Solving Combinatorial Problems Using Satisfiability

By

Aaron Barnoff

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science in Computer Science
at the University of Windsor

Windsor, Ontario, Canada

2026

Solving Combinatorial Problems Using Satisfiability

By

Aaron Barnoff

Date of final oral defence: July 31, 2026

The document has received final approval by the Master's Committee:

Co-Supervisor: Curtis Bright

Co-Supervisor: Ahmad Biniaz

Program Reader: Yung H. Tsin

Outside Program Reader: Adrian Zahariuc

DECLARATION OF CO-AUTHORSHIP/PREVIOUS PUBLICATION

I. Co-Authorship

This thesis incorporates material that is the result of joint research, as follows:

In Chapters 3 and 4 of this thesis, the findings are the result of collaborative work with Curtis Bright as co-author. The original ideas and general methodologies for both chapters were proposed by Curtis Bright, and the research was carried out collaboratively with his guidance and assistance.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-authors to include the above materials in my thesis. I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

II. Previous Publication

This thesis includes two original papers that have been previously published or submitted for publication, as follows:

- **Chapter 3:** Barnoff, A., and Bright, C. (2026). North–east lattice paths avoiding k collinear points via satisfiability. *Advances in Applied Mathematics*, 179, Article 103112. doi:10.1016/j.aam.2026.103112.

- **Chapter 4:** Barnoff, A., and Bright, C. (2026). Improving SAT solvers on orthogonal Latin square problems. To appear in the *Proceedings of the 2026 International Workshop on Satisfiability Checking and Symbolic Computation*, Oldenburg, Germany.

I certify that I have obtained a written permission from the copyright owners to include the above published materials in my thesis. I certify that the above material describes

work completed during my registration as a graduate student at the University of Windsor.

III. General

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix. I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

Many combinatorial search problems are difficult because the number of possible objects grows too quickly for naive exhaustive search. Boolean satisfiability (SAT) provides a general framework for such problems: one encodes the desired objects as satisfying assignments of a propositional formula, and then uses a SAT solver to search for solutions or prove that none exist.

This thesis studies SAT-based methods for two combinatorial search problems. The first is the Gerver–Ramsey collinearity problem for north–east lattice walks, where the goal is to determine the maximum length of a walk avoiding k collinear points. We develop SAT encodings for this problem using path constraints, non-collinearity constraints, reachability information, and parallelization. Using these methods, we enumerate all north–east lattice walks avoiding k collinear points for $k \leq 6$ and construct a north–east walk with 327 steps avoiding $k = 7$ collinear points.

The second problem concerns the construction of mutually orthogonal Latin squares. We develop a hybrid method that combines SAT solving with the Euler–Parker algorithm for finding orthogonal mates. In this approach, the SAT solver searches for Latin squares satisfying structural constraints, while an external exact-cover solver is used to test completed squares for orthogonal mates. This hybrid method substantially improves performance over a pure SAT approach on the Latin square instances considered.

ACKNOWLEDGMENTS

This work would not have been possible without the support and encouragement of many people. I am deeply grateful to those who have helped me throughout my studies, whether through mentorship, teaching, collaboration, advice, or personal support. I would especially like to thank the following people.

I would like to thank my supervisor, Dr. Curtis Bright, for his tireless mentorship, patience, and kindness throughout both my undergraduate and graduate studies. The amount of time and care he has devoted to my academic development cannot be overstated. His genuine passion for research and teaching, along with his constant encouragement, has allowed me to explore areas of research I never thought I would be able to study. Beyond my research, he has helped shape my academic path, encouraging me to pursue opportunities I would not have considered on my own and helping me make the most of them. I am especially grateful for the immense effort he has put into teaching me to approach research with clarity and precision. His incredible attention to detail and insightful feedback have shaped the way I think about problems and communicate my work. I can safely say that without him, I would not be where I am today.

I would like to thank Dr. Yung H. Tsin for instilling in me a deep appreciation for mathematics as something to be understood through proof and rigour, rather than learned by rote memorization. From the beginning, he impressed upon me that this foundation is essential for serious academic work and for a true understanding of computer science. Through his teaching and our conversations, he continually went out of his way to help me grow as a student and develop a more rigorous approach to mathematics. The confidence and skills I gained from him were essential in preparing me for my research, and his support has been invaluable in helping me continue along this academic path. I will always be grateful for his considerable effort and care.

I would like to thank Dr. Ahmad Biniaz for kindly agreeing to serve as my co-supervisor, allowing me to complete my master's studies under Dr. Bright's continued guidance. I am also grateful for what I learned from him in his course on computational

geometry, which he made interesting and rewarding to study.

I would like to thank my committee members, Dr. Curtis Bright, Dr. Ahmad Biniaz, Dr. Yung H. Tsin, and Dr. Adrian Zahariuc, for their time, helpful feedback, and support throughout the thesis process.

Lastly, I would like to thank my family for being a constant source of support throughout my studies. I am grateful to my mother Marion, my father Norman, my brother Harold, my sister-in-law Stephanie, and my nephews Jack and Sam, whose advice, encouragement, and patience over the years have meant a great deal to me. I am very fortunate to have them in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CAS	Computer algebra system.
CDCL	Conflict-driven clause learning.
CNF	Conjunctive normal form.
DIMACS	A standard plain-text format for representing CNF SAT instances, originally developed by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS).
DRAT	Deletion resolution asymmetric tautology, a proof format for unsatisfiability certificates.
GR(k)	A Gerver–Ramsey walk avoiding k collinear points.
IPASIR	Reentrant incremental SAT solver API (reverse acronym).
IPASIR-UP	IPASIR user propagators, a programmatic SAT interface used to add external propagation or clauses during solving.
k-MOLS(n)	A set of k mutually orthogonal Latin squares of order n.
KNF	At-least-k conjunctive normal form.
MOLS	Mutually orthogonal Latin squares.
SAT	The Boolean satisfiability problem, or a problem instance that is satisfiable.
SAT+CAS	SAT solving combined with computer algebra systems or symbolic computation.
TRP	Transversal representation pair.
UNSAT	A problem instance that is unsatisfiable.

CHAPTER 1

Introduction

Many problems in combinatorics can be stated in simple terms but have search spaces that are too large for direct enumeration. A typical problem asks whether there exists a finite object satisfying a collection of constraints. For small parameters, such a problem may be solvable by hand or by a straightforward computer search. As the parameters grow, however, the number of possible objects often grows exponentially, and naive exhaustive search quickly becomes infeasible.

One example is the Boolean Pythagorean triples problem, which asks whether the positive integers can be coloured red and blue so that no Pythagorean triple is monochromatic. In other words, one seeks a two-colouring of the positive integers such that there are no integers a, b, c of the same colour satisfying $a^2 + b^2 = c^2$. The problem goes back to at least the 1980s when Ronald Graham offered a prize for its solution [37]. Even for a finite interval $\{1, \ldots, n\}$, a naive search has 2^n possible colourings. The problem remained unsolved until 2016, when Heule et al. used a satisfiability solver to show that such a colouring exists for $\{1, \ldots, 7824\}$ but not for $\{1, \ldots, 7825\}$ [31].

Boolean satisfiability solving provides a general computational framework for finite-domain constraint problems of this kind. The Boolean satisfiability problem, or SAT, asks whether a propositional formula has an assignment of truth values that makes it true. To use SAT for a combinatorial problem, one introduces Boolean variables representing the choices in the problem and then adds logical constraints describing the objects being searched for. A satisfying assignment corresponds to a solution of the original problem, while an unsatisfiable formula proves that no such object

exists. SAT instances are solved using satisfiability checkers, or SAT solvers, which automatically search for satisfying assignments or prove that none exist.

Modern SAT solvers are effective because they combine systematic backtracking search with powerful inference techniques. This allows them to search large combinatorial spaces efficiently in many practical cases, despite the exponential worst-case complexity of SAT. In addition, for unsatisfiable instances, SAT solvers can produce proof certificates that can be checked independently. This is useful in computational mathematics, where a negative result often requires more justification than simply reporting that a search program found no examples.

The two topics studied in this thesis are different in their mathematical content, but they illustrate the same general principle. SAT solvers are strong general-purpose search tools, and they can be applied effectively to difficult combinatorial problems by designing computational methods that take advantage of the mathematical structure of the problem.

1.1 Case Studies in this Thesis

In this thesis, we study the use of SAT solving on two combinatorial problems:

1. The Gerver–Ramsey collinearity problem for north–east lattice walks.

2. Searching for mutually orthogonal Latin squares of a special form.

1.1.1 The Gerver–Ramsey problem

A north–east lattice walk is a path in the integer lattice using only the unit steps $(1, 0)$ and $(0, 1)$. In 1971, Tom C. Brown [17] asked whether every sufficiently long north–east lattice walk must contain k collinear points, for every fixed integer $k \geq 1$. In 1972, P. L. Montgomery [43] showed that the answer is yes, although his proof did not give an explicit bound on how long the walk must be.

Gerver and Ramsey later gave an explicit upper bound. They showed that every

north–east lattice walk of length at least

$$(k-1)2^{2^{13}(k-1)^4}$$

must contain k collinear points [29]. Around the same time, Gerver gave a lower bound by constructing walks of length greater than

$$\left(32(k-1)^{2\log_2(k-1)-7}\right)^{1/18}$$

that avoid k collinear points [28]. In a recent preprint, Korsky improved both asymptotic bounds, showing that the maximum length of a north–east lattice walk avoiding k collinear points is between

$$\exp\left(\Omega\left(k^{1/3}\right)\right) \quad \text{and} \quad \exp\left(\left(\frac{2}{e}+o(1)\right)(k-1)^2\right)$$

as k tends to infinity [36]. Although these results substantially improve the previous bounds, a large gap remains between the longest walks known to avoid k collinear points and the length at which k collinear points are guaranteed.

These bounds are extremely loose for small values of k. For example, when $k=3$, the upper bound only guarantees that every north–east walk of length at least 2^{131073} contains three collinear points. The lower bound only guarantees the existence of a walk with more than one step avoiding three collinear points. In actuality, the longest north–east walks avoiding three collinear points have three steps.

In this thesis, we study the problem of determining these path lengths exactly for small values of k, a problem posed by A. Meir in 1979 [23]. Our results on the Gerver–Ramsey collinearity problem are discussed in Chapter 3. The work in Chapter 3 is based on joint work with Curtis Bright which appeared in [7].

1.1.2 Finding mutually orthogonal Latin squares

A *Latin square* of order n is an $n \times n$ array containing n symbols, where each symbol occurs exactly once in each row and exactly once in each column. Latin squares are classical objects in combinatorics and appear in areas such as experimental design, finite geometry, algebra, and recreational mathematics [22].

Two Latin squares of the same order are called *orthogonal* if, when the two squares are overlaid, every ordered pair of symbols occurs exactly once. A set of k pairwise orthogonal Latin squares of order n is called a k-MOLS(n).

The study of orthogonal Latin squares has a long history. They were of interest to Euler, who posed his famous 36 officers problem in 1782 [26]. In this problem, one tries to arrange 36 officers from six regiments and six ranks in a 6×6 grid so that every row and column contains each regiment and each rank exactly once. Equivalently, one Latin square records the regiment of each officer and another records the rank. The requirement that every regiment-rank pair occurs exactly once is precisely the orthogonality condition. Thus, the 36 officers problem is equivalent to asking whether a pair of orthogonal Latin squares of order 6 exists. It turns out that no such arrangement exists.

After constructing pairs of orthogonal Latin squares for orders that are odd or divisible by 4, Euler was unable to construct a pair of order 6. He conjectured that pairs of orthogonal Latin squares do not exist for any order of the form $4t+2$ [26]. This conjecture is trivially true for $n = 2$, and Tarry proved it true for $n = 6$ in 1900 [53]. Several later works attempted to prove Euler's conjecture in general [48, 55, 41]. However, Bose, Shrikhande, and Parker showed in 1959–1960 that the conjecture is false in all remaining cases: pairs of orthogonal Latin squares exist for every order except 2 and 6 [11, 47, 12].

Although this settles the general existence problem for pairs of orthogonal Latin squares, computational search problems involving Latin squares remain difficult. The difficulty is especially pronounced when the desired squares must satisfy additional structural constraints. In such cases, general existence theorems do not necessarily provide the required examples, and direct search through all possible Latin squares is usually infeasible. For example, the existence of a 3-MOLS(10) remains open [15]. This problem has attracted interest since at least the 1950s, and order 10 is the smallest order for which the maximum number of mutually orthogonal Latin squares is not known and the only order n for which it is unknown if a 3-MOLS(n) exists.

In this thesis, we study structured orthogonal Latin square search problems using

a hybrid method that combines SAT solving with an external algorithm exploiting mathematical structure that is not easily visible to a SAT solver from a standard Boolean encoding. This allows the solver to combine the general search capabilities of SAT with the specialized structure of Latin squares. We discuss our results on mutually orthogonal Latin squares in Chapter 4. The work in Chapter 4 is based on joint work with Curtis Bright, and will appear in the proceedings of the 2026 International Workshop on Satisfiability Checking and Symbolic Computation.

CHAPTER 2

Background

This chapter gives the background used in the two main parts of this thesis. We first describe the Boolean satisfiability problem, the input format used by modern SAT solvers, and the main ideas behind SAT solving. We then discuss several SAT-based techniques used later in the thesis, including cardinality constraints, proof certificates, cube-and-conquer parallelization, and programmatic SAT. The chapter concludes with background on exact cover and Myrvold's order-10 MOLS framework, which are used in the second case study.

2.1 Satisfiability Solving

The Boolean satisfiability problem, usually abbreviated SAT, asks whether a Boolean formula has an assignment of truth values to its variables that makes the formula evaluate to true. A Boolean formula is built from propositional variables such as p, q, and r, together with logical operations such as AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). For example, the formula

$$(p \wedge q) \vee \neg r$$

is satisfiable, since assigning p and q to true makes the first term true, while assigning r to false makes the second term true; either choice is enough to make the whole formula evaluate to true.

SAT was the first problem shown to be NP-complete, by the Cook–Levin theorem [20]. This means, in particular, that every problem in NP can be reduced to SAT in polynomial time. In practical terms, a reduction to SAT consists of constructing

a Boolean formula whose satisfying assignments correspond exactly to the objects being searched for in the original problem. If the formula is satisfiable, a satisfying assignment can be decoded into a solution of the original problem. If the formula is unsatisfiable, then no object satisfying the encoded constraints exists.

This makes SAT useful as a general framework for finite combinatorial search. Instead of writing a specialized search program for each new problem, one can express the problem using Boolean variables and constraints, and then use an existing SAT solver to perform the search.

2.1.1 SAT solvers

A SAT solver is a program that takes a Boolean formula as input and determines whether the formula is satisfiable. As mentioned previously, SAT is NP-complete, so no polynomial-time algorithm is known for solving all SAT instances. Nevertheless, modern SAT solvers have proved highly effective on many structured instances arising in practice [13]. In particular, they can often solve instances containing hundreds of thousands of variables and millions of clauses [45]. However, formula size does not always correspond directly to difficulty: the Boolean Pythagorean triples problem required fewer than 8000 variables and 20,000 clauses, yet took approximately four CPU-years of computation [31].

In this thesis, SAT solvers are used in two different ways. In the Gerver–Ramsey problem, the SAT solver directly searches for long north–east lattice walks avoiding collinear points. In the MOLS problem, the SAT solver searches for Latin squares satisfying structural constraints, while a separate exact cover based procedure is used to test completed squares for orthogonal mates.

2.1.2 Conjunctive normal form

Modern SAT solvers typically require their input to be written in conjunctive normal form, or CNF. A literal is either a Boolean variable p or its negation $\neg p$. A clause is a disjunction of literals, and a CNF formula is a conjunction of clauses. For example,

$p \wedge (q \vee \neg r)$ is in CNF. It has two clauses: the unit clause p, and the clause $q \vee \neg r$. We use implication notation as shorthand, with $p \to q$ standing for $\neg p \vee q$. Thus, an implication can be written directly as a clause.

SAT solvers commonly use the DIMACS format to represent CNF formulas. Variables are represented by positive integers, and negated variables are represented by negative integers. Each clause is written as a list of integers ending in 0. For example, the formula $p \wedge (q \vee \neg r)$ can be represented in DIMACS as

```
p cnf 3 2
1 0
2 -3 0
```

where variables p, q, and r are represented by 1, 2, and 3, respectively.

Not every Boolean formula is naturally in CNF. Direct conversion to CNF using distributivity can cause an exponential increase in formula size. For example, the formula

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$$

expands into a CNF formula with 2^n clauses

$$(x_1 \vee x_2 \cdots \vee x_n) \wedge (y_1 \vee x_2 \vee \cdots \vee x_n) \wedge \cdots,$$

because each clause must choose one literal from each pair $\{x_i, y_i\}$.

To avoid this exponential blowup, one usually uses the Tseitin transformation [54]. The Tseitin transformation introduces auxiliary variables representing subformulas. The resulting formula is not logically equivalent to the original formula, because it contains new variables, but it is equisatisfiable: it is satisfiable if and only if the original formula is satisfiable.

For example, consider the formula $\neg(p \wedge q)$. Introduce a new variable x to represent $p \wedge q$, and a new variable y to represent $\neg x$. We then enforce

$$x \leftrightarrow (p \wedge q) \qquad \text{and} \qquad y \leftrightarrow \neg x.$$

The first equivalence can be encoded by the clauses

$$(\neg x \vee p), \qquad (\neg x \vee q), \qquad (x \vee \neg p \vee \neg q),$$

and the second by

$$(\neg y \vee \neg x), \qquad (y \vee x).$$

Finally, since y represents the whole formula, we add the unit clause (y). The resulting CNF formula is satisfiable exactly when the original formula is satisfiable. Since only a constant number of clauses is needed for each gate or subformula, the Tseitin transformation produces a CNF formula whose size is linear in the size of the original formula.

2.1.3 The DPLL algorithm

The basic search procedure underlying SAT solving is the Davis–Putnam–Logemann–Loveland algorithm, usually called DPLL. DPLL is a backtracking algorithm that repeatedly applies Boolean constraint propagation and branches on variables when propagation alone is not enough [21].

Boolean constraint propagation can be viewed as repeated applications of unit resolution. Suppose a formula contains a unit clause $\{u\}$. Then any satisfying assignment must set the literal u to true. Once this is done, every clause containing u is already satisfied and can be ignored. Every occurrence of $\neg u$ in the remaining clauses can be removed, since $\neg u$ is false. If this process produces an empty clause, then a conflict has been found, because the current partial assignment cannot be extended to a satisfying assignment.

If unit propagation does not determine the satisfiability of the formula, DPLL chooses an unassigned variable and branches on it. One recursive branch sets the variable to true, and the other sets it to false. The algorithm continues until it either finds a satisfying assignment or proves that all branches lead to conflicts.

Although DPLL is an older algorithm, its core operation, Boolean constraint propagation, remains central to modern SAT solving. Modern SAT solvers spend much of their running time propagating consequences of partial assignments.

2.1.4 Conflict-driven clause learning

Modern SAT solvers such as CaDiCaL [8] and Kissat [10] are usually based on conflict-driven clause learning, or CDCL. CDCL extends DPLL by analyzing conflicts and learning new clauses that prevent the same kind of conflict from recurring [42]. During the search, each branching decision made by the solver creates a new *decision level*, consisting of the chosen decision literal together with all assignments forced by unit propagation before the next decision is made. CDCL also uses non-chronological backtracking, meaning that the solver can backtrack directly to the decision level relevant to the conflict, instead of undoing assignments one level at a time.

Suppose, for example, that the solver discovers that the assignments

$$p \equiv \mathbf{T}, \qquad q \equiv \mathbf{F}, \qquad r \equiv \mathbf{T}$$

lead to a contradiction. The solver may learn the clause $(\neg p \vee q \vee \neg r)$, which is the negation of the conjunction of those assignments. This clause thus rules out the same combination of assignments in the future. If the solver later assigns p and r to true again, then the learned clause becomes the unit clause q, forcing q to be true. Thus, the learned clause can cause unit propagation and prevent the solver from repeating the same mistake.

CDCL solvers analyze conflicts using an implication graph. This graph records which assignments were decisions and which were implied by unit propagation. By examining the graph, the solver identifies a reason for the conflict and derives a learned clause. The same analysis determines how far the solver should backtrack.

CDCL is the basis of most high-performance SAT solvers. Modern solvers also include many additional techniques, such as branching heuristics, clause deletion, restarts, preprocessing, and inprocessing [42, 24, 33].

2.1.5 Cardinality constraints

Many combinatorial problems contain constraints of the form “at most one”, “exactly one”, or more generally “at most k”. These are known as cardinality constraints. If

$x_1, \ldots, x_s$ are Boolean variables, then the constraint

$$\sum_{i=1}^{s} x_i \leq k$$

means that at most k of the variables may be assigned true. Similarly,

$$\sum_{i=1}^{s} x_i = k$$

means that exactly k of the variables are assigned true.

Cardinality constraints are not themselves clauses, but they can be encoded into CNF. The simplest encoding of an at-most-one constraint uses one binary clause for every pair of variables:

$$\neg x_i \vee \neg x_j \qquad \text{for all } i < j.$$

This says that no two variables may both be true. The resulting encoding contains $\binom{n}{2}$ clauses. An exactly-one constraint can be obtained by adding the at-least-one clause

$$\bigvee_{i=1}^{n} x_i.$$

This states that at least one of the variables must be true. For larger cardinality constraints, more compact encodings are often used, such as sequential counters or totalizer encodings [6].

Cardinality constraints appear in both case studies of this thesis. In the Gerver–Ramsey problem, they are used to ensure that no line contains too many points from the walk. In the MOLS problem, they are used to enforce Latin square and orthogonality constraints.

2.1.6 Proofs of unsatisfiability

When a SAT solver reports that a formula is satisfiable, it can provide a satisfying assignment. This assignment can be checked directly by evaluating the formula. The unsatisfiable case is more subtle, because the solver is claiming that no satisfying assignment exists.

Modern SAT solvers can produce proof certificates for unsatisfiable instances. These certificates are not usually intended to be read by humans, but they can be checked by an independent proof verifier. This improves trust in computational results, because the SAT solver itself does not need to be trusted as a black box.

One common proof format is DRAT [19], which stands for deletion resolution asymmetric tautology. A DRAT proof consists of a sequence of clause additions and deletions. Clause additions must be redundant with respect to the current formula, meaning that they do not change satisfiability. Clause deletions remove clauses that are no longer needed, which helps keep the proof manageable. A proof of unsatisfiability ends by deriving the empty clause. Since the empty clause cannot be satisfied by any truth assignment, deriving it proves that the original formula is unsatisfiable.

In the Gerver–Ramsey part of this thesis, unsatisfiability certificates are used to certify that certain walks do not exist (see Section 3.2.2). This is important when proving that north–east walks of a certain length are maximal, because one must show that no longer walks exist.

2.1.7 Parallelization

Some SAT instances are too difficult to solve efficiently on a single processor. A common method for parallelizing SAT solving is cube-and-conquer. The idea is to divide the search space into many disjoint subproblems, solve those subproblems independently, and then combine the results.

The splitting phase produces a collection of cubes. A cube is a conjunction of literals, such as $x_1 \wedge \neg x_2 \wedge x_3$. Each cube describes a subproblem obtained by fixing those literals to true. If the cubes partition the search space, then the original formula is satisfiable if and only if at least one cube subproblem is satisfiable. Similarly, the original formula is unsatisfiable if and only if every cube subproblem is unsatisfiable.

Cube-and-conquer typically uses a lookahead solver to choose useful branching variables. A lookahead solver spends more time than a CDCL solver evaluating the effect of possible branches, with the goal of producing subproblems of roughly balanced difficulty. In this thesis, the lookahead solver MARCH [30] is used for cube generation.

After a branching variable is selected, MARCH creates two subproblems, one where the variable is true and one where it is false, applies Boolean constraint propagation, and then repeats this process recursively until sufficiently many cubes have been produced.

Cube-and-conquer is especially useful when there are many available processors. Each processor can be given the original SAT instance together with one or more cubes. The processor then solves its assigned subproblem by assuming the literals in the cube.

2.1.8 Programmatic SAT

In a standard SAT workflow, the entire problem is encoded in CNF before the solver begins. This is effective when all relevant constraints are naturally expressible as clauses. However, some problems contain subproblems that are better handled by a specialized algorithm. Programmatic SAT provides a way to combine ordinary SAT solving with external computation during the search.

The SAT solver CaDiCaL supports a programmatic interface based on IPASIR-UP [27]. This interface allows user-defined callback functions to interact with the solver. In particular, it can report when selected variables are assigned, report when backtracking occurs, and add externally generated clauses during solving. Some important IPASIR-UP notification functions and callbacks include:

- `notify_assignment()`, which reports that a variable has been assigned;
- `notify_backtrack()`, which reports that the solver has backtracked and some assignments have been undone; and
- `has_external_clause()`, which indicates that an external clause is ready to be added to the solver.

This kind of interface is useful in SAT+CAS methods, where SAT solving is combined with symbolic computation. The SAT solver handles the general search and learns clauses from conflicts, while an external mathematical routine handles structured subproblems.

In the MOLS part of this thesis, the IPASIR-UP interface is used to facilitate the communication between the SAT solver and an external implementation of the Euler–

Parker algorithm (see Section 4.1.1). When the SAT solver completes one of the Latin squares, the external implementation is notified through `notify_assignment()`, and it attempts to find an orthogonal mate. If a mate is found, the external implementation communicates its cell assignments to the SAT solver as a set of unit clauses through `has_external_clause()`. The solver then propagates these clauses and discovers the satisfying assignment, possibly after backtracking. If the completed square does not have an orthogonal mate, the external implementation instead passes a blocking clause to the SAT solver through the same callback. This forces the solver to backtrack far enough to invalidate the completed square and continue the search.

The IPASIR-UP interface can also be useful for understanding how the SAT solver approaches a problem internally. For the Gerver–Ramsey problem, discussed in Section 3, the functions `notify_new_decision_level()` and `notify_assignment()` can be used to visualize how the solver constructs north–east paths by revealing the general order in which points are assigned to lie on the path. Figure 1 shows a heatmap of the sequence of assignments leading to a satisfying assignment for a 239-step walk avoiding 7 collinear points. Rather than beginning at the origin and extending the path sequentially, as a typical backtracking search might, the solver instead built the path from the middle and extended outwards in both directions.

Combining SAT solving and symbolic computation

Interfaces like IPASIR-UP are useful when part of the instance is more naturally handled outside the base SAT encoding. For example, one may invoke a computer algebra system (CAS) on a structured subproblem determined by the current partial assignment. This "SAT+CAS" paradigm has shown to be beneficial in a number of mathematical problems [16].

More generally, combinations of satisfiability checking (SC) and symbolic computation (SC) have led to a number of successes, resulting in what has become known as the "SC^2" paradigm [25]. The paradigm had its inception around 2015, when Ábrahám [1] and Zulkoski et al. [58] proposed combining the fields of satisfiability checking and symbolic computation to solve mathematical problems. Both fields

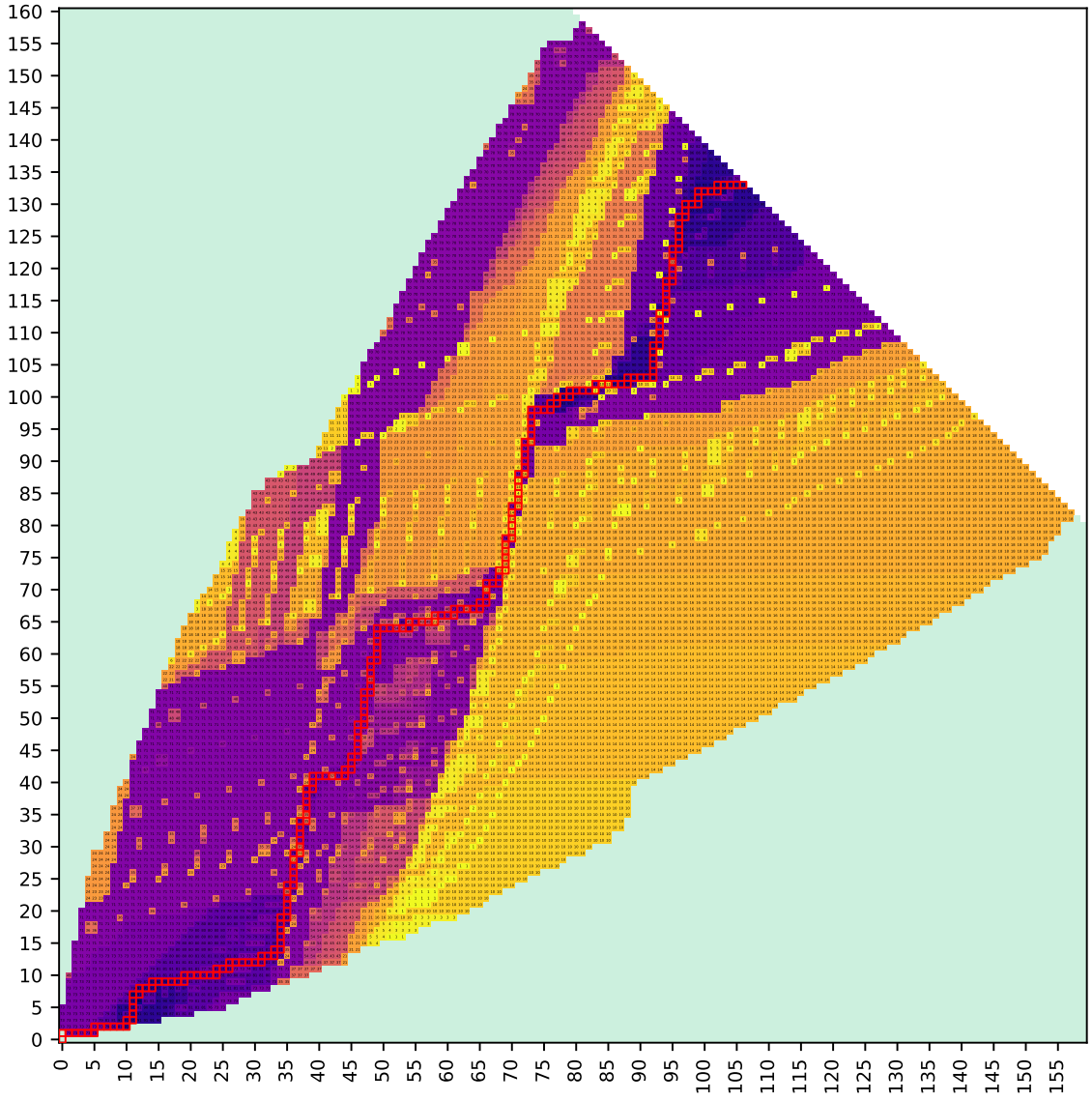


Fig. 1: Heatmap showing the decision level at which the literal corresponding to each grid point was assigned in a 239-step path avoiding 7 collinear points. The decision levels range from the root level, 0, to level 96, where the satisfying assignment was found. Points fixed at the root level by unit clauses are shown in light green. Levels 1 through 96 follow a yellow-purple gradient, with level 1 shown in the brightest yellow and level 96 in the darkest purple. Points on the path are highlighted with a red border. The first non-trivial point added to the path is $(93, 113)$.

were mature with well-established methods for solving mathematical problems, but there had been limited interaction between the fields up to that point, despite the fields' complementary strengths. Satisfiability checking is effective at general search and learning, while symbolic computation is effective at solving problems that admit clever solutions by exploiting mathematical structure. The SC^2 approach therefore excels at solving problems that can benefit from both effective search and exploiting mathematical structure.

The SC^2 workshop, now in its eleventh year, hosts work that combines satisfiability checking and symbolic computation. What is particularly notable is the sheer variety of problems that benefit from the SC^2 framework. For example, SC^2 approaches have been used on graph enumeration up to isomorphism [38, 39], factorizing integers with known bits of the factors [3, 4], searching for hash function collisions [5], and proving theorems in extremal combinatorics [2].

2.2 Exact Cover

An exact cover problem consists of a set X and a family of subsets $S_1, \ldots, S_N \subseteq X$. The goal is to decide whether there is a subfamily whose sets are pairwise disjoint and whose union is all of X.

Exact cover can be represented as a linear Diophantine system. Let A be a 0-1 matrix whose rows correspond to constraints and whose columns correspond to choices. The entry a_{ij} is 1 precisely when choice j covers constraint i. Then an exact cover can be encoded as

$$A\boldsymbol{x} = \mathbf{1}, \qquad \mathbf{0} \leq \boldsymbol{x} \leq \mathbf{1},$$

where $x_j = 1$ means that choice j is selected and $x_j = 0$ means that it is not selected. The equation $A\boldsymbol{x} = \mathbf{1}$ enforces that every constraint is covered exactly once.

In this thesis, our exact cover subproblems are solved using LIBEXACT [34]. More generally, LIBEXACT solves systems of the form

$$A\boldsymbol{x} = \boldsymbol{b}, \qquad \mathbf{0} \leq \boldsymbol{x} \leq \boldsymbol{u},$$

where A is a 0-1 matrix and the solution vector $\boldsymbol{x}$ is integral. For exact cover, the vectors $\boldsymbol{b}$ and $\boldsymbol{u}$ are taken to be all-one vectors.

For example, consider the two exact cover problems used in the Euler–Parker approach to find an orthogonal mate, discussed in Section 4.3. Let

$$P = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{pmatrix}.$$

Step 1 of the Euler–Parker algorithm finds the transversals of P. Finding a transversal can be encoded as an exact cover problem in LIBEXACT: the rows of A correspond to row, column, and symbol constraints that we label as

$$r_0, r_1, r_2, \qquad c_0, c_1, c_2, \qquad s_0, s_1, s_2,$$

and the columns of A correspond to the cells of P. Each cell covers exactly three constraints: its row, its column, and its symbol. For instance, since $P[0,0] = 0$, the cell $(0,0)$ covers the row constraint r_0, the column constraint c_0, and the symbol constraint s_0.

One transversal of P is the main diagonal,

$$T_0 = \{(0,0), (1,1), (2,2)\},$$

whose symbols are $(0,2,1)$. This transversal is represented by the solution $\boldsymbol{x}$ to $A\boldsymbol{x} = \boldsymbol{b} = \mathbf{1}$:

	$(0,0)$	$(0,1)$	$(0,2)$	$(1,0)$	$(1,1)$	$(1,2)$	$(2,0)$	$(2,1)$	$(2,2)$		
r_0	1	1	1	0	0	0	0	0	0	1	1
r_1	0	0	0	1	1	1	0	0	0	0	1
r_2	0	0	0	0	0	0	1	1	1	0	1
c_0	1	0	0	1	0	0	1	0	0	0	1
c_1	0	1	0	0	1	0	0	1	0	1	1
c_2	0	0	1	0	0	1	0	0	1	0	1
s_0	1	0	0	0	0	1	0	1	0	0	1
s_1	0	1	0	1	0	0	0	0	1	1	1
s_2	0	0	1	0	1	0	1	0	0	1	1
										$\boldsymbol{x}$	$\boldsymbol{b}=\mathbf{1}$

A

For step 2, suppose the transversals found in step 1 include

$$T_0 = \{(0,0),(1,1),(2,2)\}, \quad T_1 = \{(0,1),(1,2),(2,0)\}, \quad T_2 = \{(0,2),(1,0),(2,1)\}.$$

The second step in the Euler–Parker algorithm selects n pairwise disjoint transversals. This is again an exact cover problem: the rows correspond to the cells of P, and the columns correspond to the transversals found in step 1. Selecting all three transversals gives $A\boldsymbol{x} = \boldsymbol{b} = \mathbf{1}$:

$$\underbrace{\begin{array}{c|ccc} & T_0 & T_1 & T_2 \\ \hline (0,0) & 1 & 0 & 0 \\ (0,1) & 0 & 1 & 0 \\ (0,2) & 0 & 0 & 1 \\ (1,0) & 0 & 0 & 1 \\ (1,1) & 1 & 0 & 0 \\ (1,2) & 0 & 1 & 0 \\ (2,0) & 0 & 1 & 0 \\ (2,1) & 0 & 0 & 1 \\ (2,2) & 1 & 0 & 0 \end{array}}_{A} \underbrace{\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}}_{\boldsymbol{b}=\mathbf{1}}.$$

Note that because these are three pairwise disjoint transversals of an order-3 square, each cell of P is covered exactly once.

2.3 Myrvold's Order-10 MOLS Framework

The existence of a 3-MOLS(10) remains an open problem. Myrvold studied a special case of this problem under the assumption that one of the three Latin squares contains a 4×4 Latin subsquare [44]. This assumption imposes strong restrictions on the other two squares in the triple.

A transversal of a Latin square of order n is a set of n cells containing exactly one cell from each row, exactly one cell from each column, and exactly one occurrence of each symbol. Instead of working directly with an orthogonal pair, Myrvold used a *transversal representation pair*, or TRP. A TRP is a pair of Latin squares (P, Q)

in which the rows of Q represent disjoint transversals of P. Thus, the rows of Q encode a transversal decomposition of P. The squares P and Q are not themselves orthogonal mates in the ordinary overlay sense, but the TRP structure is equivalent to the existence of an associated orthogonal pair.

In Myrvold's framework, there is an underlying Latin square L whose transversals are represented by both P and Q. The square L is assumed to contain a 4×4 Latin subsquare Ω in its lower-right corner, using the symbols $\{0, 1, 2, 3\}$. This assumption induces a colouring on entries of P and Q. Entries containing symbols in $\{0, 1, 2, 3\}$ are called *white*. Among the nonwhite entries in the first six columns, certain entries are coloured *dark*, and the remaining nonwhite entries are called *light*.

Myrvold classified transversals according to the number of white entries they contain in the last four columns. There are four transversal types, denoted p_1, p_2, p_3, p_4. A transversal of type p_i contains exactly i white entries in the last four columns and exactly $2i - 2$ dark entries.

Using this classification, Myrvold studied the possible ways that L could be decomposed into ten transversals. She found seven possible general decomposition types, denoted R, S, T, U, V, W, X. Since a 3-MOLS(10) would give two transversal decompositions of the relevant square, she then considered pairs of decomposition types. She showed that only eight type pairs could not be ruled out:

$$\text{UU}, \quad \text{SX}, \quad \text{UW}, \quad \text{WW}, \quad \text{VX}, \quad \text{UX}, \quad \text{WX}, \quad \text{XX}.$$

An example of a type UW TRP is shown in Figure 2.

The remaining twenty possible pairings were eliminated by theoretical arguments. The eight unresolved cases remained open for over twenty-five years. In 2026, Bright et al. used SAT solving to construct transversal representation pairs in all eight cases [15]. Therefore, these eight cases cannot be ruled out using only constraints on the corresponding orthogonal pair. Any proof eliminating the original restricted 3-MOLS(10) case would have to use additional constraints involving the third square.

These eight cases are important for this thesis because they provide structured MOLS search problems. The SAT solver can impose Myrvold's structural constraints,

type U

0	2	3	4	5	6	1	7	8	9
1	0	4	9	7	2	6	5	3	8
3	9	8	2	4	1	0	6	7	5
4	1	7	0	3	5	9	8	6	2
5	8	0	1	2	7	3	9	4	6
8	3	1	6	9	0	4	2	5	7
2	7	6	8	0	9	5	3	1	4
9	6	2	5	1	8	7	4	0	3
6	4	5	7	8	3	2	1	9	0
7	5	9	3	6	4	8	0	2	1

type W

0	1	8	3	7	9	2	4	5	6
1	6	7	4	2	0	8	3	9	5
2	5	4	6	3	1	7	9	8	0
3	0	2	7	9	6	5	8	4	1
5	9	6	0	1	3	4	7	2	8
6	8	3	5	4	2	9	0	1	7
7	2	1	9	8	5	3	6	0	4
8	4	9	2	0	7	1	5	6	3
9	3	0	8	5	4	6	1	7	2
4	7	5	1	6	8	0	2	3	9

Fig. 2: A type-UW transversal representation pair, with white, light, and dark colouring. Each row of each square represents a transversal of the other square. For example, consider the first row of the first square $[0, 2, 3, 4, \dots]$. Going through the second square column-by-column and extracting the cell from each column containing these symbols in sequence yields the cells $(0, 0)$, $(6, 1)$, $(5, 2)$, $(1, 3)$, $\dots$ which form a transversal of the second square (the cells highlighted with a red border).

such as type constraints, colour constraints, symmetry breaking, and Ω-compatibility constraints. The Euler–Parker algorithm can then be used as a specialized subroutine for constructing orthogonal mates from completed Latin squares.

CHAPTER 3

North–East Lattice Walks

A *north–east lattice walk* is a path in the integer lattice whose steps are restricted to $(1, 0)$ and $(0, 1)$. Thus each step is either east or north. If such a walk starts at the origin and has m steps, then it contains $m + 1$ lattice points and ends on the line $x + y = m$. The Gerver–Ramsey collinearity problem asks how long a north–east lattice walk can be while avoiding k collinear points.

We write $a(k)$ for the smallest integer such that every north–east lattice walk with $a(k)$ steps contains k collinear points. Equivalently, $a(k) - 1$ is the maximum number of steps in a north–east walk avoiding k collinear points. We call a walk avoiding k collinear points a GR(k) walk, and we say that a GR(k) walk is *maximal* if it has $a(k) - 1$ steps.

For example, Meir noted that $a(3) = 4$ [23]. This means that there exist walks with three steps that avoid three collinear points, but every walk with four steps contains three collinear points. More generally, determining $a(k)$ requires both an existence result and a nonexistence result. To prove that $a(k) = m$, one must exhibit an $(m - 1)$-step walk avoiding k collinear points, and also prove that no m-step walk avoiding k collinear points exists.

The known exact values for $4 \leq k \leq 6$ were determined by Shallit [51] to be

$$a(4) = 9, \qquad a(5) = 29, \qquad a(6) = 97.$$

The first unknown value is $a(7)$. Before the work in this thesis, the best known lower bound was obtained by Shallit, who found a 260-step GR(7) walk, showing that $a(7) \geq 261$.

The problem is computationally difficult because the number of possible walks grows exponentially with the number of steps. An m-step walk has two choices at each step, so there are 2^m possible step sequences before accounting for symmetries or any additional structure. This makes naive exhaustive search infeasible for larger values of m.

In this thesis, we treat the existence of a GR(k) walk of fixed length as a satisfiability problem. Boolean variables are used to represent the points appearing on the walk. Some constraints ensure that these points form a valid north–east walk, while other constraints ensure that no line contains k selected points. A satisfying assignment gives a GR(k) walk, while an unsatisfiability proof certifies that no such walk exists.

3.1 Encoding a North–East Walk

We now describe our SAT encoding asserting the existence of a GR(k) walk with m steps. The values of k and m are taken to be fixed in advance, and we let $n = m + 1$ denote the number of points in the walk. Without loss of generality we take our starting point as the origin $(0, 0)$, so an m-step north–east lattice path ends on the line $y = m - x$. In order to describe such a walk, we use the Boolean variable $v_{x,y}$ to represent that the point (x, y) appears on the walk. There are $i + 1$ points reachable from the origin in a north–east lattice path with i steps, so there are a total of $\sum_{i=0}^{n-1}(i + 1) = n(n + 1)/2$ point variables in our instance.

3.1.1 Path constraints

Next, we describe the constraints on the variables $v_{x,y}$ that must hold in north–east lattice paths. In what follows, (x, y) is one of the points that might appear on an m-step (n-point) north–east lattice path (i.e., $(x, y) \in \mathbb{N}^2$ with $x + y \leq m$). First, we know that if point (x, y) is on the path then either $(x + 1, y)$ or $(x, y + 1)$ is on the path, unless (x, y) is the final point. We encode this constraint using the clauses

$$v_{x,y} \to (v_{x+1,y} \vee v_{x,y+1}) \qquad \text{for } x + y \neq m.$$

Second, we know that if point $(x, y) \neq (0, 0)$ is on the path then either $(x-1, y)$ or $(x, y-1)$ is on the path. We encode this using the clauses

$$v_{x,y} \to (v_{x-1,y} \vee v_{x,y-1}), \quad v_{0,y} \to v_{0,y-1}, \quad v_{x,0} \to v_{x-1,0} \qquad \text{for } x, y \geq 1.$$

Third, we know that a path never splits into two directions: both $(x+1, y)$ and $(x, y+1)$ can never both be on the path at the same time. We encode this using the clauses

$$\neg v_{x+1,y} \vee \neg v_{x,y+1} \qquad \text{for } x + y \neq m.$$

Lastly, we assert that the origin is on the path with the unit clause $v_{0,0}$. In total, we have $O(n^2)$ path constraints. A satisfying assignment of these constraints provides a north–east lattice path starting from the origin and ending after m steps.

3.1.2 Non-collinearity constraints

In order to assert that the path is a GR(k) walk, we need to assert that it does not contain k collinear points. To do this, we use a generalization of clauses known as cardinality constraints. An at-most-k cardinality constraint over a set of literals $X = \{x_1, \ldots, x_s\}$ says that no more than k of the literals in X can be assigned true, and we use the notation $\sum_{i=1}^{s} x_i \leq k$ to represent this constraint. Note that cardinality constraints are not natively in conjunctive normal form. However, there are a number of efficient ways of converting cardinality constraints into CNF such as the sequential counter encoding [52] and the totalizer encoding [6]. Moreover, other formats like at-least-k conjunctive normal form [49] have native support for cardinality constraints (see Section 3.1.5).

Now, we assert that no k collinear points appear on the path. This requires determining all ways in which the points in our instance (i.e., $(x, y) \in \{0, \ldots, n-1\}^2$ with $x+y < n$) might lie on the same line. First, consider the case of avoiding k points on the same vertical line. Avoiding k points on the line $x = i$ can be accomplished with the cardinality constraint

$$\sum_{j=0}^{n-i-1} v_{i,j} \leq k - 1,$$

and avoiding k points on the horizontal line $y = j$ can be accomplished with the cardinality constraint

$$\sum_{i=0}^{n-j-1} v_{i,j} \leq k-1.$$

Generalizing this, avoiding k points on the line with slope s and y-intercept b can be accomplished with the cardinality constraint

$$\sum_{\substack{i,si+b\in\mathbb{N} \\ i<(n-b)/(s+1)}} v_{i,si+b} \leq k-1.$$

The slope s will be of the form r/d where $r \in \mathbb{N}$ is the rise of the slope and $d \in \mathbb{N}$ is the run of the slope. We can assume that $r+d \leq (n-1)/(k-1)$, since otherwise there will necessarily be fewer than k points on a line with slope $s = r/d$ in the relevant region, as indicated in Proposition 1.

Proposition 1. *Suppose the region* $\{\,(x,y) \in \mathbb{N}^2 : x+y < n\,\}$ *contains* k *points on a line with slope* r/d*. Then* $r+d \leq (n-1)/(k-1)$*.*

Proof. Order the k points on a line with slope r/d so that $(x_{i+1}, y_{i+1}) = (x_1+id, y_1+ir)$ for $0 \leq i < k$. The number of steps it takes to walk from (x_1, y_1) to (x_k, y_k) is $x_k - x_1 + y_k - y_1 = d(k-1) + r(k-1) = (r+d)(k-1)$. It is possible to take up to $n-1$ north–east steps while remaining in the region $\{\,(x,y) \in \mathbb{N}^2 : x+y < n\,\}$, so if all points are in this region then $(r+d)(k-1) \leq n-1$, and $r+d \leq (n-1)/(k-1)$. □

We can also assume that the slope is written in lowest terms so that r and d are coprime. The probability two integers in $[1, n)$ are coprime tends to $6/\pi^2$ when $n \to \infty$, so there are asymptotically $O((n/k)^2)$ slopes to consider when the rise and run are each bounded by n/k (cf. Proposition 3 below).

Each slope $s = r/d$ forms a line $y = sx$ for which we add the constraint $\sum_{i=0}^{\lceil n/(s+1)\rceil - 1} v_{i,si} \leq k-1$. There are $O(n^2/k^2)$ slopes to consider, so there are $O(n^2/k^2)$ lines to consider with zero y-intercept. Next, consider lines with positive y-intercepts $b \in \mathbb{Q}$. Note $b \leq n$, otherwise $y = si + b > n$. Also, the denominator of b in lowest terms must divide d, otherwise $y = (r/d)i + b$ would not be an integer. Thus we have $b = \alpha/d$ where $\alpha \leq nd$ is a positive integer. Thus, there are $O(nd) = O(n^2/k)$ lines

with positive y-intercepts to consider. By symmetry, there are also $O(n^2/k)$ lines with positive x-intercepts to consider (which are equivalent to the lines with $b < 0$). Thus, in total there are $O(n^2/k)$ values of b to consider.

Since there are $O(n^2/k^2)$ slopes s to consider and $O(n^2/k)$ y-intercepts b to consider, there are $O(n^4/k^3)$ cardinality constraints in total. Not all these constraints are necessary to include; e.g., if there are strictly less than k variables $v_{x,y}$ corresponding to points on the line $y = sx+b$, then the constraint associated with this line is unnecessary, since the constraint $\sum_{y=sx+b} v_{x,y} < k$ will always be satisfied. Proposition 2 below provides an additional restriction on the line slopes, showing certain non-collinearity constraints to be unnecessary.

Proposition 2. *A north–east lattice path without $k-1$ consecutive steps in the same direction does not have k points on a line with slope $s > k-2$.*

Proof. For contradiction, suppose a north–east lattice path without $k-1$ consecutive steps in the same direction has k points (x_1, y_1), $\ldots$, (x_k, y_k) on a line with slope $s > k-2$. Suppose $x_k - x_1 = r$ and let V_i denote the number of north steps taken along the line $x = i$, noting that $V_i \leq k-2$ since the path never has $k-1$ consecutive north steps. The total number of north steps taken from (x_1, y_1) to (x_k, y_k) is $y_k - y_1 = \sum_{i=x_1}^{x_k} V_i$. Since $V_i \leq k-2$ and there are $r+1$ summands, $\sum_{i=x_1}^{x_k} V_i \leq (r+1)(k-2)$.

Also note that $y_{i+1} - y_i = s(x_{i+1} - x_i)$ since (x_i, y_i) and (x_{i+1}, y_{i+1}) are both on a line with slope s. Since $s > k-2$, we have $y_{i+1} - y_i > (k-2)(x_{i+1} - x_i)$, and since both sides are integers, we have $y_{i+1} - y_i \geq (k-2)(x_{i+1} - x_i) + 1$. Summing this from $i = 1$ to $k-1$ we obtain

$$\begin{aligned}\sum_{i=1}^{k-1}(y_{i+1} - y_i) &\geq \sum_{i=1}^{k-1}\big((k-2)(x_{i+1} - x_i) + 1\big) \\ &= (k-2)(x_k - x_1) + (k-1) \\ &= r(k-2) + (k-1) = (r+1)(k-2) + 1.\end{aligned}$$

However, the left-hand side equals $y_k - y_1 = \sum_{i=x_1}^{x_k} V_i$. Putting our bounds on $y_k - y_1$

together,

$$(r+1)(k-2)+1 \leq y_k - y_1 \leq (r+1)(k-2),$$

a contradiction. □

As a result of Proposition 2, we do not need to consider cardinality constraints corresponding to lines with slopes $s > k-2$. Symmetrically, we also do not need to consider constraints corresponding to lines with slopes $s < 1/(k-2)$. Even when the encoding is optimized to remove constraints proven to be unnecessary, the cardinality constraints still dominate the encoding size. For example, with $n = 325$ and $k = 7$ there are about 953,000 non-collinearity constraints and about 158,000 path constraints.

Finally, Proposition 2 allows the bounds on the rise and run in Proposition 1 to be slightly tightened.

Proposition 3. *Suppose the region* $\{ (x,y) \in \mathbb{N}^2 : x+y < n \}$ *contains* $k > 2$ *points on a line with slope* r/d *and these points are all on a north–east lattice path without* $k-1$ *consecutive steps in the same direction. Then* $r < (n-1)/k$.

Proof. By Proposition 2, we have $r/(k-2) \leq d$, and by Proposition 1, we have $r + d \leq (n-1)/(k-1)$. Combining these, we have $r(1+1/(k-2)) \leq (n-1)/(k-1)$. Multiplying both sides by $(k-2)/(k-1)$ gives $r \leq (n-1)(k-2)/(k-1)^2 < (n-1)/k$. □

Similarly, it also follows that $d < (n-1)/k$. Thus we need only consider lines whose slopes have rise and run in $\{1, \ldots, \lfloor (n-1)/k \rfloor\}$.

Horizontal/vertical non-collinearity optimization The only way it is possible to have k vertical collinear points in a north–east lattice path is by taking a north step $k-1$ times in a row. Thus, it is possible to block the existence of k vertical points on the line $x = i$ via the binary clauses

$$v_{i,j} \to \neg v_{i,j+k-1} \qquad \text{for all } j = 0, \ldots, n-i-k.$$

Similarly, it is possible to block the existence of k horizontal points on the line $y = j$ via the binary clauses

$$v_{i,j} \to \neg v_{i+k-1,j} \qquad \text{for all } i = 0, \ldots, n-j-k.$$

Although a minor optimization, experimentation showed that this alternate encoding of the vertical and horizontal non-collinearity constraints tended to be preferable to the cardinality encodings described above.

3.1.3 Symmetry breaking

There are three nontrivial operations that when applied to a GR(k) walk produce another GR(k) walk: complement the steps (switch north steps with east steps and vice versa), reverse the steps, and a complement + reverse combination. We consider the paths generated by these operations as equivalent, and to shrink the search space it is desirable to remove such paths from the search space—assuming that *up to equivalence* we don't remove paths. The process of adding extra constraints that remove solutions that can be assumed without loss of generality is known as *symmetry breaking*.

The complementation symmetry is simple to break by enforcing the first step to be north, since a Gerver–Ramsey walk without a north first step can be complemented to make an equivalent walk with a north first step. We encode this by adding the unit clause $v_{0,1}$ into our SAT encoding (which then implies $\neg v_{1,0}$).

We also tried breaking the reversal and reversal+complement symmetries, but the SAT encoding to do this was more involved. Ultimately, experiments revealed that the overhead of adding more clauses into the encoding was more trouble than it was worth. Thus, the SAT encodings we used ignored the reversal symmetries, only breaking the complementation symmetry via a north first step. Exhaustive enumeration of GR(k) walks was possible for $k \leq 6$, and after the enumeration was complete we checked for and removed GR(k) walks that were duplicates up to equivalence (see Section 3.2.2).

3.1.4 Blocking extremal points

Points that are too close to the x-axis or y-axis can quickly be shown to never occur in a GR(k) walk and therefore can be blocked directly in the encoding. In particular, because a GR(k) walk can never take $k-1$ consecutive steps in the north direction, a

GR(k) walk can never cross the line $y = (k-2)x + (k-1)$. Thus, we add in the unit clauses

$$\neg v_{x,(k-2)x+k-1} \qquad \text{for } 0 \leq x < n/(k-1) - 1$$

which blocks the points $(0, k-1)$, $(1, 2k-3)$, $(2, 3k-5)$, ... from appearing on the path.

Similarly, a GR(k) walk can never take $k-1$ consecutive steps in the east direction, so a GR(k) walk whose first step is north can never cross the line $y = \frac{1}{k-2}(x-1)$. Thus, we add in the unit clauses

$$\neg v_{(k-2)y+1,y} \qquad \text{for } 0 \leq y < (n-1)/(k-1)$$

which blocks the points $(1, 0)$, $(k-1, 1)$, $(2k-3, 2)$, ... from appearing on the path.

3.1.5 At-least-*k* conjunctive normal form

It is not always convenient to write a logical expression in CNF, and a number of more expressive extensions of CNF have been proposed. One such extension proposed by Reeves, Heule, and Bryant [50], called *at-least-k conjunctive normal form* (KNF), augments CNF with constraints of the form $l_1 + \cdots + l_s \geq k$ where l_1, ..., l_s are literals. Such a constraint is known as a *klause* and is satisfied by an assignment if at least k of the literals l_1, ..., l_s are true under that assignment.

Klauses are by definition lower bounds, but upper bounds can also be expressed as klauses since the lower bound $l_1 + \cdots + l_s \geq k$ is equivalent to the upper bound $\bar{l}_1 + \cdots + \bar{l}_s \leq s - k$ where $\bar{x}$ denotes the negation of x. Thus, we are able to express the non-collinearity constraints from Section 3.1.2 natively using klauses. For example, the constraint that there are at most $k-1$ points on the line $y = x$ is represented as the klause

$$\sum_{i=0}^{\lceil n/2 \rceil - 1} \bar{v}_{i,i} \geq \lceil n/2 \rceil - k + 1.$$

Reeves et al. [50] provide a KNF solver based on the SAT solver CaDiCaL [9] called Cardinality-CaDiCaL. Their code is available at `https://github.com/jreeves3/`

`Cardinality-CDCL/`. Their solver is able to reason about klauses natively and incorporates a cardinality-based propagation routine for deriving consequences of partial assignments. They report that Cardinality-CaDiCaL performs particularly well on satisfiable instances having many large cardinality constraints. Intuitively, cardinality-based propagation allows solving satisfiable instances faster because it bypasses the auxiliary variables used to convert cardinality constraints into CNF in a compact way. On the other hand, they report that for unsatisfiable instances converting klauses into CNF cardinality constraint encodings tends to result in improved performance. Intuitively, the auxiliary variables used in the CNF conversion tend to be important for finding short proofs of unsatisfiability.

3.1.6 Encoding the unreachability of points

Section 3.1.4 provides upper and lower bounds on GR(k)-reachable points; in particular, GR(k)-reachable points always lie between the lines $y = (k-2)x + (k-1)$ and $y = \frac{1}{k-2}(x-1)$. However, these bounds are not tight for large x. Given that if a point (x, y) can be shown to be GR(k)-unreachable then the Boolean variable $v_{x,y}$ can be fixed to false, it is desirable to determine the reachability of as many points as possible.

The problem of determining the reachability of a point can be phrased using a variant of the SAT encoding we've already described. Say we want to determine if (x, y) is a GR(k)-reachable point. We generate the SAT encoding specifying the existence of an $(x+y)$-step GR(k) walk, except we add the additional unit clause $v_{x,y}$ into the encoding. The presence of $v_{x,y}$ ensures the point (x, y) must appear on the path. If such an instance is satisfiable, the satisfying assignment will provide a GR(k) walk from $(0, 0)$ to (x, y). Otherwise, if such an instance is unsatisfiable, this implies that (x, y) is a GR(k)-unreachable point.

Once it is known that (x, y) is GR(k)-unreachable, the unit clause $\neg v_{x,y}$ is included in all larger instances specifying the existence of GR(k) walks with more than $x + y$ steps. Such unit clauses help the solver, because the solver now no longer needs to consider walks passing through (x, y). In fact, we can say more: if (x, y) is GR(k)-

unreachable when starting from the origin, it must also be the case that $(x+x_0, y+y_0)$ is GR(k)-unreachable when starting from (x_0, y_0). Thus, instances asserting the existence of GR(k) walks of length n may also include clauses of the form

$$v_{x_0,y_0} \to \neg v_{x_0+x,y_0+y} \qquad \text{for all } (x_0, y_0) \in \mathbb{N}^2 \text{ with } x_0 + y_0 < n - x - y \tag{1}$$

where (x, y) is a GR(k)-unreachable point with $x + y < n - 1$.

Some care must be taken in order to use the unreachability clauses in conjunction with the symmetry breaking described in Section 3.1.3. Recall our symmetry breaking assumes the first step is north. If (x, y) is determined to be unreachable in a GR(k) walk using $x + y$ steps (the first of which is north), it follows that (x, y) does not appear on all longer GR(k) walks using our symmetry breaking (i.e., with a north first step). However, the clauses in (1) can only be added if it is known that (x, y) is unreachable from the origin in walks with *either* a north or east first step. Note that walks from the origin to (x, y) with a north first step are equivalent to walks from the origin to (y, x) with an east first step. Thus, we add clauses of the form (1) when both (x, y) and (y, x) were determined to be unreachable from the origin in GR(k) walks with a north first step.

3.1.7 Constraint-removal heuristic

Although not always the case, SAT solvers may perform better if redundant constraints are not used in the encoding. During solving, modern SAT solvers store all clauses in memory and most of their time is spent performing constraint propagation, a task whose running time is proportional to the number of clauses stored in memory. Thus, reducing the number of stored clauses often improves the performance of the solver, particularly when the removed clauses are redundant.

In the Gerver–Ramsey collinearity problem, we observed that many of the non-collinearity constraints described in Section 3.1.2 were not useful in practice. That is, many of these constraints could be removed and the solver could still either find correct GR(k) paths (in satisfiable instances) or prove that no GR(k) paths exist (in unsatisfiable instances).

Note that the technique of removing some non-collinearity constraints is heuristic. On the one hand, if constraints are removed from the instance and the solver reports an UNSAT result, we know for certain that the original instance was also unsatisfiable (as removing constraints can only *increase* the number of satisfying assignments, never decrease it). On the other hand, if constraints are removed from the instance and the solver reports a SAT result, there is no guarantee that the satisfying assignment returned by the solver is actually a GR(k) path. However, in such cases we can explicitly check that the returned path has no k collinear points on it. To do this check efficiently, we iterate over all pairs of points on the path and ensure that the line through the two points in the pair contains fewer than k points on the path.

In practice, we found that the majority of the non-collinearity constraints corresponded to lines having a relatively small number of points (x, y) in the relevant region of $\{ (x, y) \in \mathbb{N}^2 : x + y < n \}$. For example, for $k = 7$ and $n = 300$, about 35% of the constraints contain exactly 7 points in the relevant region. In practice these constraints are unlikely to be useful, as it is unlikely for a satisfying assignment to pass through all 7 points simultaneously. Thus, we removed constraints corresponding to lines with a small number of points in the relevant region. For $k = 7$ and $n \geq 150$, we removed all constraints corresponding to lines containing 16 or fewer points in the relevant region. Despite removing the majority of the non-collinearity constraints in this way (for example, about 94% for $n = 300$), most satisfying assignments found by the solver produced a valid GR(k) walk, with fourteen exceptions among several hundred cases. This heuristic also produced a speedup in the solver's efficiency.

3.2 Results

We now discuss our experimental results. Our source code is available at `https://github.com/aaronbarnoff/Collinear`. We start with a description of the benchmarking we did in order to determine the effectiveness of the encodings from Section 3.1 (see Section 3.2.1). We then describe how we used our SAT encoding to enumerate all GR(k) walks up to $k = 6$ and produce certificates demonstrating that there do not

exist GR(3), GR(4), GR(5), and GR(6) walks with 4, 9, 29, and 97 steps, respectively (see Section 3.2.2). Unless otherwise mentioned, experiments in Sections 3.2.1, 3.2.2 and 3.2.3 were run on the Digital Research Alliance of Canada Fir cluster, a high performance computing cluster consisting of AMD EPYC processors, most of which run at 2.7 GHz. The experiments of Section 3.2.4 used the Digital Research Alliance of Canada Nibi cluster with Intel Xeon processors at 2.4 GHz. A few experiments were run on a desktop computer with an AMD Ryzen 9950X 4.3 GHz processor and 64 GiB of memory and these are indicated separately.

3.2.1 Benchmarking

Because modern SAT solvers use a number of heuristics (e.g., to decide which variable to branch on when solving) their solving times on the same instance tend to vary widely. This is especially true if the instance is satisfiable, since the solver will stop as soon as a single satisfying assignment is found, and depending on the heuristic choices this will sometimes happen much quicker than usual. Unsatisfiable instances usually have more consistent solve times (since in these cases the solver always needs to prove there are no satisfying assignments) but even unsatisfiable instances have variance in their solve times. To mitigate the effect of this variance, we solved each benchmark 15 times using 15 different random seeds and report the median running time.

KNF vs. CNF: Performance

Our first set of benchmarks explore the performance of a KNF encoding versus a CNF encoding. We experimented with all of the CNF cardinality encodings supported by the Python library PySAT [32], and we found the sequential counter encoding had the best performance for this problem, so our CNF instances used the sequential counter encoding for the non-collinearity cardinality constraints. The SAT solver used to solve the CNF instances was CaDiCaL [9], and the KNF solver was Cardinality-CaDiCaL [50].

Four satisfiable benchmarks were chosen and four unsatisfiable benchmarks were

Table 1: Median solve times (in seconds) across 15 trials for two encodings (a CNF encoding and a KNF encoding) of eight different benchmarks.

k	n	Endpoint	Type	CNF time	KNF time
6	97	—	SAT	102.5	25.5
7	220	—	SAT	2347.4	76.2
7	240	—	SAT	6802.7	374.6
7	261	—	SAT	26 499.8	2750.0
6	98	—	UNSAT	616.6	361.0
7	122	$(33, 88)$	UNSAT	363.8	702.2
7	151	$(46, 104)$	UNSAT	1529.3	10 826.6
7	180	$(56, 123)$	UNSAT	2842.6	46 439.4

chosen (in each case, one benchmark had $k = 6$, and the other three benchmarks had $k = 7$). The unsatisfiable instances with $k = 7$ had an endpoint (x, y) of the path added as a unit clause $v_{x,y}$, with the point (x, y) chosen to be GR(k)-unreachable. Each benchmark was solved 15 times using 15 random seeds, and the median times (in seconds) are presented in Table 1. CaDiCaL used at most 7743 MiB of memory on the satisfiable benchmarks and 1685 MiB on the unsatisfiable benchmarks. For the KNF encoding, Cardinality-CaDiCaL required at most 887 MiB on the satisfiable benchmarks and 492 MiB on the unsatisfiable benchmarks.

The results show that the KNF encoding tends to perform better on satisfiable instances, while the CNF encoding tends to perform better on unsatisfiable instances, matching the observation of Reeves et al. [50]. Thus, in our future results when we know or expect the instance to be satisfiable we use the KNF encoding and otherwise we use the CNF encoding.

Unreachable points encoding: Performance

The next set of benchmarks examines the performance of the unreachable point encoding described in Section 3.1.6. When solving an instance asserting the existence

of an m-step GR(k) walk, if there are known points (x, y) with $x + y < m$ that are GR(k)-unreachable, we exploit this in our encoding. For now, we assume the reachability of points in $\{ (x, y) \in \mathbb{N}^2 : x + y < n - 1 \}$ is known; in Section 3.2.3 we describe the computations performed to determine the reachability of points. As described in Section 3.1.6, for each GR(k)-unreachable point (x, y), we add the unit clause $\neg v_{x,y}$ (stating that (x, y) is not on the path) and the binary clauses from (1).

We used the same eight benchmarks from Section 3.2.1, and solved them with and without the unreachability clauses. Again, 15 trials were run with 15 different random seeds and the median running time is given in Table 2. CaDiCaL used at most 970 MiB on the unsatisfiable benchmarks, whereas Cardinality-CaDiCaL used at most 597 MiB on the satisfiable benchmarks.

The results show that the unreachability clauses tend to help the solver, especially for unsatisfiable instances, where adding the unreachability clauses improved the solver's median running time, sometimes dramatically: for example, the solver was able to show there is no GR(7) walk from the origin to $(56, 123)$ about 110 times faster when the unreachable point clauses were included. For satisfiable instances, the results were less dramatic, but the unreachability clauses still tended to improve the performance of the solver.

Constraint-removal heuristic: Performance

We now examine the performance of the constraint-removal heuristic described in Section 3.1.7. For $k = 6$ with the heuristic enabled, we ignored all non-collinearity constraints corresponding to lines with 13 or fewer points in the region $\{ (x, y) \in \mathbb{N}^2 : x + y < n \}$ and the solver was still able to prove there are no GR(k) walks with $n = 98$ points. For $n = 97$, the solver found paths containing k collinear points, indicating that some of the ignored constraints were not redundant. Since the $k = 6$ instances were quickly solvable without the heuristic anyway, for $k = 6$ we only enabled the heuristic on the final unsatisfiable case (for $n = 98$ points).

For $k = 7$, we ignored all non-collinearity constraints corresponding to lines with 16 or fewer points in the region $\{ (x, y) \in \mathbb{N}^2 : x + y < n \}$. For $n \geq 150$ the solver was

Table 2: A table summarizing solve times using our encoding with and without the reachability clauses. The reported times are the median time (in seconds) across 15 trials, each using a different random seed. Satisfiable (SAT) instances use the KNF encoding and unsatisfiable (UNSAT) instances use the CNF encoding.

k	n	Endpoint	Type	Without	With
6	97	—	SAT	25.5	22.9
7	220	—	SAT	76.2	59.3
7	240	—	SAT	374.6	361.3
7	261	—	SAT	2750.0	2397.4
6	98	—	UNSAT	616.6	508.7
7	122	$(33, 88)$	UNSAT	363.8	134.4
7	151	$(46, 104)$	UNSAT	1529.3	35.2
7	180	$(56, 123)$	UNSAT	2842.6	25.8

able to solve our previous benchmarks correctly—for the satisfiable instances, valid GR(k) walks were found and the unsatisfiable instances were determined to have no GR(k) walks. For the unsatisfiable $n = 122$ benchmark, the solver found a satisfying assignment having k collinear points on the corresponding path, meaning the ignored constraints were not redundant. This is an indication that the heuristic works better for larger n which are the instances that are the most difficult to solve.

Table 3 contains a summary of the results we found using our constraint-removal heuristic on several satisfiable and unsatisfiable benchmarks. The constraint-removal heuristic significantly lowered memory usage, bringing CaDiCaL down to at most 391 MiB on the unsatisfiable cases (2.5× reduction) and Cardinality-CaDiCaL to at most 344 MiB on the satisfiable ones (1.7× reduction).

3.2.2 Enumeration of GR(k) walks for $k \leq 6$

Here we describe our enumeration of all GR(k) walks up to isomorphism for $k \leq 6$. For this, we use the basic CNF encoding described in Section 3.1 along with the

Table 3: A table summarizing solve times with and without using our constraint-removal heuristic. The reported times are the median time (in seconds) across 15 trials, each using a different random seed. Satisfiable (SAT) instances use the KNF encoding and unsatisfiable (UNSAT) instances use the CNF encoding.

k	n	Endpoint	Type	Heuristic Off	Heuristic On
7	220	—	SAT	59.3	48.4
7	240	—	SAT	361.3	245.2
7	261	—	SAT	2397.4	1710.3
6	98	—	UNSAT	508.7	395.9
7	151	$(46, 104)$	UNSAT	35.2	26.4
7	180	$(56, 123)$	UNSAT	25.8	18.3

unreachable point encoding of Section 3.1.6 (but not the constraint-removal heuristic).

For a fixed k, an incremental approach was used to enumerate all GR(k) walks. A variable m was used to control the number of steps in the walk. Given k and m, the enumeration of all m-step GR(k) walks was accomplished by generating $m + 1$ SAT instances, one for each ending point $(x, m - x)$ with $x \in \{0, 1, \ldots, m\}$. For each such point, a version of CaDiCaL that exhaustively finds *all* solutions of a SAT instance was used to find all GR(k) walks ending at the point $(x, m - x)$. Once all GR(k) walks with m steps were known they were filtered up to isomorphism using the equivalence operations described in Section 3.1.3.

The equivalence filtering was done by converting each GR(k) walk into a normal form in such a way that all equivalent walks produce the same normal form. To do this, each m-step walk is represented as a binary string of length m, with `1` representing a north step and `0` representing an east step. The normal form of a GR(k) walk is the walk whose binary string is the lexicographically greatest of all walks in the same equivalence class.

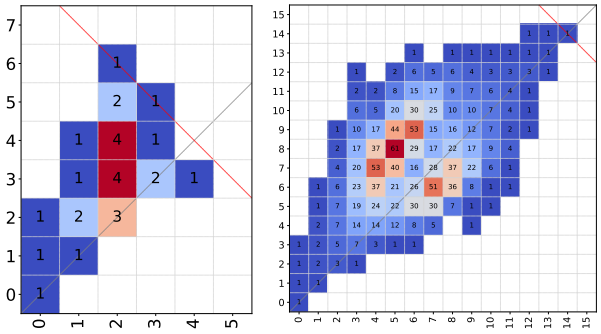
Once all m-step GR(k) walks had been determined, if there was at least one m-step GR(k) walk then m was incremented by 1 and the enumeration process was repeated.

Eventually, no m-step GR(k) walks were found. Once this happens, we have that $a(k) = m$ and the length of the maximal GR(k) walk(s) is $m - 1$. For example, we found that $a(4) = 9$, $a(5) = 29$, and $a(6) = 97$, confirming the results of Shallit [51]. The computations for $k = 4$ and $k = 5$ completed in under a second of CPU time, while the $k = 6$ case required 118,990 seconds.

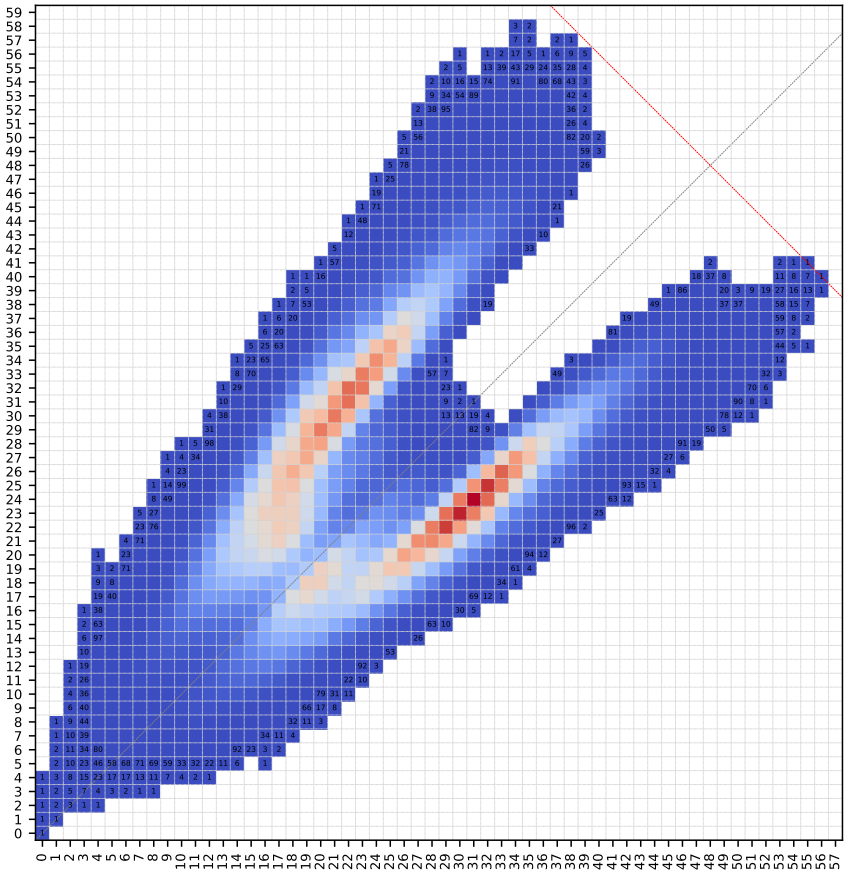
Visual heatmap diagrams depicting our GR(k) walk enumeration results for $k = 4$ to $k = 6$ are given in Figure 3. These diagrams visually show how many GR(k) walks (in normal form) exist from the origin to a point (x, y) in the plane. Only walks in normal form are counted, so it is possible a point has walks to it when both the point below and the point to the left do not. For example, $(9, 4)$ is reachable using a GR(5) walk, but $(9, 3)$ and $(8, 4)$ are not reachable by walks in normal form. $(8, 4)$ is GR(5)-reachable, but only using a walk which in normal form ends at $(4, 8)$.

Once the value of $a(k) = m$ has been determined, we produce unsatisfiability certificates demonstrating the nonexistence of m-step GR(k) walks. The certificates are available at `https://doi.org/10.5281/zenodo.17645678`. When combined with the known GR(k) walks of length $m - 1$ (which are easy to check for correctness) this provides a certificate that $a(k) = m$. The unsatisfiability certificates are in the DRAT format [19], a standard format for unsatisfiability proofs in modern SAT solving. A DRAT proof consists of a list of clause additions and deletions. Added clauses are redundant with respect to the current formula, meaning they may be added without changing the formula's satisfiability. Deletions discard clauses when they are deemed to not be useful (in order to keep the memory footprint of the solver manageable). The last added clause in the DRAT proof is the empty clause. The empty clause having the same satisfiability status as the original formula proves the original formula to be unsatisfiable, since no truth assignments satisfy the empty clause. The proofs were validated with the proof verifier DRAT-TRIM [56]. This tool verifies that each clause addition is indeed a logical consequence of previously derived clauses. Thus, the SAT solver itself does not need to be trusted; only the proof verifier—a much simpler piece of software—needs to be trusted.

Table 4 summarizes the running times of the proof generation and proof verification



(a) $k = 4$ (b) $k = 5$



(c) $k = 6$

Fig. 3: Exhaustive enumeration of GR(k) walks for $k = 4$ to $k = 6$ in normal form. Color intensity indicates the number of distinct walks reaching each point, with red indicating the greatest number of paths (the deepest red for $k = 6$ corresponding to 324,571 walks), blue indicating the least number of paths, and white indicating the point is unreachable using a GR(k) walk in normal form. The red antidiagonal line corresponds to $y = (a(k) - 1) - x$.

Table 4: Results for proving $a(k) \leq m$ for $3 \leq k \leq 6$. CaDiCaL was used for solving and proof generation and DRAT-trim was used for proof verification. Solving and verification times are given in seconds.

k	m	Solve time	Proof size	Verification time
3	4	< 1 sec	< 1 KiB	< 1 sec
4	9	< 1 sec	< 1 KiB	< 1 sec
5	29	< 1 sec	23 KiB	< 1 sec
6	97	311 sec	583 MiB	516 sec

steps for $3 \leq k \leq 6$, and these were done on the Ryzen machine. The proof that $a(6) \leq 97$ was generated by CaDiCaL in 311 seconds and was verified by DRAT-trim in 516 seconds. This nonexistence certificate provides more trust when compared to a traditional search program—because a bug in a search program could cause GR(6) walks to be missed, and there is no way to tell after-the-fact if there are no certificates that can be examined for correctness. However, for the purposes of double-checking and runtime comparison, we wrote a custom backtracking search program in C++ that we used to search for GR(6) walks of length 97, and this search took 2344 seconds on the Ryzen machine to confirm that there are no GR(6) walks of length 97 (i.e., 7.5× slower than the SAT solver). Although with more work the speed of the backtracking search program could likely be improved, this is an indication that not only are SAT solvers more trustworthy than custom search, they can also be more efficient.

3.2.3 Reachability bounds for GR(7) walks

As explained in Section 3.1.6, if a point (x, y) is known to be GR(k)-unreachable then we can add clauses encoding the unreachability of (x, y), and such clauses were shown to be helpful in Section 3.2.1. In Section 3.2.2, we determined all GR(k)-reachable points for $k \leq 6$. For $k = 7$, the difficulty of the problem prevented us from determining all GR(7)-reachable points. However, we were able to determine upper and lower reachability bounds for GR(7) walks of up to 267 steps.

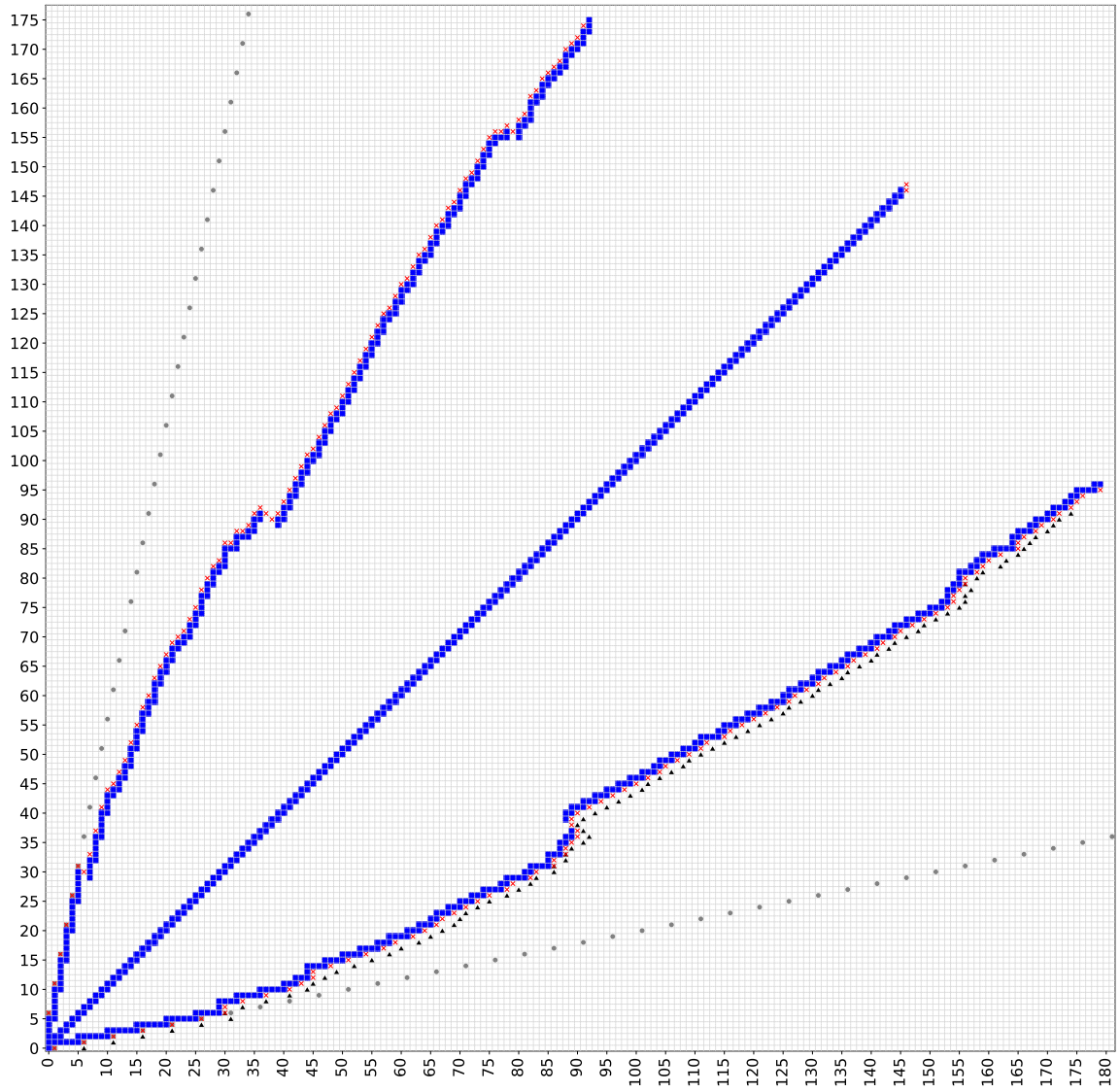


Fig. 4: A partial reachability diagram for $k = 7$. Blue squares are GR(7)-reachable points, and red crosses are GR(7)-unreachable points from the origin using a north first step. The black triangles show the reflected upper unreachability bound, and gray circles show the extremal bounds from Section 3.1.4.

A diagram showing the upper and lower bounds we found is provided in Figure 4. These bounds were computed incrementally starting from the origin and using no symmetry breaking except assuming the first step is north. For the upper bound, when a point (x, y) was found to be reachable the point to solve was updated to $(x, y + 1)$ and the process restarted. Conversely, if the point (x, y) was found to be unreachable the point to solve was updated to $(x + 1, y - 1)$. For the lower bound, when a point (x, y) was found to be reachable the next point to solve was set to $(x + 1, y)$, and if the point (x, y) was found to be unreachable the next point to solve was set to $(x - 1, y + 1)$.

In order to determine the upper and lower GR(7)-reachability bounds, we must solve both satisfiable and unsatisfiable instances. Furthermore, it is not known in advance which instances are satisfiable and which are unsatisfiable. A priori, it is not clear whether to use the CNF or KNF encoding, so we tried both CNF and KNF, as well as a third "hybrid" mode of Cardinality-CaDiCaL. The hybrid mode switches between solving with clauses and klauses, spending approximately half the time doing traditional Boolean constraint propagation (i.e., operating on clauses) and the other half using special cardinality-based propagation (i.e., operating on klauses). For the purposes of benchmarking, we solved all satisfiable and unsatisfiable reachability instances on the upper and lower boundary corresponding to GR(7) walks with $n \leq 180$ points. Each instance was solved 15 times, using 15 random seeds for each instance, and the results are given in Table 5. The constraint-removal heuristic was not used for benchmarking, but it was used to determine the points in Figure 4. In twelve instances the satisfying assignment produced an invalid GR(k) walk due to the constraint removal heuristic. In each of these cases the solver was rerun and a valid GR(k) walk was found on the next run of the solver.

The results show that KNF performed better on satisfiable instances and CNF performed better on unsatisfiable instances. However, the satisfiable instances were unknown in advance, so CNF was a better choice overall, given that KNF performed poorly on unsatisfiable instances and CNF outperformed the hybrid mode on both satisfiable and unsatisfiable instances. Thus, the reachability of the upper and lower

Table 5: Performance of the CNF (CaDiCaL), KNF (Cardinality-CaDiCaL with `ccdclMode=0`), and Hybrid (Cardinality-CaDiCaL with `ccdclMode=1`) modes for computing all reachable (SAT) and unreachable (UNSAT) boundary points across instances with $n \leq 180$. Times are reported as the total solve times (in seconds), with each instance solved 15 times (each time with a different random seed), and the median solve time used in the total.

Type	CNF	KNF	Hybrid
SAT	2601.6	2261.6	8151.3
UNSAT	4268.5	35 295.1	7063.0
Total	6870.1	37 556.7	15 214.2

bounds in Figure 4 was computed using the CNF encoding. Overall, it took 375.05 CPU days to determine the reachability of the upper and lower boundaries in Figure 4. The most difficult single instance that was successfully solved was the point $(176, 94)$. The solver took 63.5 days and used up to 5.5 GiB of memory to determine $(176, 94)$ was GR(7)-unreachable.

Given the shape of the GR(6)-reachability diagram in Figure 3c, we suspect there are GR(7)-unreachable points on and around the midline $y = x$ taking significantly fewer than $a(7)$ steps to reach. We made an effort to find such unreachable points by using CaDiCaL to determine the GR(7)-reachability of points along the line $y = x + 1$. Without using parallelization, the last point we were able to successfully determine the reachability of was $(138, 139)$. A GR(7) walk to $(138, 139)$ was found in 10.4 hours using the CNF encoding. The CNF encoding was able to solve instances for larger n than the KNF encoding; perhaps an indication that the instances are becoming "closer" to unsatisfiable in the sense that fewer satisfying assignments are present. Using the KNF encoding, the solver was unable to determine the reachability of points with $n > 270$ and on the line $y = x + 1$ using a week of compute time.

We incorporated parallelization in order to solve midline reachability instances with $n \geq 280$. We split the SAT instances into subinstances by enumerating all points

on lines of the form $y = sn - x$, where s is chosen to be a simple slope such as $\frac{1}{2}$. Each subinstance fixes one point on the line to true and the remaining points on the line to false. Using this approach, the instance with $n = 294$ corresponding to the final point $(146, 147)$ was determined to be GR(7)-unreachable (assuming an initial north step). This computation required a two-stage splitting process. The instance was first split by selecting all possible ways of selecting a point (x_0, y_0) on the line $y = \frac{1}{3}n - x$, and a point (x_1, y_1) on the line $y = \frac{2}{3}n - x$. Subinstances leading to trivially unsatisfiable instances were disregarded, such as when (x_0, y_0) was GR(7)-unreachable, or when (x_1, y_1) was unreachable from (x_0, y_0). This resulted in 1752 subinstances, corresponding to all simultaneously feasible choices of (x_0, y_0) and (x_1, y_1). Each of the 1752 subinstances ran for 7 days on the Nibi cluster, and afterwards 213 subinstances remained unresolved. The unsolved instances were split a second time by selecting all points (x_2, y_2) on the line $y = \frac{1}{2}n - x$ that were GR(7)-reachable from both (x_0, y_0) and (x_1, y_1) simultaneously. The second split produced 4554 subinstances, all of which were determined to be unsatisfiable in under 7 days. The total running time across all subinstances was about 18.0 CPU years.

The same two-stage procedure was used to determine that the instance with $n = 293$ corresponding to the final point $(146, 146)$ on the line $y = x$ is also GR(7)-unreachable. In this case, the first split produced 1753 subinstances, of which 131 remained unresolved after 7 days. These unresolved instances were then split a second time, producing 2786 subinstances, all of which were determined to be unsatisfiable in under 7 days. The total running time across all subinstances was about 7.12 CPU years.

Note the final GR(6)-reachable point on the line $y = x + 1$ is $(30, 31)$, and north–east walks to this point contain $n = 62$ points. The maximal GR(6) walk contains $n = 97$ points, so $a(6)$ is about 56% larger than the number of points on the longest GR(6) walk ending on the line $y = x + 1$. If this was also the case for $k = 7$, we would expect $a(7)$ to be around 457.

3.2.4 Searching for long GR(7) walks

The difficulty of the SAT instances in the Gerver–Ramsey collinearity problem with $k = 7$ prevented us from determining the exact value of $a(7)$. The SAT instances asserting the existence of a GR(7) walk with n points were feasible to solve without parallelization up to around $n = 300$. In order to go farther, we tried several strategies of incorporating parallelization.

The simplest parallelization strategy is simply to run multiple independent copies of the solver on the same instance, with the only change being the random seed passed to Cardinality-CaDiCaL. Setting different random seeds prevents the solver from making the same choices each time, permitting different parts of the search space to be explored. For the GR(7) instance with $n = 305$ points, we used 150 random seeds and ran each instance of the solver for three days. Twenty-six of the 150 solver instances found 305-point GR(7) walks, and all GR(7) walks found were distinct.

Ultimately, we had better results employing the cube-and-conquer parallelization strategy described in Section 2.1.7 and creating cubes with the lookahead solver MARCH. The cube-and-conquer paradigm is typically used on unsatisfiable SAT instances. However, in our case we are aiming to find long GR(7) walks, and therefore are looking to find satisfying assignments for instances for as large n as possible. Thus, even if MARCH does its job of partitioning the SAT instance into subinstances that are roughly of equal difficulty, there is no guarantee that this partition will be useful for finding satisfying assignments. For example, MARCH could yield subinstances for which all but one are unsatisfiable—this would limit the benefit of using cube-and-conquer if the goal is to find a satisfying assignment. We generated 150 cubes using MARCH's default parameters and none of the subinstances were found to be satisfiable after 3 days. Examining the cubes produced, most of the literals in the cubes corresponded to points that were close to the lower boundary and consequently most subinstances focused on searching for GR(7) walks close to the lower boundary—not optimal for finding long GR(7) walks.

By modifying MARCH to branch on variables corresponding to points around $(70, 70)$,

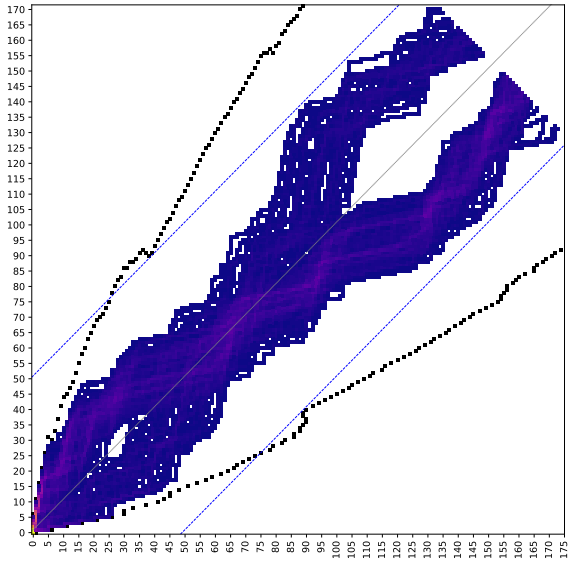


Fig. 5: Heatmap of points visited by 125 distinct 305-point walks found using parallelization. Also included in the plot is the midline $y = x + 1$ as well as the lines $y = x + 1 \pm 50$.

we computed two other sets of 150 cubes each. Altogether, including the GR(7) walks found by the random seed approach and a fourth set of cubes described below, we found 125 distinct 305-point GR(7) walks (discarding two satisfying assignments that were invalid due to the constraint removal heuristic). A heatmap of the points visited by these walks is provided in Figure 5. Streamlining unit clauses were included in these instances to enforce that the path did not stray more than 25 points diagonally from the line $y = x + 1$. This streamlining was added after preliminary experimentation found twenty GR(7) walks with $n \geq 310$ points and all these walks did not stray more than 25 points from $y = x + 1$.

A fourth set of cubes ultimately led to the longest GR(7) walk we found. They were produced using a SAT instance not containing the streamlining unit clauses and with some boundary points mistakenly encoded incorrectly. Some cubes became trivially unsatisfiable once the instance was corrected and the streamlining unit clauses were added, leaving 133 useful cubes. Despite this, these cubes ultimately resulted in the longest GR(7) walks we found. A summary of the GR(7) walks found with this set of cubes is provided in Table 6. The two distinct solutions for $n = 323$ are visually

Table 6: Summary of the GR(7) walks found using the best performing set of cubes with the streamlining technique. There were 133 non-trivial cubes and each cube was used to produce a KNF instance on which Cardinality-CaDiCaL was run for 72 hours. Times are in seconds.

n	Solutions	Min	Max	Median
323	2	43762.2	59735.7	51748.9
317	7	7962.2	88252.3	54885.6
311	19	2100.1	90337.3	53017.2
305	38	334.3	104144.6	51792.3

shown in Figure 6. Counting the total time spent across all 133 cubes, these solutions took 346.0 CPU days to find.

An examination of the two 323-point GR(7) walks found using cube-and-conquer revealed that a 188-step component was shared between the two walks. Given this, we ran additional searches to see if the common subpath could be extended to GR(7) walks with more than 323 points. Ultimately, up to isomorphism we found ten 328-point GR(7) walks containing this subpath or its complement via an exhaustive search with CaDiCaL in 40,931 seconds. Using 6.6 CPU days on the Ryzen desktop computer, CaDiCaL was also able to show that no 329-point GR(7) walk exists containing the common subpath or its complement, even with 20 points removed from both ends of the common subpath.

Finding extensions of a given subpath was accomplished using an extension of our SAT encoding. In addition to the variables and constraints given in Section 3.1, we also add new variables r_i representing that the ith step of the walk is east (where $0 \leq i < m$). If both (x, y) and $(x+1, y)$ are on the path, the $(x+y)$th step is eastward. Similarly, if $(x + 1, y)$ is on the path and the $(x + y)$th step was east, the previous point was (x, y). Thus, we use the clauses

$$(v_{x,y} \wedge v_{x+1,y}) \rightarrow r_{x+y} \quad \text{and} \quad (v_{x+1,y} \wedge r_{x+y}) \rightarrow v_{x,y}$$

for all $x \geq 0$, $y \geq 0$, and $x + y < n - 1$. When r_i is false, this represents that the ith

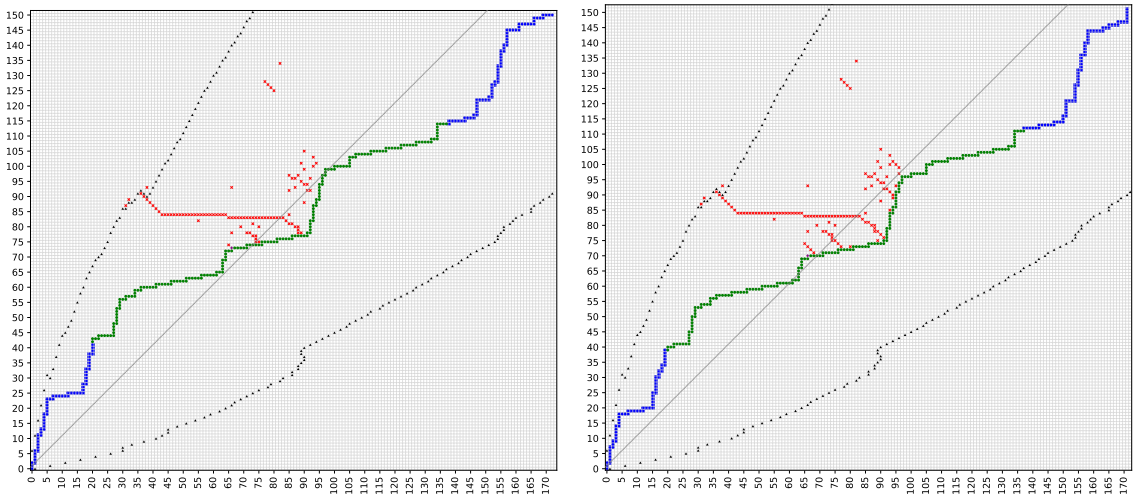


Fig. 6: Two 323-point GR(7) walks, with their associated cubes. The positive literal in the cube is shown as a purple plus marker, and the negative literals as red crosses. The upper and lower GR(7) unreachable boundary points are shown as black triangles. The path points shown as green circles are common to both GR(7) walks.

step was north. This case is handled similarly, using the clauses

$$(v_{x,y} \wedge v_{x,y+1}) \to \neg r_{x+y} \quad \text{and} \quad (v_{x,y+1} \wedge \neg r_{x+y}) \to v_{x,y}$$

for all $x \geq 0$, $y \geq 0$, and $x + y < n - 1$.

Say that $B := b_0 b_1 \dots b_{\ell-1}$ is a binary string (with `1` denoting a north step) representing the ℓ-step path that we want to extend. In other words, the steps in B must appear as a subpath in every GR(k) walk produced by a satisfying assignment. We also introduce the variables s_i representing that the subpath B starts immediately following the ith step (where $0 \leq i < n - \ell$). If the subpath B starts after the ith step, that means r_{i+j} is true exactly when $b_j = \texttt{0}$ for $0 \leq j < \ell$. Thus, we use the conjunction of clauses

$$\bigwedge_{\substack{0 \leq j < \ell \\ b_j = \texttt{0}}} (s_i \to r_{i+j}) \qquad \text{and} \qquad \bigwedge_{\substack{0 \leq j < \ell \\ b_j = \texttt{1}}} (s_i \to \neg r_{i+j}) \qquad \text{for all } 0 \leq i < n - \ell$$

to encode that the path B appears after the ith step when s_i is true. Then we can enforce that the path B appears somewhere in the GR(k) walk by the clause

$s_0 \vee s_1 \vee \cdots \vee s_{n-1-\ell}$, which says that the subpath B must start somewhere in the path.

This encoding determined that the 188-step common subpath in our two 323-point GR(7) walks can be extended to GR(7) walks with up to 328 points, but no more. One of the 328-point GR(7) walks we found is visually shown in Figure 7. The binary sequence of steps in the walk is

1100000100000100010000010001011111011111001000001000001001000001011011111000010000010001101110011011111011110111101111101111010110000100011011111011111011111010111110111110100001000001001000001000100111011111000110111110111111011111101111101111101100000111011111011111011100100000111000001000001000001000010000111110110111111000100.

By definition, there are necessarily no lines passing through 7 points on this walk, but there are 196 lines passing through 6 points on the walk and these lines are also drawn in Figure 7.

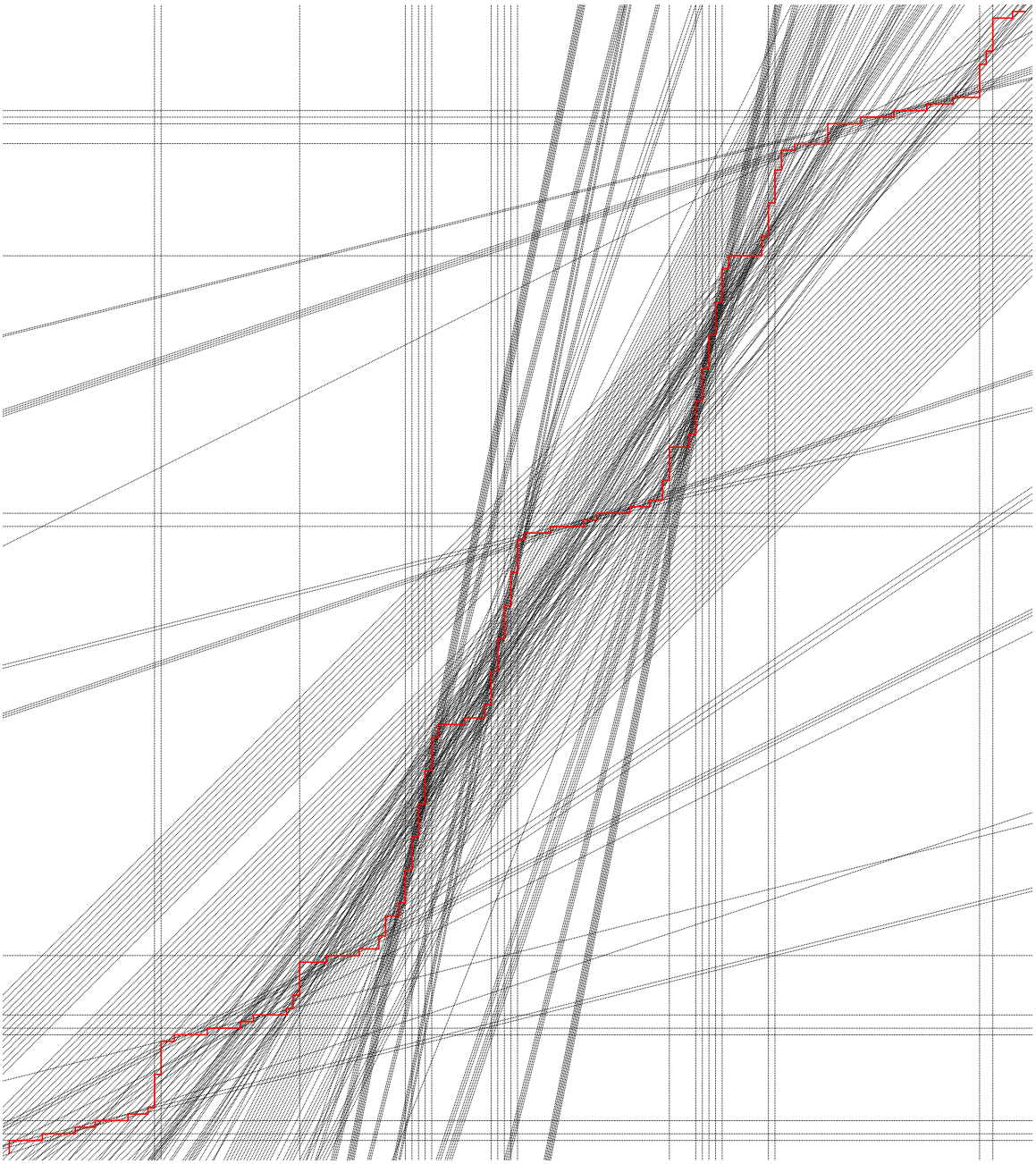


Fig. 7: A visual depiction of a 328-point GR(7) walk found by our approach, along with all 196 lines passing through 6 lattice points on the walk.

CHAPTER 4

Mutually Orthogonal Latin Squares

A *Latin square* of order n is an $n \times n$ array containing n symbols, where each symbol occurs exactly once in each row and exactly once in each column (see Figure 8a). Two Latin squares of the same order are called *orthogonal* if, when the two squares are overlaid, every ordered pair of symbols occurs exactly once (see Figure 8b and 8c). A k-MOLS(n) is a collection of k Latin squares of order n in which every pair of squares is orthogonal.

A *transversal* of a Latin square of order n is a set of n cells containing exactly one cell from each row, exactly one cell from each column, and exactly one occurrence of each symbol. For example, the five colour classes in Figure 8a are five disjoint transversals of the square.

Transversals are closely related to orthogonal mates. Euler observed that a Latin square has an orthogonal mate if and only if its cells can be partitioned into n disjoint transversals. To see this, suppose L has an orthogonal mate M. Fix a symbol s in M, and consider the n cells in which M contains s. Since M is a Latin square, these cells contain exactly one cell from each row and exactly one cell from each column. Since M is orthogonal to L, the entries of L in these cells must contain each symbol exactly once. Therefore, the cells containing s in M form a transversal of L. Repeating this for every symbol of M partitions the cells of L into n disjoint transversals.

Conversely, suppose the cells of L can be partitioned into n disjoint transversals. Construct a new square M by assigning a distinct symbol to each transversal. Since each transversal contains one cell from each row and column, each symbol appears exactly once in each row and column of M. Thus M is a Latin square. Since each

0	1	2	3	4
1	2	3	4	0
2	3	4	0	1
3	4	0	1	2
4	0	1	2	3

(a) Original square

0	1	4	2	3
2	3	0	1	4
1	4	2	3	0
3	0	1	4	2
4	2	3	0	1

(b) Orthogonal mate

00	11	24	32	43
12	23	30	41	04
21	34	42	03	10
33	40	01	14	22
44	02	13	20	31

(c) Overlay

Fig. 8: (a) A Latin square of order 5 decomposed into five non-overlapping transversals, with each transversal highlighted in a separate colour. (b) The orthogonal mate produced by assigning a distinct symbol to each transversal. (c) The overlay of the two squares, showing that every ordered pair of symbols appears exactly once.

transversal contains each symbol of L exactly once, overlaying L and M produces every ordered pair of symbols exactly once. Therefore M is an orthogonal mate of L. For example, assigning a distinct symbol to each of the five transversals in Figure 8a produces the orthogonal mate in Figure 8b.

4.1 The Euler–Parker Algorithm

Euler's transversal observation leads to a natural algorithm for finding orthogonal mates. Given a Latin square L of order n, instead of constructing an orthogonal mate cell by cell, one can search for a decomposition of L into n disjoint transversals. This is the basis of the Euler–Parker algorithm, named in part after Parker, who used Euler's method to compute orthogonal Latin squares of order 10 in 1959 [46]. Knuth estimates that the Euler–Parker approach reduced the search cost for finding order-10 orthogonal mates by a factor of 10^{12} [35].

The Euler–Parker algorithm has two stages. First, compute the transversals of L. Second, select n pairwise disjoint transversals from this set. If such a collection is

found, then these transversals partition the cells of L, and an orthogonal mate is constructed by assigning a distinct symbol to each transversal.

The Euler–Parker algorithm is very effective when one is given a Latin square and wants to determine whether it has an orthogonal mate. It is less directly applicable when the goal is to find orthogonal Latin squares satisfying additional structural constraints. In that setting, the difficulty is not only to construct a mate of a given square, but also to find a starting square whose mate has the desired form.

Many problems involving orthogonal Latin squares require the squares to satisfy additional structure. Examples include diagonal orthogonal Latin squares [57], doubly self-orthogonal Latin squares [40], orthogonal Latin squares with Latin subsquares [44], and self-orthogonal Latin squares with a symmetric orthogonal mate [18]. In these problems, there may not be an obvious way to choose a starting Latin square for the Euler–Parker algorithm, because most Latin squares with orthogonal mates may not satisfy the additional structure being sought.

This is where SAT solving is useful. Extra structural requirements can often be encoded directly as Boolean constraints. The SAT solver can then search only within the structured space of interest. However, a standard SAT solver does not automatically exploit specific domain knowledge about Latin squares, such as Euler's observation. It only sees the Boolean constraints in the encoding.

4.1.1 The hybrid SAT + Euler–Parker approach

A pure SAT approach can encode the fact that the symbols of one Latin square partition another Latin square into disjoint transversals. This gives a declarative form of the Euler–Parker approach: the SAT instance asserts that the required transversal decomposition exists. This encoding can work well in practice [15], but it does not force the solver to use the two-stage Euler–Parker strategy.

The goal of our hybrid method is to combine these two strengths. The SAT solver is used to search through the constrained space of Latin squares, while an external implementation of the Euler–Parker algorithm is used to test completed squares for orthogonal mates. This allows the search to use both conflict-driven SAT solving and

the transversal structure exploited by the Euler–Parker method.

In our approach, when the SAT solver finds a partial solution that represents a complete Latin square L, we call an implementation of the Euler–Parker algorithm to determine whether L has an orthogonal mate. If no mate exists, then a blocking clause is added to the SAT solver, forcing it to backtrack and preventing the same square from being considered again.

This differs from a pure SAT approach. In a pure SAT encoding, once the solver finds a completed Latin square L, it may begin searching for n disjoint transversals of L. However, the solver does not know that it should first enumerate all transversals and then search among them for a disjoint set. If the solver encounters a conflict while searching for transversals, it may backtrack away from L before the transversal search has been exhausted. Thus, even when L has an orthogonal mate, a pure SAT solver may fail to find it before moving on to another square.

The hybrid approach avoids this issue by treating the orthogonal-mate test as a separate structured computation. Whenever a completed square is found, the Euler–Parker algorithm is run to completion on that square. If the algorithm succeeds, then an orthogonal mate has been found. If it fails, then the square is blocked and the SAT solver continues its search.

4.2 Encoding 2-MOLS(n)

First, we describe the SAT encoding used to ensure an $n \times n$ matrix P is a Latin square. To represent the entries of P, we introduce variables $P_{i,j,k}$ for $0 \leq i, j, k < n$ denoting that cell (i, j) of P contains symbol k. To ensure that P is a Latin square, we impose *exactly-one* constraints requiring that each cell contains exactly one symbol and that each symbol appears exactly once in each row and column. Exactly-one constraints on n variables can be straightforwardly encoded into CNF using a single *at-least-one* clause of length n and $\binom{n}{2}$ *at-most-one* clauses of length 2, but Bright et al. [15] found that using an exactly-one CNF encoding based on the "totalizer" encoding [6] performed better in practice, and so this is the encoding we use in our

work. We denote a cardinality constraint saying that exactly x of the variables in a set S are true by $\sum_{s \in S} s = x$. For example, enforcing that cell (i, j) of P has exactly one symbol can be done with the exactly-one constraint $\sum_{s \in S} s = 1$, where $S = \{P_{i,j,k} \mid 0 \leq k < n\}$.

Orthogonality of two squares P and R can be encoded directly into the SAT instance, but a naive encoding uses $\Theta(n^6)$ clauses of length 4 [14, Eq. 4]. In practice, a better way of enforcing the orthogonality of P and R is to instead assert that for all symbols $0 \leq k < n$ the cells containing the symbol k in R form a transversal in P. In order to do this, we introduce a new square Q whose kth row represents the symbols of P contained in the cells having symbol k in R. This is done with the constraints

$$(R_{i,j,k} \wedge P_{i,j,l}) \rightarrow Q_{k,j,l} \qquad \text{for all } 0 \leq i, j, k, l < n.$$

Once Q has been introduced, asserting that the cells with symbol k in R form a transversal of P is done by asserting that each row of Q contains each symbol exactly once. The transversals formed in this way are necessarily disjoint, so the existence of Q ultimately implies that R is an orthogonal mate of P. In fact, Q itself will be a Latin square, and constraints enforcing this are additionally included in the encoding. Bright et al. [15] derive additional structure on the squares in terms of a composition operation. The composition is performed independently within each column and is defined by

$$(QR)[i, j] = Q[R[i, j], j].$$

Thus, the value $R[i, j]$ specifies which row of column j of Q is used. For example, if $R[1, 2] = 0$ and $Q[0, 2] = 2$, then $(QR)[1, 2] = 2$. They prove that when P, Q, and R are Latin squares the composition equation $QR = P$ is equivalent to (P, R) being an orthogonal pair and (P, Q) being a transversal representation pair.

4.3 Programmatic Euler–Parker

The orthogonality encoding described in Section 4.2 can be considered a "static" implementation of the Euler–Parker approach. However, a standard backtracking SAT

solver will never stop to compute the set of *all* transversals of a square and therefore (as described in Section 4.1.1) will not fully exploit the Euler–Parker approach. This motivated us to augment a SAT solver with a programmatic implementation of the Euler–Parker algorithm that can be called whenever the SAT solver finds a complete Latin square during the search.

The hybrid method still uses the SAT encoding from Section 4.2 in order to direct the solver towards orthogonal squares, but it also has an external Euler–Parker subroutine added in conjunction with the CNF encoding. Whenever the SAT solver reaches a complete Latin square, the programmatic Euler–Parker implementation is invoked to determine whether that square admits an orthogonal mate. If the square is determined to not have an orthogonal mate by Euler–Parker, the solver will immediately backtrack (which is earlier than it would have otherwise) and otherwise an orthogonal mate is produced. We now describe how we implemented the two stages of the Euler–Parker method using two calls to the library LIBEXACT. In the following, say L is a Latin square of order n and we want to compute all orthogonal mates of L.

4.3.1 Stage 1: Construct all transversals

A transversal of L is a set of n cells containing exactly one cell from each row, exactly one cell from each column, and with L containing each symbol exactly once amongst the transversal's cells. Constructing a transversal naturally reduces to solving a linear Diophantine system where a solution of the system provides a transversal t of L. Let $x_{i,j} \in \{0, 1\}$ be a variable that will be 1 when cell (i, j) appears in t. Since t contains exactly one cell from row i, we have $\sum_{j=0}^{n-1} x_{i,j} = 1$; since t contains exactly one cell from column j, we have $\sum_{i=0}^{n-1} x_{i,j} = 1$; and since t contains an occurrence of symbol k in L exactly once, we have $\sum_{L[i,j]=k} x_{i,j} = 1$.

Since there are n rows, n columns, and n symbols, this provides a linear Diophantine system with n^2 variables and $3n$ equations. Given L, our implementation symbolically forms the above system and uses LIBEXACT to find all integer 0–1 solutions of the system. Each solution is transformed into a transversal and the input to the second stage is the set of all transversals of L.

4.3.2 Stage 2: Construct n disjoint transversals

Given a set T of transversals of a Latin square L, constructing a set of n disjoint transversals also naturally reduces to solving a linear Diophantine system, where a solution of the system provides n disjoint transversals $T^* \subseteq T$. Let $x_t \in \{0, 1\}$ be a variable that will be 1 when transversal $t \in T$ is one of the n transversals in T^*. Since every cell (i, j) appears in exactly one transversal in T^* (because the transversals are disjoint and there are n of them), we have the equations

$$\sum_{t \in T, (i,j) \in t} x_t = 1 \qquad \text{for every cell } (i, j).$$

This is a linear system with $|T|$ variables and n^2 equations. For $n = 10$, we found that $|T|$ is approximately 825 on average, but it grows rapidly with n, reaching over 3 million for $n = 15$. Our implementation symbolically forms this system and uses LIBEXACT to solve it exhaustively. Each solution has exactly n variables with $x_t = 1$, providing n disjoint transversals. Denoting these n transversals as $t_0, \ldots, t_{n-1}$, we create an orthogonal mate to L by assigning symbol i to the cells of t_i for all $0 \leq i < n$.

4.3.3 Injecting Euler–Parker into the SAT solver

The hybrid aspect of the method comes from using IPASIR-UP to call LIBEXACT to perform the above two-stage process during SAT search. Using the callback routines described in Section 2.1.8, the interface tracks the assignment status of the variables encoding a Latin square. Once all variables encoding a square have been assigned, the interface takes the complete square (say L) and begins stage 1. If fewer than n transversals are found, then L cannot admit an orthogonal mate. In this case, the interface constructs a blocking clause from the literals encoding L and returns it to the SAT solver. This forces the solver to backtrack and prevents L from being considered again.

If at least n transversals are found, they are passed to stage 2, which attempts to select a disjoint set of n transversals. If this second stage fails, the complete square L is again blocked. If it succeeds, then an orthogonal mate has been found. If there happen

to be additional constraints on the orthogonal mate, they can be checked at this point. Alternatively, it may be possible to run the Euler–Parker algorithm in such a way that it will *always* construct orthogonal mates that satisfy the additional constraints. For example, in Section 4.4 we show how to do this for Myrvold's 2 MOLS(10) instances.

When the values in square P have been completely assigned to form a square L which has been determined to have no orthogonal mates, the blocking clause passed to the solver is

$$\bigvee_{L[i,j]=k} \neg P_{i,j,k}$$

which naively contains n^2 literals, since i and j both range over $\{0, \ldots, n-1\}$. In fact, it is sufficient to take i and j only over $\{0, \ldots, n-2\}$, since the upper-left $(n-1) \times (n-1)$ subsquare of L uniquely determines the final row and column. This shrinks the length of the blocking clause to $(n-1)^2$ literals, though this clause still only blocks the square L.

4.4 Specialization to Myrvold's 2-MOLS(10) Instances

The encoding described so far is for the general 2 MOLS(n) problem. Our experimentation of this general encoding as n grows reveals the augmented SAT solver dramatically outperforms a pure SAT solver (see Section 4.5). While this is a promising sign of the usefulness of the hybrid approach, the general 2 MOLS(n) problem can be solved without a SAT solver and so this is a somewhat artificial benchmark.

To highlight the utility of our hybrid approach, we use it to solve 2 MOLS(10) instances with structure that seems intractable to handle purely theoretically. The eight 2 MOLS(10) cases left unsolved by Myrvold (see Section 2.3) are suitable for this purpose, since she was unable to solve the eight cases by theoretical arguments. Currently, the cases have only been solved with the assistance of a SAT solver [15]. In order to enforce the additional structure considered by Myrvold, we add more constraints on the pair (P, Q) from Section 4.2. For example, there are colour

constraints, subsquare Ω compatibility constraints, and symmetry breaking constraints. There are also constraints stating that the pair (P, Q) is one of Myrvold's eight unsolved types. For example, (P, Q) being of type XX means that both P and Q contain four transversals of type p_1 and six transversals of type p_2. A transversal is of type p_i when it contains exactly i white entries in the last four columns, so the type constraints can be implemented by adding appropriate cardinality constraints into the SAT instance. Because these constraints are not an essential component of our new hybrid approach, we do not go into them in more detail and instead refer the reader to [15, Sec. 4].

The first stage of the Euler–Parker implementation does not need modification to work on the instances encoding Myrvold's problem. However, after the first stage, some transversals can be discarded because they cannot be part of a valid solution. For example, in the case of XX, we are looking to find 10 disjoint transversals of the first square: 4 of type p_1 and 6 of type p_2. So, any transversals of type p_3 found by stage 1 will be discarded before starting stage 2. Similarly, transversals that violate the Ω compatibility constraints will also be discarded. This filtering reduced the number of transversals in the average square found by the SAT solver from around 825 to around 140. Without loss of generality, there are two possibilities for the Ω subsquare (Ω_1 and Ω_2), and the SAT instance permits solutions from either possibility—so long as solutions only contain transversals all compatible with Ω_1 or all compatible with Ω_2. In order to enforce this in our hybrid approach, we run the second Euler–Parker stage twice, once only with transversals compatible with Ω_1, and once only with transversals compatible with Ω_2.

In the second stage of the Euler–Parker implementation we make use of Myrvold's constraints explicitly. For example, in case XX we specify that a valid solution requires exactly four transversals of type p_1 and six transversals of type p_2. Letting $T_i \subseteq T$ denote the set of transversals of type p_i, we form the equations $\sum_{t \in T_1} x_t = 4$ and $\sum_{t \in T_2} x_t = 6$ and pass them to LIBEXACT. There are also colour constraints that can be directly incorporated: a valid solution must possess exactly two dark entries in each of the first six columns. Let $D_i \subseteq T$ denote the transversals whose entry in the ith column is coloured dark. We form the six equations $\sum_{t \in D_i} x_t = 2$ for $0 \leq i < 6$

and pass them all to the linear system provided to LIBEXACT.

4.5 Results

In this section, we evaluate the performance of our hybrid SAT + Euler–Parker method. As a first experiment, we test its performance on the unrestricted 2 MOLS(n) problem (see Section 4.5.1). Our primary experiment is solving each of the eight 2 MOLS(10) cases that Myrvold left unsolved (see Section 4.5.2). All instances were solved using the SAT solver CADICAL version 3.0, and the Euler–Parker implementation used LIBEXACT version 1.0. Experiments were run on the Digital Research Alliance of Canada's Fir cluster, a high-performance computing cluster consisting of AMD EPYC processors, most of which run at 2.7 GHz. Our source code is available at https://github.com/aaronbarnoff/MOLS_EP.

4.5.1 Results for 2 MOLS(n) instances

As mentioned at the start of Chapter 4, the unrestricted 2 MOLS(n) problem has been solved by purely theoretical constructions, and therefore one does not need to use a SAT solver for this problem. Even still, it forms a useful experiment to demonstrate that providing SAT solvers with static Euler–Parker constraints (i.e., constraints asserting the symbols of one square decompose the other square into n disjoint transversals) may not be as effective as running an external Euler–Parker implementation.

Table 7 provides a detailed timing breakdown of our unrestricted 2 MOLS(n) results for $8 \leq n \leq 12$. The table shows that as n grows the hybrid SAT + Euler–Parker approach is orders of magnitude faster than a standard SAT solver provided with static Euler–Parker constraints. For example, for $n = 12$, the median time across 15 tests for the hybrid approach is at least 150,000 times faster than the median time for the pure SAT approach (for which the solver did not finish within four days).

These results conclusively demonstrate that off-the-shelf SAT solvers do not exploit the Euler–Parker approach to its full potential. As n increased, our experimentation

Table 7: Runtime breakdown for the 2 MOLS(n) experiments, showing median times in seconds for the pure SAT and the hybrid SAT + Euler–Parker approach across 15 random instances. The median number of Euler–Parker calls before an orthogonal mate was found is also provided. The timeout was four days.

Order n	8	9	10	11	12
Pure SAT	0.09	168.19	3486.03	39 383.54	timeout
Hybrid	0.26	0.42	0.78	1.20	2.23
# EP calls	160	31	2	1	1

showed that the static Euler–Parker constraints made the SAT solver find a square with an orthogonal mate quickly. For $n \geq 11$, in the median case the *very first square* the solver found was one having an orthogonal mate. Despite this, the pure SAT solver was unable to find its mate, evidently backtracking before finding n disjoint transversals in the square. Of course, the pure SAT solver is only provided static constraints telling it to find n disjoint transversals; it does not attempt to find *all* transversals and then from within these try to find n disjoint transversals. This two-stage methodology is exploited by the hybrid approach and explains why it can dramatically outperform the pure SAT approach.

These instances had CaDiCaL's `--shufflerandom` option enabled, which randomizes the initial variable scores and causes the solver to branch on different variables in each experiment. We noticed that otherwise the hybrid solver would always make the same decisions and find the same solutions every time, even when run using different random seeds. This seems to be an artifact of the fact that the hybrid solver is so fast it solves the instances before the impact of the random seed is felt. The hybrid solver also had CaDiCaL's `--preprocesslight` disabled because otherwise in some orders the solver would find a solution instantly as a result of the preprocessing, leading to the same decisions being made every time, even with variable shuffling enabled.

4.5.2 Results for Myrvold's 2-MOLS(10) instances

We now describe the results of our hybrid SAT + Euler–Parker approach on Myrvold's eight unresolved cases, and compare them with the pure SAT approach. For both approaches, we ran 15 instances of each of the eight cases for seven days, using a different random seed for each instance so that CaDiCaL would make different decisions each time. These results are summarized in Table 8, with a visual comparison shown in Figure 9.

Table 8: Comparison of the hybrid and pure-SAT approaches for each of the eight Myrvold pair types. Times are in seconds. Each pair type had 15 randomly seeded instances that ran for one week.

pair type	method	# solved	median	min	max
UU	Hybrid	15	5077.36	509.92	38 323.47
	Pure SAT	15	31 721.01	2706.00	366 291.48
SX	Hybrid	15	2882.27	499.79	22 652.56
	Pure SAT	14	59 447.47	255.08	timeout
UW	Hybrid	15	11 304.32	903.01	147 568.43
	Pure SAT	12	116 856.78	751.59	timeout
WW	Hybrid	15	35 638.79	1360.37	412 264.03
	Pure SAT	6	timeout	66 699.78	timeout
VX	Hybrid	15	20 789.28	484.36	62 917.17
	Pure SAT	13	174 215.01	46 195.26	timeout
UX	Hybrid	15	27 033.18	205.98	68 228.84
	Pure SAT	13	243 781.81	9314.35	timeout
WX	Hybrid	15	20 612.80	491.44	53 664.08
	Pure SAT	3	timeout	279 253.68	timeout
XX	Hybrid	15	5105.64	491.25	44 225.28
	Pure SAT	0	timeout	timeout	timeout

We found that the best performance was obtained by limiting how often a call to

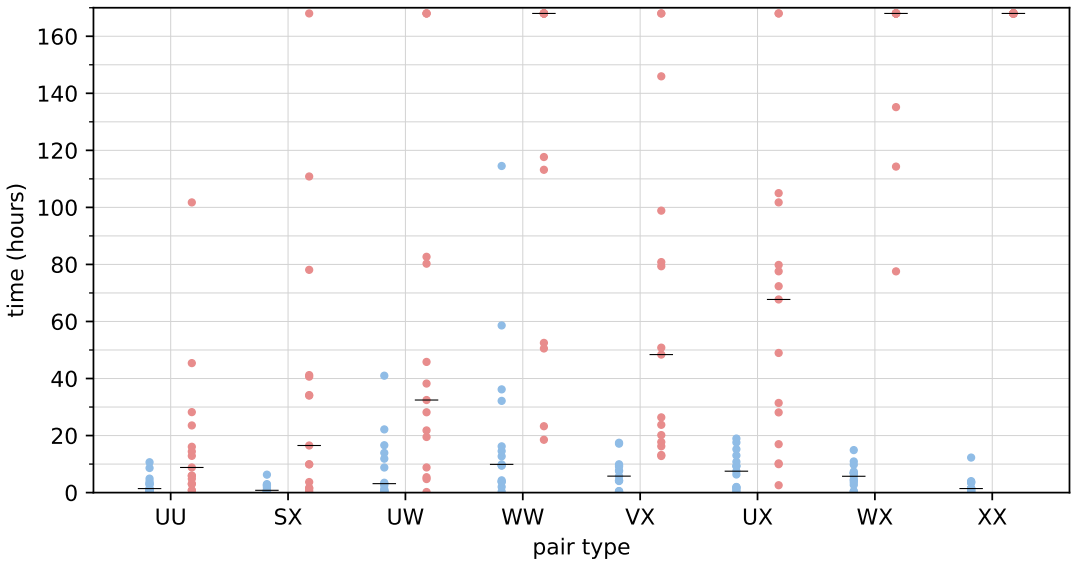


Fig. 9: Scatterplot comparing the hybrid approach (left) to the pure SAT approach (right), for the eight Myrvold pair types. The median time is indicated with a black bar.

Euler–Parker could be invoked. This was due to how quickly CaDiCaL produces completed squares, each of which makes a relatively expensive call to libexact in the first exact cover stage. When the call to Euler–Parker was unrestricted, it was called so often that the SAT solver had little opportunity to make progress of its own through conflict-driven search, as reflected in a low number of internal (non-programmatic) conflicts and restarts. The blocking clauses generated by Euler–Parker were added to the solver as forgettable clauses (meaning that the solver was allowed to discard them to keep memory usage low), but CaDiCaL only performs reductions after a certain number of non-programmatic conflicts. Since programmatic blocking clauses were being created so rapidly, memory usage increased steadily and typically exceeded 8 GiB after running for a few days.

To address this, we configured the solver so that the solver generates at least one non-programmatic conflict clause before calling Euler–Parker again. This threshold gave the SAT solver more time to progress through the search space between Euler–Parker invocations. This resulted in far fewer squares being passed to Euler–Parker

between clause database reductions, substantially reducing memory usage. The highest recorded memory usage by the hybrid solver was 0.99 GiB (after running for about 5 days). We also found that the threshold improved the performance of the solver in most cases, particularly UW (see Table 9). A breakdown of the median time spent in the SAT solver and in each stage of the Euler–Parker algorithm is shown in Table 10.

Table 9: Comparison of the hybrid method in solving the eight Myrvold cases with and without limiting the frequency of Euler–Parker (EP) calls. Time is median solve time in seconds across 15 randomly seeded instances.

pair type	UU	SX	UW	WW	VX	UX	WX	XX
Limited EP	5077.36	2882.27	11 304.32	35 638.79	20 789.28	27 033.18	20 612.80	5105.64
Unlimited EP	20 871.53	27 065.39	230 812.17	87 203.51	38 492.24	18 606.29	14 058.23	8672.98

Table 10: Runtime breakdown for the eight Myrvold cases, showing median times in seconds before a 2 MOLS(10) was found. The total runtime is divided into the SAT solving time and the time for the two stages of the Euler–Parker (EP) method. The median number of calls to Euler–Parker is also provided.

pair type	UU	SX	UW	WW	VX	UX	WX	XX
SAT solver	2036.67	960.89	6788.22	12 107.96	6847.76	8957.87	5366.88	1095.49
EP stage 1	2698.06	1753.94	3929.43	20 634.07	12 179.40	15 456.45	12 957.77	3274.35
EP stage 2	372.91	173.16	586.67	3116.00	1762.12	2618.85	2288.15	625.46
# EP calls	1 253 736	677 695	1 565 150	8 148 047	4 688 434	5 974 377	4 928 770	1 254 684

For these instances, we call Euler–Parker whenever the SAT solver finds a complete assignment of either the variables defining the square P or the square Q. When P has been determined, we run Euler–Parker to construct Q, and when Q has been determined, we run Euler–Parker to construct P. Without loss of generality, the first row of P can be taken to be one of three possibilities [15, Thm. 13], and the SAT encoding enforces the first row of P to be one of these three possibilities. In cases when Q has been determined, it is also possible to add constraints into the second LIBEXACT instance enforcing that one of the three possibilities appears in the

transversal decomposition of Q. However, it was more efficient to *not* provide this symmetry breaking constraint to LIBEXACT, thereby making the stage 2 instance less constrained. The downside is that it makes the LIBEXACT call less efficient. However, in general it results in fewer calls to Euler–Parker, as it allows the solver to find solutions that it otherwise would discard.

CHAPTER 5

Conclusion

In this thesis, we studied the use of satisfiability solving as a tool for solving finite combinatorial search problems. The two main case studies were the Gerver–Ramsey collinearity problem for north–east lattice walks and the construction of mutually orthogonal Latin squares. Although these problems arise from different areas of combinatorics, both can be formulated as searches for finite objects satisfying a collection of constraints.

5.1 Summary of Results

For the Gerver–Ramsey problem, we developed a SAT encoding for north–east lattice walks avoiding k collinear points. The encoding represents the points of the walk using Boolean variables, enforces the structure of a north–east path, and uses cardinality constraints to rule out k collinear points. We used this approach to enumerate GR(k) walks for small values of k, verify known values of $a(k)$ for $k \leq 6$, and produce proof certificates for the corresponding nonexistence results. We also applied SAT solving and parallelization techniques to the case $k = 7$, improving the best known lower bound by finding a GR(7) walk with 327 steps, and therefore showing that $a(7) \geq 328$.

For the mutually orthogonal Latin squares problem, we developed a hybrid method combining SAT solving with the Euler–Parker algorithm. Our hybrid approach uses the SAT solver to search through constrained Latin squares, and then calls an external exact-cover based implementation of the Euler–Parker algorithm whenever a completed square is found. This approach combines the general search capabilities of

SAT with the mathematical structure exploited by Euler's transversal decomposition method. This hybrid approach significantly outperformed a pure SAT encoding in both unconstrained 2-MOLS(n) and in a highly constrained variant of 2-MOLS(10) arising from Myrvold's work.

REFERENCES

[1] Erika Abraham. Building bridges between symbolic computation and satisfiability checking. In *Proceedings of the 2015 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC'15, page 1–6. ACM, 2015. doi:10.1145/2755996.2756636.

[2] Tanbir Ahmed, Lamina Zaman, and Curtis Bright. Symbolic sets for proving bounds on Rado numbers. In *SC^2 2025: 10th International Workshop on Satisfiability Checking and Symbolic Computation*, volume 4116 of *CEUR Workshop Proceedings*, pages 55–75, 2025. URL: https://ceur-ws.org/Vol-4116/paper139.pdf.

[3] Yameen Ajani and Curtis Bright. A hybrid SAT and lattice reduction approach for integer factorization. In *SC^2 2023: 8th International Workshop on Satisfiability Checking and Symbolic Computation*, volume 3455 of *CEUR Workshop Proceedings*, pages 39–43, 2023. URL: https://cs.uwaterloo.ca/~cbright/reports/sc2-fact-update.pdf.

[4] Yameen Ajani and Curtis Bright. SAT and lattice reduction for integer factorization. In *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation*, ISSAC '24, page 391–399. ACM, 2024. doi:10.1145/3666000.3669712.

[5] Nahiyan Alamgir, Saeed Nejati, and Curtis Bright. SHA-256 collision attack with programmatic SAT. In *SC^2 2024: 9th International Workshop on Satisfiability*

Checking and Symbolic Computation, volume 3717 of *CEUR Workshop Proceedings*, pages 91–110, 2024. URL: https://ceur-ws.org/Vol-3717/paper5.pdf.

[6] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-45193-8_8.

[7] Aaron Barnoff and Curtis Bright. North–east lattice paths avoiding k collinear points via satisfiability. *Advances in Applied Mathematics*, 179, 2026. Article 103112. doi:10.1016/j.aam.2026.103112.

[8] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

[9] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In A. Gurfinkel and V. Ganesh, editors, *Computer Aided Verification – CAV 2024*, volume 14681 of *Lecture Notes in Computer Science*, page 133–152, Cham, 2024. Springer. doi:10.1007/978-3-031-65627-9_7.

[10] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki, 2024.

[11] R. C. Bose and S. S. Shrikhande. On the falsity of Euler's conjecture about the non-existence of two orthogonal Latin squares of order $4t + 2$. *Proceedings of the National Academy of Sciences*, 45(5):734–737, May 1959. doi:10.1073/pnas.45.5.734.

[12] R. C. Bose, S. S. Shrikhande, and E. T. Parker. Further results on the construction of mutually orthogonal Latin squares and the falsity of Euler's conjecture. *Canadian Journal of Mathematics*, 12:189–203, 1960. doi:10.4153/CJM-1960-016-5.

[13] Curtis Bright, Jürgen Gerhard, Ilias Kotsireas, and Vijay Ganesh. Effective problem solving using SAT solvers. In J. Gerhard and I. Kotsireas, editors, *Maple in Mathematics Education and Research*, volume 1125 of *Communications in Computer and Information Science*, page 205–219, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-41258-6_15.

[14] Curtis Bright, Amadou Keita, and Brett Stevens. Orthogonal Latin squares of order 10 with two relations: A SAT investigation. *Discrete Mathematics, Algorithms and Applications*, November 2025. To appear. doi:10.1142/s1793830925501563.

[15] Curtis Bright, Amadou Keita, and Brett Stevens. Myrvold's results on orthogonal triples of 10×10 Latin squares: A SAT investigation. *The Electronic Journal of Combinatorics*, 33(1):P1.30, 2026. doi:10.37236/13960.

[16] Curtis Bright, Ilias Kotsireas, and Vijay Ganesh. When satisfiability solving meets symbolic computation. *Communications of the ACM*, 65(7):64–72, June 2022. doi:10.1145/3500921.

[17] Tom C. Brown. Advanced problem 5811. *The American Mathematical Monthly*, 78(7):798, 1971. doi:10.1080/00029890.1971.11992858.

[18] A. P. Burger, M. P. Kidd, and J. H. van Vuuren. Enumerasie van self-ortonogale Latynse vierkante met simmetriese ortogonale maats. *LitNet Akademies (Natuurwetenskappe)*, 9(2):1–24, April 2012. doi:10.10520/EJC125908.

[19] Sam Buss and Neil Thapen. DRAT proofs, propagation redundancy, and extended resolution. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, volume 11628 of *Lecture Notes in Computer Science*, pages 71–89, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-24258-9_5.

[20] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*, STOC '71, page 151–158. ACM Press, 1971. doi:10.1145/800157.805047.

[21] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. doi:10.1145/368273.368557.

[22] József Dénes and A. Donald Keedwell. *Latin Squares and their Applications*. North-Holland, 2 edition, 2015. doi:10.1016/C2014-0-03412-0.

[23] Michael W. Ecker. Problem 408. *Crux Mathematicorum*, 10:294–296, Dec 1979. URL: https://cms.math.ca/wp-content/uploads/crux-pdfs/Crux_v5n10_Dec.pdf.

[24] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005. doi:10.1007/11499107_5.

[25] Matthew England. SC-square: Overview to 2021. In *SC^2 2021: 6th International Workshop on Satisfiability Checking and Symbolic Computation*, volume 3273 of *CEUR Workshop Proceedings*, pages 1–6, 2021. URL: https://ceur-ws.org/Vol-3273/invited1.pdf.

[26] Leonhard Euler. Recherches sur une nouvelle espèce de quarrés magiques. *Verhandelingen uitgegeven door het zeeuwsch Genootschap der Wetenschappen te Vlissingen*, pages 85–239, 1782.

[27] Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: User Propagators for CDCL. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:13, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SAT.2023.8.

[28] Joseph L. Gerver. Long walks in the plane with few collinear points. *Pacific Journal of Mathematics*, 83(2):349–355, 1979. doi:10.2140/pjm.1979.83.349.

[29] Joseph Leonide Gerver and Lawrence Thom Ramsey. On certain sequences of lattice points. *Pacific Journal of Mathematics*, 83(2):357–363, 1979. doi:10.2140/pjm.1979.83.357.

[30] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In H.H. Hoos and D.G. Mitchell, editors, *Theory and Applications of Satisfiability Testing – SAT 2004*, volume 3542 of *Lecture Notes in Computer Science*, page 345–359. Springer Berlin Heidelberg, 2005. doi:10.1007/11527695_26.

[31] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. *Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer*, page 228–245. Springer International Publishing, 2016. doi:10.1007/978-3-319-40970-2_15.

[32] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437, Cham, 2018. Springer. doi:10.1007/978-3-319-94144-8_26.

[33] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012. doi:10.1007/978-3-642-31365-3_28.

[34] Petteri Kaski and Olli Pottonen. libexact user's guide, version 1.0. 2008. HIIT Technical Reports 2008–1, Helsinki Institute for Information Technology HIIT. URL: `https://pottonen.kapsi.fi/hiit-tr-2008-1.pdf`.

[35] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.* Addison-Wesley, Upper Saddle River, NJ, 2011.

[36] Samuel Korsky. North-east lattice paths with few collinear vertices, 2026. `arXiv:2607.02832`, `doi:10.48550/arXiv.2607.02832`.

[37] Evelyn Lamb. Two-hundred-terabyte maths proof is largest ever. *Nature*, 534(7605):17–18, May 2016. `doi:10.1038/nature.2016.19990`.

[38] Zhengyu Li, Curtis Bright, and Vijay Ganesh. An SC-square approach to the minimum Kochen–Specker problem. In *SC^2 2022: 7th International Workshop on Satisfiability Checking and Symbolic Computation*, volume 3458 of *CEUR Workshop Proceedings*, page 55–66, 2022. URL: `https://ceur-ws.org/Vol-3458/paper6.pdf`.

[39] Zhengyu Li, Curtis Bright, and Vijay Ganesh. A SAT solver + computer algebra attack on the minimum Kochen–Specker problem. In Kate Larson, editor, *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 1898–1906. International Joint Conferences on Artificial Intelligence Organization, 8 2024. Main Track. `doi:10.24963/ijcai.2024/210`.

[40] Runming Lu, Sheng Liu, and Jian Zhang. Searching for doubly self-orthogonal Latin squares. In *Principles and Practice of Constraint Programming – CP 2011*, volume 6876 of *Lecture Notes in Computer Science*, page 538–545. Springer Berlin Heidelberg, 2011. `doi:10.1007/978-3-642-23786-7_41`.

[41] Harris F MacNeish. Euler squares. *Annals of Mathematics*, 23(3):221–227, 1922. `doi:10.2307/1967920`.

[42] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh,

editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2 edition, 2021. doi:10.3233/FAIA200987.

[43] P. L. Montgomery. Collinear points on a monotonic polygon. *The American Mathematical Monthly*, 79(10):1143–1144, 1972. doi:10.1080/00029890.1972.11993206.

[44] Wendy Myrvold. Negative results for orthogonal triples of Latin squares of order 10. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 29:95–105, 1999. URL: https://combinatorialpress.com/jcmcc-articles/volume-029/negative-results-for-orthogonal-triples-of-Latin-squares-of-order-10/.

[45] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. *Propagation = Lazy Clause Generation*, page 544–558. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-74970-7_39.

[46] E. T. Parker. A computer search for Latin squares orthogonal to Latin squares of order ten, abstract 564-71. *American Mathematical Society Notices*, 6:798, 1959.

[47] E. T. Parker. Orthogonal Latin squares. *Proceedings of the National Academy of Sciences*, 45(6):859–862, June 1959. doi:10.1073/pnas.45.6.859.

[48] Julius Petersen. Les 36 officiers. *Annuaire des Mathématiciens*, pages 413–427, 1902.

[49] Joseph E. Reeves. *Cardinality Constraints in Boolean Satisfiability Solving*. PhD thesis, Carnegie Mellon University, 2025.

[50] Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From clauses to klauses. In A. Gurfinkel and V. Ganesh, editors, *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*,

page 110–132, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-65627-9_6.

[51] Jeff Shallit. On-line encyclopedia of integer sequences entry A231255. https://oeis.org/A231255, 2013.

[52] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In P. van Beek, editor, *Principles and Practice of Constraint Programming – CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, page 827–831. Springer Berlin Heidelberg, 2005. doi:10.1007/11564751_73.

[53] Gaston Tarry. Le problème des 36 officiers. *Association Française pour l'Avancement des Sciences: Compte Rendu de la 29^{me} session, Paris 1900*, 2:170–203, 1901.

[54] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, 1983.

[55] P Wernicke. Das problem der 36 offiziere. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 19:264–267, 1910.

[56] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-09284-3_31.

[57] Oleg Zaikin, Eduard Vatutin, and Curtis Bright. Enumerating extended self-orthogonal diagonal Latin squares of order up to 10. *Journal of Integer Sequences*, 28, 2025. Article 28.7.4. URL: https://cs.uwaterloo.ca/journals/JIS/VOL28/Zaikin/zaikin4.html.

[58] Edward Zulkoski, Vijay Ganesh, and Krzysztof Czarnecki. MathCheck: A math assistant via a combination of computer algebra systems and SAT solvers. In

Automated Deduction - CADE-25, volume 9195 of *Lecture Notes in Computer Science*, page 607–622. Springer International Publishing, 2015. doi:10.1007/978-3-319-21401-6_41.

VITA AUCTORIS

NAME: Aaron Barnoff

EDUCATION:

University of Windsor, B.Sc. Chemistry, Windsor, ON, 2017

University of Windsor, B.Sc. Computer Science, Windsor, ON, 2024

University of Windsor, M.Sc. Computer Science, Windsor, ON, 2026