Avoiding Additive Squares

Thomas Finn Lidbetter
CS 662 Course Project

April 4, 2018

1 Introduction

A word w over a finite alphabet Σ is said to avoid squares if there is no non-empty word x such that xx is a subword of w. A natural question that arises from just this definition is whether there exists an infinite word over Σ that avoids squares. When $|\Sigma| = 2$ it is clear that there is no word of length greater than 3 that avoids squares. On the other hand, if $|\Sigma| \geq 3$ there is an infinite word avoiding squares. Proof of this fact can be found, for example, in the proof of Theorem 2.5.2 in [1]. A stronger avoidance property is that of abelian squares. A word w over alphabet Σ is said to avoid abelian squares if there is no pair of words $x, y \in \Sigma^*$ such that x is a permutation of y, i.e., for all $a \in \Sigma$ we have $|x|_a = |y|_a$, and xy is a subword of w. Again, a natural question that one can ask is whether or not there exists an infinite word over Σ avoiding abelian squares. This question was first posed by Erdős in [2, p.240] where it was noted that there can be no word of length greater than 7 avoiding abelian squares when $|\Sigma| \leq 3$. In the case where $|\Sigma| = 25$, the existence of an infinite word avoiding abelian words was established in [3] with the construction of such a word. This result was later improved to show that where $|\Sigma| = 5$ there is still an infinite word avoiding abelian squares [4]. The remaining case of $|\Sigma| = 4$ was later resolved in [5], again by giving a construction for an infinite word over 4 letters avoiding abelian squares.

The subject of this report concerns one further generalization of words avoiding squares. In particular, we are interested in words avoiding *additive squares*. Where the alphabet Σ is some finite subset of the natural numbers, a word w defined over Σ avoids additive squares if there is no pair of words $x, y \in \Sigma^*$ with $|x| = |y| = n$ and $x = x_1 x_2 \cdots x_n$ and $y = y_1 y_2 \cdots y_n$ such that $\sum_{i=1}^n x_i = \sum_{i=1}^n y_i$ and xy is a subword of w. The question of whether there exists an infinite word over a finite subset of the natural numbers avoiding additive squares was posed by Brown and Freedman in [6], by Pirillo and Varricchio in [7], and also by Halbeisen and Hungerbühler in [8].

Related to this question, work has been done to show that there exists an infinite word over the alphabet $\{0, 1, 3, 4\}$ that avoids additive cubes [9], i.e., that there is an infinite word $\mathbf{w}$ over $\Sigma = \{0, 1, 3, 4\}$ such that $\mathbf{w}$ has no subword $xyz \in \Sigma^*$ with $|x| = |y| = |z| = n$ and $\sum_{i=1}^n x_i = \sum_{i=1}^n y_i = \sum_{i=1}^n z_i$. For alphabets $\Sigma = \{a, b, c, d\}$ of size 4 where $a + b = c + d$, Freedman and Brown [10] show that there is no word of length greater than 60 avoiding additive squares. Additionally, for all finite subsets Σ of $\mathbb{N}$, Brown [11] shows that there exists

a constant C such that every infinite word over Σ has arbitrarily long subwords $x, y \in \Sigma^*$ with $|x| = |y| = n$ and $|\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i| \leq C$. In the same paper, Brown also shows that for all subsets Σ of the natural numbers, and all infinite words $\mathbf{w}$ over Σ, and all positive integers $k > 1$, there exist words $X_1, \ldots, X_k \in \Sigma^*$ such that $X_1 X_2 \cdots X_k$ is a subword of $\mathbf{w}$ and

$$\frac{1}{|X_1|} \sum X_1 = \cdots = \frac{1}{|X_k|} \sum X_k,$$

where $\sum X_i$ is the sum of the numeric values of the symbols in X_i. That is, there exist arbitrarily long subwords of $\mathbf{w}$ that can be broken into blocks with the same average. However, it is not apparent that these results can be used to show that all infinite words have some additive square subword.

In this report I will discuss the work that I conducted on this problem throughout the Winter 2018 semester. Section 2 is dedicated to the work that I did on using SAT solvers to attempt to compute the lengths of longest words avoiding additive squares over specific alphabets. This section constitutes the bulk of this report and it is split into subsections discussing: use of SAT solvers on a similar problem (Section 2.2), generating the input to a SAT solver for the additive square problem (Section 2.3), the choice of SAT solvers used in this project (Section 2.4), the approach to using these SAT solvers (Section 2.5), and the results obtained (Section 2.6). Section 3 briefly discusses some alternative approaches to this problem that I considered but ultimately did not end up devoting a lot of time to, and Section 4 concludes the report by discussing ways in which the ideas presented here may be further developed.

2 Using SAT Solvers

2.1 Definitions

Before proceeding it will be useful to have the following terms laid out. A *literal* is a boolean variable or its negation. A *conjunctive clause* is the conjunction, or logical AND, of one or more literals. A *disjunctive clause*, also referred to just as a *clause*, is the disjunction, or logical OR, of one or more literals. In this setting a *formula* refers to either the disjunction of one or more conjunctive clauses, or the conjunction of one or more disjunctive clauses. A formula is said to be in *disjunctive normal form* (DNF) if it is the disjunction of one or more conjunctive clauses, and a formula is said to be in *conjunctive normal form* (CNF) if it is the conjunction of one or more disjunctive clauses. The conjunctive normal form will be important in the following discussion as it is the input format required by most SAT solvers. For a variable x, and a formula, F, we use the notation $\overline{x}$ and $\overline{F}$ to refer to the negation of x and the negation of F respectively.

2.2 Previous use of SAT solvers

Given a boolean formula, the satisfiability problem asks if there exists an assignment of boolean values to the variables in the formula such that the formula evaluates to TRUE. While this problem is NP-complete, heuristics have been developed that make it feasible to

solve SAT instances involving thousands of variables and clauses. A program that solves a satisfiability instance is referred to as a *SAT solver*. Recently, SAT solvers have been used to prove mathematical theorems. A notable example is the use of a SAT solver to resolve the *boolean Pythagorean triples problem*, a problem in Ramsey theory. The problem can be stated as follows. Can the natural numbers be partitioned into 2 sets, such that neither set contains a triple (a, b, c) with $a^2 + b^2 = c^2$? This can also be stated in terms of colourings: does there exist a 2-colouring of the natural numbers admitting no monochromatic Pythagorean triples? Heule, Kullman, and Marek [12] used a SAT solver to answer this question by demonstrating that there is no 2-colouring of $\{1, \ldots, 7825\}$ that avoids monochromatic Pythagorean triples, but that $\{1, \ldots, 7824\}$ can be coloured in this way. The computations proving these statements required 4 years of CPU time and produced an output file, serving as a verifiable proof of unsatisfiability, 200 terabytes in size. Despite the large resources required in proving the theorem, the successful application of SAT solvers to a problem in combinatorial number theory serves as a compelling proof of concept for the use of this method on similar problems. Hence, we may hope to use SAT solvers to gain insight on the problem of avoiding additive squares.

Following this model I wrote a program that would generate input, specific to the problem of avoiding additive squares, for a SAT solver. The remainder of this section will be devoted to describing how the input is generated and the results obtained from this SAT solving approach in contrast with a depth-first-search implementation. The source code for the depth-first-search implementation and the program for generating the SAT solver input can be found here: `https://github.com/FinnLidbetter/AvoidingAdditiveSquares`

2.3 CNF Formula Generation

The high level idea for producing a conjunctive normal form formula representing the constraints on some word w of length n over a finite alphabet $\Sigma \subset \mathbb{N}$ avoiding additive squares is as follows. First we create variables that correspond to each symbol in the word. This is done by creating one variable for each bit that could be set in the canonical base 2 representation of any symbol in Σ. This means that if m is the largest value in Σ, then $\lfloor \log_2(m) \rfloor + 1$ variables are needed for each symbol in the word. We refer to each collection of variables corresponding to a number or symbol in the word as a *variable grouping*. We then create clauses that constrain assignments to these variables to assignments that correspond only to symbols in Σ. Then, clauses are created that enforce the avoidance of additive squares. To do this we create new variable groupings that correspond to sums of symbols in subwords of w. Clauses are added to make sure that these new variables are constrained to match the values given by the sums of the initial variables. It is then just a matter of ensuring that no variable grouping corresponding to the sum of the symbols in some subword of w is equal to the variable grouping corresponding to an adjacent subword of the same length, where two variable groupings are equal if they correspond to the same numeric value.

We can now discuss in a little more depth how clauses can be created for each of these constraints. In developing these clauses there are a couple of operations that will be used multiple times, so we examine these first.

In creating a formula for this problem there are instances where the constraints can be represented more naturally in disjunctive normal form. For example, if we want to ensure

that the variables $x_0, x_1, x_2, \ldots, x_m$ are not all each equal to the variables with matching indexes $y_0, y_1, \ldots, y_m$, then it is straightforward to write this constraint as

$$(x_0 \wedge \overline{y_0}) \vee (\overline{x_0} \wedge y_0) \vee (x_1 \wedge \overline{y_1}) \vee (\overline{x_1} \wedge y_1) \vee \cdots \vee (x_m \wedge \overline{y_m}) \vee (\overline{x_m} \wedge y_m). \tag{1}$$

Given this representation we want to find a CNF representation that preserves the satisfiability of the formula. However, the naïve transformation from disjunctive normal form to conjunctive normal form can result in an exponential growth in the number of clauses. To avoid this we perform a transformation that only adds a linear number of clauses, but at the cost of also adding a linear number of additional variables—one new variable for each clause in the disjunction. We create a clause that is the disjunction of all of the new variables, ensuring that at least one of the new variables is true in any satisfying assignment for the transformation. Then for each literal in each clause in the disjunctive normal form formula we create a new clause that is the disjunction of the literal and the negation of the new variable corresponding to the clause that the literal is in. The result of applying the transformation to the disjunctive normal form formula in (1) is given in (2), where $d_0, d_1, \ldots d_{2m-1}$ are the new "dummy" variables.

$$\begin{aligned}(d_0 \vee d_1 \vee d_2 \vee \cdots \vee d_{2m-2} \vee d_{2m-1}) \wedge (\overline{d_0} \vee x_0) \wedge (\overline{d_0} \vee \overline{y_0}) \wedge (\overline{d_1} \vee \overline{x_0}) \wedge (\overline{d_1} \vee y_0) \wedge (\overline{d_2} \vee x_1) \\ \wedge (\overline{d_2} \vee \overline{y_1}) \wedge (\overline{d_3} \vee \overline{x_1}) \wedge (\overline{d_3} \vee y_1) \wedge \cdots \wedge (\overline{d_2} \vee x_m) \wedge (\overline{d_2} \vee \overline{y_m}) \wedge (\overline{d_3} \vee \overline{x_m}) \wedge (\overline{d_3} \vee y_m).\end{aligned} \tag{2}$$

One can observe that this transformation preserves satisfiability. That is, there is a satisfying assignment for the transformed CNF formula if and only if there is a satisfying assignment for the original disjunctive normal form formula. To see this, simply maintain the assignments for all non-dummy variables and assign each d_i corresponding to a clause that evaluates to TRUE in (1) to TRUE as well, and assign FALSE to all other dummy variables.

Another operation that will be carried out in developing a CNF formula for the constraints is the assignment of some variable to have the same truth value as a formula. To constrain a variable x to have the same truth value as some formula, F, we add the constraint formula $x \Leftrightarrow F = (x \vee \overline{F}) \wedge (\overline{x} \vee F)$ after converting it to conjunctive normal form. If we have both a CNF and a DNF representation for F we can use DeMorgan's laws and distributivity to easily get a CNF representation for $(x \vee \overline{F}) \wedge (\overline{x} \vee F)$ with a number of literals that is linear in the sum of the number of the literals in the DNF and CNF representations for F.

2.3.1 Constraining to the alphabet

For generating CNF clauses for the additive square problem I implemented two ways through which an alphabet can be specified. The first way of specifying the alphabet is for alphabets made up of all natural numbers less than or equal to some given value m. The second way allows for arbitrary finite alphabets, where the elements can be specified individually. When just a maximum value, m, is specified we are able to constrain a variable grouping to assignments corresponding to values less than or equal to m by using at most $\lfloor \log_2 m \rfloor + 1$ clauses each with at most $\lfloor \log_2 m \rfloor + 1$ literals, assuming that the variable grouping being constrained is also made up of $\lfloor \log_2 m \rfloor + 1$ variables. However, for an arbitrary alphabet

with maximum element m, we need $O(|\Sigma| \log m)$ clauses to constrain each variable grouping to values in the alphabet.

To constrain each variable grouping to correspond only to symbols in some arbitrary alphabet, we can easily come up with a disjunctive normal form representation. For example, to constrain the variable grouping $x = x_1x_0$ to the values $0, 1, 3$ we could use the clauses $(\overline{x_1} \wedge \overline{x_0}) \vee (\overline{x_1} \wedge x_0) \vee (x_1 \wedge x_0)$. To get a conjunctive normal form representation we then just use the transformation described above, adding dummy variables as necessary. Before the transformation, there is at most one conjunctive clause for every symbol in the alphabet and each clause has $\lfloor \log_2 m \rfloor + 1$ literals. This means that after applying the transformation we get $|\Sigma|(\lfloor \log_2 m \rfloor + 1)$ clauses with 2 literals each and one clause with $|\Sigma|(\lfloor \log_2 m \rfloor + 1)$ literals—the disjunction of the dummy variables added in the transformation.

To constrain each variable grouping to correspond to any value less than or equal to some maximum value, m, we will need one clause for each 0 in the base 2 representation of m. This is because in order for a number n, with the same number of bits as m, to be greater than m there must be at least one digit that is a 0 in m but a 1 in n, while all more significant digits match in n and m. So we can create one clause for each 0 digit in m. For example, suppose $m = 44 = 101100_2$. Then we can constrain the variable grouping $x = x_5x_4x_3x_2x_1x_0$ to correspond to a value less than or equal to $m = 44$ with the formula $\overline{(x_5 \wedge x_4)} \wedge \overline{(x_5 \wedge x_3 \wedge x_2 \wedge x_1)} \wedge \overline{(x_5 \wedge x_3 \wedge x_2 \wedge x_0)} = (\overline{x_5} \vee \overline{x_4}) \wedge (\overline{x_5} \vee \overline{x_3} \vee \overline{x_2} \vee \overline{x_1}) \wedge (\overline{x_5} \vee \overline{x_3} \vee \overline{x_2} \vee \overline{x_0})$.

2.3.2 Adding variable groupings

As mentioned in the high level description, we establish constraints that ensure the avoidance of additive squares by first defining new variable groupings to correspond to each possible subword sum that could make up part of an additive square. That is, there will be a variable grouping for the sum of the numbers from index i to index j in the word w of length n if either $i - (j - i) - 1 \geq 1$, or $j + (j - i) + 1 \leq n$. To construct constraints for each of the new variable groupings, it suffices to be able to add a pair of variable groupings. So suppose that we have variable groupings $a = a_m a_{m-1} \cdots a_1 a_0$ and $b = b_m b_{m-1} \cdots b_1 b_0$ where each a_i and each b_i is a boolean variable. Assume that a_m and b_m each correspond to the most significant bits of the numbers or sums that the variable groupings correspond to and a_0 and b_0 correspond to the least significant bit. Note that these variable groupings may be given variable assignments that result in leading zeros.

If we wish to define the variable grouping $z = z_{m+1}z_m \cdots z_1z_0$ such that it corresponds to the sum of groupings a and b, then we perform operations as a serial bitwise adder would. This requires keeping track of carry bits. So, in determining z we will also compute clauses for carry variables $c_0, c_1, \ldots c_m$. Below we use uppercase symbols to refer to formulas. We will then define single variables (given by lowercase symbols) to have the same truth value as the formulas.

For the base case of this addition, we have

$$C_0 := a_0 \wedge b_0, \tag{3}$$

$$Z_0 := (a_0 \wedge \overline{b_0}) \vee (\overline{a_0} \wedge b_0) = (a_0 \vee b_0) \wedge (\overline{a_0} \vee \overline{b_0}). \tag{4}$$

Then for the inductive step, for $1 \leq i \leq m$, we have

$$\begin{aligned} C_i :&= (a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1}) \\ &= (a_i \vee b_i) \wedge (a_i \vee c_{i-1}) \wedge (b_i \vee c_{i-1}), \qquad (5) \\ Z_i :&= (a_i \wedge \overline{b_i} \wedge \overline{c_{i-1}}) \vee (\overline{a_i} \wedge b_i \wedge \overline{c_{i-1}}) \vee (\overline{a_i} \wedge \overline{b_i} \wedge c_{i-1}) \vee (a_i \wedge b_i \wedge c_{i-1}) \\ &= (a_i \vee \overline{b_i} \vee \overline{c_{i-1}}) \wedge (\overline{a_i} \vee b_i \vee \overline{c_{i-1}}) \wedge (\overline{a_i} \vee \overline{b_i} \vee c_{i-1}) \wedge (a_i \vee b_i \vee c_{i-1}). \qquad (6) \end{aligned}$$

We then define $Z_{m+1} := c_m$ if it is possible that a and b corresponded to values that will sum to a number greater than $2^{m+1} - 1$. We determine if this is possible by looking at the number of bits needed to represent the largest number in the alphabet multiplied by the length of the subword that the variable grouping corresponds to.

Both the disjunctive normal form and conjunctive normal form representations are given because in order to use c_{i-1} in the calculation of z_i and to also use $z = z_{m+1}z_m \cdots z_1 z_0$ in future summations or as part of a constraint, we need to create clauses that constrain each c_i and each z_i to those given in (3–6). Hence we need to add constraints $z_i \Leftrightarrow Z_i = (z_i \vee \overline{Z_i}) \wedge (\overline{z_i} \vee Z_i)$ and $c_i \leftrightarrow C_i = (c_i \vee \overline{C_i}) \wedge (\overline{c_i} \vee C_i)$. So, to assign z_0 and c_0 to have the same truth value as the formulas given above we add the CNF constraints

$$\begin{aligned} &(z_0 \vee \overline{a_0} \vee \overline{b_0}) \wedge (z_0 \vee a_0 \vee \overline{b_0}) \wedge (\overline{z_0} \vee a_0 \vee b_0) \wedge (\overline{z_0} \vee \overline{a_0} \vee \overline{b_0}), \\ &(c_0 \vee \overline{a_0} \vee \overline{b_0}) \wedge (\overline{c_0} \vee a_0) \wedge (\overline{c_0} \vee b_0). \end{aligned}$$

Similarly for $1 \leq i \leq m$ we assign each z_i and each c_i to have the correct truth values by adding the following CNF constraints

$$\begin{aligned} &(c_i \vee \overline{a_i} \vee \overline{b_i}) \wedge (c_i \vee \overline{a_i} \vee \overline{c_{i-1}}) \wedge (c_i \vee \overline{b_i} \vee \overline{c_{i-1}}) \wedge (\overline{c_i} \vee a_i \vee b_i \vee c_{i-1}) \wedge (\overline{c_i} \vee \overline{a_i} \vee b_i \vee c_{i-1}) \\ &\wedge (\overline{c_i} \vee a_i \vee \overline{b_i} \vee c_{i-1}) \wedge (\overline{c_i} \vee a_i \vee b_i \vee \overline{c_{i-1}}), \\ &(z_i \vee \overline{a_i} \vee b_i \vee c_{i-1}) \wedge (z_i \vee a_i \vee \overline{b_i} \vee c_{i-1}) \wedge (z_i \vee a_i \vee b_i \vee \overline{c_{i-1}}) \wedge (z_i \vee \overline{a_i} \vee \overline{b_i} \vee \overline{c_{i-1}}) \\ &\wedge (\overline{z_i} \vee a_i \vee b_i \vee c_{i-1}) \wedge (\overline{z_i} \vee \overline{a_i} \vee \overline{b_i} \vee c_{i-1}) \wedge (\overline{z_i} \vee \overline{a_i} \vee b_i \vee \overline{c_{i-1}}) \wedge (\overline{z_i} \vee a_i \vee \overline{b_i} \vee \overline{c_{i-1}}) \end{aligned}$$

Now that we can create new variable groupings that correspond to the sum of values corresponding to other variable groupings, and we can constrain the variable groupings to correspond to values in the alphabet, it just remains to constrain the sums such that they avoid additive squares.

2.3.3 Avoiding additive squares

Given variable groupings $a = a_m a_{m-1} \cdots a_1 a_0$ and $b = b_m b_{m-1} \cdots b_1 b_0$ that correspond to adjacent subwords of equal length, we want to ensure that a is not equal to b. This constraint is given exactly by the CNF formula in the example of the transformation from disjunctive normal form to conjunctive normal form after relabeling x to a and y to b.

2.4 The SAT Solvers

I tried running the CNF clause inputs that I generated on a variety of SAT solvers. I chose to try several different SAT solvers in the hope that one or more of them would

use a search heuristic that is particularly effective for the inputs generated for the additive square avoidance problem. Due to technical difficulties encountered in compiling many of the solvers that I found, I ended up investing a significant amount of time into troubleshooting compilation errors. The SAT solvers that I used in this project are listed below. These particular solvers were chosen largely for the ease with which I was able to install them (at least in the case of Minisat and Riss), and because they were mentioned and recommended in a meeting with Curtis Bright. Additionally, each of these SAT solvers has been a medal winner in the SAT competition (see `http://www.satcompetition.org/`) held in the year that they were each submitted. The years listed with the SAT solvers below indicate the year of the version of the SAT solver used in this project.

- Minisat (2010): `http://minisat.se/`
- Riss (2016): `https://github.com/kammoh/riss`
- Maplesat (2016): `https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/`
- MaplesatCOMSPS_LRB (2017): `https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/`
- MaplesatCOMSPS_CHB (2017): `https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/`

I was able to install Minisat and Riss with relative ease using the Homebrew package manager. However, in order to compile and install maplesat, maplesatCOMSPS_LRB, and maplsesatCOMSPS_CHB I had to manually add, remove, and modify multiple lines of source code. These changes were necessary as I was working on a Mac operating system and the Minisat basis for the Maplesat variants does not compile correctly on Mac machines. The changes that I made are described in the github repository for this project found here: `github.com/FinnLidbetter/AvoidingAdditiveSquares/blob/master/MacCompileMaplesat.md`

2.5 Method

To evaluate the potential of the use of SAT solvers for the additive square avoidance problem, I also implemented a program that performs a depth-first-search over the state space of additive square avoiding words for a given alphabet. The program uses recursive backtracking and tries each possible symbol in the alphabet at the current position in the word. If the symbol doesn't create an additive square in the word, then the program continues on to the next index in the word. However, if an additive square is created, the algorithm backtracks and tries the next symbol at the index backtracked to. Care is taken to avoid re-computing sums as much as is possible by keeping track of the sums of all subwords up to the current index and updating these sums as symbols are added to and removed from the word. Source code for this program can be found here: `https://github.com/FinnLidbetter/AvoidingAdditiveSquares/blob/master/AdditiveSquareDFS.java`

This depth-first search based program also served as a valuable testing tool, to make sure that the CNF formulas generated as input to the SAT solvers were being constructed correctly.

To establish an idea of whether or not a there might exist an infinite word avoiding additive squares, we attempt to find the longest word avoiding additive squares (if such a word exists) over each alphabet of size k of the form $\Sigma = \{0, 1, \ldots, k-1\}$. If we are able to find a longest word for every k, then this would demonstrate that there is no infinite word avoiding additive squares. To this end, we run both the depth-first search and the SAT solvers on each of the alphabets of this form for which the computation is feasible.

In order to improve the efficiency of the depth-first search and to reduce the amount of time required by the SAT solvers we also try restricting the maximum length of an additive square. The intuition behind this is that most of the additive square "conflicts" created should be short relative to the length of the word being tested. So, we test a number of different alphabets and vary the maximum length for an additive square subword to determine the smallest maximum length ℓ needed to give a finite length longest word which avoids all additive square subwords with block length at most ℓ. For a fixed alphabet Σ, getting an idea of the size of ℓ as compared to the length of the longest word over Σ avoiding all additive squares may inform the choice of a maximum additive square length to check in instances where searching the entire state space is too inefficient.

2.6 Results and Analysis

Unfortunately, I have not yet been able to produce any results with the SAT solving approach that cannot also be produced faster by the depth-first-search approach. After running the depth-first-search algorithm over several days, a word of length 163 avoiding additive squares over the alphabet $\Sigma = \{0, 1, 2, 3, 4\}$ was found. Running maplesat for over four days of CPU time on a CNF formula that is satisfiable if and only if there is a word of length 164 over Σ avoiding additive squares yielded no determination of satisfiability. However, this is not surprising, given the amount of CPU time required to find satisfying assignments corresponding to shorter words over Σ. Figure 1 compares the performance of each of the SAT solvers on deciding the satisfiability of CNF formulas corresponding to words of varying lengths over Σ. It seems that the maplesat SAT solver performed best for this alphabet. However, if the trend of CPU time required to solve the instance continues to grow at the rate seen in Figure 1, solving an instance corresponding to a word over of length 164 over Σ with maplesat could reasonably be expected to require at least 2 million seconds (23 days) of CPU time. Since it is quite possible, or even likely, that there is a satisfying assignment to such an instance, even more computational work would have to be done to attempt to establish the maximum length, if it exists, of a word over Σ avoiding additive squares.

As mentioned in the Method section, one way of reducing the number of clauses in a SAT instance for this problem is to restrict the length of the subwords that are checked for additive squares. Table 2 summarizes the effect that the maximum block length, m, has on the length of words avoid additive squares of length at most m. An example of an infinite word is included for each shortest maximum block length m that allows a word of unbounded length avoiding additive squares with block length at most m. The results in Table 2 were all obtained using the depth-first-search implementation.

It was hypothesized that, for a particular alphabet Σ, most additive square conflicts would be caused by additive squares with small block lengths with respect to the length of the longest word over Σ avoiding all additive squares. However, this does not seem to be the

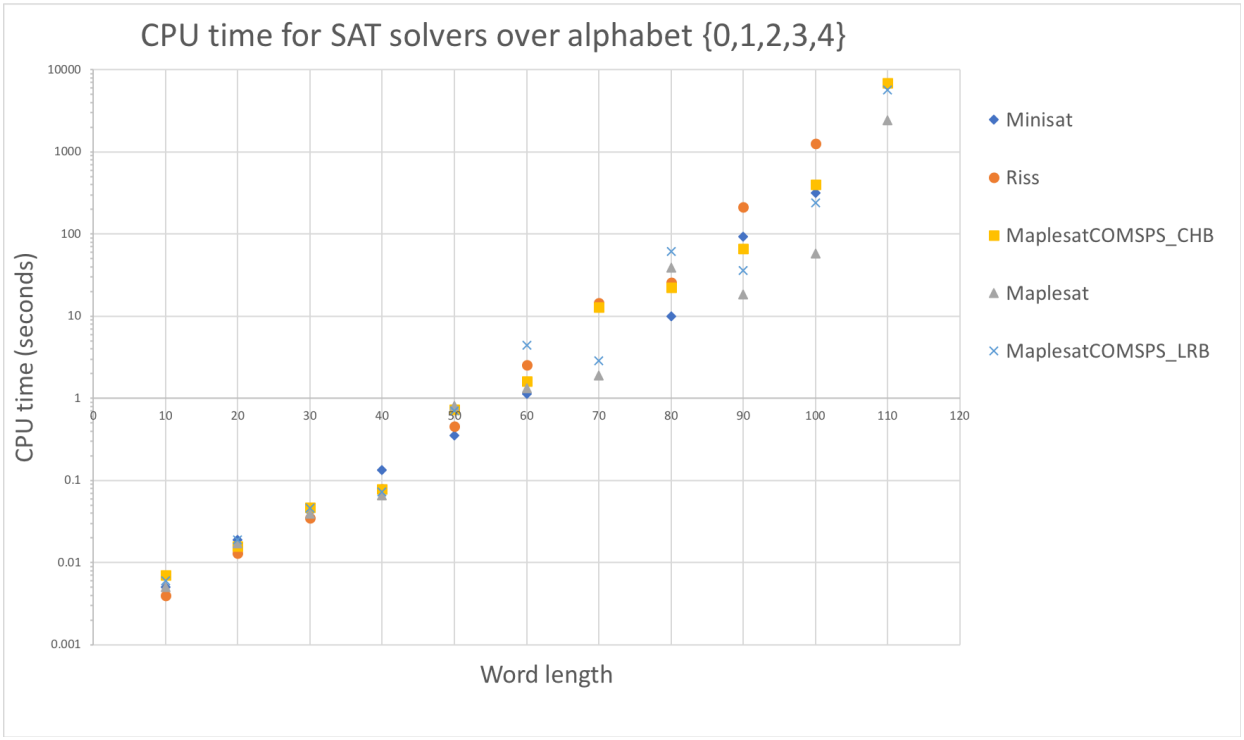

Figure 1: Comparison of SAT solver performance on CNF formulas for alphabet $\{0, 1, 2, 3, 4\}$. Computations were performed on macOS HighSierra version 10.13.3 on a 2.3 GHz Intel Core i5 processor. Raw data is included in the github repository here: `https://github.com/FinnLidbetter/AvoidingAdditiveSquares/blob/master/SATsolverComparison.xlsx`

Word length	Number of Variables	Number of Clauses
10	454	1664
20	2188	8161
30	5438	20176
40	10418	38371
50	17178	62846
60	25688	93496
70	36202	131162
80	48832	176137
90	63462	228012
100	80092	286787
110	98722	352462

Table 1: Number of variables and clauses in each of the SAT instances used in generating the data for Figure 1.

case. For all alphabets tested and shown in Table 2, infinite words could be found when the block length is restricted to as much as one third of the length of the longest word avoiding additive squares. While it is not clear that this should always be the case, computational limits made it prohibitive to test alphabets with more symbols and alphabets with larger maximum values. In any case, it seems that seeking upper bounds on the lengths of longest words avoiding additive squares by restricting the length of the largest block length to check will not give a significant improvement in efficiency over performing the search with no such restriction on the alphabet $\{0, 1, 2, 3, 4\}$.

Alphabet	Max. Block Length	Longest word length	Infinite word
$\{0, 1\}$	Unbounded	3	N/A
$\{a, b\}$	Unbounded	3	N/A
$\{0, 1, 2\}$	Unbounded	7	N/A
$\{a, b, c\}$	Unbounded	7	N/A
$\{0, 1, 2, 3\}$	Unbounded	50	N/A
$\{0, 1, 2, 3\}$	21	50	N/A
$\{0, 1, 2, 3\}$	20	Unbounded	$(01020103102310313 2313)^\omega$
$\{0, 1, 2, 4\}$	Unbounded	62	N/A
$\{0, 1, 2, 4\}$	26	62	N/A
$\{0, 1, 2, 4\}$	25	Unbounded	$(0102040140424142402410 2402)^\omega$
$\{0, 1, 3, 4\}$	Unbounded	55	N/A
$\{0, 1, 3, 4\}$	21	55	N/A
$\{0, 1, 3, 4\}$	20	Unbounded	$(0103010410341041434 14)^\omega$
$\{0, 2, 3, 4\}$	Unbounded	62	N/A
$\{0, 2, 3, 4\}$	26	62	N/A
$\{0, 2, 3, 4\}$	25	Unbounded	$(0203020402324042434 2402324)^\omega$
$\{0, 1, 2, 5\}$	Unbounded	86	N/A
$\{0, 1, 2, 5\}$	42	86	N/A
$\{0, 1, 2, 5\}$	41	90	N/A
$\{0, 1, 2, 5\}$	40	90	N/A
$\{0, 1, 2, 5\}$	39	90	N/A
$\{0, 1, 2, 5\}$	38	90	N/A
$\{0, 1, 2, 5\}$	37	90	N/A
$\{0, 1, 2, 5\}$	36	91	N/A
$\{0, 1, 2, 5\}$	35	91	N/A
$\{0, 1, 2, 5\}$	34	91	N/A
$\{0, 1, 2, 5\}$	33	93	N/A
$\{0, 1, 2, 5\}$	32	Unbounded	$(01020521205251525$ $0521202510120105)^\omega$
$\{0, 1, 2, 3, 4\}$	Unbounded	≥ 163	Unknown

Table 2: The effect of the largest block length checked, m, on the length of the longest word avoiding squares of length less than or equal to $2m$, over various small alphabets.

3 Other Approaches

If we assume that an infinite word avoiding additive squares exists, an alternative approach to resolving the additive square problem is to search for a morphism that generates such a word. Before I began working with SAT solvers on this problem I started by looking at the morphisms used to generate infinite words avoiding abelian squares. Since any word avoiding additive squares must also avoid abelian squares, any morphism generating an infinite word avoiding additive squares must be a morphism whose iterated application does not produce abelian squares either. As such, one might hope to take advantage of some of the existing work on infinite words avoiding abelian squares by finding an appropriate alphabet to apply directly to a known morphism. Alternatively, one might be able to search for some other morphism for some fixed alphabet that generates an infinite word avoiding additive squares. If a candidate morphism is found through either of these approaches, it may be possible to use techniques from [9] to prove that the resulting infinite word avoids additive squares. I did not make any progress with this approach, but I also did not invest much time in this approach over the course of the semester.

On the other hand, if we assume that we should be able to prove that no infinite word avoiding additive squares exists, then one possible way of approaching this is through using ideas similar to those in the proof of Van der Waerden's theorem in the area of Ramsey theory. We can state the additive square problem in a "Ramsey theory-like way" in the following sense. Define $a(r)$, if it exists, to be the least integer such that for all $n \geq a(r)$, every r-colouring of $\{1, 2, 3, \ldots, n\}$ defined by the function $\chi : \{1, 2, \ldots, n\} \to \{0, 1, \ldots, r-1\}$ gives an additive square, i.e., there exist i, k such that

$$\sum_{j=i}^{i+k-1} \chi(j) = \sum_{j=i+k}^{i+2k-1} \chi(j).$$

This is another area in which I have not yet invested much time, although I did read the first two chapters of *Ramsey Theory on the Integers* [13], as well as the last two sections of the last chapter. I read these parts of the book in particular as the first two chapters were devoted to preliminaries and the proof of Van der Waerden's theorem, the penultimate section of the last chapter was devoted to squarefree and cubefree colourings, and the last section of the last chapter was dedicated to *zero-sum* sequences. This last section caught my attention as it concerns summing the "values of the colours" of some subset of terms of a larger sequence. That is, given $k, r \geq 1$ and $\chi : \mathbb{Z}^+ \to \{0, 1, \ldots, r-1\}$, the sequence $(x_1, x_2, \ldots, x_k)$ for $x_1, x_2, \ldots, x_k \in \mathbb{Z}^+$ is a zero-sum sequence if $\sum_{i=1}^{k} \chi(x_i) \equiv 0 \pmod{r}$ [13]. I have not yet investigated the extent to which "summing colours" has otherwise been studied beyond zero-sums in Ramsey theory.

4 Future Directions

In addition to further pursuing the non-SAT solver approaches discussed above, there is more that could be done with the SAT solver approach. Despite not achieving any improvements in running time by using the SAT solver approach to find long words avoiding additive squares

over various different alphabets, I still think that there is some promise in this approach and that more can be done to generate CNF input for which the satisfiability can be determined in a reasonable amount of time for longer words over the alphabet $\{0, 1, 2, 3, 4\}$. For example, some speedup may be attained by performing some pre-processing steps to reduce the number of clauses and variables in the CNF input formula. While a simplification step is taken in the execution of each of the SAT solvers used, there may be some further simplifications that can be made that take greater advantage of the structure of the CNF formulas for the additive square problem.

Another way in which one might hope to get further results from the SAT solver approach is by making use of parallelized SAT solvers and the cube and conquer paradigm used in [12]. To take full advantage of the benefits of the cube and conquer approach, it is likely that the input formulas would have to be simplified as in [12] to give better performance on the cube and conquer steps.

It is also possible that the SAT solvers considered as part of this project may be able to be tuned to perform better on this particular problem. There are a large number of parameters that can be specified upon execution for each of the SAT solvers and it is conceivable that by setting the parameters correctly one might gain a significant improvement in performance. While I did try adjusting some of these parameters, I did not find adjustments that consistently outperformed the default settings. A more systematic study of the effects of changes to the tuning parameters may yield a configuration resulting in consistent performance improvements.

Lastly, although not specifically related to the additive square problem, it may be worth considering the development of a tool to allow CNF formulas to be generated for a wider range of problems. For example, during this term I wrote a program to generate CNF formulas that were satisfiable if and only if given integers m, n, ℓ there exist words $x, y \in \{0, 1\}^*$ with $|x| = n$ and $|y| = m$ such that for all $t \in \{0, 1\}^*$ with $|t| = \ell$ we have that x is a subword of yty, and x is not a subword of y, and there exists $t' \in \Sigma^*$ with $|t'| = \ell + 1$ such that x is not a subword of $yt'y$. I was then able to use a SAT solver to determine the existence of such an x, y pair more quickly than might have been achievable using a depth-first-search style algorithm. It is easy to imagine that a tool allowing easy generation of CNF formulas could be useful for other applications as well.

References

[1] J. Shallit, *A Second Course in Formal Languages and Automata Theory*. New York, NY, USA: Cambridge University Press, 1 ed., 2008.

[2] P. Erdős, "Some unsolved problems," *Magyar Tud. Akad. Mat. Kutato Int. Kozl. 6*, pp. 221–254, 1961.

[3] A. A. Evdokimov, "Strongly asymmetric sequences generated by a finite number of symbols," *Doklady Akademii Nauk SSSR*, vol. 179, no. 6, 1968.

[4] P. A. Pleasants, "Non-repetitive sequences," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 68, pp. 267–274, Cambridge University Press, 1970.

[5] V. Keränen, "Abelian squares are avoidable on 4 letters," in *International Colloquium on Automata, Languages, and Programming*, pp. 41–52, Springer, 1992.

[6] T. Brown and A. Freedman, "Arithmetic progressions in lacunary sets," *The Rocky Mountain Journal of Mathematics*, pp. 587–596, 1987.

[7] G. Pirillo and S. Varricchio, "On uniformly repetitive semigroups," in *Semigroup Forum*, vol. 49, pp. 125–129, Springer, 1994.

[8] L. Halbeisen and N. Hungerbühler, "An application of van der waerdens theorem in additive number theory," *Integers*, p. A7, 2000.

[9] J. Cassaigne, J. D. Currie, L. Schaeffer, and J. Shallit, "Avoiding three consecutive blocks of the same size and same sum," *Journal of the ACM (JACM)*, vol. 61, no. 2, p. 10, 2014.

[10] T. Brown and A. Freedman, "Sequences on sets of four numbers," *INTEGERS: Electronic Journal of Combinatorial Number Theory*, vol. 16, 2016.

[11] T. Brown, "Approximations of additive squares in infinite words," *INTEGERS: Electronic Journal of Combinatorial Number Theory*, vol. 12, 2012.

[12] M. J. Heule, O. Kullmann, and V. W. Marek, "Solving and verifying the boolean pythagorean triples problem via cube-and-conquer," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 228–245, Springer, 2016.

[13] B. M. Landman and A. Robertson, *Ramsey Theory on the integers*, vol. 73. American Mathematical Soc., 2014.