2 | 135248 | 2018305 |
| **15** | 3 | 630004 | 16873364 |
| **16** | 5 | 3097780 | 152233518 |
| **17** | 3 | 15991395 | 1471613387 |
| **18** | 4 | 86267557 | 15150569446 |
| **19** | 5 | 484446620 | 165269824761 |
| **20** | 6 | 2822677523 | 1901910625578 |
| **21** | 6 | 17017165987 | 23003059864006 |
| **22** | 8 | N/A | N/A |
| **23** | 9 | N/A | N/A |
| **24** | 16 | N/A | N/A |
| **25** | 16 | N/A | N/A |
| **26** | 21 | N/A | N/A |
| **27** | 29 | N/A | N/A |
| **28** | 45 | N/A | N/A |
| **29** | 50 | N/A | N/A |
| **30** | 95 | N/A | N/A |
| **31** | 136 | N/A | N/A |
| **32** | 220 | N/A | N/A |
| **33** | 392 | N/A | N/A |

Figure 1. Table of know results for Geometric, Modular and all lattices

constraints produce fewer results, the algorithm can actually generate lattices with more points then in the general case. This suggests that the more difficult models are not necessarily more difficult for the solver.

xxx

TABLE 1. Eulerian lattice

| **Points** | **Solutions** |
|---|---|
| 2 | 1 |
| 3 | 0 |
| 4 | 1 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 1 |
| 9 | 0 |
| 10 | 1 |
| 11 | 0 |
| 12 | 1 |
| 13 | 0 |
| 14 | 2 |
| 15 | 0 |
| 16 | 3 |
| 17 | 0 |
| 18 | 3 |
| 19 | 0 |

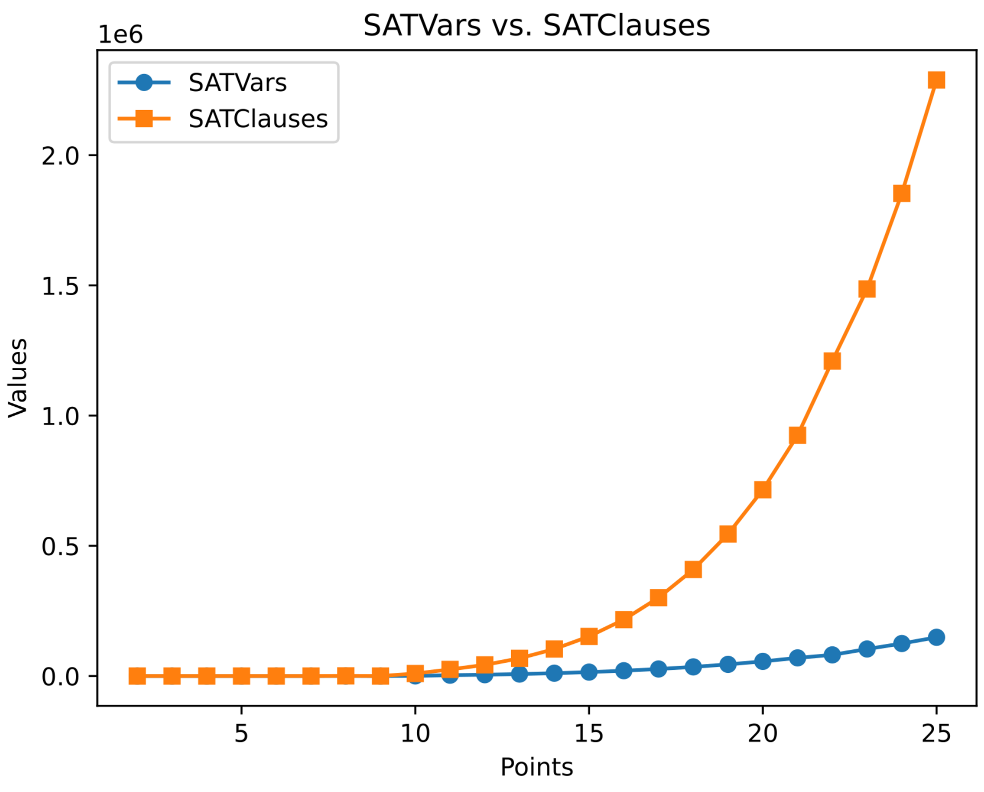

FIGURE 2. Geometric Lattice Model variables

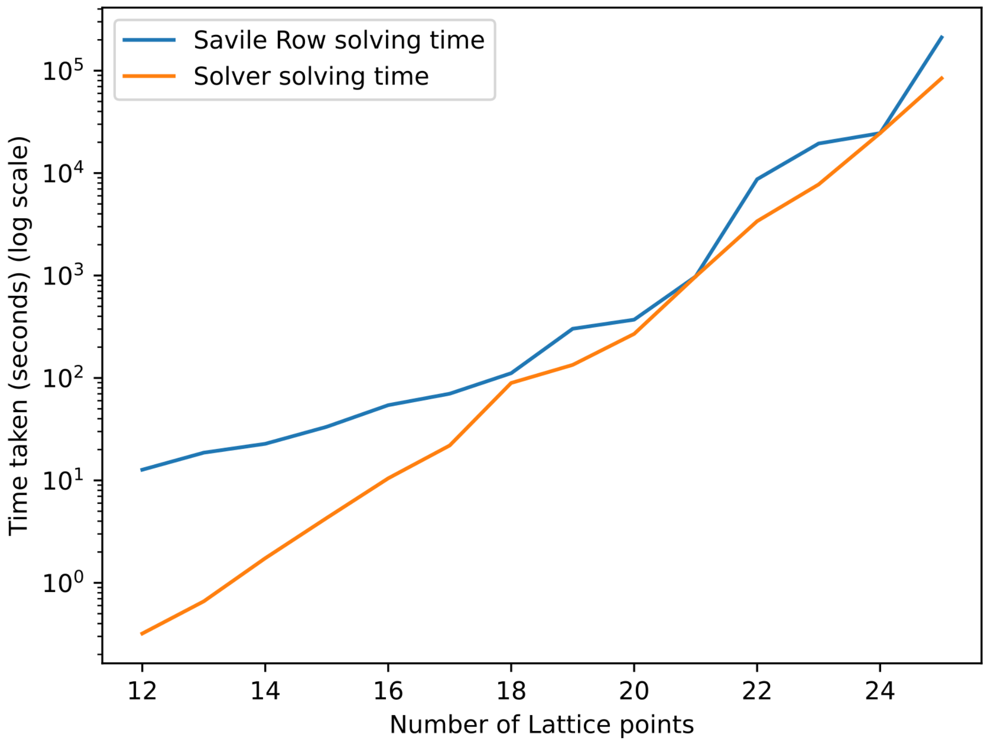

FIGURE 3. SAT solving times for Geometric Lattices

In fact, while doing research on lattices, I found that finite lattices can be represented with only one binary operation instead of two[73], which significantly simplified the SAT model for an arbitrary lattice. On the other hand this seemed to make the solving process significantly slower, which is why I chose to exclude it until now.

On the other hand, both models stopped solving not because the solver could not continue, but because I ran out of memory trying to store all solution files. But even if I did not have that problem, generating half a million solutions, even if solvers have been used to generate many more is still less then ideal. The number of lattices is low compared to the size of the lattice $n$, so effective symmetry breaking is key to this problem. Unfortunately, there is only so much that can be done without creating a custom solver, at least purely in SAT. More complicated static symmetry breaking is not always effective.

Symmetry breaking can also be done dynamically, in the solution process. This requires working directly with the solver and is quite effective [19]. The problem with this approach is that it does not work well in general, therefore custom implementations need to be created for different problems, even if the approach is similar. Compare the approaches in these two papers [53], [52]. Because of this, there has been an attempt to combine SAT solvers and Algebra Systems into a single tool, see [14], which makes this process easier.

CHAPTER 6

# Hypercubes

In the previous chapters, we looked at SAT models that generate lattices and used their ordering relations to break symmetry in the process, demonstrating how SAT solvers are a good tool to enumerate these structures. In this section, we will instead use lattices to model a problem from biology, thus illustrating how lattice theory can be applied in fields beyond mathematics and computer science.

## 6.1. Hypercubes, Boolean Lattices and their connection to biology

A hypercube is an extension of the definition of a cube in the same way a cube is an extension of the definition of a square.1

Definition 6.1. *Let $Q_0$ be the graph with one point and no edges. Let $Q_{n+1}$ be the graph, with vertices the disjoint union of two copies of $Q_n$, that is $V[Q_n] \sqcup V[Q_n] := \{ (v,1),(v,2) \mid v \in V[Q_n] \}$ and edge set*
$E := \{ ((v,1),(v,2)) \mid$ *for all* $(v,1),(v,2) \in V[Q_{n+1}] \}$.
*The $Q_n$, or any isomorphic graph to $Q_n$ is called a hypercube.*

Theorem 6.1. *Let $Q_n$ be a graph, with vertex set all binary sequene of length $n$, with edges between any two binary sequences differing in only one index. Then $Q_n$ is a hypercube.*

Proof. See [71] □

Definition 6.2. *A directed hypercube $\mathcal{Q}_n$ is a DAG with simplification graph a hyper cube $Q_n$.*

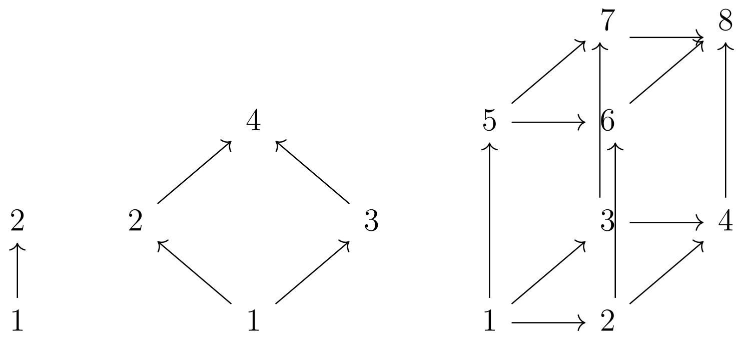

Figure 1. Hypercubes

This recursive definition of a hypercube is inconvenient when generating SAT models for hypercubes. Therefore, consider the following equivalent definitions of a hypercube.

DEFINITION 6.3. *A* Boolean Lattice *$Q$ is a lattice with the following extra conditions:*

*Distributivity For all $a, b, c \in Q$, we have $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$. Similarly, $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.*

*Negation For all $a \in Q$, there exists $-a \in Q$, such that $a \vee -a = 1$ and $a \wedge -a = 0$.*

THEOREM 6.2. *Every finite boolean lattice has a representation graph a hypercube.*

PROOF. See [22] □

THEOREM 6.3. *Let $G$ be a simple graph with $2^n$ vertices and $2^{n-1} * n$ edges, with no odd cycles and every vertex has degree $n$. Then $G$ is a hypercube.*

PROOF. See [71] □

We may use a hypercube to model the geometric theory of basic gene interactions, using the model suggested by[68]. I will make a slight change in the terminology and call the boolean lattices from that paper - Epistasis Fitness Graph or EFG for short.

DEFINITION 6.4. *An Epistasis Fitness Graph(EFG) has the same underlying graph of a hyper cube, except that the edges can be directed in any which way, assuming the resulting graph is a DAG.*

DEFINITION 6.5. *An RSE face of an EFG is a subgraph isomorphic to the graph with edge set* $\{ (0, 1), (0, 2), (1, 3), (2, 3) \}$.

DEFINITION 6.6. *A peak is a vertex with no edges pointing to it, while a valley is a vertex where all neighbouring edges point to it.*

In the paper[71], the authors took this model, first introduce in:[24] and created an algorithm that "glued" smaller lattices to generate bigger ones. The way they achieved this is by using the recursive definition of a hypercube and a bitstring indexing. The definitions introduced here are the same, except that I chose the name Directed Hypercube, instead of the non standard Boolean Lattice.

Given two EFGs with the same number of points, they proceed to generate a bigger EFG by generating all possible ways the edges may be directed. They start with the following set of hypercubes[71]. Once they have all possible orientations of a given length (they were able to generate all possible EFGs up to size $2^4$), they do a search and count the number of RSE faces, peaks and valleys.

This method is not particularly efficient, since isomorphic images are generated repeatedly, which is unnecessary since the number of all solutions can easily be calculated by the number of isomorphic classes, by calculating the size of the automorphism groups of the underlying lattice, for which there is a closed form formula[39]. Since the number of RSE faces, peaks and valley is invariant under graph isomorphism, we can do a simple calculation, instead of searching through $2^{28}$ solutions.

Another solution is to use a SAT solver to enumerate the number of lattices with a fixed number of peaks, valleys and RSE faces. That way, we can generate a smaller subset of solutions or conclude that there are no solutions with the given configuration.

## 6.2. SAT Method

In this section, I will explain my attempt to recreate and extend the results introduced in these papers[24][68]. I will introduce methods for enumerating hypercubes with a fixed number of peaks, valleys and RSE faces. Three methods are presented, all of which start by constructing the simplified graph of the directed hypercube and directing the edges.

### 6.2.1. Using the definition of a Directed Hypercube

In this section, I used the alternative definition of a hypercube to generated the non-directed edges and add additional constraints to direct them later. To achieve this, I simply give the solver a function $C$ that keeps track of the edges. Constraints that break even cycles are added by a simple script in python. For example, the constraints to disallow a triangle are as follows:

```
        language ESSENCE 2.4.0

$ No triangles

forAll i:N. (forAll j,k:N. ((C((i,j)) = 1) \/
(C((j,i)) = 1)) /\ ((C((i,k)) = 1) \/ (C((k,i)) = 1)) ->

((C((j,k)) = 0) /\ (C((k,j)) = 0))),
```

Similarly, constraints that break pentagons and other odd cycles are generated similarly. Notice the nesting of `for all` clauses. This is an easy sign that this formulation of the problem is going to be difficult to solve. Indeed, this model breaks even before adding other constraints. Sometimes adding extra constraints makes the model easier for the solver, as the search algorithm can get to the solutions faster but this did not work in this case.

### 6.2.2. Generating the DAG as a boolean lattice

Since the graphs we want to generate are DAGs, we can use a very nice results, called the topological ordering of a DAG.

DEFINITION 6.7. *A topological ordering of the vertices of a DAG, is a labeling where for any edge* $(v, u)$ *in the DAG, we require* $v \leq u$.

THEOREM 6.4. *Any DAG may be topologically ordered.*

PROOF. See [7] $\square$

The general model is as before with slight modifications.

```
        language ESSENCE 2.4.0

given n : int
letting N be domain int(0..n-1)

find M : function (total) (N,N) --> N
find J : function (total) (N,N) --> N
find compliment : function (total) N --> N
find suprema : N
find infima : N
find C : function (total) (N,N) --> int(0..1)
find D : function (total) (N,N) --> int(0..1)
```

The only difference from the previous lattice models is that I include a function that keeps tract of the directedness of the DAG.

```


$Compliment
forAll i: N. J((i,compliment(i))) = suprema,
forAll i: N. M((i,compliment(i))) = infima,

$Direction
forAll i,j:N. ((C((i,j)) = 1) /\ ((J((i,j)) = i) /\ !(i = j)
/\
!(exists k : N. !(i = k) /\
!(j = k) /\ (J((i, k)) = i) /\ (J((k,j)) = k)))) \/
              (!(C((i,j)) = 1) /\ !((J((i,j)) = i) /\ !(i = j)
              /\
              !(exists k : N. !(i = k) /\ !(j = k) /\
              (J((i, k)) = i) /\ (J((k,j)) = k)))),

forAll i,j:N. ((C((i,j)) = 1) \/ (C((j,i)) = 1)) <->
((D((i,j)) = 1) \/ (D((j,i)) = 1)),
forAll i,j:N. (D((i,j)) = 1) ->  (D((j,i)) = 0),
forAll i,j:N. (D((i,j)) = 1) -> (i > j),
forAll i:N. D((i,i)) = 0,

$Locked
infima = 0,
```

This model is already very difficult to solve using a SAT solver. For $n > 3$ it will not work, even on university machines. Adding the additional constraints makes the model crash even faster. I will show how to model the entire directed hypercube in the next section.

In this case I used the topological ordering to aid the generating process.

### 6.2.3. Generating the DAG with the underlying graph

Here, I use the definition of a boolean lattice to generate the its cover graph and look for the patterns while the lattice is generating.

As before, I declare a function that keeps track of the directed edges, called C. The size of the graph to be generated is n, the underlying, non-directed graph is given as a parameter called lattice.

```
language ESSENCE 2.4.0

given n : int,
letting N be domain int(0..n-1),
given lattice : function (total) (N, N) --> int(0..1)
find C : function (total) (N,N) --> int(0..1)

find p: N
find f: int(1..n*n*n*n)

such that
```

The graph is then required to be directed. The RSE faces are f, by requiring the sum of all variables that form an RSE face to be a given number, in this model f=1. The peeks are similar, we require the sum of all vertices, that have no edge pointing to them to be some number, in this case p=2.

```
$Directed
forAll i,j: N. (lattice((i,j)) = 1) <-> ((C((i,j)) = 1)
\/ (C((j,i)) = 1)),
forAll i,j: N. (C((i,j)) = 1) -> (C((j,i)) = 0),

$Peaks
(sum a:N. toInt(forAll k:N. (C((a,k)) = 0) ) ) = p /\

$RSE faces

(sum a1, a2, a3, a4 : N.
    toInt( (a2 < a3) /\ (a1 < a4) /\ allDiff([a1,a2,a3,a4])
    /\
((C((a1, a2)) = 1) /\ (C((a1, a3)) = 1) /\
(C((a4, a2)) = 1) /\ (C((a4, a3)) = 1) ) )) = f,

$ Acyclicity
```

```
!( exists a1 , a2 , a3 , a4 : N .
(C((a1 , a2)) = 1) /\ (C((a2 , a3)) = 1) /\
(C((a3 , a4)) = 1) /\ (C((a4 , a1)) = 1) ),

p = 2,
f = 1,
```

### 6.2.4. General conclusions

None of the models were able to produce a solution for patterns with $n > 3$, except for the last approach, which involved generating the Boolean lattice graph and then directing the edges inside the solver. This approach was able to identify the nonexistence of solutions for $n = 3$. It was not able to find all solutions for $n = 3$, nor was it able to enumerate cases which did have solutions.

The problem is that generating constraints that direct the edges is something that uses nested `exists` statements, which have to be translated into Conjuctive Normal Form. Unfortunately, this is quite expensive, as it introduces many additional SAT clauses and in this case, Savile Row is not able to remove redundancies before it loses the available resources. This is not solvable with more capable machines, as even the university machines failed to execute the models.

Another interesting point is that the last method was able to produce limited results, while the others failed. Despite the fact that the solver needs to do more work to generate the edges of the graph, the Savile Role can orient the edges more efficiently using the topological labeling[67].

In my opinion, enumerating all solutions using SAT solving is most likely impossible for $n > 3$. Perhaps finding the existence or non existence of solutions is possible, using a custom tool to generate the SAT clauses and thereby bypassing Savile Row.

A better approach might be to incorporate acyclicity directly to Savile Row instead. This is because acyclicity is a common constraint when working with graphs in SAT solvers. Perhaps introducing a special data structure `graph` that can the user of the SAT solver can ask to be connected, acyclic, etc.

CHAPTER 7

# Permutation Patterns

In this chapter I will introduce different types of permutation patterns and build upon the previous work on their generation using SAT solving[70].

## 7.1. Set-up

DEFINITION 7.1. *We say that a sequence $\sigma = [\sigma(1), \sigma(2)...\sigma(n)]$ **is order isomorphic** to a sequence $\pi = [\pi(1), \pi(2)...\pi(n)]$ if $\sigma(i) \leq \sigma(j) \iff \pi(i) \leq \pi(j)$ for all $1 \leq i \leq n$. We say that a permutation $\sigma$ **contains** a permutation $\pi$ as a pattern, if $\sigma$ contains a subsequence, order isomorphic to $\pi$ and we write $\pi \leq \sigma$. Otherwise, we say that $\sigma$ avoids $\pi$.*

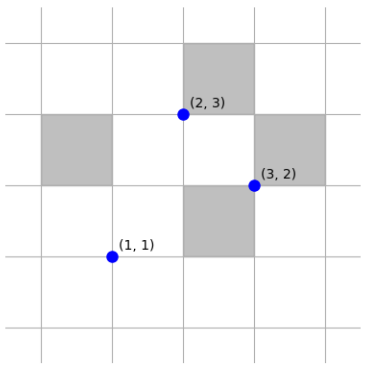

FIGURE 1. Mesh Pattern, $S = \{ (2,1), (3,2), (0,2), (2,3) \}$

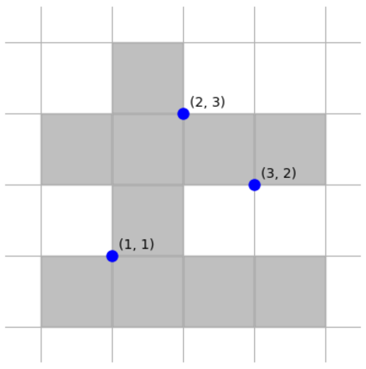

FIGURE 2. Bivincular Pattern, 1 3 2, $S = \{ 1 \} \times \{ 1 \}$

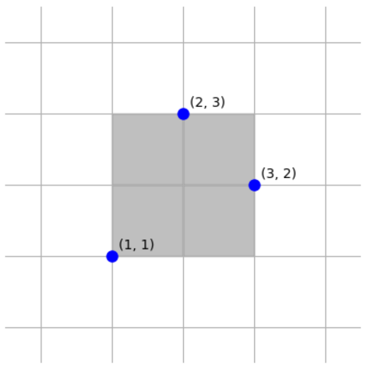

FIGURE 3. Box Pattern, 1 3 2

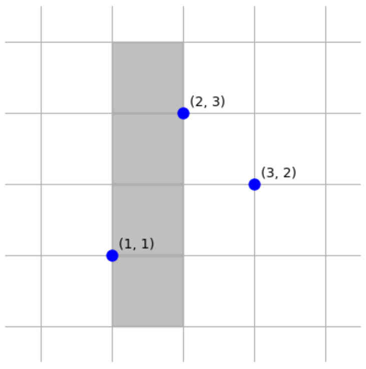

FIGURE 4. Vincular Pattern, 1 3 2, $S = \{1\}$

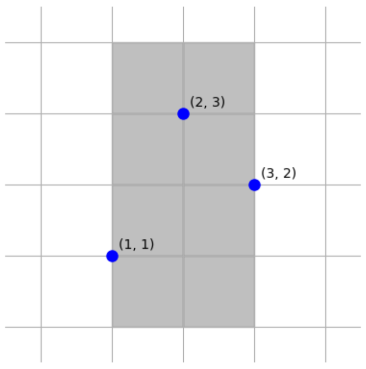

FIGURE 5. Consecutive Pattern, 1 3 2

With this definition, the set $\Pi$ of all bijections over all $n$-sets forms a poset, with **contains** as the ordering relation[62]. Moreover, if we restrict $n$ to be a specific natural number them the group of finite permutation over an $n$-set $S_n$, is a lattice. The minima and suprema are the permutations (1 2 3 ... n) and (n n-1 ... 3 2 1) respectively, the join and meet operators are the least and greatest upper/lower bounds respectively[31]. All of these structures, as well as their congruence lattices[31] have been heavily studied in finite geometry[16], group theory[78] and representation theory of $S_n$ and $A_n$[47].

Computer science, the theory of permutation patterns has been used to develop sorting algorithms using stacks, networks and deques[75]. A basic result is that the permutations(as an input sequence of numbers) that avoid the pattern 2 3 1 are stack sortable[28]. The patterns 2 4 1 3 and 3 1 4 2 also have an interesting, as permutations that avoid them are known to be *pop-stack*-sortable[2], they are also known to be separable[23] and counted by the *Schröder numbers.*

Perhaps it is not too surprising that the number of permutations of a given length $n$ avoiding 1 2 3 are counted by the Catalan numbers $\mathcal{C}_n$[13]. One of the many formulations of $\mathcal{C}_n$ is the number of binary trees with $n+1$ vertices. What is interesting about this is a bijection between binary trees of length $n$, patterns avoiding 1 2 3 and patterns avoiding 2 3 1. This is one of the first enumerative results about permutation patterns.

The following definitions are a tightening of the previous definition, which is also refered to as the **classic** avoidance/containment pattern. As in the classic case, a (vincular/bivincular/consecutive/mesh/box pattern) $\sigma$ which does not contain $\pi$ is said to avoid it.

Definition 7.2. *A **vincular pattern** $(\pi, A)$ consists of a permutation $\pi = \pi(1), \pi(2) \dots \pi(n)$ and a set $A \subseteq \{1, 2 \dots n\}$. We say that a permutation $\sigma = \sigma(1), \sigma(2) \dots \sigma(k)$ contains $(\pi, A)$ if $\sigma$ contains $\pi$ in a subsequence $\sigma(i_1), \sigma(i_2) \dots \sigma(i_n)$ and for all indices $i$ we have that $i_{a+1} = i_a + 1$ for all $a \in A$.*

*In other words, a permutation contains a vincular pattern, if it contains a pattern classically and avoids the columns specified in $A$. For an example see figure:4.*

Definition 7.3. *A **consecutive pattern** $(\pi, A)$ is in particular a vincular pattern, where every entry in $A$ is consecutive.*

Definition 7.4. *A **bivincular pattern** $(\pi, A \times B)$ is is one that contains the vincular pattern $(\pi, A)$ but in addition, for all indices $i \in B$ we have $\sigma(a_{i+1}) = \sigma(i_a) + 1$.*

*This is essentially the same as a vincular pattern, except that we also want the pattern to avoid rows specified in $B$.*

Definition 7.5. *A **mesh pattern** $(\pi, R)$ of length $k$, where $R \in [0, k] \times [0, k]$ is a pattern that classically avoids $\pi$ and in addition, for all $(x, y) \in R$ there does not exist $i \in \{1 \dots n\} \mid i_x < i < i_{x+1} \wedge \sigma(i_{\pi^{-1}(y)}) < \sigma(i) < \sigma(i_{\pi^{-1}(y+1)})$.*

*This is a technical way to say that a mesh pattern avoids $\pi$ classically, but in addition, other points must avoid the boxes specified by the coordinates in $R$.*

Definition 7.6. ***Box Pattern*** *is a mesh pattern that encloses every square of* $\pi$. *That is,* $R = [1, k-1] \times [1, k-1]$.

$$\begin{tikzcd}
 & \text{Mesh} & \\
\text{Bivincular} \arrow[ur] & \text{Consecutive} \arrow[u] & \text{Boxed Mesh} \arrow[ul] \\
\text{Vincular} \arrow[u] \arrow[ur] & & \\
 & \textit{Classic} \arrow[ul] \arrow[uu] \arrow[uur] &
\end{tikzcd}$$

For visual examples of vincular, bivincular, boxed mesh, consecutive and mesh, see figures: 4, 2, 3, 5, 1, respectively.

As I explained before, permutation patterns play an interesting role in enumerative combinatorics. For instance, consider the following: $Av : \Pi \to \mathbb{N} \mid Av(\pi) := |\{\, \sigma \in \Pi \mid \sigma \text{ avoids } \pi \,\}|$,
or $St : \Pi \to \mathbb{N} \mid St(\pi) := |\{\, \sigma \in \Pi \mid \sigma \text{ has not fixed points as a bijection } \}|$ but more generally any function of the form $st : \Pi \to \mathbb{N}$ is called a pattern statistic. The properties of these functions are studied extensively[4]. Vincular and bivincular patterns were first introduced in the study of specific permutation pattern statistics. Latter, Mesh patterns were proposed as a way to generalize known results about Vincular and Bivincular patterns. All of the presented patterns are a subclass of the mesh patterns[54].

### 7.2. Models

To model different permutation patterns, SAT solvers have been used. I will first present the current models introduced here[70] and then give an improved and more general version of them, which I have called mixed patterns.

#### 7.2.1. Mesh

```
language ESSENCE 2.4.0

$ This of mesh patterns to avoid
given mesh_avoidance :
set of (sequence(injective) of int, relation of (int * int))

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

$ this is only used for mesh avoidance/containment
find permPadded : matrix indexed by [int(0..length+1)] of
    int(0..length+1)
such that permPadded[0] = 0, permPadded[length+1] = length+1
such that forAll i : int(1..length) . perm(i) = permPadded[i]
```

Let us go through the model. First we take the mesh avoidance pattern we wish to look for as a parameter. Instead of a pair of a permutation and a set relations, the model uses an injective (therefore in this case also injective) function and a set of relations. The length is also specified as a parameter.

The solver is asked to find a permutation that satisfies a set of clauses which are defined later. A "patted" version of the permutation is also constructed to start at index zero and end at index $length + 1$, this is done purely for convenience so that later constraints are easier to state.

```
such that
$ pattern is the pattern, mesh is the mesh as a relation
forAll (pattern, mesh) in mesh_avoidance .
exists patterninv:
matrix indexed by
    [int(0..|pattern|+1)] of int(0..|pattern|+1),
            patterninv[0] =
                0 /\ patterninv[|pattern|+1] = |pattern|+1 /\
            (forAll i: int(1..|pattern|) .
            patterninv[pattern(i)] = i).

    $ Look for all places where the pattern can occur
    forAll ix :
        matrix indexed by
            [int(0..|pattern|+1)] of int(0..length+1),
        and([ ix[0]=0
            , ix[|pattern|+1]=length+1
            , forAll i : int(0..|pattern|) . ix[i] < ix[i+1]
            , forAll n1, n2 : int(1..|pattern|) , n1 < n2 .
                pattern(n1) < pattern(n2) <->
                permPadded[ix[n1]] < permPadded[ix[n2]]
            ]) .
```

The following looks complicated, but the idea is fairly simple. In order to look for appropriate patterns, it is natural to ask for a sequence that is order isomorphic to the parameter sequence and then check each mesh square to see if the mesh constraints are broken or not. This is done by "simulating" a for-loop, which we model as a sequence (in this case a matrix) *ix* that keeps track of the *i* coordinate.

In this model we want to avoid the mesh, so we require the existence of at least one relation $(i, j)$ for which the pattern is broken. The way this is implemented is slightly confusing, but essentially, we need to find the corresponding $i$ coordinate of the $j$ coordinate in the permutation we were given. The way this is done is by constructing the inverse of that permutation. Another way to say it is that we wish to find the index in the permutation *permPatted* with value $j$.

```
(
    $ If we find the pattern, make sure there is at least one
      value in some cell of the mesh
      exists (i,j) in mesh.
          exists z: int(ix[i]+1..ix[i+1]-1).
          (permPadded[ix[patterninv[j]]] <= permPadded[z] /\
          permPadded[z] <= permPadded[ix[patterninv[j+1]]])
)
```

All other models follow a similar structure, with only slight implementational details that are not too important. The difference between avoidance and containment models is almost irrelevant from a modeling perspective as we just negate the containment clause. The important difference comes from the last constraint. For vincular, bivincular and consecutive patterns, we have a much simpler constraint. I will present the constraint for vincular patterns to illustrate.

Consider a vincular pattern $(\pi, A)$. The appropriate avoidance constraint would be: $\bigwedge_{\forall ix \in [1..|pattern|]} \bigwedge_{\forall a \in A} ix[a] + 1 = ix[a+1]$. The other patterns are modeled similarly. The difference between the avoidance constraints and containment is just a negation and nothing else.

### 7.2.2. Mixed

There are a few things that can be done to improve the solving process. The first consideration is that Savile Row seems to do a low of work in this particular model. This is good because the actual solving process is often less then a second.

On the other hand Savile Row itself is not resource efficient, on my personal machine with 16GB of ram, either my computer would crash Savile Row, or the solver would exit due to lack of memory. Even on the university machine where memory is abundant, this might make it impossible to compute larger patterns, as the resources required to execute the program increase exponentially.

One optimization is to compute the inverse of the parameter permutation in another process like a python script and then supply it to the solver as a parameter. Similarly, we can compute the padded form of the pattern and also add that as a parameter. Both of these computations are done inside Savile Row and never make it to the final solution process. Moreover, Savile Row itself was not designed to do things like this. In testing on my local machine, this provided a significant (up to triple reduction of solution time for small patterns of length 4 or less). On the university machines, the difference was not as dramatic but still consistently faster and more efficient. To keep things simple however, I have chosen to not include this change in later tests.

Another way we can optimize the models is to "quotient out" vincular, bivincular or box pieces out of a general mesh pattern. This can be done efficiently before the solving process with a simple parcing algorithm. Indeed, from the definition of the vincular and bivincular patterns, all we need to do is count columns and rows that are filled with mesh squares, then save the new parameter file with the vincular or bivincular part extracted and the other pieces left as the mesh part of the mixed model. We can do the same for the consecutive and box patterns.

The mixed model is essentially the same as the mesh model, except that vincular and bivincular constraints may be evaluated in addition to mesh constraints. This does not mean that the models are more expressive, indeed anyone of the models presented here can be constructed with suitable mesh pieces. However combining constraints seems to be more efficient for Savile Row to solve. Bellow is a version that uses mesh and vincular constraints. Note that we may use all constraints at the same time and create a *mega model* that can do all of them. While running testing, I came to the conclusion that this is not as efficient as having the models separated.

```
$ Mesh pieces
(
$ If we find the pattern,
$make sure there is at least one value in some cell of the mesh
exists (i,j) in mesh.
    exists z: int(ix[i]+1..ix[i+1]-1).
    (permPadded[ix[patterninv[j]]] <= permPadded[z]
    /\ permPadded[z] <= permPadded[ix[patterninv[j+1]]])
        ) \/
    $ Bivincular pieces
    !(  (forAll bar in ind_bars . ix[bar] + 1 = ix[bar+1])
    /\
    forAll bar in val_bars .
    permPadded[ix[patterninv[bar]]]+1 =
    permPadded[ix[patterninv[bar+1]]])
            )
```

From looking at the models above, it might seem like the box mesh and mesh patterns are idenitical from the perspecive of the solver. Similarly, the consecutive, vincular and bivincular patterns look quite similar. to each other In fact they are, however, the box mesh model is consistently faster then the more general mesh model, even if both models check the exact same indices. The only difference is that Savile Row will generate the indices for the box model, instead of taking them from the parameter file. The situation is almost identical with the vincular and consecutive models. The consecutive model is consistently faster then the more general vincular model. Perhaps slightly less surprising is that the bivincular model is slightly slower then the less general vincular model in the vincular case. The difference on my machine is no more then a 30 second difference, which does not seem to scale with larger patterns.

## 7.3. Results

To test the mixed model against the mesh model, I ran an experiment with 16 parameter files for avoidance, that ran the mesh and mixed avoidance models. The mesh version was able to evaluate 12 of them, 4 of them crashed inside Savile Row and finished solving in 305.97 seconds. In comparison, the mixed model was able to solve 13 of the models, with 3 crashing in the span of 234.10 seconds. The difference is about 30.77% improvement. The solver used was mini SAT, which took less then a second in both cases. The major difference is the Savile Row solving step.

I ran a similar experiment to test the containment model, but this time removing parameter files which did not evaluate in Savile Row. This time the mixed model was faster by 33.67%.

As can be seen using the mixed models whenever possible is a big improvement from using the mesh model alone, especially on a personal computer.

Not only was solving faster, in some situations, it made the computation possible. This makes using the SAT solver viable for larger patterns by researchers who may not have access to specialty hardware, or simply may wish to generate examples on their local machine.

Even when running the models on university machines, the speed improvement is substantial enough to warrant the more complex model.

## 7.4. Discussion

Mixing the mesh and bivincular models suggests that factoring out SAT models into simpler components can increase performance. Other "subpatterns" of the mesh pattern, can be used for further improvements, even if they themselves are of no value as models by themselves to researchers in the permutation pattern community.

Other permutation patterns might also be a good candidate for a SAT model consider the simple marked mesh models presented by Kitaev and Remmel[54].

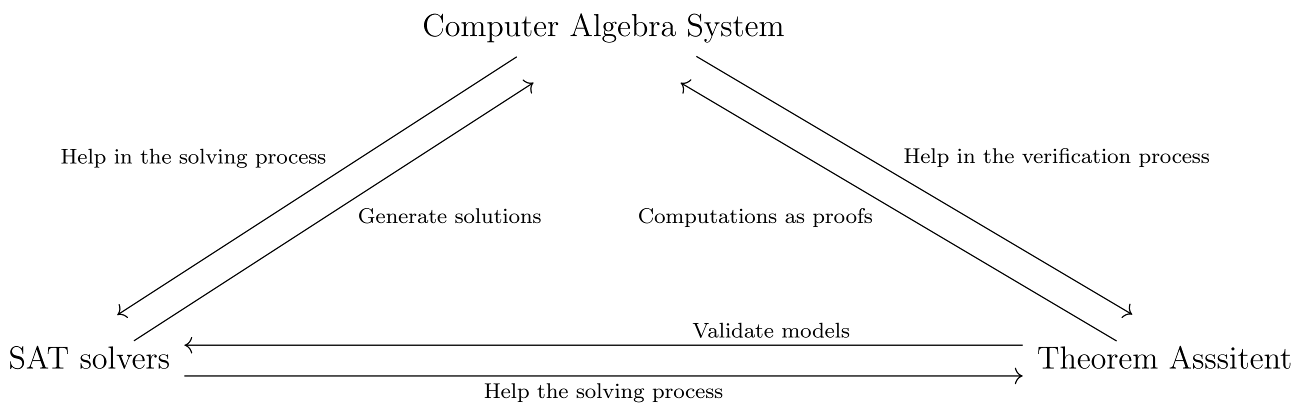

Figure 6. Connection betweem CAS, SAT and formal theorem assistants

While working on these models, I came across a slight mistake in the indexing of the box mesh model. While the mistake was simple, the model is complicated enough where it is quite possible for "off by one index" errors to occur, even if the model itself does a lot of tricks to simplify this very concern. Recently, a paper by[**20**] combined SAT solving and Formal Theorem Assistants to help check the models for their validity. Similarly, SAT solvers might be a helpful computational tools for formal verification itself. For visual summary, see: 6

CHAPTER 8

# Conclusion

This thesis delved into the fascinating realm of ordering relations, posets, and lattices, exploring their applications in various mathematical problems. Through the use of SAT solving techniques, I made significant progress in understanding and solving enumeration problems in Eulerian and Geometric lattices.

The findings have not only confirmed known results for Geometric lattices (for $n < 25$) but have also produced new results for Eulerian lattices with $n < 20$, demonstrating the effectiveness of SAT solving in this context. I encountered and successfully tackled the challenge of symmetry breaking using ordering relations, which contributed to the robustness of the approach.

One notable problem I encountered was the enumeration of directed hypercubes, where SAT solving proved to be less efficient due to the complexities involved in directing the edges and handling nested `exist` statements. This experience showed the need for further exploration and the development of specialized methods for such intricate problems.

Additionally, the research ventured into the realm of permutation patterns and their enumeration. I achieved substantial improvements in the mesh model for various sample graphs containing vincular, bivuncular, consecutive, and box subpatterns. In certain instances, the enhancements even made it possible to generate certain patterns on regular computing systems, where previous methods failed. The resource improvements make generating patterns easier for researchers not familiar with SAT solving.

Throughout this project, I gained a deeper appreciation for the synergy between SAT solvers and computer algebra systems. I envision promising opportunities for the development of tools that combine these technologies, potentially incorporating formal theorem assistance to further advance the field of mathematical theorem-solving.

In essence, this thesis has demonstrated the power of interdisciplinary approaches in mathematics and computer science, and it encourages continued exploration of innovative techniques to tackle challenging problems within the realm of combinatorics and beyond. I hope the insights and methodologies presented in this work will inspire future researchers to push the boundaries of what can be achieved in the fields of SAT solving and computer algebra.

# Bibliography

[1] Özgür Akgün, Alan M Frisch, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale, *Conjure: Automatic generation of constraint models from problem specifications*, Artificial Intelligence **310** (2022), 103751.

[2] Andrei Asinowski, Cyril Banderier, and Benjamin Hackl, *Flip-sort and combinatorial aspects of pop-stack sorting*, Discrete Mathematics & Theoretical Computer Science **22** (2021), no. Special issues.

[3] A Tepavčević B Seselja, *Collection of finite lattices generated by a poset*, Order **17** (2000), 129–139.

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

[5] Fahiem Bacchus and Jonathan Winter, *Effective preprocessing with hyper-resolution and equality reduction*, International conference on theory and applications of satisfiability testing, 2003, pp. 341–355.

[6] Clemens Ballarin, *Computer algebra and theorem proving*, Citeseer, 1999.

[7] Jørgen Bang-Jensen and Gregory Gutin, *Classes of directed graphs*, Vol. 11, Springer, 2018.

[8] Lowell W Beineke and Robin J Wilson, *Topics in algebraic graph theory*, Vol. 102, Cambridge University Press, 2004.

[9] Radim Belohlavek and Vilem Vychodil, *Residuated lattices of size less then 12*, Order **27** (2010), 147–161.

[10] Lucian Bentea and Peter Csaba Ölveczky, *A probabilistic strategy language for probabilistic rewrite theories and its application to cloud computing*, International workshop on algebraic development techniques, 2012, pp. 77–94.

[11] Garrett Birkhoff, *Lattice theory*, Vol. 25, American Mathematical Soc., 1940.

[12] Béla Bollobás and Béla Bollobás, *Random graphs*, Springer, 1998.

[13] Miklós Bóna, *Surprising symmetries in objects counted by catalan numbers*, the electronic journal of combinatorics (2012), P62–P62.

[14] Curtis Bright, Ilias Kotsireas, and Vijay Ganesh, *Sat solvers and computer algebra systems: a powerful combination for mathematics*, arXiv preprint arXiv:1907.04408 (2019).

[15] Bruno Buchberger and Rüdiger Loos, *Algebraic simplification*, Computer algebra: symbolic and algebraic computation, 1982, pp. 11–43.

[16] Nathalie Caspard, *The lattice of permutations is bounded*, International Journal of Algebra and Computation **10** (2000), no. 04, 481–489.

[17] C. Cedillo, R. MacKinney-Romero, M. A. Piza na, I. A. Robles, and R. Villarroel-Flores, *YAGS, yet another graph system, Version 0.0.5*, 2019. GAP package.

[18] Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sorensson, *Sat-solving in practice*, 2008 9th international workshop on discrete event systems, 2008, pp. 61–67.

[19] Claessen, Koen and Een, Niklas and Sheeran, Mary and Sorensson, Niklas, *Sat-solving in practice*, 2008 9th international workshop on discrete event systems, 2008, pp. 61–67.

[20] Cayden Codel, Jeremy Avigad, and Marijn Heule, *Verified encodings for sat solvers*, # placeholder_parent_metadata_value#, 2023, pp. 141–151.

[21] Michael Codish, Alice Miller, Patrick Prosser, and Peter J Stuckey, *Constraints for symmetry breaking in graph representation*, Constraints **24** (2019), 1–24.

[22] Paul Moritz Cohn and Paul Moritz Cohn, *Universal algebra*, Vol. 159, Reidel Dordrecht, 1981.

[23] Robert Cori, Enrica Duchi, Veronica Guerrini, and Simone Rinaldi, *Families of parking functions counted by the schröder and baxter numbers*, Lattice Path Combinatorics and Applications (2019), 194–225.

[24] Kristina Crona, Devin Greene, and Miriam Barlow, *The peaks and geometry of fitness landscapes*, Journal of theoretical biology **317** (2013), 1–10.

[25] Gábor Czédli, *The asymptotic number of planar, slim, semimodular lattice diagrams*, Order **33** (2016), 231–237.

[26] Gábor Czédli, Tamás Dékány, Gergő Gyenizse, and Júlia Kulin, *The number of slim rectangular lattices*, Algebra universalis **75** (2016), no. 1, 33–50.

[27] Leonardo De Moura and Nikolaj Bjørner, *Satisfiability modulo theories: An appetizer*, Brazilian symposium on formal methods, 2009, pp. 23–36.

[28] Colin Defant, *Counting 3-stack-sortable permutations*, Journal of Combinatorial Theory, Series A **172** (2020), 105209.

[29] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker, *Improved static symmetry breaking for sat*, Theory and applications of satisfiability testing–sat 2016: 19th international conference, bordeaux, france, july 5-8, 2016, proceedings 19, 2016, pp. 104–122.

[30] Peter Dobsan, *Pynauty*, 2022. GNU General Public License version 3 or any later version.

[31] Vincent Duquenne and Ameziane Cherfouh, *On permutation lattices*, Mathematical Social Sciences **27** (1994), no. 1, 73–89.

[32] Niklas Eén and Niklas Sörensson, *Temporal induction by incremental sat solving*, Electronic Notes in Theoretical Computer Science **89** (2003), no. 4, 543–560.

[33] Martin J Erickson, *Introduction to combinatorics*, John Wiley & Sons, 2013.

[34] Marcel Erné, Jobst Heitzig, and Jürgen Reinhold, *On the number of distributive lattices*, the electronic journal of combinatorics (2002), R24–R24.

[35] *GAP – Groups, Algorithms, and Programming, Version 4.12.2*, The GAP Group, 2022.

[36] Vijay K Garg, *Introduction to lattice theory with computer science applications*, John Wiley & Sons, 2015.

[37] Volker Gebhardt and Stephen Tawn, *Constructing unlabelled lattices*, Journal of Algebra **545** (2020), 213–236.

[38] Weiwei Gong and Xu Zhou, *A survey of sat solver*, Aip conference proceedings, 2017.

[39] George Grätzer, *Universal algebra*, Springer Science & Business Media, 2008.

[40] George A Gratzer and Friedrich Wehrung, *Lattice theory: special topics and applications*, Springer, 2016.

[41] Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano, *Formalization of universal algebra in agda*, Electronic Notes in Theoretical Computer Science **338** (2018), 147–166.

[42] Paul Richard Halmos, *Naive set theory*, van Nostrand, 1960.

[43] Jobst Heitzig and Jürgen Reinhold, *The number of unlabeled orders on fourteen elements*, Order **17** (2000), 333–341.

[44] Jobst Heitzig and Jurgen Reinhold, *Counting finite lattices*, Algebra universalis **48** (2002), no. 1, 43–53.

[45] Marijn JH Heule and Oliver Kullmann, *The science of brute force*, Communications of the ACM **60** (2017), no. 8, 70–79.

[46] Alexander Hulpke, *Notes on computational group theory*, Department of Mathematics. Colorado State University (2010).

[47] Gordon Douglas James, *The representation theory of the symmetric groups*, Vol. 682, Springer, 2006.

[48] Padraigh Jarvis and Alejandro Arbelaez, *Cooperative parallel sat local search with path relinking*, Evolutionary computation in combinatorial optimization: 20th european conference, evocop 2020, held as part of evostar 2020, seville, spain, april 15–17, 2020, proceedings 20, 2020, pp. 83–98.

[49] Chris Jefferson, Tom Kelsey, Steve Linton, and Karen Petrie, *Gaplex: Generalised static symmetry breaking*, Symmetry and Constraint Satisfaction Problems **17** (2006).

[50] Peter Jipsen and Nathan Lawless, *Generating all finite modular lattices of a given size*, Algebra universalis **74** (2015), 253–264.

[51] Markus Kirchweger, Tomáš Peitl, and Stefan Szeider, *Co-certificate learning with sat modulo symmetries*, arXiv preprint arXiv:2306.10427 (2023).

[52] Markus Kirchweger, Manfred Scheucher, and Stefan Szeider, *A sat attack on rota's basis conjecture*, 25th international conference on theory and applications of satisfiability testing (sat 2022), 2022.

[53] Markus Kirchweger and Stefan Szeider, *Sat modulo symmetries for graph generation*, 27th international conference on principles and practice of constraint programming (cp 2021), 2021.

[54] Sergey Kitaev and Jeffrey Remmel, *Quadrant marked mesh patterns*, J. Integer Sequences **12** (2012), no. 4.

[55] DJ Kleitman, KJ Winston, and J Srivastava, *The asymptotic number of lattices*, Combinatorical Mathematics, Optimal Designs and their Applications (J. Srivastava, ed.), Ann. Discrete Math **6** (1980), 243–249.

[56] Y Koda, *The numbers of finite lattices and finite topologies*, Bull. Inst. Combin. Appl **10** (1994), 83–89.

[57] Jukka Kohonen, *Generating modular lattices of up to 30 elements*, Order **36** (2019), no. 3, 423–435.

[58] Nathan Lawless, *Generating all modular lattices of a given size* (2013).

[59] Inês Lynce and Joël Ouaknine, *Sudoku as a sat problem.*, Ai&m, 2006.

[60] Ruben Martins, Vasco Manquinho, and Inês Lynce, *An overview of parallel sat solving*, Constraints **17** (2012), 304–347.

[61] Brendan D. McKay and Adolfo Piperno, *Practical graph isomorphism, ii*, Journal of Symbolic Computation **60** (2014), 94–112.

[62] Peter RW McNamara and Einar Steingrímsson, *On the topology of the permutation pattern poset*, Journal of Combinatorial Theory, Series A **134** (2015), 1–35.

[63] Thai Nguyen Hung, *Building a puzzle games solver using sat solver* (2017).

[64] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli, *Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t)*, Journal of the ACM (JACM) **53** (2006), no. 6, 937–977.

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

[66] C Michael Pilato, Ben Collins-Sussman, and Brian W Fitzpatrick, *Version control with subversion: next generation open source version control*, " O'Reilly Media, Inc.", 2008.

[67] Md Saidur Rahman et al., *Basic graph theory*, Vol. 9, Springer, 2017.

[68] Manda Riehl, Reed Phillips, Lara Pudwell, and Nate Chenette, *Occurrences of reciprocal sign epistasis in single-and multi-peaked theoretical fitness landscapes*, Journal of Physics A: Mathematical and Theoretical **55** (2022), no. 43, 434002.

[69] Albert Oliveras Enric Rodrıguez-Carbonell, *From dpll to cdcl sat solvers* (2019).

[70] Christopher Jepherson Ruth Hoffman Ozgun Akgun, *$Cp_2023_permutation_patterns$*, GitHub, 2023.

[71] Yousef Saad and Martin H Schultz, *Topological properties of hypercubes*, IEEE Transactions on computers **37** (1988), no. 7, 867–872.

[72] Branimir Šešelja and Andreja Tepavčević, *On generation of finite posets by meet-irreducibles*, Discrete Mathematics **186** (1998), no. 1-3, 269–275.

[73] Benjamin Steinberg et al., *Representation theory of finite monoids*, Springer, 2016.

[74] Robert Roth Stoll, *Set theory and logic*, Courier Corporation, 1979.

[75] James Andrew Storer, *An introduction to data structures and algorithms*, Springer Science & Business Media, 2001.

[76] Seselja Tepavcevic and Branimir Andreja, *Congruences on lattices and lattice-valued functions*, Computational intelligence and mathematics for tackling complex problems 2, 2022, pp. 219–228.

[77] The Sage Developers, *Sagemath, the Sage Mathematics Software System (Version x.y.z)*, 2023. `https://www.sagemath.org`.

[78] Anatolii Moiseevich Vershik and Aleksandr Nikolaevich Sergeev, *A new approach to the representation theory of the symmetric groups. iv. z2-graded groups and algebras: projective representations of the group sn*, Mosc. Math. J **8** (2008), no. 4, 813–842.

[79] Alexander A Voronov, *Notes on universal algebra*, Graphs and patterns in mathematics and theoretical physics **73** (2005), 81–103.

[80] Michael J Wester, *Computer algebra systems: a practical guide*, John Wiley & Sons, Inc., 1999.

[81] Yang Xie and Ankur Srivastava, *Anti-sat: Mitigating sat attack on logic locking*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38** (2018), no. 2, 199–207.

[82] Viktor N Zemlyachenko, Nickolay M Korneenko, and Regina I Tyshkevich, *Graph isomorphism problem*, Journal of Soviet Mathematics **29** (1985), 1426–1481.

[83] Weifeng Zhou, Qingguo Li, and Lankun Guo, *Prime, irreducible elements and coatoms in posets*, Mathematica Slovaca **63** (2013), no. 6, 1163–1178.

[84] Paul Zimmermann, Alexandre Casamayou, Nathann Cohen, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clement Pernet, et al., *Computational mathematics with sagemath*, SIAM, 2018.

[85] Mališa Žižović and Vera Lazarević, *Construction of codes by lattice valued fuzzy sets.*, Novi Sad Journal of Mathematics **35** (2005), no. 2, 155–160.