North–East Lattice Paths Avoiding k Collinear Points via Satisfiability

Aaron Barnoff
School of Computer Science
University of Windsor
Canada
barnoffa@uwindsor.ca

Curtis Bright
School of Computer Science
University of Waterloo
Canada
cbright@uwaterloo.ca

November 27, 2025

Abstract

We investigate the Gerver–Ramsey collinearity problem of determining the maximum number of points in a north–east lattice path without k collinear points. Using a satisfiability solver, up to isomorphism we enumerate all north–east lattice paths avoiding k collinear points for $k \leq 6$. We also find a north–east lattice path avoiding $k = 7$ collinear points with 327 steps, improving on the previous best length of 260 steps found by Shallit.

1 Introduction

In 1971, Tom C. Brown [6] asked the following: must every sufficiently long lattice path in the plane with steps in $\{(1, 0), (0, 1)\}$ always contain k collinear points, regardless of the value of $k \geq 1$? The following year, P. L. Montgomery [18] published a solution showing the answer to be yes: every sufficiently long north–east lattice path must contain k collinear points, regardless of the choice of k.

However, Montgomery did not provide a constructive bound on how long the walk must be before k collinear points were guaranteed. In 1979, Gerver and Ramsey [11] provided such a bound. They showed every north–east lattice path in the plane of length at least

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

must contain k collinear points. The Gerver–Ramsey bound, while explicit, is extremely loose. For example, the bound guarantees that every north–east walk of length at least 2^{131073}

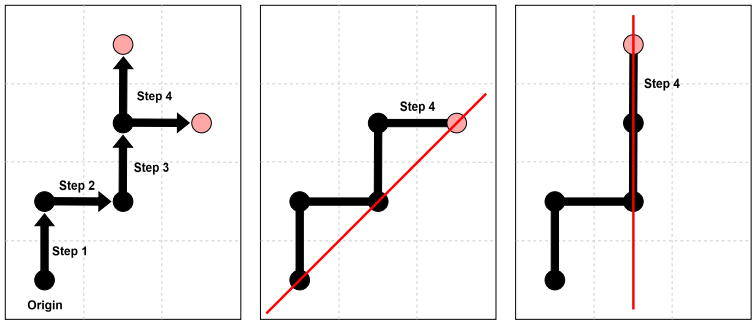


Figure 1: A visual representation of the longest north–east walk avoiding $k = 3$ collinear points. Walking two steps in the same direction introduces three collinear points, so the longest walk avoiding three collinear points alternates directions.

contains $k = 3$ collinear points, although in fact every walk with just four steps contains three collinear points (see Figure 1). In a separate paper published at the same time as Gerver and Ramsey's bound, Gerver [10] showed that there exists a north–east walk of length greater than

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

avoiding k collinear points. For example, for $k = 3$, this bound says there exists a walk with more than $(32 \cdot 2^{-5})^{1/18} = 1$ step avoiding three collinear points. Although this is a super-polynomial bound, it is quite loose for small values of k: it does not guarantee the existence of a 3-step walk avoiding k collinear points until $k = 30$.

The large gap between (1) and (2) means that precisely how long north–east lattice paths can be while avoiding k collinear points is unknown. In this paper, we study the problem of determining this length exactly for small values of k—a problem posed in 1979 by A. Meir [9]. Let $a(k)$ denote the smallest integer such that all north–east lattice paths of length $a(k)$ contain k collinear points, so that $a(k) - 1$ is the length of the longest north–east lattice path avoiding k collinear points. Meir noted that $a(3) = 4$ [9], and in 2013, J. Shallit [21] computationally determined $a(4) = 9$, $a(5) = 29$, and $a(6) = 97$. He also established the lower bound $a(7) \geq 261$ by finding a north–east lattice path of length 260 without 7 collinear points.

We call a north–east lattice path without k collinear points a *GR(k) walk* in honour of Gerver and Ramsey, and we call a point *GR(k)-reachable* if it is reachable from the origin by a GR(k) walk. In addition, if the GR(k) walk has length $a(k) - 1$ (i.e., has $a(k) - 1$ steps and therefore contains $a(k)$ points) we call the GR(k) walk *maximal*. Although the results of Montgomery and Gerver–Ramsey imply a maximal GR(k) walk exists for every k, there may exist multiple maximal GR(k) walks for given k. Up to isomorphism, we find there are two distinct maximal GR(4) and GR(6) walks and a single maximal GR(5) walk (see Section 3.2). The unique maximal GR(5) walk is visually depicted in Figure 2.

The problem of finding maximal GR(k) walks is difficult because the search space grows exponentially in the length of the walk: each m-step walk chooses north or east at each step,

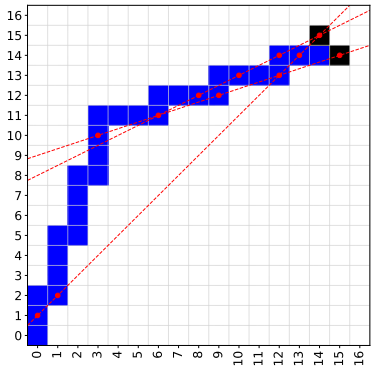


Figure 2: The unique longest GR(5) walk up to isomorphism; any additional step (black) creates five collinear points.

yielding 2^m possibilities. Establishing $a(k) = m$ requires showing both that every m-step walk contains k collinear points (thereby showing $a(k) \leq m$) and that some $(m-1)$-step GR(k) walk exists (thereby showing $a(k) \geq m$). Without loss of generality, to prove $a(k) \leq m$ only walks starting at the origin $(0, 0)$ and ending on the line $y = m - x$ need to be considered.

In this paper, we improve the lower bound on $a(7)$ by finding a GR(7) walk with 327 steps. We also enumerate all GR(k) walks for $k \leq 6$, and as a consequence confirm Shallit's computation of $a(k)$ for $k \leq 6$. Our searches and enumerations for GR(k) walks are performed using a satisfiability (SAT) solver. SAT solvers are exceptional general-purpose search tools [5] and have been surprisingly effective at solving problems in discrete geometry [15, 24, 25], finite geometry [3], infinite graph theory [23], and various puzzles in discrete mathematics [4]. In fact, certain combinatorial problems with enormous search spaces (like the Boolean Pythagorean triples problem [13]) have *only* been solved using a SAT solver, despite at first glance having nothing to do with Boolean logic. SAT solvers also have several other advantages over searching with custom-written code: from a correctness perspective, they provide *proof certificates* when the object being searched for doesn't exist. For example, when the SAT solver determines that there exists no m-step GR(k) walk, it also provides a certificate that can be certified by an independent proof verifier, a relatively simple piece of software. Consequently, only the proof verifier needs to be trusted, not the SAT solver itself. The SAT encoding also needs to be trusted, but it tends to be simpler to write a SAT encoding than it is to write optimized search code. We describe our SAT encoding for the Gerver–Ramsey collinearity problem in Section 2.

The primary contribution of this paper is a SAT-based method of finding long GR(k) walks and an experimental study the Gerver–Ramsey problem for $k \leq 7$. In particular, we enumerate all GR(k) walks up to isomorphism for $k \leq 6$. In the process, we find all maximal GR(k) walks for $k \leq 6$, accompanied by proof certificates that no longer GR(k) walks exist (see Section 3.2). We also determine the north-most and east-most GR(7)-reachable points using up to 260 steps (see Section 3.3) and find a GR(7) walk of length 327 (see Section 3.4), improving on the previously longest known GR(7) walk of length 260 [21].

2 Satisfiability Solving

The Boolean Satisfiability problem (SAT) asks whether a Boolean formula admits an assignment of truth values to its variables making the formula evaluate to true. It was the first problem proven to be NP-complete [8], and it remains a cornerstone of computational complexity theory. Over the last several decades, an active research community has developed increasingly efficient SAT solvers. Modern SAT solvers require the input formula to be specified in a format known as conjunctive normal form (CNF) defined in terms of literals and clauses. A *literal* is a Boolean variable p or its negation $\neg p$. A *clause* is a disjunction of literals. A formula is in *CNF* if it is a conjunction of clauses. For example, $(p \vee q) \wedge (p) \wedge (\neg p \vee q)$ is in CNF and contains three clauses, the second of which is a *unit clause* (consists of a single literal). We may also use the implication connective to express clauses with the understanding that $p \to q$ is shorthand for $\neg p \vee q$. A *satisfying assignment* of a formula is a true/false assignment to the variables of the formula such that the whole formula evaluates to true under that truth assignment (i.e., every clause evaluates to true under the truth assignment).

To apply a SAT solver to a search problem, the problem must be encoded as a CNF formula in such a way that the search problem has a solution if and only if the CNF formula has a satisfying assignment (i.e., the formula is *satisfiable*). Moreover, it should be possible to take a satisfying assignment of the formula and translate it into a solution of the search problem. Conversely, if the SAT solver determines the CNF formula has no satisfying assignment (i.e., the formula is *unsatisfiable*) this implies the search problem has no solution.

We describe our encoding of the Gerver–Ramsey collinearity problem into conjunctive normal form in Section 2.1, and describe an encoding of the problem into a related format called "at-least-k conjunctive normal form" in Section 2.2. We also found that using some clauses encoding the reachability of points was useful—although not strictly required, they improved the solving times in practice (see Section 2.3). Additionally, we found some of the constraints in our encoding were redundant in practice and removing them made the solver more efficient (see Section 2.4). Finally, we describe our process of parallelization using a technique known as cube-and-conquer in Section 2.5. Cube-and-conquer was necessary in order to solve the hardest SAT instances in a reasonable amount of time.

2.1 SAT encoding

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.

2.1.1 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 a 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.

2.1.2 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 [22] and the totalizer encoding [1]. Moreover, other formats like at-least-k conjunctive normal form [19] have native support for cardinality constraints (see Section 2.2).

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 and d are not larger than n/k, since if $r > n/k$ or $d > n/k$ then there will necessarily be less than k points in $\{0, \ldots, n-1\}^2$ on a line with slope $s = r/d$. 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 $6/\pi^2$ as $n \to \infty$, so there are asymptotically $\frac{6}{\pi^2}(n/k)^2$ slopes to consider.

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 is divisible by 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 with this line is unnecessary, since the constraint $\sum_{y=sx+b} v_{x,y} < k$ will always be satisfied. Proposition 1 below provides an additional restriction on the line slopes, showing certain non-collinearity constraints to be unnecessary.

Proposition 1. *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 1, 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.

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.

2.1.3 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).

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

2.2 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 [20], called *at-least-k conjunctive normal form* (KNF), augments CNF with constraints of the form $l_1 + \cdots + l_s \geq k$ where $l_1, \ldots, l_s$ are literals. Such a constraint is known as a *klause* and is satisfied by an assignment if at least k of the literals $l_1, \ldots, l_s$ 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 2.1.2 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. [20] provide a KNF solver based on the SAT solver CaDiCaL [2] called Cardinality-CDCL.[1] 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-CDCL performs 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.

2.3 Encoding the unreachability of points

Section 2.1.4 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

$$v_{x_0,y_0} \rightarrow \neg v_{x_0+x,y_0+y} \qquad \text{for all } (x_0, y_0) \in \mathbb{N}^2 \text{ with } x_0 + y_0 < n - x - y \tag{3}$$

[1] Code available at https://github.com/jreeves3/Cardinality-CDCL/.

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 2.1.3. 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 (3) 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 (3) when both (x, y) and (y, x) were determined to be unreachable from the origin in GR(k) walks with a north first step.

2.4 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 2.1.2 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 and $n \geq 150$, we removed all constraints corresponding to lines containing 16 or fewer points in the relevant region and with only two exceptions every

satisfying assignment found by the solver produced a valid GR(k) walk, despite this process removing the majority of the non-collinearity constraints (e.g., for $n = 300$, about 94% of non-collinearity constraints were removed). This heuristic also produced a speedup in the solver's efficiency (see Section 3.1.3).

2.5 Parallelization

As the number of steps in the path increases, the SAT instances tend to become more difficult. The largest instances we solve are so difficult that solving them using a single processor would take an infeasible amount of time. Thus, in order to make progress it is necessary to exploit parallelization and have multiple processors working on solving the SAT instance in parallel.

One of the most successful parallelization techniques in SAT solving is known as cube-and-conquer [14]. This technique aims to divide the search space into disjoint subproblems of roughly balanced difficulty. If this can be achieved, multiple processors can solve subproblems independently, providing a speedup proportional to the number of processors available. The method uses what is known as a lookahead solver to determine how to split the SAT instance by "branching" on a variable in the instance—setting the variable to true in one subproblem and false in another subproblem. A lookahead solver spends a significant amount of time determining which variable is best to split on in order to split the problem into two subproblems of roughly equal difficulty.

We use Heule's lookahead solver MARCH in our work [12]. After a variable is selected to branch on, MARCH generates a subproblem in which the variable is true and a subproblem in which the variable is false, and then applies Boolean constraint propagation (i.e., derives consequences of fixing the variable in each subproblem). The process then repeats recursively until a set number of cubes have been created or the subproblems have been determined to be so easy to solve that splitting them further is no longer necessary.

A *cube* is a conjunction of literals, e.g., $x_1 \wedge \neg x_2 \wedge x_3$. The above process of splitting can be viewed as generating a collection of cubes that partition the search space, with each cube defining a single subproblem. Each processor is provided the original SAT instance along with one or more cubes to solve. For each cube, the SAT solver assumes the literals in the cube are each true by adding each literal in the cube as a unit clause.

3 Results

We now discuss our experimental results.[2] We start with a description of the benchmarking we did in order to determine the effectiveness of the encodings from Section 2 (see Section 3.1). 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). Unless otherwise mentioned, experiments in Sections 3.1, 3.2, and 3.3 were run on the Digital Research Alliance

[2]Our code is available at https://github.com/aaronbarnoff/Collinear.

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 3.4 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.

3.1 Benchmarking

Because modern SAT solvers use a number of heuristics (e.g., to decide what variable to branch on when solving) their solving times on the same instance tends 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.

3.1.1 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 [16], 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 [2], and the KNF solver was Cardinality-CDCL [20] (based on CaDiCaL).

Four satisfiable benchmarks were chosen and four unsatisfiable benchmarks were chosen (in each case, one benchmark had $k = 6$, and 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-CDCL 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. [20]. 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.

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

3.1.2 Unreachable points encoding: Performance

The next set of benchmarks examines the performance of the unreachable point encoding described in Section 2.3. 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 3.3 we describe the computations performed to determine the reachability of points. As described in Section 2.3, 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 (3).

We used the same eight benchmarks from Section 3.1.1, 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-CDCL 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.

3.1.3 Constraint-removal heuristic: Performance

We now examine the performance of the constraint-removal heuristic described in Section 2.3. 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) paths with $n = 98$ points. For $n = 97$, the solver found paths containing k collinear points, indicating that some of the ignored constraints were not

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

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 able to solve our previous benchmarks correctly—the satisfiable instances valid GR(k) paths were found and the unsatisfiable instances were determined to have no GR(k) paths. 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-CDCL to at most 344 MiB on the satisfiable ones (1.7× reduction).

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

We now 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 2.1 along with the unreachable point encoding of Section 2.3 (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) points 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 find all GR(k) walks ending at the point $(x, m - x)$. Once all GR(k) walks with m steps were known they were

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	50.9
7	240	—	SAT	361.3	65.4
7	261	—	SAT	2397.4	1703.3
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

filtered up to isomorphism using the equivalence operations described in Section 2.1.3.

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 binary string of length m, with `0` representing a north step and `1` representing an east step. The normal form of a GR(k) walk is the walk whose binary string is the lexicographically least 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 [21]. 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.[3] 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 [7], a standard format for unsatisfiability proofs in modern SAT solving. A DRAT proof consists of a list of clause additions and deletions. Additions are logical consequences of the clauses in the original formula in conjunction with previously derived clauses. Deletions discard clauses when they are deemed to not be useful (in order to keep the memory footprint of

[3]The certificates are available at https://doi.org/10.5281/zenodo.17645678.

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

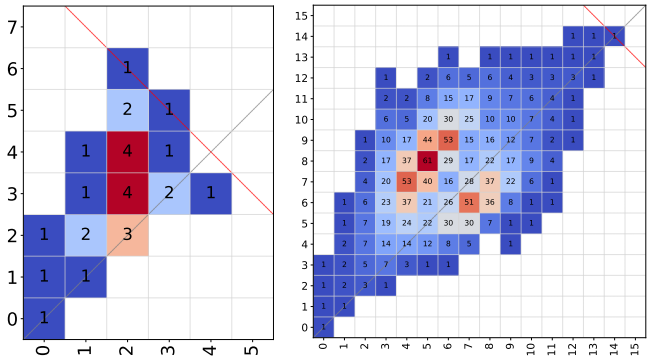
the solver manageable). The last added clause in the DRAT proof is the empty clause. The empty clause being a logical consequence of the original formula proves the formula to be unsatisfiable, since no truth assignments satisfy the empty clause. The proofs were validated with the proof verifier DRAT-trim [26]. 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 steps for $3 \leq k \leq 6$, which were computed on the Ryzen desktop computer. 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 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.

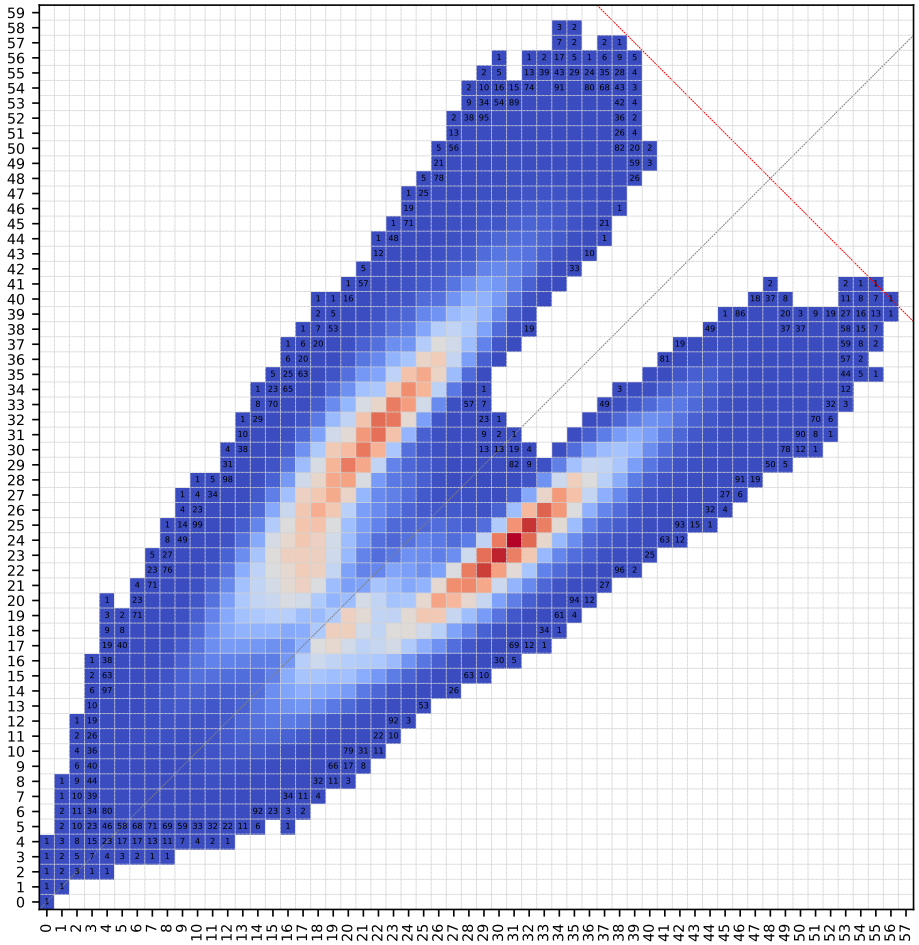
3.3 Reachability bounds for GR(7) walks

As explained in Section 2.3, 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 3.1.2. In Section 3.2, we determined all GR(k)-reachable points for $k \leq 6$. For $k = 7$, the difficulty of the problem prevented us determining all GR(7)-reachable points. However, we were able to determine upper and lower reachability bounds for GR(7) walks of up to 260 steps.

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 for 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



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



(c) $k = 6$

Figure 3: Exhaustive enumeration of GR(k) walks for $k = 4$ to $k = 6$ in normal form. Color intensity indicates number of distinct walks reaching each point, with red indicating the most 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 $y = (a(k) - 1) - x$.

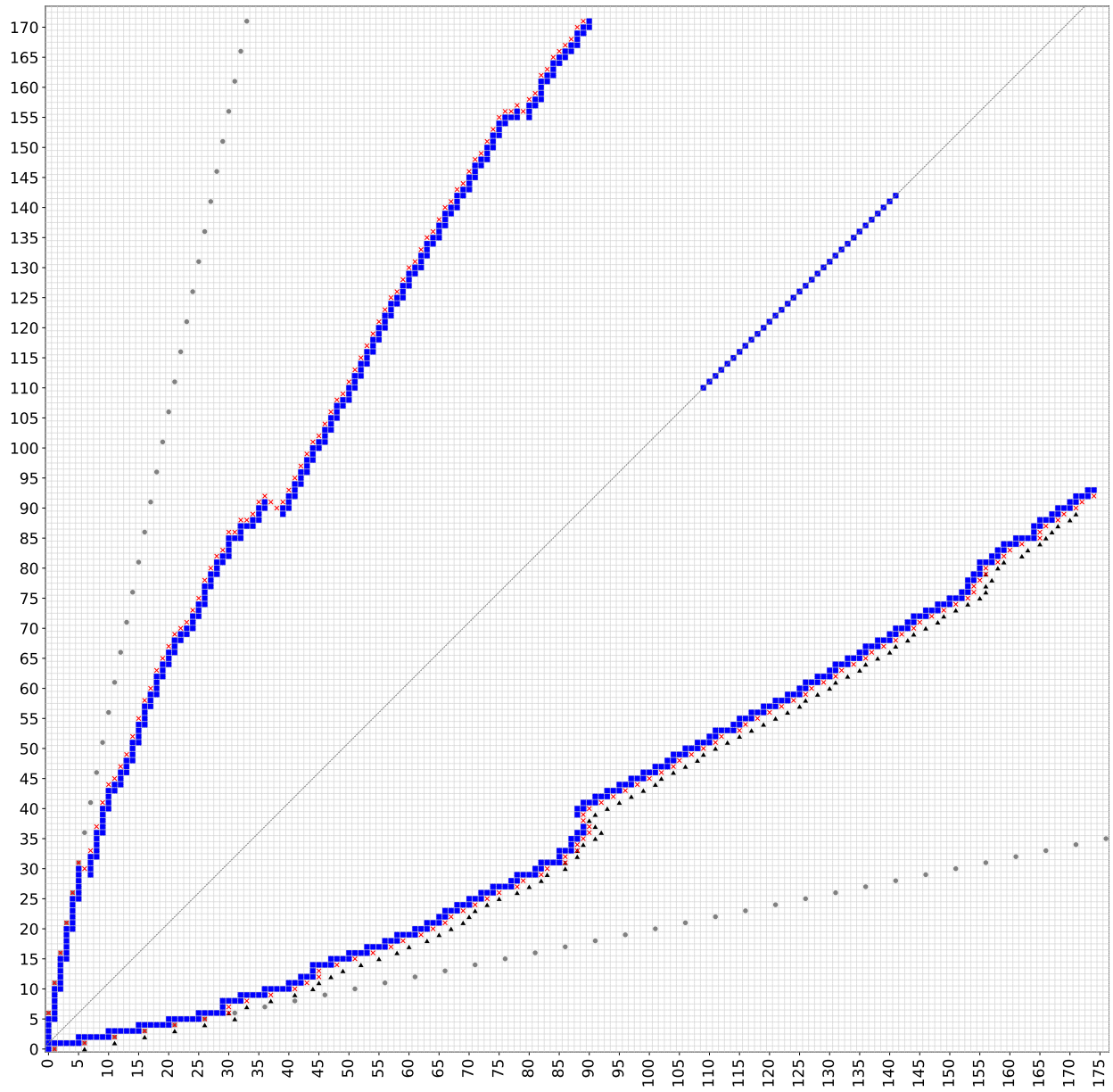


Figure 4: A partial reachability diagram for $k = 7$. Blue squares are GR(7)-reachable points, and red crosses are GR(7)-unreachable points. The black triangles show the reflected upper unreachability bound, and gray circles show the extremal bounds from Section 2.1.4.

Table 5: Performance of the CNF (CaDiCaL), KNF (Cardinality-CDCL with `ccdclMode=0`), and Hybrid (Cardinality-CDCL 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

$(x+1, y-1)$. 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-CDCL. 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.

The results show that KNF performed better on satisfiable instances and CNF performed better on unsatisfiable instances. However, given the satisfiable instances are unknown in advance, CNF was a better choice overall, as 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 bounds in Figure 4 was computed using the CNF encoding. Overall, it took 55.9 CPU days to determine the reachability of the upper and lower boundaries for instances with $n \leq 261$.

Given the shape of the GR(6)-reachability diagram in Figure 3c, we suspect that 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 $y = x + 1$ for $n \leq 300$. 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. Interestingly, the CNF encoding was able to solve instances for larger n; 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 trying all possible ways of fixing a variable corresponding to a point on the line $y = n/2 - x - 1$ to true. In this way the instance with $n = 284$ was split into 55 subinstances and a GR(7) walk was found to $(141, 142)$ in 30.7 hours of wall clock time using the Nibi cluster.

3.4 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-CDCL. 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 2.5 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 $(70, 70)$, we computed two other sets of 150 cubes each. Altogether, including the GR(7) walks found by the random seed parallelization approach and a fourth set of cubes described below, we found 125 distinct 305-point GR(7) walks (a heatmap of the points visited by these walks is

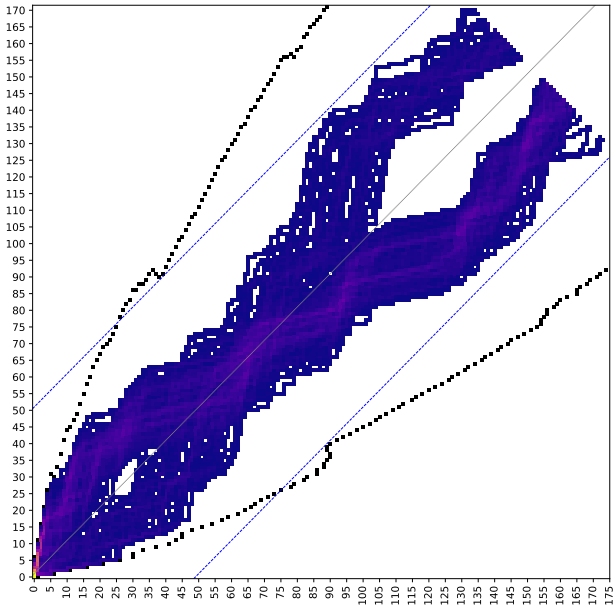


Figure 5: Heatmap of points visited by 125 distinct 305-point walks found using parallelization. Also included in the plot is the midline $y = x + 1$ as well as the lines $y = x + 1 \pm 50$.

provided in Figure 5).[4] 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 shown in Figure 6.

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

[4]There were 127 satisfying assignments, but two were invalid due to the constraint-removal heuristic.

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-CDCL 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

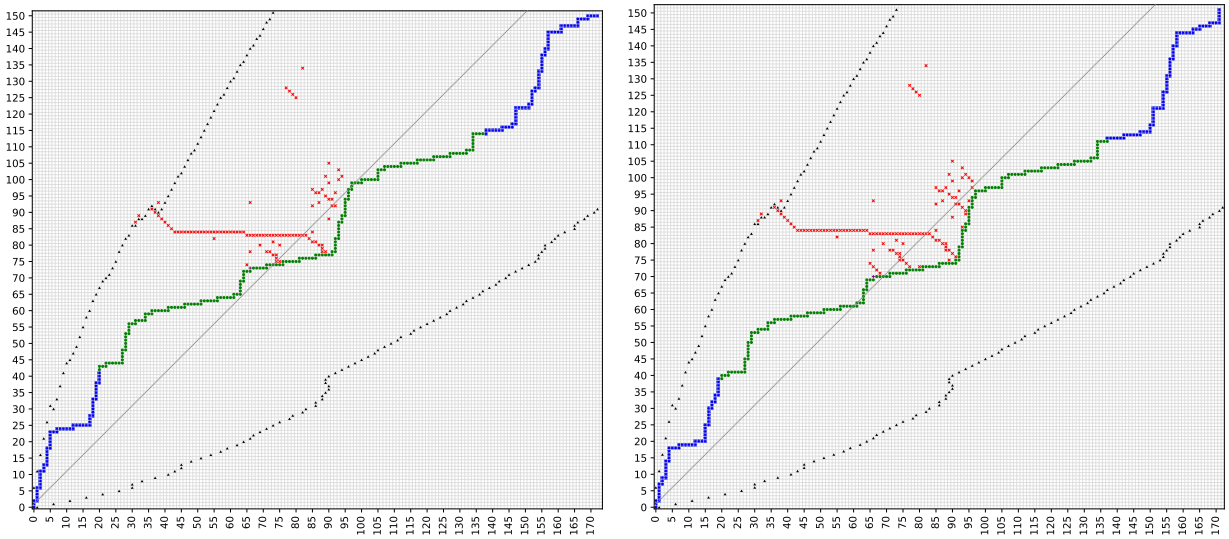


Figure 6: Two 323-point GR(7) walks, with their associated cubes. The positive literal in the cube is shown as a purple plus marker, and the negative literals as red crosses. The upper and lower GR(7) unreachable boundary points are shown as black triangles. The path points shown as green circles are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

encoding. In addition to the variables and constraints given in Section 2.1, we also add new variables r_i representing that the ith step of the walk is east (where $1 \le i < n$). If both $(x-1, y)$ and (x, y) are on the path, the $(x+y)$th step is eastward. Similarly, if (x, y) is on the path and the $(x+y)$th step was east, the previous point was $(x-1, y)$. Thus, we use the clauses

$$(v_{x-1,y} \wedge v_{x,y}) \to r_{x+y} \quad \text{and} \quad (v_{x,y} \wedge r_{x+y}) \to v_{x-1,y} \qquad \text{for all } x > 0,\ y \ge 0, \text{ and } x + y < n.$$

When r_i is false, this represents that the ith step was north. This case is handled similarly, using the clauses

$$(v_{x,y-1} \wedge v_{x,y}) \to \neg r_{x+y} \quad \text{and} \quad (v_{x,y} \wedge \neg r_{x+y}) \to v_{x,y-1} \qquad \text{for all } x \ge 0,\ y > 0, \text{ and } x + y < n.$$

Say that $B \coloneqq b_0 b_1 \dots b_{\ell-1} \in \{\mathtt{N}, \mathtt{E}\}^\ell$ is a string 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 $0 \le i < n - \ell$). If the subpath B starts after the ith step, that means r_{i+j} is true exactly when $b_j = \mathtt{E}$ for $0 \le j < \ell$. Thus, we use the conjunction of clauses

$$\bigwedge_{\substack{0 \le j < \ell \\ b_j = \mathtt{E}}} (s_i \to r_{i+j}) \qquad \text{and} \qquad \bigwedge_{\substack{0 \le j < \ell \\ b_j = \mathtt{N}}} (s_i \to \neg r_{i+j}) \qquad \text{for all } 0 \le i < n - \ell$$

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 $s_0 \vee s_1 \vee \dots \vee s_{n-1-\ell}$, 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. 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.

4 Conclusion

In this paper we devised a satisfiability-based approach for studying the Gerver–Ramsey collinearity problem on north–east lattice paths. As a result of our work we enumerated all maximal GR(k) walks for $k \le 6$ and made progress on the problem for $k = 7$, improving the longest known GR(7) walk from 260 steps to 327 steps. Although we were unsuccessful in finding a maximal GR(7) walk, we hope that the introduction of SAT solving on this problem leads to more progress and ultimately the determination of the value of $a(7)$.

In addition to determining values of $a(k)$ for larger k, there are a number of related problems that may be of interest. One variant would be to generalize the allowed steps in

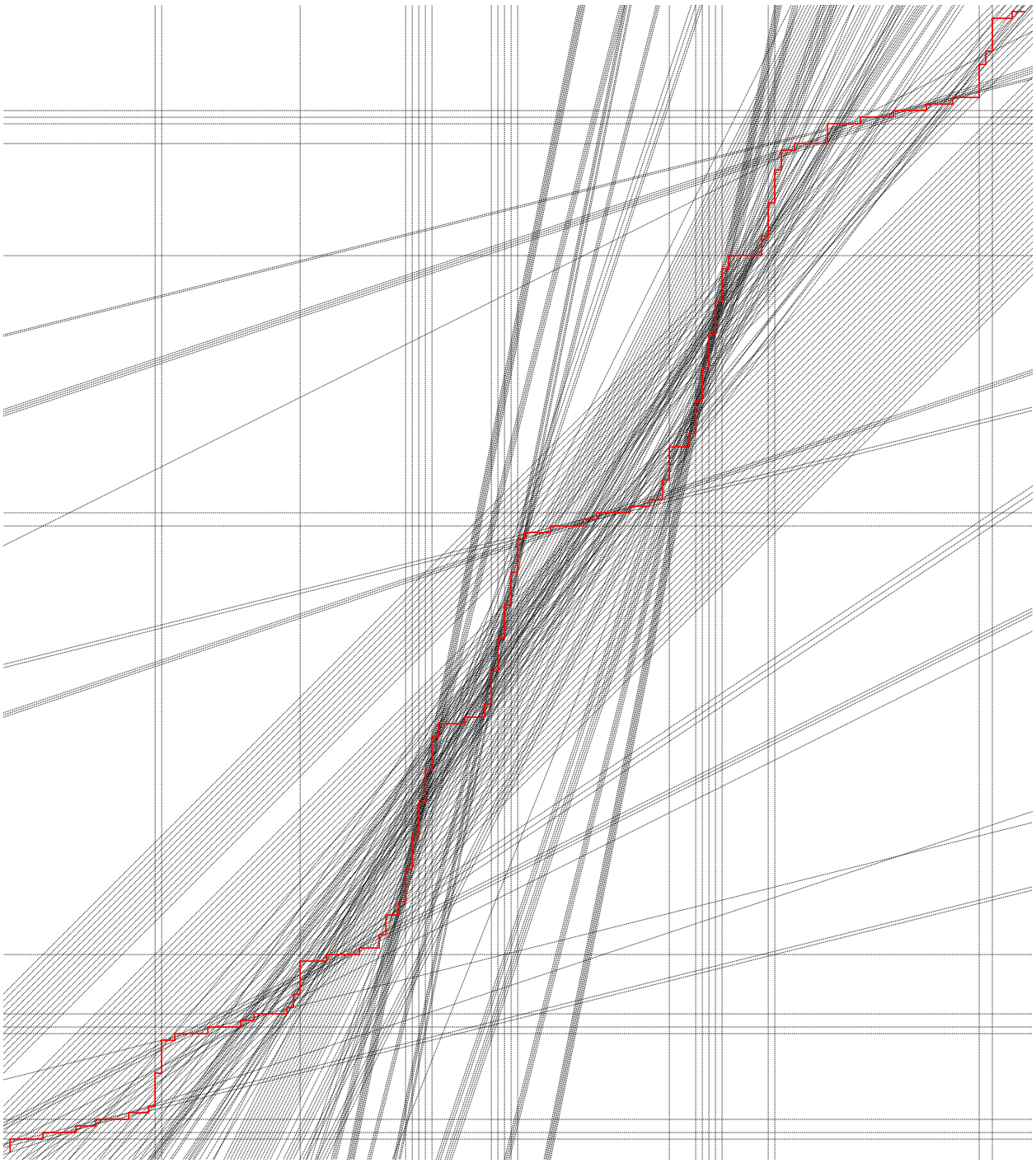


Figure 7: A visual depiction of a 328-point GR(7) walk found by our approach, along with all 196 lines passing through 6 lattice points on the walk.

the walk. In this paper we have always assumed a step set of $\{(1,0),(0,1)\}$, but Gerver and Ramsey show that regardless of the step set $S \subset \mathbb{Z}^2$ and the value of k, sufficiently long S-walks must always contain k collinear points.

Another interesting variant would be to consider a three-dimensional variant of the problem. Say $b(k)$ denotes the number of points in the longest lattice path with steps in $\{(1,0,0),(0,1,0),(0,0,1)\}$ avoiding k collinear points. In contrast with the two-dimensional case, $b(k)$ may be infinite. Gerver and Ramsey note that $b(3) = 9$, but they also prove that $b(5^{11}+1) = \infty$ by construction of an infinite walk $W \subset \mathbb{N}^3$ having at most 5^{11} collinear points. Moreover, they conjecture that W actually has at most three collinear points, which would imply $b(k) = \infty$ for all $k \geq 4$.

Recently, Lidbetter [17] determined their conjecture to be false by finding six collinear points in W. He also showed that W does not contain 189 collinear points, and as a consequence proves $b(k) = \infty$ for all $k \geq 189$. Even though Gerver and Ramsey's conjecture that W avoids four collinear points was wrong, it may still be the case that $b(4) = \infty$ because another infinite walk may avoid four collinear points. The 41st step of W is the first creating four collinear points, so $b(4) \geq 41$. We are not aware of a better bound on $b(4)$, and perhaps a SAT approach could be used to improve this bound or even determine $b(4)$, if it happened to be finite.

Acknowledgements

We thank Joesph Reeves for answering a question about configuring Cardinality-CDCL and for providing a KNF to CNF converter used in our work.

References

[1] O. Bailleux and Y. Boufkhad, Efficient CNF Encoding of Boolean Cardinality Constraints, in: F. Rossi (ed.), *Principles and Practice of Constraint Programming – CP 2003*, Springer Berlin Heidelberg, Berlin, Heidelberg, volume 2833 of *Lecture Notes in Computer Science*, 2003 pp. 108–122, doi:10.1007/978-3-540-45193-8_8.

[2] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks and F. Pollitt, CaDiCaL 2.0, in: A. Gurfinkel and V. Ganesh (eds.), *Computer Aided Verification – CAV 2024*, Springer, Cham, volume 14681 of *Lecture Notes in Computer Science*, 2024 p. 133–152, doi:10.1007/978-3-031-65627-9_7.

[3] C. Bright, K. K. H. Cheung, B. Stevens, I. Kotsireas and V. Ganesh, A SAT-based resolution of Lam's problem, volume 35, May 2021 p. 3669–3676, doi:10.1609/aaai.v35i5.16483.

[4] C. Bright, J. Gerhard, I. Kotsireas and V. Ganesh, Effective problem solving using SAT solvers, in: J. Gerhard and I. Kotsireas (eds.), *Maple in Mathematics Education and*

Research, Springer International Publishing, Cham, volume 1125 of *Communications in Computer and Information Science*, 2020 p. 205–219, doi:10.1007/978-3-030-41258-6_15.

[5] C. Bright, I. Kotsireas and V. Ganesh, When satisfiability solving meets symbolic computation, *Communications of the ACM* **65** (2022), 64–72, doi:10.1145/3500921.

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

[7] S. Buss and N. Thapen, DRAT proofs, propagation redundancy, and extended resolution, in: M. Janota and I. Lynce (eds.), *Theory and Applications of Satisfiability Testing – SAT 2019*, Springer International Publishing, Cham, volume 11628 of *Lecture Notes in Computer Science*, 2019 pp. 71–89, doi:10.1007/978-3-030-24258-9_5.

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

[9] M. W. Ecker, Problem 408, *Crux Mathematicorum* **10** (1979), 294–296, https://cms.math.ca/wp-content/uploads/crux-pdfs/Crux_v5n10_Dec.pdf.

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

[11] J. L. Gerver and L. T. Ramsey, On certain sequences of lattice points, *Pacific Journal of Mathematics* **83** (1979), 357–363, doi:10.2140/pjm.1979.83.357.

[12] M. Heule, M. Dufour, J. van Zwieten and H. van Maaren, March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver, in: H. Hoos and D. Mitchell (eds.), *Theory and Applications of Satisfiability Testing – SAT 2004*, Springer Berlin Heidelberg, volume 3542 of *Lecture Notes in Computer Science*, 2005 p. 345–359, doi:10.1007/11527695_26.

[13] M. J. H. Heule, O. Kullmann and V. W. Marek, Solving very hard problems: Cube-and-conquer, a hybrid SAT solving method, in: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, International Joint Conferences on Artificial Intelligence Organization, IJCAI-2017, August 2017 p. 4864–4868, doi:10.24963/ijcai.2017/683.

[14] M. J. H. Heule, O. Kullmann, S. Wieringa and A. Biere, Cube and conquer: Guiding CDCL SAT solvers by lookaheads, in: K. Eder, J. Lourenço and O. Shehory (eds.), *Hardware and Software: Verification and Testing*, Springer Berlin Heidelberg, Berlin, Heidelberg, volume 7261 of *Lecture Notes in Computer Science*, 2012 pp. 50–65, doi:10.1007/978-3-642-34188-5_8.

[15] M. J. H. Heule and M. Scheucher, Happy ending: An empty hexagon in every set of 30 points, in: B. Finkbeiner and L. Kovács (eds.), *30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer Nature Switzerland, Cham, volume 14570 of *Lecture Notes in Computer Science*, 2024 p. 61–80, doi:10.1007/978-3-031-57246-3_5.

[16] A. Ignatiev, A. Morgado and J. Marques-Silva, PySAT: A Python toolkit for prototyping with SAT oracles, in: O. Beyersdorff and C. M. Wintersteiger (eds.), *Theory and Applications of Satisfiability Testing – SAT 2018*, Springer, Cham, volume 10929 of *Lecture Notes in Computer Science*, 2018 pp. 428–437, doi:10.1007/978-3-319-94144-8_26.

[17] T. F. Lidbetter, Improved bound for the Gerver-Ramsey collinearity problem, *Discrete Mathematics* **347** (2024), 113718, doi:10.1016/j.disc.2023.113718.

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

[19] J. E. Reeves, *Cardinality Constraints in Boolean Satisfiability Solving*, Ph.D. thesis, Carnegie Mellon University, 2025.

[20] J. E. Reeves, M. J. H. Heule and R. E. Bryant, From clauses to klauses, in: A. Gurfinkel and V. Ganesh (eds.), *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I*, Springer-Verlag, Berlin, Heidelberg, volume 14681 of *Lecture Notes in Computer Science*, 2024 p. 110–132, doi:10.1007/978-3-031-65627-9_6.

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

[22] C. Sinz, Towards an optimal CNF encoding of boolean cardinality constraints, in: P. van Beek (ed.), *Principles and Practice of Constraint Programming - CP 2005*, Springer Berlin Heidelberg, volume 3709 of *Lecture Notes in Computer Science*, 2005 p. 827–831, doi:10.1007/11564751_73.

[23] B. Subercaseaux and M. J. H. Heule, The packing chromatic number of the infinite square grid is 15, in: S. Sankaranarayanan and N. Sharygina (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Nature Switzerland, Cham, volume 13993 of *Lecture Notes in Computer Science*, 2023 p. 389–406, doi:10.1007/978-3-031-30823-9_20.

[24] B. Subercaseaux, E. Mackey, L. Qian and M. Heule, Automated symmetric constructions in discrete geometry, in: V. de Paiva and P. Koepke (eds.), *Intelligent Computer Mathematics*, Springer Nature Switzerland, volume 16136 of *Lecture Notes in Computer Science*, October 2025 p. 29–47, doi:10.1007/978-3-032-07021-0_3.

[25] B. Subercaseaux, J. Mackey, M. J. H. Heule and R. Martins, Automated mathematical discovery and verification: Minimizing pentagons in the plane, in: A. Kohlhase and L. Kovács (eds.), *Intelligent Computer Mathematics*, Springer Nature Switzerland, Cham, volume 14960 of *Lecture Notes in Computer Science*, 2024 p. 21–41, doi:10.1007/978-3-031-66997-2_2.

[26] N. Wetzler, M. J. H. Heule and W. A. Hunt, DRAT-trim: Efficient checking and trimming using expressive clausal proofs, in: C. Sinz and U. Egly (eds.), *Theory and Applications of Satisfiability Testing – SAT 2014*, Springer International Publishing, Cham, volume 8561 of *Lecture Notes in Computer Science*, 2014 pp. 422–429, doi:10.1007/978-3-319-09284-3_31.