Computational Methods for Combinatorial and Number Theoretic Problems

by

Curtis Bright

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Dr. Erika Ábrahám Professor, RWTH Aachen University
Supervisor	Dr. Vijay Ganesh Assistant Professor
Supervisor	Dr. Krzysztof Czarnecki Professor
Internal Member	Dr. Mark Giesbrecht Professor
Internal Member	Dr. George Labahn Professor
Internal-external Member	Dr. Derek Rayside Assistant Professor

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Computational methods have become a valuable tool for studying mathematical problems and for constructing large combinatorial objects. In fact, it is often not possible to find large combinatorial objects using human reasoning alone and the only known way of accessing such objects is to use computational methods. These methods require deriving mathematical properties which the object in question must necessarily satisfy, translating those properties into a format that a computer can process, and then running a search through a space which contains the objects which satisfy those properties.

In this thesis, we solve some combinatorial and number theoretic problems which fit into the above framework and present computational strategies which can be used to perform the search and preprocessing. In particular, one strategy we examine uses state-of-the-art tools from the symbolic computation and SAT/SMT solving communities to execute a search more efficiently than would be the case using the techniques from either community in isolation. To this end, we developed the tool MathCheck2, which combines the sophisticated domain-specific knowledge of a computer algebra system (CAS) with the powerful general-purpose search routines of a SAT solver. This fits into the recently proposed SAT+CAS paradigm which is based on the insight that modern SAT solvers (some of the best general-purpose search tools ever developed) do not perform well in all applications but can be made more efficient if supplied with appropriate domain-specific knowledge. To our knowledge, this is the first PhD thesis which studies the SAT+CAS paradigm which we believe has potential to be used in many problems for a long time to come.

As case studies for the methods we examine, we study the problem of computing Williamson matrices, the problem of computing complex Golay sequences, and the problem of computing minimal primes. In each case, we provide results which are competitive with or improve on the best known results prior to our work. In the first case study, we provide for the first time an enumeration of all Williamson matrices up to order 45 and show that 35 is the smallest order for which Williamson matrices do not exist. These results were previously known under the restriction that the order was odd but our work also considers even orders, as Williamson did when he defined such matrices in 1944. In the second case study, we provide an independent verification of the 2002 conjecture that complex Golay sequences do not exist in order 23 and enumerate all complex Golay sequences up to order 25. In the third case study, we compute the set of minimal primes for all bases up to 16 as well for all bases up to 30 with possibly a small number of missing elements.

Acknowledgements

Throughout my academic life I have considered myself extremely fortunate for the opportunities I have had to study, learn, and work with incredible teachers and researchers. Without their support and assistance this thesis could not have been written. I would like to thank the following people who were instrumental in my successes and making my journey so pleasant:

- My supervisors Vijay Ganesh and Krzysztof Czarnecki who supplied ample guidance and support. Vijay's direction and vision provided the framework with which this thesis was constructed. His ability to quickly assess an issue, work around setbacks and shift course when necessary continues to be an inspiration to me. Krzysztof's unwavering support was also crucial to the completion of this thesis. He always stood behind and believed in me, even when things were uncertain.

- All the faculty members in Waterloo's symbolic computation group (SCG) and in particular George Labahn, Mark Giesbrecht, Arne Storjohann, and Ilias Kotsireas. I am greatly thankful to George and Mark for serving on my PhD committee and for reviewing my thesis. Additionally, I am grateful to George for his help and advice over many years while running the SCG lab; to Mark for teaching me the joys of symbolic computation as a beginning graduate student; to Arne for introducing me to lattice basis reduction (which unfortunately did not find its way into this thesis) as well as his support and supervision of my master's project; and to Ilias for reviewing this thesis and for many discussions in which he would provide advice and answers to questions about the technical content of this thesis.

- Derek Rayside who agreed to serve on my PhD committee and review my thesis on short notice.

- Erika Ábrahám who served as the external member of my PhD committee and reviewed my thesis. Additionally, she originally proposed the SAT+CAS paradigm which forms a major component of this thesis, and she continues to be a proponent of this paradigm in her work on the SC^2 project.

- My fantastic teachers who provided enlightening and entertaining lectures and who supported me in other ways such as through letters of recommendation. In particular, I would like to thank Kevin Hare who taught me in calculus and computational number theory, supervised my master's project, and introduced me to research in the first place; Stephen New who taught me in logic and analytic number theory;

Anna Lubiw who taught me in the theory of computation; Ian VanderBurgh who taught me in elementary number theory; Cameron Stewart who taught me in algebraic number theory and the geometry of numbers; and Jeff Shallit who taught me in formal languages and provided much support when turning my class project into a published paper which forms part of this thesis.

- My co-instructors which I have been fortunate enough to teach with during PhD studies: Gordon Cormack, Dan Holtby, Rosina Kharal, Dan Lizotte, Jeff Orchard, and Dave Tompkins. They each provided me with solid examples and inspiration that I used to improve my own teaching. Additionally, the instructional coordinators Karen Anderson, Barbara Daly, and Olga Zorin provided excellent support without which the courses could not have run as smoothly as they did.

- Jimmy Liang for answering many questions about how the SAT solver MapleSAT works and his tireless work implementing the features we requested while the work in this thesis was in progress.

- The members of Vijay Ganesh's research group for many varied conversations. In particular Murphy Berzish for helpful feedback on a talk about my research and those with which I have had the privilege of writing papers: Albert Heinle, Chunxiao Li, Saeed Nejati, and Ed Zulkoski.

- All members of the SCG lab where I spent much of my time working. In particular, Andrew Arnold and Andy Novocin for many discussions and advice; Reinhold Burger for being the first person I'd talk to if I had a question about a math problem; Albert Heinle for reading a draft of this thesis and providing many detailed comments; Colton Pauderis for often accompanying me running; Jason Peasgood for being a great housemate; and Dan Roche, Soumojit Sarkar, Mohamed Khochtali, and Connor Flood for being great officemates. Additionally, Jacques Carette, Shaoshi Chen, Kelvin Chung, Mustafa Elsheikh, Somit Gupta, Joseph Haraldson, Armin Jamshipdey, Winnie Lam, Romain Lebreton, Scott MacLean, Mirette Marzouk, Stephen Melczer, Vijay Menon, Nam Pham, Mark Prosser, Hamid Rahkooy, Hrushikesh Tilak, Shiyun Yang, and Wei Zhou for making the SCG lab a place I thoroughly enjoyed working in.

- Jean Webster and Vera Korody for their excellent help organizing my PhD defence and processing travel reimbursements, and Helen Jardine for her assistance booking rooms and getting keys.

- All the friends I've made in Waterloo's computer science program, including Adriaan, Adriel, Alan, Alexandra (R. and V.), Bahareh, Cheryl, Daniel, Daniela, Danika,

David, Dimitrios, Dipti, Divam, Hemant, Hicham, Jack, Jalaj, John, Marianna, Martin, Meng, Michael, Mina, Nathan, Navid, Nika, Par, Rafael, Sandra, Sharon, Shreya, Steven, Zak, the members of my puzzlehunt team "Hella and the Umbrellas" (Aaron, Blake, Cecylia, Dean, Hella, Oliver, Paul, Stephen, Valerie), other friends Cameron, JC, Stanley, Ty, and everyone else who I've eaten lunch with.

- All the friends I've made in Waterloo's Latin dance community, including Abdul, Aldo, Alexander, Ali (K. and L.), Alice (S. and W.), Allison, Amanda, Amin, Amy, Andrea, Angie, Anita, April, Arora, Art, Ashley, Beth, Bridjet, Carla, Carlos, Carol, Cassidee, Charles, Christina, Christine, Clarissa, Clinton, Debra, Dennis, Diana, E'Kong, Ernesto, Eryka, Faryal, Fatin, Fisayo, Frances, Freddy, Hayley, Haytham, Igor, Isaac, Janice, Jeff, Jennifer, Jerome, Jessica, Jhanvi, JJ, Joanna, Jordan, Justine, Karan, Karen, Karin, Katerina, Katie, Kenechi, Kevin, Laura, Laurian, Lexi, Lisa, Loney, Looey, Lyle, Maija, Maju, Margie, Matt, Maureen, Mays, Miguel, Mirabelle, Mohammad, Monica (L. and T.), Naresh, Nathania, Nina, Pat, Priyanka, Randy, Raquel, Richard, Ruta, Saba, Sefora, Shuntaro, Simeon, Sumit, Tarek, Teddy, Tim, Tushar, Tyler, Vesna, Wilf, William, Yue, Ziad, Zoë (N. and Q.), and any that I've regrettably failed to mention.

- Developers of the software which I use on a day-to-day basis. In particular, Donald Knuth for his extraordinary system TEX which was used (via LATEX) to typeset this thesis. Additionally, the developers and maintainers of the operating system UBUNTU, the version control system GIT, and the repository hosting service GitHub.

- The team behind the Shared Hierarchical Academic Research Computing Network (SHARCNET) and Compute/Calcul Canada whose facilities were used during the preparation of this thesis.

- The janitors for providing me with company and sharing food with me during many late nights at the SCG lab.

- Finally, my family who have always supported me and supported my decision to continue (again!) my education at the doctoral level. In particular, I lovingly thank my parents Fred and Cathy, my sister Carly, my brother Greg, and my sister-in-law Kaitlynn.

Curtis Bright
February, 2017

To all students of mathematics.

Table of Contents

Reality is a lovely place, but I
wouldn't want to live there.
Adam Young

Chapter 1

Introduction

This thesis studies several methodologies one can use when attempting to solve certain problems in mathematics, in particular in combinatorics and number theory. Problems in these fields often concern certain objects which are either conjectured to exist or to not exist. Although such questions can sometimes be solved by purely theoretical means, it is often the case that the most straightforward way of determining the existence or nonexistence of a hypothetical object is to perform a search through the space (or a subspace) where the object lives.

The advent of modern computers has made the search approach especially attractive, as computers are able to perform the task of searching a space much more effectively than would otherwise be possible. For this reason this thesis focuses on *computational* approaches to solving such problems. However, even once one has decided on using a computational approach to study a problem one still has to decide which tools and techniques to use to carry out the search. Deciding which method(s) to use is a nontrivial problem and of interest in its own right. As we will see, the choices one makes in how to structure the search can make the difference between solving the problem and not solving the problem in a reasonable amount of time.

Perhaps the most important aspect to consider when performing a computational search is how to specify the search space under consideration. Typically it is easy to define a space which trivially contains the hypothetical object in question and to write a program which will construct and enumerate the objects in the space. In the case of a finite space, this naive brute-force search is even guaranteed to answer the question of existence. The problem with this approach is that the size of the naive search space is typically at least exponential in the size of the object being searched for. In other words, even for objects of

moderate size this approach is infeasible because the search space is so large.

One way of making the search more feasible in this case is to make use of known facts about the object being searched for; such theorems or *theory lemmas* can sometimes enormously decrease the size of the space necessary to search. On the other hand, it is not always easy to encode such theoretical knowledge, or to encode it in a way such that it will make the search procedure more efficient. Often there is more than one strategy that one could use when encoding the theory lemmas and then performing the search; this thesis examines multiple strategies that have shown to be effective in practice.

1.1 Case studies in this thesis

As a way of making the strategies which we will discuss concrete, we will study three problems arising in combinatorics and number theory:

1. Computing *Williamson matrices* of a given order (or showing the nonexistence of such matrices).
2. Computing *complex Golay sequences* of a given order (or showing the nonexistence of such sequences).
3. Computing the *minimal primes* of a given base.

1.1.1 Williamson matrices

First defined in [Williamson, 1944], Williamson matrices have a long history and are studied both for their elegant theoretical properties and practical applications. In fact, in the sixties NASA's Jet Propulsion Laboratory designed space probes which used error-correcting codes based on Williamson matrices to communicate with Earth [Cooper, 2013]. To this end, JPL constructed a Williamson matrix quadruple of order 23 [Baumert et al., 1962].

Since Baumert et al.'s successful search for a Williamson matrix of order 23, there has been much work done in extending the known orders for which Williamson matrices exist. For example, in [Koukouvinos and Kounias, 1988] and [Holzmann et al., 2008]. As recounted in the latter paper:

> Most researchers working in the area had hoped that Williamson matrices of every order must exist.

However, in the nineties it was shown [Đoković, 1993] that Williamson matrices of order 35 do not exist. Fifteen years later, [Holzmann et al., 2008] provided an exhaustive search for all odd orders up to 60 and showed that Williamson matrices of orders 47, 53, and 59 also do not exist, but they do exist for all other odd orders below 60.

Our results

Note that the previous work on Williamson matrices left open the question of the existence of Williamson matrices of even order. Our work remedies this situation, as we have provided for the first time an exhaustive search of Williamson matrices for all orders up to 45, including the even orders. Our results show that it is indeed the case that $n = 35$ is the smallest integer for which Williamson matrices of order n do not exist.

We discuss in detail our results on Williamson matrices in Chapter 3 of this thesis. The work done in Chapter 3 is based on work which appeared in the papers [Bright et al., 2016b] and [Zulkoski et al., 2017] and was done in collaboration with Vijay Ganesh, Albert Heinle, Ilias Kotsireas, Saeed Nejati, Krzysztof Czarnecki, and some currently unpublished work with Chunxiao Li, Vijay Ganesh, and Ilias Kotsireas. The results were generated using the MathCheck system which is based on the SAT+CAS paradigm discussed in Section 1.2. This paradigm uses tools and techniques from the complementary fields of *satisfiability checking* and *symbolic computation*. In particular, MathCheck works by combining SAT solvers and computer algebra systems in a novel fashion. MathCheck was originally developed by Edward Zulkoski, Vijay Ganesh, and Krzysztof Czarnecki [Zulkoski et al., 2015].

1.1.2 Complex Golay sequences

Marcel Golay first introduced the sequences which now bear his name in his groundbreaking 1949 paper [Golay, 1949] on multislit spectrometry, although he did not formally define them until his 1960 paper [Golay, 1961]. Since then, Golay sequences and their generalizations have been widely studied both for their theoretical properties and because of their usefulness to a surprising number of applied domains. As recounted in [Gibson and Jedwab, 2011], they have been applied to such varied fields as optical time domain reflectometry [Nazarathy et al., 1989], power control for multicarrier wireless transmission [Davis and Jedwab, 1999], and medical ultrasound [Nowicki et al., 2003].

Although Golay defined his complementary sequences over a binary alphabet (e.g., $\{\pm 1\}$), later authors have generalized the concept and defined complementary sequences over larger alphabets. In our work, we focus on those sequences defined over the alphabet

$\{\pm 1, \pm i\}$ where i is the imaginary unit $\sqrt{-1}$. Complementary sequences defined over this alphabet are sometimes known as *quaternary* or *4-phase* Golay sequences in the literature, although we refer to them as simply *complex* Golay sequences. As we will see, complex Golay sequences share some similarities with Williamson matrices and in fact both objects can be defined in a similar manner. Of course, there are a few key differences such as the fact that complex Golay sequences are defined over a larger alphabet.

Our results

In Chapter 4 we provide a novel algorithm for enumerating all complex Golay sequences of a given order. The algorithm is based around using a computer algebra system to solve certain classes of Diophantine systems and then using these results along with an algorithm to enumerate permutations of a given form. The enumeration is exhaustive and occurs over a space much smaller than the naive search space. Despite this, we prove that it contains all possible complex Golay sequences of the given order.

Using our algorithm we have independently verified the work of [Fiedler, 2013] up to order 25 as well as the conjecture of [Craigen et al., 2002] that no complex Golay sequences of order 23 exist:

> ... we have thus far found no complex Golay sequences of length 23, and suspect that they do not exist.

We discuss in detail our results on complex Golay sequences in Chapter 4 of this thesis. The work done in Chapter 4 is based on work currently in submission and done in collaboration with Vijay Ganesh, Albert Heinle, and Ilias Kotsireas.

1.1.3 Minimal primes

In a paper published in 2000 the computer scientist Jeffrey Shallit defined a set of 26 prime numbers which he called the minimal primes [Shallit, 2000]. These prime numbers have the special property that every prime, when considered as a string in base 10, contains at least one minimal prime as a subword. In what he calls the "prime game" he suggests that you ask a friend to write down a prime number and then bet them that you can strike out 0 or more digits of that prime to obtain a minimal prime. Shallit's result proves that there is always a way for you to win, but the bet might nevertheless seem reasonable to an unsuspecting friend as the list of minimal primes is innocuous looking; it is not obvious

that you are always able to win. In fact, the entire list of minimal primes is small enough that it fits on a piece of paper the size of a business card [Shallit, 2006].

Shallit closes his original paper [Shallit, 2000] with the remark

> The reader may enjoy trying to compute $M(S)$ for some other classical sets, such as [...] the minimal elements of the primes expressed in bases other than 10.

In addition, the computer scientist David Eppstein later asked [Shallit, 2006] if there is an algorithm that takes as input a base and outputs the list of minimal primes in that base.

Our results

We are able to give a partial answer to Eppstein's question by providing a heuristic algorithm which is able to successfully compute the set of minimal primes in all bases $b \leq 16$, although we do not have a proof that the algorithm will necessarily terminate. The algorithm was also successful in computing the complete set of minimal primes in the bases $b = 18$, 20, 22, 23, 24, and 30. In the remaining bases for $b < 30$ our heuristic algorithm computed the minimal primes with at most 37 missing elements and we provide a strict characterization on the form of the missing elements if they do exist.

Furthermore, our heuristic algorithm was able to compute the set of minimal elements for primes of the form $4n + 1$ as well as for primes of the form $4n + 3$. This successfully completed the sequences A111055 and A111056 in the *Encyclopedia of Integer Sequences* [OEIS Foundation Inc., 1996]. Both of these entries had been missing elements since they were added to the encyclopedia in 2005.

We discuss in detail our results on minimal primes in Chapter 5 of this thesis. The content of Chapter 5 is based on work which appeared in the paper [Bright et al., 2016a] and was done in collaboration with Jeffrey Shallit and Raymond Devillers.

1.2 The SAT+CAS paradigm

The SAT+CAS paradigm is a novel methodology for solving problems which originated independently in two works published in 2015:

1. A paper at the *Conference on Automated Deduction* (CADE) by Edward Zulkoski, Vijay Ganesh, and Krzysztof Czarnecki [Zulkoski et al., 2015] entitled "MathCheck: A Math Assistant via a Combination of Computer Algebra Systems and SAT Solvers".

2. An invited talk at the *International Symposium on Symbolic and Algebraic Computation* (ISSAC) by Erika Ábrahám [Ábrahám, 2015] entitled "Building Bridges between Symbolic Computation and Satisfiability Checking".

Partially inspired by the usage of a SAT solver to solve a case of the Erdős discrepancy conjecture [Konev and Lisitsa, 2014], the CADE paper mentioned above presented a tool called MathCheck. This tool combined a computer algebra system (CAS) with a SAT solver for the first time. The CADE paper describes MathCheck as combining "the search capability of SAT solvers" with the "powerful domain knowledge of CAS systems" and makes the case that

> ... a SAT+CAS system combines the efficient search routines of modern SAT solvers, with the expressive power of CAS, thus complementing both.

Indeed, an immense amount of effort has gone into developing efficient SAT solvers and these tools contain some of the best general-purpose search procedures ever developed. While they do not perform well for all applications, the CADE paper showed that SAT solvers can be made more efficient if they are supplied with appropriate domain-specific knowledge, such as the knowledge available in a CAS.

MathCheck was designed to counterexample or finitely verify (i.e., verify up to some finite bound) conjectures in mathematics. In particular, the system was used to verify two conjectures in graph theory up to bounds which had previously been unobtainable. The second version of the MathCheck system formed the basis for our work on Williamson matrices, as we will discuss in Chapter 3.

Independently from the work done on MathCheck the computer scientist Erika Ábrahám made the observation in her ISSAC 2015 invited talk that the symbolic computation and satisfiability checking communities have similar goals but the way in which they approach and solve problems is rather different. She remarked that

> ... collaboration between symbolic computation and SMT solving is still (surprisingly) quite restricted...

and made the case that the communities would benefit from increased mutual discussion. Furthermore, she argued that developing algorithms and tools which combine the strengths and insights from both these fields is a promising line of research which could be beneficial to both communities.

Later that year, Erika Ábrahám along with Pascal Fontaine, Thomas Sturm, and Dongming Wang organized a "Symbolic Computation and Satisfiability Checking" seminar at the computer science research centre Dagstuhl [Ábrahám et al., 2016b]. The seminar ran from the 15th to the 20th of November 2015 and was billed as

> ... the first global meeting of the two communities of symbolic computation and satisfiability checking.

In 2016, the SC^2 project [Ábrahám et al., 2016a] (here SC standing for both Symbolic Computation and Satisfiability Checking) was started with the stated goal of creating

> ... a new research community bridging the gap between Satisfiability Checking and Symbolic Computation, whose members will ultimately be well informed about both fields, and thus able to combine the knowledge and techniques of both fields to develop new research and to resolve problems (both academic and industrial) currently beyond the scope of either individual field.

The project is funded through the *Horizon 2020* European Union funding programme for research and innovation and officially started on July 1, 2016. On August 4, 2016, James Davenport gave the project's inaugural talk at the conference *Applications of Computer Algebra* (ACA), where he introduced the project and outlined its goals [Davenport, 2016]. The SC^2 project's website `www.sc-square.org` illustrates that there is already much interest in this topic from both academia and industry.

This thesis also explores this topic as one of the methodologies which we study. In particular, part of Chapter 3 is based off of the paper [Bright et al., 2016b] which was invited to appear in the SC^2 topical session at the 2016 edition of *Computer Algebra in Scientific Computing* (CASC). Here we presented our follow-up to the original MathCheck system and described its use in computing Williamson matrices.

Additionally, we presented an extension of [Bright et al., 2016b] at the SC^2 workshop during the 2016 edition of the *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (SYNASC). In this extension we modified the inner loop of the SAT solver so that it could *programmatically* learn facts about the search space as described in the paper [Ganesh et al., 2012]. In our case, we added some functionality from a CAS directly into the SAT solver and found an increase in its efficiency to solve certain SAT instances generated by the MathCheck system. Timings which demonstrate this gain in efficiency can be found in Section 3.6 of this thesis, providing evidence of the power of the SAT+CAS paradigm. To our knowledge, this is the first PhD thesis which studies the SAT+CAS paradigm which we believe has potential to be used in many problems for a long time to come.

1.3 Implementations

In order to demonstrate the effectiveness of the previously mentioned methods, we provide implementations of our algorithms along with timings of their running times on inputs of reasonable size. The source code for each of the studied search procedures are available as specified below:

1. The tool and Python scripts which we used to generate Williamson matrices up to order 45 can be found in the MathCheck2 BitBucket repository [Bright, 2016b].

2. An implementation of our algorithm for generating complex Golay sequences written in C is available online [Bright, 2016a].

3. The C code which we used to generate the minimal primes up to order 30 can be found in the MEPN (Minimal Elements for the Prime Numbers) GitHub repository [Bright, 2014].

1.4 Historical context

Since the development of modern computers they have shaped mathematical practice by changing how mathematical theorems are proven. In some cases, computers generate data which is then used to make a conjecture that a mathematician proves in a conventional manner. In other cases, computers are used in the process of proving a theorem and steps of the proof rely on computations done by a computer. In principle these computations could be hand-checked by a mathematician but it would often be prohibitive to do so. After all, the very purpose of computers is that they allow us to compute things much more efficiently than is possible to do by hand.

Because it is not practical for a mathematician to verify the computations themselves in such *computer-assisted proofs*, some such as the philosopher of mathematics Thomas Tymoczko have argued that accepting computer-assisted proofs as legitimate changes the very sense of what it means for a theorem to be proven [Tymoczko, 1979]. He has argued that such proofs are no longer an *a priori* deduction but an *experimental* argument, since it requires trust that the computer performed the computations correctly and trust that the computer was programmed correctly.

The impetus for Tymoczko's paper referred to above was the publication of a computer-assisted proof of the four colour theorem [Appel and Haken, 1976] which was the first major

theorem to require the usage of a computer to perform the necessary computations. The proof used mathematical arguments to reduce the number of cases to check to 1,936 and then used a computer program to verify each case in turn. A simpler proof was later published, albeit one still relying on the usage of computers [Robertson et al., 1997].

The computations done in Appel and Haken's proof were independently checked using different programs and hardware, making the possibility of error less likely. However, to fully accept the proof one must still trust that the implementations are free of error. To remedy this situation and further increase the confidence in the result, in 2005 Benjamin Werner and Georges Gonthier completed a formal proof of the four colour theorem [Gonthier, 2008] which was subsequently verified by the theorem prover Coq [Barras et al., 1997]. Thus, one only needs to trust the implementation of the Coq kernel (which is small and well tested) to have confidence that the theorem is correct. Of course, one also needs to ensure that the formalization of the theorem's statement in Coq is correct, but this should be much easier than verifying the proof itself.

Another famous theorem proven using computer assistance was Kepler's conjecture that the most efficient way of packing spheres in three dimensions is in a pyramid shape. Although the conjecture was first stated by Johannes Kepler in 1611 and later included in David Hilbert's famous 1900 list of 23 unsolved problems, it was not proven until 1998 by the mathematician Thomas Hales [Hales, 2005]. Like in the proof of the four colour theorem, Hales' proof reduced the problem to a large but finite number of cases and a computer was used to individually check each case. The mathematician Jeffrey Lagarias described the reviewing process [Lagarias, 2011] as follows:

> The nature of this proof, consisting in part of a large number of inequalities having little internal structure, and a complicated proof tree, makes it hard for humans to check every step reliably. In this process detailed checking of many specific assertions found them to be essentially correct in every case. This result of the reviewing process produced in these reviewers a strong degree of conviction of the essential correctness of this proof...

The issue of the *Annals of Mathematics* in which Hales' paper appeared included the following "Statement by the Editors":

> The computer part may not be checked line-by-line, but will be examined for the methods by which the authors have eliminated or minimized possible sources of error...

Because the reviewers could not claim 100% confidence in Hales' result, in 2003 he began a project to produce a proof of the result which could be formally verified by a computer. After over a decade of work, the project was completed in 2015 [Hales et al., 2015] and the formal proof was successfully verified by a combination of the proof assistants HOL Light [Harrison, 1996] and Isabelle [Nipkow et al., 2002].

Like in the computer-assisted proofs just described, the work in this thesis also relies on computer programs to perform computations which would be infeasible to perform in any other way. Thus, the confidence in our results is only as strong as the confidence we hold in the implementors of the programs, operating systems, and hardware that we used. We used independent implementations of the algorithms we describe whenever possible; this decreases the likelihood that our results contain errors but it unfortunately does not completely rule out the possibility. Ideally, we would produce a formal proof to accompany our results, but we currently do not have this capability. Although some theorems which were proven using SAT solvers can produce certificates which can then be formally verified [Heule et al., 2016], our work relies on additional results (e.g., Theorem 3.2) which would also have to be formally verified.

1.5 Structure of this thesis

In Chapter 2 we give background on SAT/SMT solvers, the MathCheck system, and the Hadamard conjecture, which is necessary knowledge to understand our work on Williamson matrices as presented in Chapter 3. In Chapter 4 we present our work on complex Golay sequences, and in Chapter 5 we present our work on minimal primes. Finally, in Chapter 6 we conclude the thesis with what we believe are the main takeaways of our work. In particular, we describe properties of problems which make them suitable for solving with the SAT+CAS paradigm.

If we can know, it surely would
be intolerable not to know.
Edward Titchmarsh

Chapter 2

Background

In this chapter we provide context and background which will help the reader understand our work on the problems we discuss in the following chapters. In particular, we provide background on SAT/SMT solvers, give an overview of the MathCheck system, and describe the Hadamard conjecture.

Many conjectures in number theory and combinatorial mathematics are simple to state but exceptionally hard to verify. For example, the Hadamard conjecture asserts the existence of certain combinatorial objects in an infinite number of cases [Colbourn and Dinitz, 2007], making exhaustive search impossible. In such cases, mathematicians often resort to finite verification (i.e., verification up to some finite bound) in the hopes of learning some meta property of the class of combinatorial structures they are investigating, or discover a counterexample to such conjectures. However, even finite verification of combinatorial conjectures can be challenging because the search space for such conjectures is often exponential in the size of the structures they refer to. This makes straightforward brute force search impractical. However, it is sometimes possible to use "inspired" brute force search by using domain-specific knowledge to decrease the size of the search space coupled with a general-purpose search procedure. As Doron Zeilberger has put it [Zeilberger, 2015],

Brute-brute force has no hope. But clever, inspired brute force is the future.

In recent years, conflict-driven clause learning (CDCL) Boolean SAT solvers (see for example [Biere et al., 2009], [Marques-Silva et al., 1999], [Moskewicz et al., 2001]) have become very efficient general-purpose search procedures for a large variety of applications. Indeed, SAT solvers are probably the best general-purpose search procedures we currently

have. Despite this remarkable progress these algorithms have worst-case exponential time complexity, and may not perform well by themselves for many search applications—but they can become more efficient with appropriately encoded domain-specific knowledge. By contrast, computer algebra systems (CAS) such as MAPLE [Char et al., 1986], MATHEMATICA [Wolfram, 1999], and SAGE [Biere et al., 2009] are often a rich storehouse of domain-specific knowledge, but do not generally contain sophisticated general-purpose search procedures.

As argued by the SC^2 project [Ábrahám et al., 2016a], the strengths of modern SAT solvers and CAS are complementary, i.e., the domain-specific knowledge of a CAS can be crucially important in cutting down the search space associated with combinatorial conjectures, while at the same time the clever heuristics of SAT solvers, in conjunction with CAS, can efficiently search a wide variety of such spaces.

2.1 SAT/SMT solvers

A *propositional formula* is an expression involving Boolean variables and logical connectives such as AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). The *satisfiability problem* is to determine if a given formula is *satisfiable*, i.e., if there exists an assignment to the variables of the formula which makes the formula true. One way of solving this problem is simply to try every possible assignment of true and false to all variables. For example, Figure 2.1 shows all possible assignments to the variables in the formula

$$\phi \coloneqq (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y) \wedge z$$

in a tree diagram; the leaves of the tree contain the result if that particular assignment satisfies the formula or not.

A *SAT solver* is a program which solves the satisfiability problem and produces a satisfying assignment in the case that the given formula is satisfiable. In practice, SAT solvers usually accept formulae given in *conjunctive normal form* (CNF), i.e., formulae of the form $\bigwedge_i C_i$ where each C_i is a *clause* (of the form $\bigvee_j l_{ij}$ where each l_{ij} is either a Boolean variable or its negation). For example, the formula ϕ defined above is a CNF formula.

Most modern SAT solvers are based on the Davis–Putnam–Logemann–Loveland (DPLL) algorithm. At its core this algorithm performs a depth-first search through the space of possible assignments, stopping when a satisfying assignment has been reached or when the space has been exhausted. Naturally, there are many details and optimizations which make

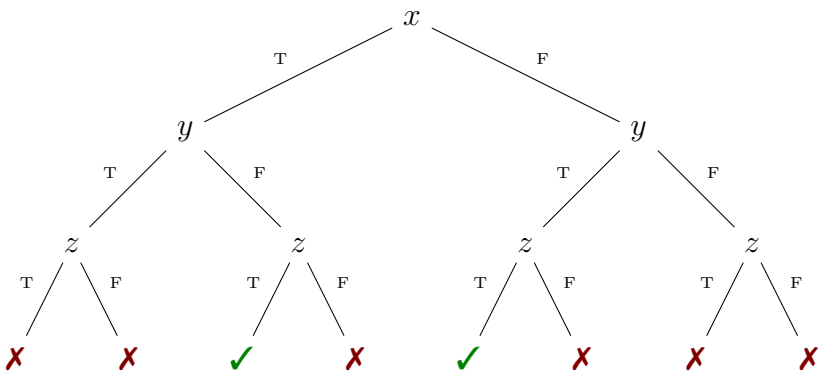

Figure 2.1: Tree of possible assignments for the formula $(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y) \wedge z$.

this idea more practical, such as employing conflict-driven clause learning. In more detail, the DPLL algorithm repeatedly performs the following steps:

- Decide: Choose an unassigned variable and assign it a value. If all variables have been assigned without a conflict, return the satisfying assignment.
- Deduce: Perform simplifications on the clauses in the given formula to detect conflicts and infer values of variables. The inference process (*Boolean Constant Propagation*) is used repeatedly until no new inferences are made.
- Resolve: If a conflict occurs, learn a clause prohibiting the current assignment and perform a "backjump" by undoing the variable choices leading to the conflict and continuing with another assignment. If the conflict occurs when there are no variable assignments to undo, return UNSAT.

These steps are outlined in a flowchart diagram in Figure 2.2.

It is also possible to consider the satisfiability problem for quantifier-free formulas of first-order logic over various theories. Some theories which are commonly used include equality logic with uninterpreted functions, array theory, bitvector theory, and various theories of arithmetic such as integer or real arithmetic (or their linear arithmetic fragments). Programs which are able to solve the satisfiability problem in this context are known as *SAT-modulo-theories* (SMT) solvers.

Modern SMT solvers typically combine a SAT solver with appropriate theory solver(s) to determine satisfiability of first-order formulas. Given an existentially-quantified first-order formula in conjunctive normal form it is possible to abstract the formula into a pure

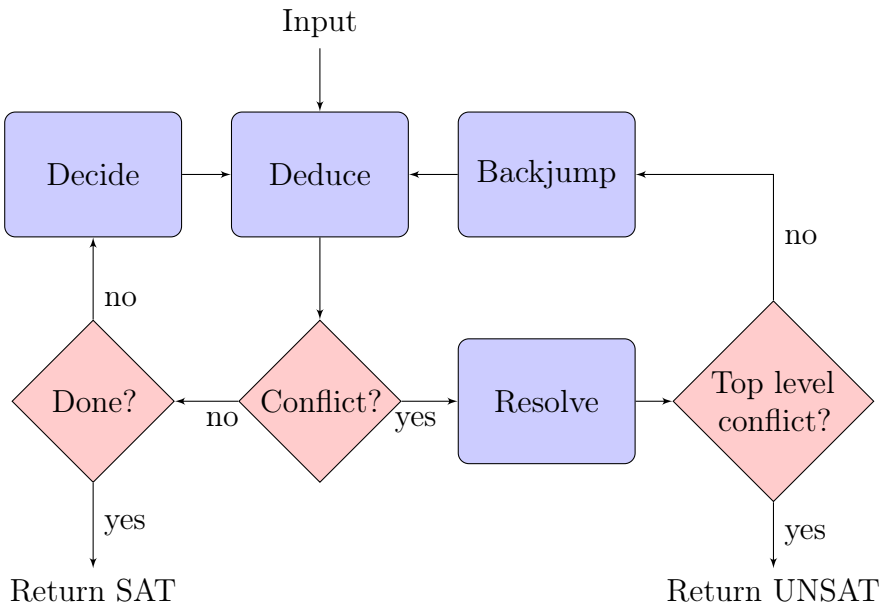

Figure 2.2: Flowchart outline of the DPLL algorithm.

propositional formula by replacing each theory constraint with a fresh propositional variable. A SAT solver is then used to determine if the abstracted formula is satisfiable. If so, the assignment is given to the theory solver to determine if the assignment yields a genuine solution of the original formula. If the assignment does not give rise to a solution then the theory solver determines a clause which encodes a reason why the assignment is prohibited by the theory and passes that to the SAT solver so the search can be resumed. These steps are outlined in a flowchart diagram in Figure 2.3.

The above procedure forms the basis of the DPLL(T) algorithm (here T represents the theory under consideration). The procedure is called *lazy* if it waits until all propositional variables have been assigned until querying the theory solver. *Less lazy* variants are possible which also query the theory solver with partial assignments and therefore pass theory lemmas to the SAT solver more frequently.

2.2 The MathCheck system

In this section we outline a proof-of-concept system we call MathCheck which uses SAT and CAS functionalities to finitely verify conjectures in mathematics. The first version of

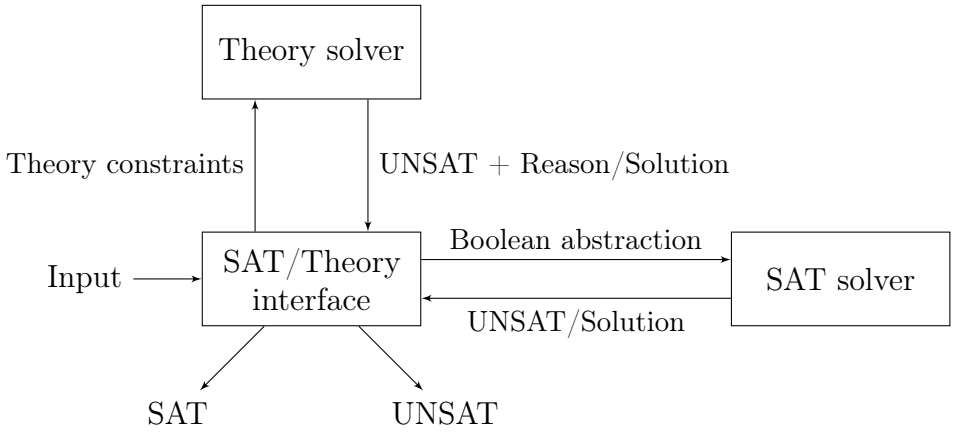

Figure 2.3: Flowchart outline of SMT solvers.

MathCheck [Zulkoski et al., 2015] was developed by Edward Zulkoski, Vijay Ganesh, and Krzysztof Czarnecki and was successfully used to verify two conjectures from graph theory up to bounds which had previously been unobtainable. A SAT solver was the primary search tool used, but the system would periodically make queries to a CAS to learn facts about the remaining search space. These facts were then translated into a form the SAT solver could use and the search was resumed with these additional clauses.

The second version of MathCheck was developed by the author with Vijay Ganesh, Albert Heinle, Ilias Kotsireas, Saeed Nejati, and Krzysztof Czarnecki. It was used to study conjectures in combinatorial design theory, as we describe in Sections 2.3 and 3. The main addition made in the second version of MathCheck is the addition of a more sophisticated *generator* script. This script generates the SAT instance(s) and also queries a CAS for information about the domain space and encodes relevant facts directly into the generated instance(s).

The MathCheck system can be viewed as a parallel systematic generator of combinatorial structures referred to by the conjecture under verification C. It uses a CAS to supply domain-specific knowledge to prune away structures that do not satisfy C, while the SAT solver is used to verify whether any of the remaining structures satisfy C. In addition, if the SAT solver used supports the generation of UNSAT cores (concise encodings of why a formula is unsatisfiable) we can use them to further prune the search in a CDCL-style learning feedback loop.

The architecture of the MathCheck2 system is outlined in Figure 2.4. At its heart is a generation script written in Python. This script generates SAT instances containing

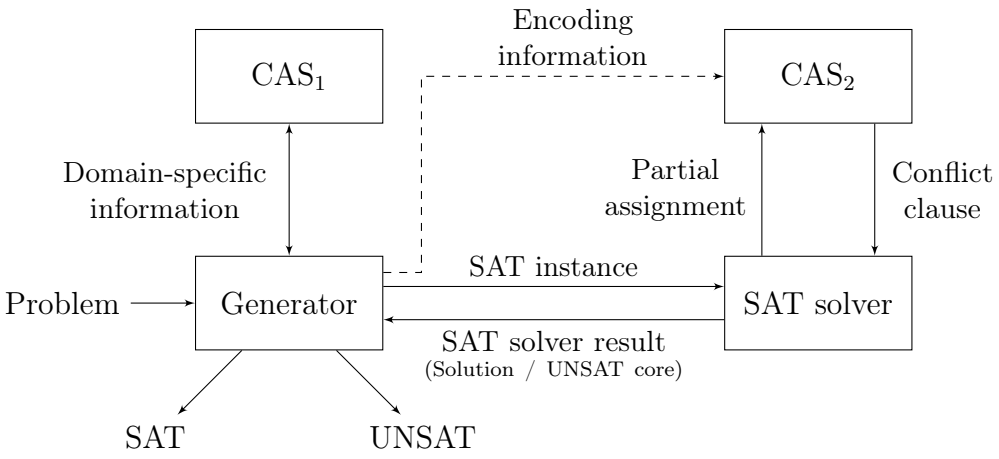

Figure 2.4: Outline of the architecture of MathCheck2. The boxes on the left correspond to the preprocessing which encodes and decomposes the original problem into SAT instance(s). The boxes on the right correspond to an SMT-like setup with the CAS playing the role of the theory solver.

constraints defining the combinatorial structures of the conjecture in question. It also uses data provided to it by CAS functions to prune the search space and it interfaces with SAT solvers to discern if the certain subspaces of the search space contain an instance of the combinatorial structure being searched for. For example, in Chapter 3 we show how to use Theorem 3.2 to split the search space into partitions and then use a CAS to remove some of those partitions from consideration. The generation script contains functions useful for translating combinatorial conditions into clauses which can be read by a SAT solver. The generator is currently optimized to deal with problems defined in terms of an *autocorrelation* function. Such problems arise from the Hadamard conjecture from coding and combinatorial design theory. For example, we discuss the the problem of computing Williamson matrices (which can be used to construct Hadamard matrices) in Chapter 3.

Once the class of combinatorial objects has been determined, the script accepts a parameter n which determines the size of the object to search for. For example, when searching for Williamson matrices, the parameter n denotes the dimension (i.e., the number of rows and columns) of the matrix. The generation script then queries the CAS it is interfaced with for properties that any order n instance of the combinatorial object in question must satisfy. The domain-specific information returned by the CAS is read by the generator and then used to prune the space which will be searched by the SAT solver.

Once the generator determines the space to be searched it splits the space into distinct

subspaces in a divide-and-conquer fashion. After the partitioning of the search space has been completed, the script generates two types of files:

1. A single "master" file in DIMACS CNF (conjunctive normal form) format which contains the conditions specifying the combinatorial object being searched for. These are encoded as propositional formulas in conjunctive normal form. An assignment to the variables which makes all of them true would give a valid instance of the object being searched for (and a proof that no such assignment exists proves that there are no instances of the object in question).

2. A set of files which contain partial assignments of the variables in the master file. Each file corresponds to exactly one subspace of the search space produced by the generator.

There are at least 2 advantages of splitting up the problem in such a way:

1. It easily facilitates parallelization. For example, once the instances are generated they can be given to a cluster of SAT solvers running in parallel.

2. It allows domain-specific knowledge to be used in the splitting process; partitioning the space in a fortuitous manner can considerably speed up the search, as it is possible that the CAS can use its domain-specific knowledge to immediately rule out certain subspaces that the SAT solver would have difficulty searching.

Furthermore, in cases that an instance is found to be unsatisfiable, SAT solvers such as MapleSAT [Liang et al., 2016] that support the generation of a so-called *UNSAT core* can be used to further prune away other similar structures that do not satisfy the conjecture-under-verification. Given an unsatisfiable instance ϕ, its UNSAT core is a set of clauses that concisely characterizes the reason why ϕ is unsatisfiable and thus encodes an unsatisfying subspace of the search space.

Lastly, we modify the inner loop of the SAT solver with the addition of a *callback* function. This function accepts the current partial assignment under consideration inside the SAT solver and uses CAS functionality to try to show that the partial assignment cannot be extended to a complete satisfying assignment. If the CAS is successful in ruling out a class of assignments which includes the current partial assignment then a conflict clause is generated encoding this fact and added into the SAT solver's clause database. This idea of *programmatically* generating conflict clauses was introduced in [Ganesh et al., 2012].

We describe a callback function for the case of Williamson matrices in Section 3.4 and in Chapter 3.6 show that this function is useful in practice.

The CAS which the SAT solver interfaces with does not necessarily need to be the same CAS that the generator used when constructing the SAT instances, hence we refer to them as CAS_1 and CAS_2 in Figure 2.4. Also, CAS_2 needs to know the meaning of the variables in the SAT instances so that it can appropriately determine which assignments to rule out. This encoding information can either be given to CAS_2 in a file or the generator can always label variables in a manner such that this information can be implicitly inferred. For example, in the Williamson case study the variables encoding the entries of the Williamson matrices were always the variables in the SAT instances with the lowest indices.

2.3 The Hadamard conjecture

As motivation for studying autocorrelation problems, we formally introduce the *Hadamard conjecture* from combinatorial design theory. First, we give the definition of a Hadamard matrix.

Definition 2.1. A matrix $H \in \{\pm 1\}^{n \times n}$, $n \in \mathbb{N}$, is called a *Hadamard matrix*, if for all $i \neq j \in \{1, \ldots, n\}$, the dot product between row i and row j in H is equal to zero. We call n the *order* of the Hadamard matrix.

First studied by Hadamard [Hadamard, 1893], he showed that if n is the order of a Hadamard matrix, then either $n = 1$, $n = 2$ or n is a multiple of 4. In other words, he gave a *necessary* condition on n for there to exist a Hadamard matrix of order n. The Hadamard conjecture is that this condition is also *sufficient*, so that there exists a Hadamard matrix of order n for all $n \in \mathbb{N}$ where n is a multiple of 4.

Conjecture 2.1. *There exists a Hadamard matrix of order n for all n divisible by 4.*

Hadamard matrices play an important role in many widespread branches of mathematics, for example in coding theory [Muller, 1954, Reed, 1954, Walsh, 1923] and statistics [Hedayat et al., 1978]. Because of this, there is a high interest in the discovery of different Hadamard matrices up to equivalence. Two Hadamard matrices H_1 and H_2 are said to be *equivalent* if H_2 can be generated from H_1 by applying a sequence of negations/permutations to the rows/columns of H_1, i.e., if there exist signed permutation matrices U and V such that $U \cdot H_1 \cdot V = H_2$.

There are several known ways to construct sequences of Hadamard matrices. One of the simplest such constructions is by Sylvester [Sylvester, 1867]: given a known Hadamard matrix H of order n,

$$\begin{bmatrix} H & H \\ H & -H \end{bmatrix}$$

is a Hadamard matrix of order $2n$. This process can of course be iterated, and hence one can construct Hadamard matrices of order $2^k n$ for all $k \in \mathbb{N}$ from H.

There are other methods which produce infinite classes of Hadamard matrices such as those by Paley [Paley, 1933]. However, there is no known method which can provably construct a Hadamard matrix of order n for arbitrary multiples of 4, although there are some methods which are conjectured to have this property [Kotsireas, 2013b]. The smallest unknown order is currently $n = 4 \cdot 167 = 668$ [Colbourn and Dinitz, 2007]. A database with many known matrices is included in the computer algebra system MAGMA [Bosma et al., 1997] and further collections are available online [Sloane, 2004, Seberry, 1999]. In addition, we have submitted over 500 pairwise inequivalent Hadamard matrices which were generated by MATHCHECK2 to the MAGMA database as well as making them available through the MATHCHECK webpage [Ganesh et al., 2015].

Because there are 2^{n^2} matrices of order n with ± 1 entries, the search space of possible Hadamard matrices grows extremely quickly as n increases, and brute-force search is not feasible. Because of this, researchers have defined special types of Hadamard matrices which can be searched for more efficiently because they lie in a small subset of the entire space of Hadamard matrices.

2.3.1 The Williamson construction

The so-called *Williamson method* is one way to generate large Hadamard matrices while searching through a much smaller space than the space of all Hadamard matrices. In fact, Hadamard matrices of size $4n \times 4n$ generated by Williamson's method are defined by only approximately $2n$ Boolean variables.

Theorem 2.1 (cf. [Williamson, 1944]). *Let $n \in \mathbb{N}$ and let A, B, C, $D \in \{\pm 1\}^{n \times n}$. Further, suppose that*

1. *A, B, C, and D are symmetric;*
2. *A, B, C, and D commute pairwise (i.e., $AB = BA$, $AC = CA$, etc.);*

3. $A^2 + B^2 + C^2 + D^2 = 4nI_n$, *where* I_n *is the identity matrix of order* n.

Then

$$\begin{bmatrix} A & B & C & D \\ -B & A & -D & C \\ -C & D & A & -B \\ -D & -C & B & A \end{bmatrix}$$

is a Hadamard matrix of order $4n$.

For practical purposes, one considers A, B, C, and D in the Williamson construction to be *circulant* matrices, i.e., those matrices in which every row is the previous row shifted by one entry to the right (with wrap-around, so that the first entry of each row is the last entry of the previous row). Such matrices are completely defined by their first row $[x_0, \ldots, x_{n-1}]$ and always satisfy the commutativity property. If the matrix is also symmetric, then we must further have $x_1 = x_{n-1}$, $x_2 = x_{n-2}$, and in general $x_i = x_{n-i}$ for $i = 1, \ldots, n-1$. Therefore, if a matrix is both symmetric and circulant its first row must be of the form

$$\begin{array}{ll} [x_0, x_1, x_2, \ldots, x_{(n-1)/2}, x_{(n-1)/2}, \ldots, x_2, x_1] & \text{if } n \text{ is odd} \\ [x_0, x_1, x_2, \ldots, x_{n/2-1}, x_{n/2}, x_{n/2-1}, \ldots, x_2, x_1] & \text{if } n \text{ is even.} \end{array} \tag{2.1}$$

Definition 2.2. A *symmetric* sequence of length n is one of the form (2.1), i.e., one which satisfies $x_i = x_{n-i}$ for $i = 1, \ldots, n-1$.

Williamson matrices are circulant matrices A, B, C, and D which satisfy the conditions of Theorem 2.1. Since they must be circulant, they are completely defined by their first row. In light of this, we may simply refer to them as if they were sequences, as we will do in Chapter 3. Furthermore, since they are symmetric the Hadamard matrix generated by these matrices is completely specified by the $4\left\lceil \frac{n+1}{2} \right\rceil$ variables

$$a_0, a_1, \ldots, a_{\lfloor n/2 \rfloor}, b_0, b_1, \ldots, b_{\lfloor n/2 \rfloor}, c_0, c_1, \ldots, c_{\lfloor n/2 \rfloor}, d_0, d_1, \ldots, d_{\lfloor n/2 \rfloor}.$$

Given an assignment of these variables, the rest of the entries of the matrices A, B, C, and D may be chosen in such a way that conditions 1 and 2 of Theorem 2.1 always hold. There is no trivial way of enforcing condition 3, but in Chapter 3 we will derive consequences of this condition which will simplify the search for matrices which satisfy it.

2.3.2 Complexity

In the course of our research the question was raised concerning the computational complexity of computing a Hadamard matrix. After all, if the problem was in P (i.e., solvable in polynomial time) then trying to solve it by translating it into a SAT instance would be somewhat questionable, as the problem of solving a SAT instance is NP-complete and widely expected to not be solvable in polynomial time.

The problem is easily stated as a function problem as follows:

Problem 2.1. *Given a natural number n, output an $n \times n$ Hadamard matrix if one exists and "does not exist" otherwise.*

This problem can be solved in nondeterministic polynomial time in n since one can easily verify in polynomial time if a given $n \times n$ matrix is a Hadamard matrix. To ask if the problem is NP-complete one must first formally state the problem as a decision problem. There are multiple ways one could do this, but the obvious translation of "Given n, does a Hadamard matrix of order n exist?" is not a good candidate, since if the Hadamard conjecture is true then this problem has a simple solution of outputting *yes* if $n = 1$, 2, or if 4 divides n, and outputting *no* otherwise.

We propose the following as a suitable formulation as a decision problem:

Problem 2.2. *Given an $n \times n$ matrix with entries either empty or ± 1 does there exist some way of filling in the empty entries to generate a Hadamard matrix?*

It is straightforward to see that if one can solve this problem in polynomial time then one can construct a Hadamard matrix of order n in polynomial time, assuming one exists. To accomplish this, one simply needs to make repeated queries to the procedure which solves Problem 2.2, starting with an empty $n \times n$ matrix and arbitrarily filling in one extra entry after each query. If after adding an entry one ever determines that the current matrix is no longer extendable to a Hadamard matrix then one flips the last entry added. This new matrix will be extendable to a Hadamard matrix since there are only two possibilities for each entry, and following this procedure one will generate a Hadamard matrix with only n^2 queries.

The complexity of Problem 2.2 is unknown. Indeed, there seems to be no previous work attempting to answer it. We are not aware of any work even formally stating the problem as a decision problem as we have done. Clearly Problem 2.2 is in NP, since an affirmative answer can easily be verified by an actual assignment of ± 1 to the empty entries which produces a Hadamard matrix.

There is also no known polynomial time algorithm for solving Problem 2.2. It would be a phenomenal development if someone did find such an algorithm, since currently all of the known methods which can provably construct Hadamard matrices in polynomial time only work in very specific orders. The running times of the methods used for constructing Hadamard matrices of order n in practice can only be provably bounded by an exponential in n. Even with the heuristics used they are not expected to run in polynomial time. Additionally, they focus on matrices of very special forms making them useless for answering Problem 2.2. The best known algorithm for provably answering Problem 2.2 uses brute-force search to try all possible ways of filling in the empty entries. There are $O\left(2^{n^2}\right)$ possibilities for this and checking if a matrix is Hadamard can be done using $O(n^3)$ arithmetic operations so the brute-force algorithm uses $O\left(n^3 2^{n^2}\right)$ arithmetic operations.

It is straightforward to see that Problem 2.2 is equivalent to solving certain very specific quadratic Diophantine systems where the variables are restricted to lie in $\{\pm 1\}$, in particular, Diophantine systems containing the variables h_{ij} for $1 \leq i, j \leq n$ and the constraints

$$h_{i1}h_{j1} + \cdots + h_{in}h_{jn} = 0 \qquad \text{for } i \neq j,$$

possibly along with some of the form $h_{ij} = 1$ or $h_{ij} = -1$. The complexity of *general* quadratic Diophantine equations has been studied and they are known to be NP-complete [Manders and Adleman, 1976].

One might think that the restriction that variables take on at most two values is severe enough that Diophantine solving over such systems is not NP-complete, but this is not the case. The book [Gopalakrishnan, 2006] gives a simple reduction from the problem of solving SAT to the problem of solving Diophantine equations whose variables are restricted to lie in $\{0, 1\}$, showing that the latter problem is NP-hard. It is NP-complete since it is also in NP, the actual solution being a certificate of an affirmative answer.

The reduction mentioned above is to Diophantine systems which are not necessarily quadratic. However, it is also known [Skolem, 1938] that the solvability of any Diophantine equation can be reduced to the solvability of a system of quadratic Diophantine equations. Also, the reduction is to systems with $\{0, 1\}$ variables instead of $\{\pm 1\}$ variables, but this is not an issue as the solvability of $\{0, 1\}$-systems reduces to the solvability of $\{\pm 1\}$-systems by replacing each variable x in the system with $(x + 1)/2$.

In summary, we can state the following result:

Theorem 2.2. *The problem of deciding whether a system of quadratic Diophantine equations whose variables lie in* $\{\pm 1\}$ *has a solution is NP-complete.*

It should be stressed that this does not show that Problem 2.2 is NP-complete, as the quadratic Diophantine equations which arise in Problem 2.2 are of a very specific form which may potentially be easier to solve than general quadratic Diophantine equations. Theorem 2.2 at least tells us that the equations being quadratic and having variables lying in $\{\pm 1\}$ cannot be used to rule out the problem from being NP-complete.

If a statement is false, that's the
worst thing you can say about it.
Paul Graham

Chapter 3

Computation of Williamson Matrices

As recounted in the introduction, the search for Williamson matrices has a long history and they have been studied by many mathematicians, who have tried to construct them in as many orders as possible. In fact, the mathematician Richard Turyn constructed an infinite family of Williamson matrices [Turyn, 1972]. However, his construction only applies to orders n such that $2n - 1$ is a prime power congruent to 1 (mod 4).

As described in Chapter 2.3.1, Williamson matrices of order n are used to construct Hadamard matrices of order $4n$. In Turyn's 1972 paper cited above he makes the following remarks:

> It has been conjectured that an[1] Hadamard matrix of this type might exist of every order $4t$, at least for t odd.

However, this conjecture was shown to be false in [Đoković, 1993] when it was demonstrated with the help of a computer search that no Williamson matrices exist for order $n = 35$. Đoković remarked that 35 was

> . . . the first odd integer, found so far, with this property.

Williamson matrices of even order were not studied until recently [Bright et al., 2016b], although *Williamson-type matrices* (a generalization of Williamson matrices defined in a similar way) of even order have been studied [Wallis, 1974]. The work [Bright et al., 2016b] constructed Williamson matrices in all orders up to 35, including the even orders. This

[1] This quote uses the French pronunciation of Hadamard.

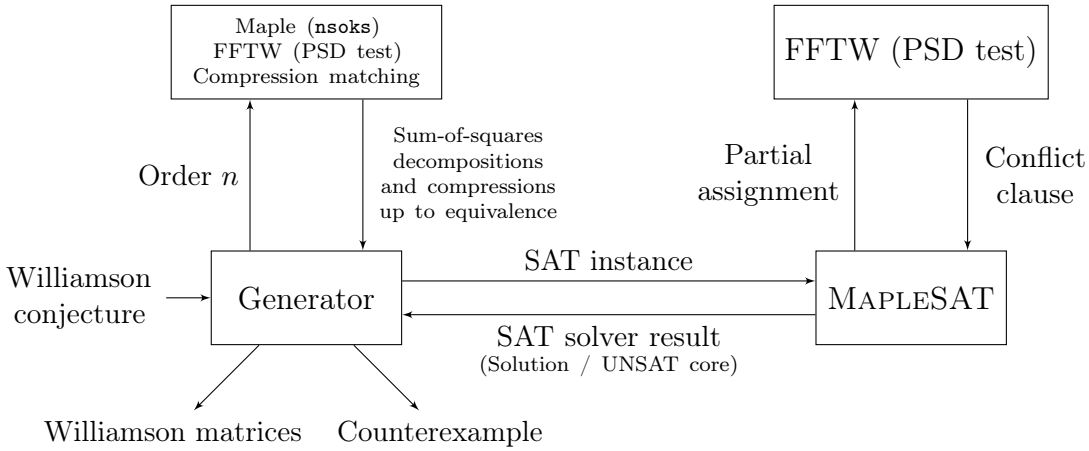

Figure 3.1: Outline of the architecture of MathCheck2 as applied to the Williamson conjecture (that a Williamson matrix of order n exists for all n).

finally showed that 35 is indeed the smallest integer with the property that Williamson matrices do not exist in that order. In this chapter we explain the work done to obtain this result and report on additional results on Williamson matrices of even order. In particular, we provide for the first time a count of the number of Williamson matrices which exist (up to equivalence as described in Section 3.1.2) in all even orders less than 45.

An outline of the MathCheck2 system applied to the case study of computing Williamson matrices is given in Figure 3.1 (this contains the same outline as in Figure 2.4 except is specific to this case study). The techniques and tools referred to in Figure 3.1 will be described throughout this chapter. The work done in this chapter was done in collaboration with Vijay Ganesh, Albert Heinle, Ilias Kotsireas, Saeed Nejati, Krzysztof Czarnecki, and Chunxiao Li.

3.1 Mathematical preliminaries

Williamson matrices were already defined in Section 2.3.1, where it was pointed out that the matrices could really be viewed as if they were sequences because of the large amount of symmetry they contain. For convenience, in this section we will use a definition of Williamson matrices which is equivalent to the previous definition but it makes no reference to matrices at all. Because of this we will refer to the computation of Williamson *sequences* for the remainder of this chapter.

3.1.1 Periodic autocorrelation

Before defining Williamson sequences we require the notion of *periodic autocorrelation*.

Definition 3.1. The *periodic autocorrelation function* of $A = [a_0, \ldots, a_{n-1}]$ is the periodic function given by

$$\mathrm{PAF}_A(s) \coloneqq \sum_{k=0}^{n-1} a_k a_{(k+s) \bmod n} \qquad \text{for } s \in \mathbb{Z}.$$

Note that this function is symmetric and periodic. In particular, one has $\mathrm{PAF}_A(s) = \mathrm{PAF}_A(n-s)$ and $\mathrm{PAF}_A(s) = \mathrm{PAF}_A(s \bmod n)$ (see [Kotsireas, 2013a]), so that the PAF_A only needs to be computed for $s = 0, \ldots, \lfloor n/2 \rfloor$. The other values can be computed through symmetry and periodicity.

Many other combinatorial conjectures can also be stated in terms of the PAF. For example, the chapter "Algorithms and Metaheuristics for Combinatorial Matrices" in the *Handbook of Combinatorial Optimization* [Kotsireas, 2013a] contains a list of at least eleven different types of combinatorial objects which are simply defined using an autocorrelation function. If a conjecture can be recast in such a way it can be beneficial to do so since it can allow the reusing of code written for dealing with PAF values.

The definition of Williamson sequences is straightforward now that the PAF has been defined.

Definition 3.2. Four symmetric sequences A, B, C, $D \in \{\pm 1\}^n$ are known as *Williamson sequences* if they satisfy the equations

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s) = 0 \tag{3.1}$$

for $s = 1, \ldots, \lfloor n/2 \rfloor$.

To see that this is an equivalent definition to the one for Williamson matrices, note that symmetric sequences generate symmetric circulant matrices. This is accomplished by setting the first row of the matrix to be equal to the sequence, and every other row of the matrix to be equal to the previous row shifted down and to the right by one position (with a wrap-around). In other words, there is an equivalence between symmetric circulant matrices and symmetric sequences.

There is also an equivalence between condition 3 of Theorem 2.1 which is the matrix equation

$$A^2 + B^2 + C^2 + D^2 = 4nI_n$$

and the sum-of-PAFs equation (3.1). To see this, note that the sth entry in the first row of $A^2 + B^2 + C^2 + D^2$ is exactly

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s).$$

Condition 3 requires that this entry should be $4n$ when $s = 0$ and that it should be 0 when $s = 1, \ldots, n-1$. The sum of the $\mathrm{PAF}_X(0)$s being $4n$ is not actually relevant to the definition because it is guaranteed to hold as $\mathrm{PAF}_X(0) = \sum_{k=0}^{n-1}(\pm 1)^2 = n$ for any $X \in \{\pm 1\}^n$.

Additionally, the first row of $A^2 + B^2 + C^2 + D^2$ will be symmetric as each matrix in the sum has a symmetric first row. Thus ensuring that

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s) = 0 \quad \text{for } s = 1, \ldots, \lfloor n/2 \rfloor \tag{3.2}$$

guarantees that every entry in the first row of $A^2 + B^2 + C^2 + D^2$ is 0 besides the first. Since $A^2 + B^2 + C^2 + D^2$ will also be circulant, ensuring that (3.2) holds will ensure condition 3 of Theorem 2.1.

3.1.2 Williamson equivalences

There are three types of operations which can be applied to a Williamson sequence quadruple (A, B, C, D) to produce another Williamson sequence quadruple (A', B', C', D'). If (A', B', C', D') can be generated from (A, B, C, D) by a sequence of such operations then we say that these two Williamson sequences are *equivalent*. For our purposes, generating just one of the equivalent quadruples will be sufficient, so we impose additional constraints on the search space to cut down on extraneous equivalent solutions and hence speed up the search. Such constraints are often referred to as *symmetry breaking* constraints.

1. Ordering

Note that the conditions on the Williamson sequences are symmetric with respect to A, B, C, and D. In other words, those four sequences can be permuted amongst themselves and they will still form a valid Williamson sequence quadruple. Given this, we enforce the constraint that

$$|\mathrm{rowsum}(A)| \leq |\mathrm{rowsum}(B)| \leq |\mathrm{rowsum}(C)| \leq |\mathrm{rowsum}(D)|,$$

where $\mathrm{rowsum}(X)$ denotes the sum of the entries of X. The sequences of any quadruple (A, B, C, D) can be permuted so that this condition holds.

2. Negation

The entries in the sequences A, B, C, or D can be negated and the sequences will still form a valid Williamson sequence quadruple. Given this, we do not need to try both possibilities for the sign of the rowsum of A, B, C, and D. For example, we can choose to enforce that the rowsum of each of sequence is nonnegative (when the rowsum is 0 this condition removes no equivalences so we can instead enforce that the sequence's first entry is positive). This is what we do when n is even, but when n is odd there is an alternative strategy which is useful. In such a case we choose the signs so that they satisfy $\text{rowsum}(X) \equiv 1 \pmod 4$ for $X \in \{A, B, C, D\}$. In this case, the following lemma allows us to fix the first entries in any Williamson sequence (A, B, C, D).

Lemma 3.1. *If (A, B, C, D) is a Williamson sequence of order $n \equiv 1 \pmod 4$ whose rowsums are all congruent to $1 \pmod 4$ then $a_0 = b_0 = c_0 = d_0 = 1$.*

Proof. By symmetry, one has that $\text{rowsum}(A) = a_0 + 2\sum_{k=1}^{(n-1)/2} a_k$ and

$$\sum_{k=1}^{(n-1)/2} a_k \equiv \sum_{k=1}^{(n-1)/2} 1 \equiv (n-1)/2 \pmod 2,$$

so $2\sum_{k=1}^{(n-1)/2} a_k \equiv n - 1 \pmod 4$. Thus

$$a_0 \equiv \text{rowsum}(A) - (n-1) \equiv 1 \pmod 4$$

and then $a_0 \in \{\pm 1\}$ implies $a_0 = 1$, and the same identity holds for each of b_0, c_0, and d_0. □

By the same argument, if $n \equiv 3 \pmod 4$ then one can show that $a_0 = b_0 = c_0 = d_0 = -1$. Furthermore, the following result of Williamson allows us to fix the value of $a_k b_k c_k d_k$ for all k.

Theorem 3.1 (cf. [Williamson, 1944])**.** *If (A, B, C, D) is a Williamson sequence of odd order n then we have $a_k b_k c_k d_k = -a_0 b_0 c_0 d_0$ for all $1 \le k \le (n-1)/2$.*

In summary, when n is odd we enforce the convention that all rowsums are taken to be equivalent to $1 \pmod 4$ so that $a_k b_k c_k d_k = -1$ for all $1 \le k \le (n-1)/2$ and

$$a_0 = b_0 = c_0 = d_0 = \begin{cases} 1 & \text{if } n \equiv 1 \pmod 4, \\ -1 & \text{if } n \equiv 3 \pmod 4. \end{cases}$$

3. Permuting entries

We can reorder the entries of the sequences A, B, C, and D with the rule $a_i \mapsto a_{ki \bmod n}$ where k is any number coprime with n, and similarly for b_i, c_i, d_i (the *same* reordering must be applied to each sequence for the result to still be equivalent). Such a rule effectively applies an automorphism of $\mathbb{Z}_n$ to the Williamson sequence quadruple.

Example 3.1. The sequence $A \coloneqq [1, -1, 1, 1, 1, 1, -1]$ has order 7 and can be permuted to $[1, 1, -1, 1, 1, -1, 1]$ using the rule $a_i \mapsto a_{2i \bmod 7}$ and to $[1, 1, 1, -1, -1, 1, 1]$ using the rule $a_i \mapsto a_{3i \bmod 7}$. The rules $a_i \mapsto a_{ki \bmod 7}$ for $k \in \{4, 5, 6\}$ produce exactly the same permutations. In general the symmetry property implies that $a_i \mapsto a_{ki \bmod n}$ is equivalent to $a_i \mapsto a_{(n-k)i \bmod n}$.

3.1.3 Power spectral density

Because the search space for Williamson sequences is so large, it is advantageous to describe properties which any Williamson sequence must satisfy. Such properties can speed up a search by significantly reducing the size of the necessary space. One such set of properties for Williamson sequences is derived using the *discrete Fourier transform* from Fourier analysis, i.e., the periodic function $\mathrm{DFT}_A(s) \coloneqq \sum_{k=0}^{n-1} a_k \omega^{ks}$ for a sequence $A = [a_0, a_1, \ldots, a_{n-1}]$, where $s \in \mathbb{Z}$ and $\omega \coloneqq e^{2\pi i/n}$ is a primitive nth root of unity. Because $\omega^{ks} = \omega^{ks \bmod n}$ one has that $\mathrm{DFT}_A(s) = \mathrm{DFT}_A(s \bmod n)$, so that only n values of DFT_A need to be computed and the remaining values are determined through periodicity. In fact, when A consists of real entries, it is well-known that $\mathrm{DFT}_A(s)$ is equal to the complex conjugate of $\mathrm{DFT}_A(n - s)$. Hence only $\left\lceil \frac{n+1}{2} \right\rceil$ values of DFT_A need to be computed.

The *power spectral density* of the sequence A is given by

$$\mathrm{PSD}_A(s) \coloneqq |\mathrm{DFT}_A(s)|^2 \qquad \text{for } s \in \mathbb{Z}.$$

Example 3.2. Let $\omega = e^{2\pi i/5}$. The discrete Fourier transform and power spectral density of the sequence $A = [1, 1, -1, -1, 1]$ are given by:

$$\begin{aligned}
\mathrm{DFT}_A(0) &= 1 + 1 - 1 - 1 + 1 = 1 && \mathrm{PSD}_A(0) = 1 \\
\mathrm{DFT}_A(1) &= 1 + \omega - \omega^2 - \omega^3 + \omega^4 \approx 3.236 && \mathrm{PSD}_A(1) \approx 10.472 \\
\mathrm{DFT}_A(2) &= 1 + \omega^2 - \omega^4 - \omega^6 + \omega^8 \\
&= 1 - \omega + \omega^2 + \omega^3 - \omega^4 \approx -1.236 && \mathrm{PSD}_A(2) \approx 1.528
\end{aligned}$$

3.1.4 Compression

Because the size of the space in which a combinatorial object lies is generally exponential in the size of the object, it is advantageous to instead search for *smaller* objects when possible. Recent theorems on so-called "compressed" sequences allow us to do that when searching for Williamson sequences.

Definition 3.3 (cf. [Đoković and Kotsireas, 2015])**.** Let $A = [a_0, a_1, \ldots, a_{n-1}]$ be a sequence of length $n = dm$ and set

$$a_j^{(d)} = a_j + a_{j+d} + \cdots + a_{j+(m-1)d}, \qquad j = 0, \ldots, d-1.$$

Then we say that the sequence $A^{(d)} = [a_0^{(d)}, a_1^{(d)}, \ldots, a_{d-1}^{(d)}]$ is the *m-compression* of A.

Example 3.3. Consider the sequence $A = [1, 1, -1, -1, -1, 1, -1, 1, 1, -1, 1, -1, -1, -1, 1]$ of length 15. Since 15 factors uniquely as $15 = 3 \cdot 5$, there are two non-trivial choices for the tuple (d, m), namely $(d, m) = (3, 5)$ and $(d, m) = (5, 3)$. The sequence A then has the two compressions

$$A^{(3)} = [-3, 1, 1] \qquad \text{and} \qquad A^{(5)} = [3, -1, -1, -1, -1].$$

As we will see, the space of the compressed sequences that we are interested in will be much smaller than the space of the uncompressed sequences. What makes compressed sequences especially useful is that we can derive conditions that the compressed sequences must satisfy using our known conditions on the uncompressed sequences. To do this, we utilize the following theorem which is a special case of a result from [Đoković and Kotsireas, 2015].

Theorem 3.2. *Let A, B, C, and D be sequences of length $n = dm$ which satisfy*

$$\mathrm{PAF}_A(s) + \mathrm{PAF}_B(s) + \mathrm{PAF}_C(s) + \mathrm{PAF}_D(s) = \begin{cases} 4n, & s = 0 \\ 0, & 1 \le s < \mathrm{len}(A). \end{cases} \tag{3.3}$$

Then for all $s \in \mathbb{Z}$ we have

$$\mathrm{PSD}_A(s) + \mathrm{PSD}_B(s) + \mathrm{PSD}_C(s) + \mathrm{PSD}_D(s) = 4n. \tag{3.4}$$

Furthermore, both (3.3) and (3.4) hold if the sequences A, B, C, D are replaced with their compressions $A^{(d)}$, $B^{(d)}$, $C^{(d)}$, $D^{(d)}$.

Since $\mathrm{PSD}_X(s)$ is always nonnegative, equation (3.4) implies that $\mathrm{PSD}_{A^{(d)}}(s) \leq 4n$ (and similarly for B, C, D). Therefore if a candidate compressed sequence $A^{(d)}$ satisfies $\mathrm{PSD}_{A^{(d)}}(s) > 4n$ for some $s \in \mathbb{Z}$ then we know that the uncompressed sequence A can never be part of a Williamson sequence and we have the following Corollary.

Corollary 3.1. *If X is a sequence which satisfies $\mathrm{PSD}_X(s) > 4n$ for some $s \in \mathbb{Z}$ then X or any uncompression of X is not part of a Williamson sequence.*

Similarly, if $\mathrm{PSD}_X(s) + \mathrm{PSD}_Y(s) > 4n$ for some $s \in \mathbb{Z}$ then X and Y cannot both occur in the same Williamson sequence.

Useful compression properties

Lastly, we derive some properties that the compressed sequences which arise in our context must satisfy. For a concrete example, note that the compressed sequences of Example 3.3 fulfill these properties.

Lemma 3.2. *If A is a sequence of length $n = dm$ with ± 1 entries, then the entries $a_i^{(d)}$, $i \in \{0, \ldots, d-1\}$, have absolute value at most m and $a_i^{(d)} \equiv m \pmod 2$.*

Proof. For all $0 \leq j < d$ we have, using the triangle inequality, that

$$\left|a_j^{(d)}\right| = \left|\sum_{k=0}^{m-1} a_{j+kd}\right| \leq \sum_{k=0}^{m-1} |a_{j+kd}| = m.$$

Additionally, $a_j^{(d)} \equiv \sum_{k=0}^{m-1} 1 \equiv m \pmod 2$ since $a_{j+kd} \equiv 1 \pmod 2$. □

In the course of our research we discovered the following useful property of compressed sequences which significantly reduces the number of possible compressions of any Williamson sequence.

Lemma 3.3. *The compression of a symmetric sequence is also symmetric.*

Proof. Suppose that A is a symmetric sequence of length $n = dm$. We want to show that $a_j^{(d)} = a_{d-j}^{(d)}$ for $j = 1, \ldots, d-1$. By reversing the sum defining $a_j^{(d)}$ and then using the fact that $n = md$, we have

$$\sum_{k=0}^{m-1} a_{j+kd} = \sum_{k=0}^{m-1} a_{j+(m-1-k)d} = \sum_{k=0}^{m-1} a_{n+j-d(k+1)}.$$

By the symmetry of A, $a_{n+j-d(k+1)} = a_{d(k+1)-j}$, which equals a_{d-j+dk}. The sum in question is therefore equal to $\sum_{k=0}^{m-1} a_{d-j+dk} = a_{d-j}^{(d)}$, as required. □

3.2 SAT encoding

An attractive property of Williamson sequences when encoding them in a SAT context is that each of their entries is one of two possible values, namely ± 1. We choose the encoding that 1 is represented by true and -1 is represented by false. We call this the *Boolean value* or BV encoding. Under this encoding, the multiplication function of two x, $y \in \{\pm 1\}$ becomes the XNOR function in the SAT setting, i.e., $\mathrm{BV}(x \cdot y) = \mathrm{XNOR}(\mathrm{BV}(x), \mathrm{BV}(y))$.

If the intention is clear from the context then we may just use the variable names

$$a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, c_0, \ldots, c_{n-1}, d_0, \ldots, d_{n-1}$$

to refer to either the ± 1 entries of a Williamson sequence or to the Boolean variables in our SAT instances which encode those ± 1 entries.

3.2.1 Encoding the PAF values

Recall that the periodic autocorrelation function is defined by

$$\mathrm{PAF}_A(s) \coloneqq \sum_{k=0}^{n-1} a_k a_{(k+s) \bmod n}.$$

In a SAT context the products in the summand $a_k a_{(k+s) \bmod n}$ can be encoded by defining the new 'product' variables

$$P^A_{k,s} \coloneqq \mathrm{XNOR}(a_k, a_{(k+s) \bmod n}).$$

Note that since Williamson sequences are symmetric there are only $\lfloor (n+1)/2 \rfloor$ distinct variable choices for each a_k and $a_{(k+s) \bmod n}$. In other words, this definition requires the introduction of approximately $n^2/4$ new variables for each sequence A, B, C, and D.

Once the product variables have been defined, we want to find some way of encoding the PAF condition

$$\sum_{k=0}^{n-1} \left(P^A_{k,s} + P^B_{k,s} + P^C_{k,s} + P^D_{k,s} \right) = 0,$$

where we are thinking of the product variables as ± 1 values; since the sum of the variables is zero there must be an equal number of $+1$s and -1s in this sum. Since there are $4n$ terms in the sum there must be exactly $2n$ variables with the value $+1$ and $2n$ variables

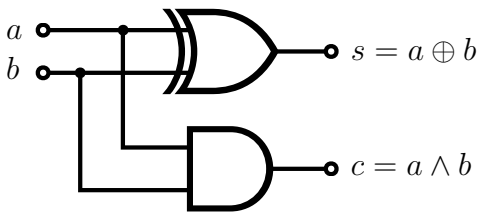

Figure 3.2: A schematic diagram of a half adder.

with the value -1. In other words, if we instead think of the product variables as Boolean values then the PAF condition is equivalent to the cardinality constraint

$$\left| \left\{ x \in \bigcup_{k=0}^{n-1} \left\{ P^A_{k,s}, P^B_{k,s}, P^C_{k,s}, P^D_{k,s} \right\} : x \text{ true} \right\} \right| = 2n. \tag{3.5}$$

This condition may be encoded in a SAT instance using what is known as the *binary adder method*.

A *binary adder* is a circuit which accepts a number of variables as input and outputs a count of how many input variables were set to true. If there are n variables of input then there will be $\lfloor \log_2 n \rfloor + 1$ output variables and they will encode in binary the number of variables set to true. A digram of a 2-bit binary adder, also known as a *half adder*, is shown in Figure 3.2. A 3-bit binary adder (or a *full adder*) is the same except it has 3 input variables.

Repeatedly using full and half adders one can construct arbitrary n-bit adders and we use such a network to encode the constraint (3.5). The input to the binary adder network will be the $4n$ variables $\bigcup_{k=0}^{n-1} \left\{ P^A_{k,s}, P^B_{k,s}, P^C_{k,s}, P^D_{k,s} \right\}$ and the output to the network will be $\lfloor \log_2 4n \rfloor + 1$ variables. To ensure that the cardinality constraint is satisfied, i.e., to ensure that exactly $2n$ input variables are true, we explicitly assign values to the output variables so that they encode $2n$ in binary. As an example, if we wanted to ensure that 2 of the inputs were true in Figure 3.2 (i.e., both of them) then we would set c to true and s to false, since 2 encoded in binary is $[1, 0]$ or $[\text{true}, \text{false}]$.

3.3 Techniques for improved efficency

We call the SAT encoding that we have just described the "naive" encoding since it is basically just a straight translation of the definition of Williamson sequences into a SAT

context. The largest Williamson sequence we were able to find using the naive encoding and an off-the-shelf SAT solver with a timeout of 24 hours had order 31 (see Section 3.6.1). However, in this section we describe three techniques which allow us to prune the size of the search space and increase the efficiency of our search.

3.3.1 Sum-of-squares decomposition

As a special case of compression, consider what happens when $d = 1$ and $m = n$. In this case, the compression of A is a sequence with a single entry whose value is $\sum_{k=0}^{n-1} a_k = \text{rowsum}(A)$. If A, B, C, and D are $\{\pm 1\}$-sequences which satisfy the conditions of Theorem 3.2, then the theorem applied to this m-compression says that

$$\text{PAF}_{A^{(1)}}(0) + \text{PAF}_{B^{(1)}}(0) + \text{PAF}_{C^{(1)}}(0) + \text{PAF}_{D^{(1)}}(0) = 4n.$$

Since $\text{PAF}_{[x]}(0) = x^2$ by definition, this simplifies to

$$\text{rowsum}(A)^2 + \text{rowsum}(B)^2 + \text{rowsum}(C)^2 + \text{rowsum}(D)^2 = 4n.$$

Additionally, each rowsum must have the same parity as n by Lemma 3.2.

In other words, the rowsums of the sequences A, B, C, and D decompose $4n$ into the sum of four perfect squares whose parity matches the parity of n. Since there are usually only a few ways of writing $4n$ as a sum of four perfect squares this severely limits the number of sequences which could satisfy the hypotheses of Theorem 3.2. Furthermore, some computer algebra systems contain functions for explicitly computing what the possible decompositions are (e.g., `PowersRepresentations` in MATHEMATICA and `nsoks` by Joe Riel of Maplesoft [Riel, 2006]). We can query such CAS functions to determine all possible values that the rowsums of A, B, C, and D could possibly take. For example, when $n = 35$ we find that there are exactly three ways to write $4n$ as a sum of four positive odd squares in ascending order, namely,

$$1^2 + 3^2 + 3^2 + 11^2 = 1^2 + 3^2 + 7^2 + 9^2 = 3^2 + 5^2 + 5^2 + 9^2 = 4 \cdot 35.$$

As described in Section 3.1.2, any Williamson quadruple is equivalent to another quadruple whose rowsum sum-of-squares decomposition

$$\text{rowsum}(A)^2 + \text{rowsum}(B)^2 + \text{rowsum}(C)^2 + \text{rowsum}(D)^2$$

is of one of the above three types.

This is where the power of the SAT+CAS combination begins to be seen, because although it is necessarily true that the rowsums of all Williamson sequences will provide a sum-of-squares decomposition of $4n$, it is unlikely that the SAT solver would discern this on its own. After all, the SAT solver does not have any knowledge of the mathematical background of the problem domain. The main drawback of translating such problems into SAT is that the problem context is lost when everything is replaced with Boolean constraints. This is where CAS knowledge can be extremely useful as the CAS can determine all possible sum-of-squares decompositions for a given order, and that information can be provided to the SAT solver to help guide its search.

For example, suppose we are searching for Williamson sequences whose rowsum sum-of-squares decomposition is the first type given above, $1^2 + 3^2 + 3^2 + 11^2$. Then we have that

$$\text{rowsum}(A) = 1, \quad \text{rowsum}(B) = 3, \quad \text{rowsum}(C) = 3, \quad \text{rowsum}(D) = 11.$$

Similarly to the addition encoding from Section 3.2.1 this information can be encoded using four binary adder networks, one with each of the variables in A, B, C, and D. In fact, as a small performance gain we can give variables which appear twice in the sequences (due to symmetry) a weight of 2 in the adder network. For example, when n is odd we have $\text{rowsum}(A) = a_0 + 2\sum_{k=1}^{(n-1)/2} a_k$ so the variables a_k for $k = 1, \ldots, (n-1)/2$ can be given a weight of 2.

3.3.2 Divide-and-conquer via compression

Because each instance can take a significant amount of time to solve, it is beneficial to divide instances into multiple partitions, each instance encoding a subset of the search space. Not only does this allow us to use parallelism, it also allows us another opportunity to use the domain-specific knowledge in a CAS. In certain cases the functionality provided by a CAS can immediately show that an instance can be ignored.

In our case, we found that an effective splitting method was to split by compressions, i.e., to have each instance contain one possibility of the compressions of A, B, C, and D. To do this, we first need to know all possible compressions of A, B, C, and D. These can be generated by applying Lemmas 3.2 and 3.3. For example, when $n = 35$ and $d = 5$ (so $m = 7$) there are 28 possible m-compressions of A with $\text{rowsum}(A) = 1$. In addition, we can use the filtering condition in Corollary 3.1 to discard some of these sequences as those which have a PSD value large enough that they can never be the compression of a Williamson sequence. In fact, only 12 of the 28 possibilities for $A^{(5)}$ satisfy $\text{PSD}_{A^{(5)}}(s) \leq 4n$

$$\begin{array}{cccc} [-7,1,3,3,1] & [-7,3,1,1,3] & [-3,-1,3,3,-1] & [-3,1,1,1,1] \\ [-3,3,-1,-1,3] & [1,-3,3,3,-3] & [1,-1,1,1,-1] & [1,1,-1,-1,1] \\ [1,3,-3,-3,3] & [5,-3,1,1,-3] & [5,-1,-1,-1,-1] & [5,1,-3,-3,1] \end{array}$$

Figure 3.3: Possible 7-compressions of symmetric $\{\pm 1\}$-sequences of order 35 with a rowsum of 1 and which satisfy Lemmas 3.2, 3.3, and Corollary 3.1.

for all $s \in \mathbb{Z}$; these sequences are given in Figure 3.3. For efficiency, the calculation of the PSD values was performed using the C library FFTW [Frigo and Johnson, 2005] instead of querying a CAS.

There are also 12 possible 7-compressions for each of B, C, and D with $\text{rowsum}(B) = \text{rowsum}(C) = 3$ and $\text{rowsum}(D) = 11$. (As previously mentioned, in practice we actually use rowsums of -3 and -11 so that Lemma 3.1 applies.) Thus there are 12^4 total instances which would need to be generated for this selection of rowsums. However, only 80 of them satisfy the sum-of-PSDs condition in Theorem 3.2. Furthermore, only 23 of those 80 are pairwise inequivalent; these sequences are given in Figure 3.4. It suffices to only consider these 23 pairwise inequivalent 7-compressions, as if one 7-compression uncompresses to a Williamson sequence (A, B, C, D) then any other equivalent 7-compression will uncompress to a Williamson sequence equivalent to (A, B, C, D). (More precisely, if σ is an equivalence operation then $\sigma(A^{(5)}, B^{(5)}, C^{(5)}, D^{(5)})$ will uncompress to $\sigma(A, B, C, D)$.) Since we are only interested in finding Williamson sequences up to equivalence it suffices to use just one 7-compression from each equivalence class.

Using two compression factors

If n has two nontrivial divisors m and d then we can find all possible m- and d-compressions of A, B, C, and D. In this case, each instance can set both the m- and d-compression for each of A, B, C, and D. The downside of using two compressions in each instance is that it leads to many more possible instances necessary to check. Since one does not know in advance which (if any) m- and d-compression will lead to a solution, one has to check all possible ways of combining the m- and d-compressions.

In this case removing equivalences from *both* the m- and d-compressions does not work. As an example, say that (A, B, C, D) is a Williamson sequence which has the compression $(A^{(d)}, B^{(d)}, C^{(d)}, D^{(d)})$ in the equivalence class Γ and the compression $(A^{(m)}, B^{(m)}, C^{(m)}, D^{(m)})$ in the equivalence class Δ. Then by applying a suitable equivalence operation to (A, B, C, D) there is a Williamson sequence which m-compresses to any member of Γ, as well as

$A^{(5)}$	$B^{(5)}$	$C^{(5)}$	$D^{(5)}$
$[5,1,-3,-3,1]$	$[-3,3,-3,-3,3]$	$[-3,1,-1,-1,1]$	$[1,-3,-3,-3,-3]$
$[5,1,-3,-3,1]$	$[-3,3,-3,-3,3]$	$[1,-3,1,1,-3]$	$[-3,-3,-1,-1,-3]$
$[5,1,-3,-3,1]$	$[5,-3,-1,-1,-3]$	$[1,-3,1,1,-3]$	$[-3,-1,-3,-3,-1]$
$[5,1,-3,-3,1]$	$[-3,1,-1,-1,1]$	$[-3,-1,1,1,-1]$	$[-7,1,-3,-3,1]$
$[1,3,-3,-3,3]$	$[1,1,-3,-3,1]$	$[5,-3,-1,-1,-3]$	$[1,-3,-3,-3,-3]$
$[1,3,-3,-3,3]$	$[1,1,-3,-3,1]$	$[1,-3,1,1,-3]$	$[-3,1,-5,-5,1]$
$[1,3,-3,-3,3]$	$[-3,1,-1,-1,1]$	$[-7,1,1,1,1]$	$[1,-3,-3,-3,-3]$
$[1,3,-3,-3,3]$	$[1,-3,1,1,-3]$	$[-7,1,1,1,1]$	$[-3,-3,-1,-1,-3]$
$[5,-1,-1,-1,-1]$	$[1,-1,-1,-1,-1]$	$[-7,1,1,1,1]$	$[-7,-1,-1,-1,-1]$
$[1,1,-1,-1,1]$	$[5,-1,-3,-3,-1]$	$[1,1,-3,-3,1]$	$[-7,1,-3,-3,1]$
$[1,1,-1,-1,1]$	$[1,1,-3,-3,1]$	$[1,1,-3,-3,1]$	$[5,-5,-3,-3,-5]$
$[1,1,-1,-1,1]$	$[-3,3,-3,-3,3]$	$[1,-3,1,1,-3]$	$[-7,-3,1,1,-3]$
$[1,1,-1,-1,1]$	$[1,-3,1,1,-3]$	$[-7,1,1,1,1]$	$[-3,-5,1,1,-5]$
$[1,1,-1,-1,1]$	$[-7,1,1,1,1]$	$[-7,1,1,1,1]$	$[-3,-1,-3,-3,-1]$
$[-3,3,-1,-1,3]$	$[1,3,-5,-5,3]$	$[1,-1,-1,-1,-1]$	$[1,-3,-3,-3,-3]$
$[-3,3,-1,-1,3]$	$[1,-1,-1,-1,-1]$	$[-7,1,1,1,1]$	$[1,-1,-5,-5,-1]$
$[-3,1,1,1,1]$	$[1,3,-5,-5,3]$	$[1,-1,-1,-1,-1]$	$[1,-5,-1,-1,-5]$
$[-3,1,1,1,1]$	$[5,-1,-3,-3,-1]$	$[5,-3,-1,-1,-3]$	$[1,-3,-3,-3,-3]$
$[-3,1,1,1,1]$	$[5,-1,-3,-3,-1]$	$[1,-3,1,1,-3]$	$[-3,1,-5,-5,1]$
$[-3,1,1,1,1]$	$[-3,3,-3,-3,3]$	$[-3,1,-1,-1,1]$	$[-7,-3,1,1,-3]$
$[-3,1,1,1,1]$	$[-3,3,-3,-3,3]$	$[-3,-3,3,3,-3]$	$[1,-3,-3,-3,-3]$
$[-3,1,1,1,1]$	$[-3,1,-1,-1,1]$	$[-7,1,1,1,1]$	$[-3,-5,1,1,-5]$
$[-7,3,1,1,3]$	$[1,1,-3,-3,1]$	$[1,1,-3,-3,1]$	$[-3,-3,-1,-1,-3]$

Figure 3.4: Possible 7-compressions of Williamson sequences with sums-of-squares decomposition $1^2 + 3^2 + 3^2 + 11^2$ which are pairwise inequivalent.

a Williamson sequence which d-compresses to any member of Δ, but not necessarily a Williamson sequence which compresses to any member of Γ and Δ *simultaneously*. Therefore, even if we are only interested in searching for inequivalent Williamson sequences we cannot remove equivalences from *both* the m- and d-compressions because we do not know in advance which representatives from the m- and d-compressions will *simultaneously* lead to a solution. Using a set of inequivalent representatives for just one compression is fine, as in that case every representative of an equivalence class will yield a solution if just one does.

Even though using two compressions factors has these drawbacks, we still found that technique was advantageous to use when we could because it allows the encoding of much more information into each instance. This information allows the instances to be solved much faster than those generated using information from just one compression factor. The main drawback of this technique was that it could lead to a huge number of instances to generate. Because of this, we only used two compression factors when there would be a reasonable number of SAT instances generated. We always used compression by the smallest nontrivial divisor of the order n and only used compression by another nontrivial divisor of n when there would be under 10,000 instances generated.

3.3.3 UNSAT core

After using the divide-and-conquer technique one obtains a collection of instances which are almost identical. For example, the instances will contain variables which encode the rowsums of A, B, C, and D. Since there are multiple possibilities of the rowsums (as discussed in Section 3.3.1), not all instances will set those variables to the same values. However, since the instances are the same except for those variables, it is sometimes possible to use an *UNSAT core* result from one instance to learn that other instances are unsatisfiable.

Provided a master instance and a set of assumptions (variables which are set either true or false), the UNSAT core contains a subset of the assumptions which make the master instance unsatisfiable. Thus, any other instance which sets the variables in the UNSAT core in the same way must also be unsatisfiable and we do not need to make another call to a SAT solver to discover this. MapleSAT [Liang et al., 2016], based on MiniSat [Eén and Sörensson, 2004], is one SAT solver which supports the generation of UNSAT cores.

For example, our instances for $n = 35$ contained 14,483 clauses which were identical among all instances. The instances contained 3120 variables of which only 84 were given as assumptions and assigned differently in each instance.

3.4 Programmatic filtering

The paper [Ganesh et al., 2012] introduced the idea of a programmatic SAT solver. Such a SAT solver can generate conflict clauses *programmatically*, i.e., by a piece of code included with the SAT solver which is executed as the SAT solver carries out its search. Such code can be custom to the specific problem under consideration and in certain cases the clauses which are programmatically learned can dramatically increase the efficiency of the search. The usage of a programmatic SAT solver also fits nicely into the SAT+CAS paradigm, as the programmatic code can use any method to generate the learned clauses, including querying CAS functions.

In our work we used a version of MapleSAT [Liang et al., 2016] modified with the addition of a callback function which is called inside the inner loop of MapleSAT, just before the decision step when an unassigned variable is assigned a truth value. The callback function accepts the current partial assignment and if it can be determined that the partial assignment cannot be extended into a satisfying assignment then it returns a conflict clause encoding that fact. Not only can this increase the efficiency of the search it also allows for increased expressiveness, as some constraints are difficult to express as a CNF propositional formula but can be naturally expressed in a programmatic fashion.

Writing an appropriate callback function is a delicate task, as ideally one would want the learned clauses to encode something which satisfies each of the following criteria:

(a) Nontrivial; something the SAT solver would not discover on its own.

(b) Useful; something which will increase the efficiency of the search.

(c) Efficiently computable; something which can be computed quickly and with little overhead.

In our case, we found that an effective callback function was based on the PSD filtering criteria of Theorem 3.2. In detail, our programmatic filtering callback function performs the following:

1. Checks the variables defining the sequences A, B, C, and D to see which sequences, if any, have had all their entries assigned.

2. Computes $\mathrm{PSD}_X(s)$ for $s = 0, \ldots, \lfloor n/2 \rfloor$ for the sequences $X \in \{A, B, C, D\}$ which have all of their entries set.

3. If any computed $\mathrm{PSD}_X(s)$ is strictly larger than $4n$ then learn a clause discounting the variables in X from being set the way that they currently are. In other words, we learn the clause

$$\neg(x_0^{\mathrm{cur}} \wedge x_1^{\mathrm{cur}} \wedge \cdots \wedge x_{\lfloor n/2 \rfloor}^{\mathrm{cur}}) \equiv \neg x_0^{\mathrm{cur}} \vee \neg x_1^{\mathrm{cur}} \vee \cdots \vee \neg x_{\lfloor n/2 \rfloor}^{\mathrm{cur}} \tag{3.6}$$

where x_i^{cur} is the literal x_i when x_i is currently assigned to true and is the literal $\neg x_i$ when x_i is currently assigned to false.

4. Otherwise, compute $\sum_X \mathrm{PSD}_X(s)$ for $s = 0, \ldots, \lfloor n/2 \rfloor$ where the sum is over the $X \in \{A, B, C, D\}$ which have all of their entries set.

5. If any computed $\sum_X \mathrm{PSD}_X(s)$ is strictly larger than $4n$ then learn a clause discounting the variables in the sequences which have all their entries set from being set the way they currently are.

This callback function meets the three criteria we outlined above:

(a) Nontrivial; the uncustomized SAT solver has no knowledge of Theorem 3.2 or that the variables are being used to encode Williamson sequences.

(b) Useful; the clauses learned dramatically cut down the search space. As a rough estimate, if the search space has about $2^{4(n/2)}$ total assignments then learning a clause of the form (3.6) removes about $2^{3(n/2)}$ assignments from consideration.

(c) Efficiently computable; we used the C library FFTW [Frigo and Johnson, 2005] to compute the PSD values which this library has been fined-tuned to do extremely efficiently. The direct usage of C libraries is preferable to making calls to an external CAS when possible, as such calls tend to have large overhead.

3.4.1 Numerical accuracy

The library FFTW which we used to compute the PSD values uses floating point numbers and thus is inherently imprecise. The FFTW website [Frigo and Johnson, 2014] makes the following claim:

> ... floating-point arithmetic is not exact, so different FFT algorithms will give slightly different results (on the order of the numerical accuracy; typically a fractional difference of 1e−15 or so in double precision).

However, we were not able to find any *provable guarantee* on the accuracy of FFTW which makes it problematic to use for mathematical results of nonexistence. Conceivably, FFTW could produce a result with the computed value of $\mathrm{PSD}_X(s)$ larger than $4n + \epsilon$ (where ϵ is whatever tolerance we use when checking the PSD condition) but the true value of $\mathrm{PSD}_X(s)$ smaller than $4n$. In such a case, the learned clause could block a satisfying assignment and the SAT solver could return UNSAT erroneously. This is less of an issue in the cases where the SAT solver returns a satisfying assignment, since in those cases the assignment can easily be checked to satisfy the instance constraints without relying on FFTW.

To address this problem we also computed the PSD values using a method with a provable guarantee of accuracy. Note that it follows from the Wiener–Khinchin theorem (see e.g., [Ricker, 2012]) that the PSD values of X are equal to the values of the discrete Fourier transform of the PAF values of X. In other words,

$$\mathrm{PSD}_X(s) = \sum_{k=0}^{n-1} \mathrm{PAF}_X(k) e^{2\pi i k s/n} = \sum_{k=0}^{n-1} \mathrm{PAF}_X(k) \cos(2\pi k s/n), \tag{3.7}$$

the later equality following since the values of X are real. We can use this expression to evaluate $\mathrm{PSD}_X(s)$ to a provable guarantee of accuracy. First, we use arbitrary precision arithmetic to evaluate $\cos(2\pi k s/n)$ to whatever level of accuracy we desire, e.g., double precision (53 bits). Note that since $\cos(2\pi k s/n) = \cos(2\pi (k s \bmod n)/n)$ we only need to make n evaluations of the cosine function.

Once the values of $\cos(2\pi k s/n)$ are known to the required precision one can use standard error analysis techniques to get a bound on the error introduced by evaluating the summation in (3.7) using floating point arithmetic. These bounds depend on the *unit roundoff* μ of the floating point system used (e.g., in double precision one has $\mu = 2^{-53}$). As summarized in [Higham, 1993] the absolute error introduced by performing each multiplication in the summands of (3.7) will be at most

$$|\mathrm{PAF}_X(k) \cos(2\pi k s/n)| \mu \leq n\mu$$

and the absolute error introduced by computing the summation will be at most

$$\frac{(n-1)\mu}{1-(n-1)\mu} \sum_{k=1}^{n} |\mathrm{PAF}_X(k) \cos(2\pi k s/n)| \leq \frac{n^2(n-1)\mu}{1-(n-1)\mu}.$$

When $\mu = 2^{-53}$ and $n \leq 2^6$ the total absolute error will be at most 10^{-10} and thus if our PSD filtering criterion tests that the computed value of $\mathrm{PSD}_X(s)$ is larger than $4n + \epsilon$ for $\epsilon > 10^{-10}$ it guarantees that the true value of $\mathrm{PSD}_X(s)$ is larger than $4n$. The computation

of $\sum_X \mathrm{PSD}_X(s)$ introduces similar floating point errors, but these are also irrelevant if the tolerance ϵ is large enough. In our implementation we used $\epsilon := 10^{-2}$ which is multiple orders of magnitude larger than strictly necessary.

3.5 Matching method

In order to compare the SAT+CAS method with a different approach we also examine a method which proceeds by finding Williamson sequences by solving a matching problem. This method also has the advantage that it is straightforward to use it to find all solutions in the search space. One way to state the matching problem is as follows:

Problem 3.1. *Given multiple lists of vectors with integer entries, find one vector from each list such that their sum is the zero vector.*

A solution of the problem is referred to as a *matching* of the lists. That is, a matching is a set of vectors (one from each list) summing to the zero vector.

The matching problem is easily seen to be closely related to the problem of finding Williamson sequences, for we can make a list of the vectors

$$\begin{bmatrix} \mathrm{PAF}_A(1) & \mathrm{PAF}_A(2) & \cdots & \mathrm{PAF}_A(\lfloor n/2 \rfloor) \end{bmatrix}$$

for all symmetric $A \in \{\pm 1\}^n$, and similar lists for B, C, and D. Then any matching of those four lists yields a Williamson sequence by taking the $\{\pm 1\}$-sequences which were used to generate the vectors in the matching.

Of course, it is advantageous to use filtering theorems to decrease the size of the lists as much as possible before solving the matching problem. To do this, we again use the PSD criterion of Theorem 3.2 and the sums-of-squares technique of Section 3.3.1. In other words, we only populate the lists with vectors

$$\begin{bmatrix} \mathrm{PAF}_X(1) & \mathrm{PAF}_X(2) & \cdots & \mathrm{PAF}_X(\lfloor n/2 \rfloor) \end{bmatrix}$$

for X with appropriate rowsum and for which $\mathrm{PSD}_X(s) \leq 4n$ for all s. Let L_A, L_B, L_C, and L_D be the list of vectors formed in this way for some particular sums-of-squares decomposition.

The naive way of finding a matching would be to search through every member of the Cartesian product $L_A \times L_B \times L_C \times L_D$ and find those members whose vectors sum to the

zero vector. When there are not many distinct values in the entries of the vectors a more sophisticated way of running the search is to use a binning process, as we outline below.

Let $\text{vals}_k(L_X)$ be the set of distinct values in the kth entry of the vectors in L_X. To start off with, we form the set

$$\text{vals}_1(L_A) \times \text{vals}_1(L_B) \times \text{vals}_1(L_C) \times \text{vals}_1(L_D)$$

and search it for members whose four values sum to 0. If there are no such matchings then there are no matchings to the original problem either, as there are not even any entries in the first entry which sum to 0. Otherwise, for every matching (v_A, v_B, v_C, v_D) found we form the filtered lists

$$L'_X \coloneqq \{\text{the vectors in } L_X \text{ with first entry equal to } v_X\}.$$

Now we repeat the above procedure except using the second entries of the filtered lists, i.e., we form the set

$$\text{vals}_2(L'_A) \times \text{vals}_2(L'_B) \times \text{vals}(L'_C) \times \text{vals}(L'_D)$$

and search it for members whose four values sum to 0. If there are no such members then there are no matchings to the original problem whose first entries are the matching (v_A, v_B, v_C, v_D). If there was another possible matching for the first entry then we have to backtrack and try that selection of first entries as well. Otherwise, we inductively continue on in this fashion, filtering entries of $L_A \times L_B \times L_C \times L_D$ so that the first k entries of their vectors all sum to zero and then solving the matching problem on the distinct values in the $(k+1)$th entry.

To find Williamson sequences using this method it is necessary to store a mapping between the original sequences A, B, C, D and the vectors containing their PAF values. This is because the result of the matching process will be PAF vectors which sum to the zero vector, not the actual sequences which were used to generate the PAF vectors, and of course it is the latter sequences which will be the Williamson sequences we are interested in.

We implemented this method for finding Williamson sequences using a solver for the matching problem as a black box. We used code written in C++ by Chunxiao Li and publicly available on GitHub [Li, 2016]. The matching code can either stop after finding one matching or continue until finding all matchings in the space; we provide timings for both methods in Section 3.6.5. Furthermore, when constructing the lists L_A, L_B, L_C, and L_D we guarantee that every Williamson sequence of the given order will be equivalent to at least one Williamson sequence which generates a PAF vector in each of the lists. Thus, finding all matchings of the lists we construct allows us to determine the exact number of

Williamson sequences of a given order. In some cases multiple matchings will be generated by equivalent Williamson sequences, so we need to run an equivalence checking script which discards equivalent Williamson sequences before finalizing the total count.

3.6 Results and timings

In this section we provide our results and timings of each of the methods we have discussed. The timings were run on the "Shared Hierarchical Academic Research Computing Network", a high-performance computing cluster known as SHARCNET [Bauer, 2016] run by a consortium of 18 academic partners located in Ontario, Canada. Specifically, the cluster we used ran CentOS 6.7 and used 64-bit AMD Opteron processors running at 2.2 GHz. Each SAT instance was generated using the MathCheck2 system with the appropriate parameters and the instance was submitted to SHARCNET to solve by running MapleSAT.

3.6.1 Naive method results

First, we present timings for the straight SAT encoding described in Section 3.2. These SAT instances were generated by MathCheck2 without any of the techniques in Section 3.3 enabled. In Table 3.1 we give timings for both an unmodified version of MapleSAT and a version of MapleSAT with the programmatic functionality described in Section 3.4 enabled; in both cases, a timeout of 24 hours was used. Every instance which successfully returned a result in the allotted time was satisfiable.

In most cases, using the programmatic filtering was beneficial to the search. We generated cases up to order 45 and the largest order the normal MapleSAT was able to solve was 31, while the largest order the programmatic MapleSAT was able to solve was 40. Additionally, the programmatic version often ran over 10 times faster than the normal version, and in one case over 200 times faster. In a few cases the programmatic version was several times slower than the normal version, but this was not due to the extra computations in the callback function. Logging showed that the amount of time spent in the programmatic code was not a significant amount of the total time. The orders in which the programmatic version was unusually slow were those with only a few solutions and in these cases luck tends to play an increased role in the runtimes. Evidently the programmatic version searched some areas of the search space which was ignored by the normal MapleSAT because the normal version had discovered a solution before searching them.

n	Normal MapleSAT	Programmatic MapleSAT	Programmatic Speedup Factor
10	0.04	0.03	1.33
11	0.04	0.04	1.00
12	0.14	0.13	1.08
13	0.03	0.04	0.75
14	0.12	0.05	2.40
15	0.21	0.07	3.00
16	24.56	0.26	94.46
17	0.30	0.19	1.58
18	1.50	0.06	25.00
19	1.06	1.39	0.76
20	3.09	0.06	51.50
21	390.55	6.60	59.17
22	34.90	0.70	49.86
23	545.71	7.19	75.90
24	3116.93	13.72	227.18
25	591.78	42.62	13.89
26	6238.15	46.98	132.78
27	2485.84	719.32	3.46
28	6234.42	118.14	52.77
29	7053.56	25850.39	0.27
30	29881.94	441.49	67.68
31	20313.47	68538.98	0.30
32	–	3309.02	–
33	–	8549.17	–
34	–	2986.61	–
35	–	–	–
36	–	639.58	–
37	–	–	–
38	–	–	–
39	–	–	–
40	–	15835.62	–

Table 3.1: Timings in seconds for using MapleSAT to search for Williamson sequences of order $10 \leq n \leq 40$ using the naive method, both with and without programmatic filtering enabled. A '–' denotes a timeout after 24 hours. All instances returned SAT.

3.6.2 Sums-of-squares decomposition method results

Now we present timings for instances encoded using the sum-of-squares decomposition technique described in Section 3.3.1. We used MathCheck to generate instances up to order 45, and the resulting timings are given in Tables 3.2, 3.3, and 3.4. We give timings using both the normal version of MapleSAT and the version with programmatic functionality as described in Section 3.4. Again, the programmatic version generally performed better than the normal version. The largest order the normal version was able to solve was 30 and the largest order than the programmatic version was able to solve was 42.

The largest UNSAT case which was returned occurred in order 29; i.e., the nonexistence of Williamson sequences of order 35 was not able to be shown by this method alone. Our UNSAT instances tended to be harder than SAT cases of the same order, as the solver could not 'get lucky' by finding a solution and stopping before the entire space is searched. This makes UNSAT cases especially useful for comparing methods, and the programmatic version did tend to perform especially well on UNSAT cases.

3.6.3 Divide-and-conquer method results

Next, we provide timings for the divide-and-conquer technique discussed in Section 3.3.2; note that these instances also use the sum-of-squares decomposition technique and so the timing for each instance is given along with the instance's sum-of-squares decomposition. We used MathCheck to generate instances up to order 45, and the resulting timings are given in Tables 3.5 and 3.6.

Because the dividing technique splits the search space into many distinct subspaces, it is possible to use parallelization to solve the resulting instances. In our case, we used 50 cores on SHARCNET and divided the instances evenly amongst those 50 cores. In the case of SAT results, the timing indicates the time of the first core to return a SAT result. In the case of UNSAT results, the timing indicates the time it took for all cores to finish and return UNSAT.

Since our divide-and-conquer method works by splitting the search space based on possible compressions it can only be applied when there is a nontrivial factor by which to compress by, i.e., when the order is not prime. Hence, we do not provide timings for the prime orders 29, 31, 37, 41, and 43.

Again, the programmatic filtering technique was demonstrated to be useful in most cases. However, many instances finished very fast (in under a second) both with and without programmatic filtering. In cases which did not finish so fast the programmatic filtering

n	Decomposition	Normal MapleSAT	Programmatic MapleSAT	Programmatic Speedup Factor	Result
10	$0^2+0^2+2^2+6^2$	0.03	0.03	1.00	SAT
10	$2^2+2^2+4^2+4^2$	0.04	0.03	1.33	SAT
11	$1^2+3^2+3^2+5^2$	0.04	0.03	1.33	SAT
12	$0^2+4^2+4^2+4^2$	0.10	0.05	2.00	UNSAT
12	$2^2+2^2+2^2+6^2$	0.04	0.04	1.00	SAT
13	$1^2+1^2+1^2+7^2$	0.05	0.03	1.67	SAT
13	$1^2+1^2+5^2+5^2$	0.04	0.03	1.33	SAT
13	$3^2+3^2+3^2+5^2$	0.04	0.03	1.33	SAT
14	$0^2+2^2+4^2+6^2$	0.08	0.04	2.00	SAT
15	$1^2+1^2+3^2+7^2$	0.15	0.05	3.00	SAT
15	$1^2+3^2+5^2+5^2$	0.21	0.07	3.00	SAT
16	$0^2+0^2+0^2+8^2$	1.75	0.12	14.58	UNSAT
16	$4^2+4^2+4^2+4^2$	0.15	0.06	2.50	SAT
17	$1^2+3^2+3^2+7^2$	0.26	0.13	2.00	SAT
17	$3^2+3^2+5^2+5^2$	0.19	0.06	3.17	SAT
18	$0^2+0^2+6^2+6^2$	0.43	0.04	10.75	SAT
18	$0^2+2^2+2^2+8^2$	0.26	0.07	3.71	SAT
18	$2^2+4^2+4^2+6^2$	2.61	0.04	65.25	SAT
19	$1^2+1^2+5^2+7^2$	0.30	0.55	0.55	SAT
19	$1^2+5^2+5^2+5^2$	62.52	5.81	10.76	UNSAT
19	$3^2+3^2+3^2+7^2$	0.58	0.14	4.14	SAT
20	$0^2+0^2+4^2+8^2$	14.89	79.36	0.19	UNSAT
20	$2^2+2^2+6^2+6^2$	1.82	0.13	14.00	SAT

Table 3.2: Timings in seconds for using MapleSAT to search for Williamson sequences of order $10 \leq n \leq 20$ using the sum-of-squares decomposition method, both with and without programmatic filtering enabled.

n	Decomposition	Normal MapleSAT	Programmatic MapleSAT	Programmatic Speedup Factor	Result
21	$1^2+1^2+1^2+9^2$	95.13	0.22	432.41	SAT
21	$1^2+3^2+5^2+7^2$	73.27	1.46	50.18	SAT
21	$3^2+5^2+5^2+5^2$	15.69	0.83	18.90	SAT
22	$0^2+4^2+6^2+6^2$	162.70	1.02	159.51	SAT
22	$2^2+2^2+4^2+8^2$	44.39	0.22	201.77	SAT
23	$1^2+1^2+3^2+9^2$	12595.27	102.03	123.45	UNSAT
23	$3^2+3^2+5^2+7^2$	481.19	30.41	15.82	SAT
24	$0^2+4^2+4^2+8^2$	1690.09	6.36	265.74	SAT
25	$1^2+1^2+7^2+7^2$	57.29	13.29	4.31	SAT
25	$1^2+3^2+3^2+9^2$	8051.75	42.68	188.65	SAT
25	$1^2+5^2+5^2+7^2$	421.95	17.04	24.76	SAT
25	$5^2+5^2+5^2+5^2$	68.14	28.39	2.40	SAT
26	$0^2+0^2+2^2+10^2$	1685.26	19.12	88.14	SAT
26	$0^2+2^2+6^2+8^2$	2078.38	6.74	308.36	SAT
26	$4^2+4^2+6^2+6^2$	60284.93	8.86	6804.17	SAT
27	$1^2+1^2+5^2+9^2$	12997.81	44.92	289.35	SAT
27	$1^2+3^2+7^2+7^2$	32998.14	201.38	163.86	SAT
27	$3^2+3^2+3^2+9^2$	–	2103.05	–	UNSAT
27	$3^2+5^2+5^2+7^2$	4543.09	147.52	30.80	SAT
28	$2^2+2^2+2^2+10^2$	35768.54	48.03	744.71	SAT
28	$2^2+6^2+6^2+6^2$	1030.11	12.38	83.21	SAT
28	$4^2+4^2+4^2+8^2$	–	–	–	–
29	$1^2+3^2+5^2+9^2$	–	1189.22	–	SAT
29	$3^2+3^2+7^2+7^2$	–	12144.50	–	UNSAT

Table 3.3: Timings in seconds for using MapleSAT to search for Williamson sequences of order $21 \leq n \leq 29$ using the sum-of-squares decomposition method, both with and without programmatic filtering enabled. A '$-$' denotes a timeout after 24 hours.

n	Decomposition	Normal MapleSAT	Programmatic MapleSAT	Programmatic Speedup Factor	Result
30	$0^2 + 2^2 + 4^2 + 10^2$	85258.48	127.09	670.85	SAT
30	$2^2 + 4^2 + 6^2 + 8^2$	10269.38	73.21	140.27	SAT
31	$1^2 + 5^2 + 7^2 + 7^2$	–	10491.08	–	SAT
31	$5^2 + 5^2 + 5^2 + 7^2$	–	1971.16	–	SAT
32	$0^2 + 0^2 + 8^2 + 8^2$	–	100.66	–	SAT
33	$1^2 + 1^2 + 3^2 + 11^2$	–	21332.12	–	SAT
33	$1^2 + 5^2 + 5^2 + 9^2$	–	7474.67	–	SAT
33	$3^2 + 5^2 + 7^2 + 7^2$	–	47245.16	–	SAT
34	$0^2 + 0^2 + 6^2 + 10^2$	–	550.86	–	SAT
34	$0^2 + 6^2 + 6^2 + 8^2$	–	373.74	–	SAT
34	$2^2 + 2^2 + 8^2 + 8^2$	–	402.70	–	SAT
34	$2^2 + 4^2 + 4^2 + 10^2$	–	3345.30	–	SAT
36	$2^2 + 2^2 + 6^2 + 10^2$	–	687.05	–	SAT
36	$6^2 + 6^2 + 6^2 + 6^2$	–	555.97	–	SAT
38	$0^2 + 2^2 + 2^2 + 12^2$	–	30178.19	–	SAT
38	$0^2 + 4^2 + 6^2 + 10^2$	–	12810.39	–	SAT
38	$4^2 + 6^2 + 6^2 + 8^2$	–	23925.97	–	SAT
40	$0^2 + 0^2 + 4^2 + 12^2$	–	22969.16	–	SAT
40	$4^2 + 4^2 + 8^2 + 8^2$	–	1864.23	–	SAT
42	$0^2 + 2^2 + 8^2 + 10^2$	–	11233.80	–	SAT

Table 3.4: Timings in seconds for using MapleSAT to search for Williamson sequences of order $30 \leq n \leq 42$ using the sum-of-squares decomposition method, both with and without programmatic filtering enabled. A '–' denotes a timeout after 24 hours. Cases for which no results were generated are not shown.

was usually significantly faster; for example, in order 35 the programmatic filtering was consistently more than 30 times faster than without using programmatic filtering. In fact, in order 35 with 150 processors (50 on each of the three sum-of-squares decompositions) the divide-and-conquer method with programmatic filtering was able to show that Williamson sequences of order 35 do not exist in under 21 seconds.

3.6.4 UNSAT core method results

Next, we provide timings for the UNSAT core technique discussed in Section 3.3.3. This technique was somewhat hit-or-miss in that it was usually either significantly faster (if the UNSAT cores which were generated provided useful information) or slightly slower (due to the overhead of having to read UNSAT cores from disk). The timings for this method incorporate the previous two techniques, as the UNSAT core method requires the search space to be divided into multiple instances as a prerequisite. Thus, we provide the sum-of-squares decomposition along with each time in Table 3.7.

The usage of UNSAT cores was seen to significantly improve the performance of three cases for orders between 35 and 45: two cases in order 36, and one case in order 44. In these cases, the UNSAT cores which were computed consisted of a single variable and were used to discard many instances without making an additional call to MapleSAT.

To understand the meaning behing these UNSAT cores which were computed, we examined what the variable in the UNSAT core was encoding. To explain the meaning of the variable, recall that we computed compression values using binary adders and therefore those values were encoded by variables representing bits in the binary encoding of the compression values. In the cases we examined where the UNSAT cores were found to be extremely effective the UNSAT cores encoded the fact that setting the low bit of the middle entry in a 2-compression to be 1 (i.e., true) would cause the instance to be unsatisfiable. We can explain this behaviour using the following lemma:

Lemma 3.4. *Let A be a member of a Williamson sequence of order $n = 4k$ with $A^{(2k)}$ its 2-compression. Then $a_k^{(2k)} \neq 0$.*

Proof. By definition, we have $a_k^{(2k)} = a_k + a_{3k}$ and by symmetry $a_{3k} = a_{4k-3k} = a_k$. Thus $a_k^{(2k)} = 2a_k \in \{\pm 2\}$. □

This shows that when n is divisible by 4 the middle entry in a 2-compression is either 2 or -2 and the inputs to the binary adder calculating that value are the same (i.e., both

n	Decomposition	Normal MapleSAT	Programmatic MapleSAT	Programmatic Speedup Factor	Result
25	$1^2+1^2+7^2+7^2$	0.37	0.10	3.70	SAT
25	$1^2+3^2+3^2+9^2$	2.69	0.04	67.25	SAT
25	$1^2+5^2+5^2+7^2$	0.61	0.12	5.08	SAT
25	$5^2+5^2+5^2+5^2$	3.34	0.04	83.50	SAT
26	$0^2+0^2+2^2+10^2$	0.02	0.02	1.00	SAT
26	$0^2+2^2+6^2+8^2$	0.02	0.02	1.00	SAT
26	$4^2+4^2+6^2+6^2$	0.03	0.03	1.00	SAT
27	$1^2+1^2+5^2+9^2$	0.03	0.05	0.60	SAT
27	$1^2+3^2+7^2+7^2$	0.19	0.03	6.33	SAT
27	$3^2+3^2+3^2+9^2$	7.29	0.35	20.83	UNSAT
27	$3^2+5^2+5^2+7^2$	0.12	0.03	4.00	SAT
28	$2^2+2^2+2^2+10^2$	0.10	0.07	1.43	SAT
28	$2^2+6^2+6^2+6^2$	0.11	0.05	2.20	SAT
28	$4^2+4^2+4^2+8^2$	0.22	0.22	1.00	UNSAT
30	$0^2+2^2+4^2+10^2$	0.07	0.02	3.50	SAT
30	$2^2+4^2+6^2+8^2$	0.03	0.02	1.50	SAT
32	$0^2+0^2+8^2+8^2$	4.31	4.18	1.03	SAT
33	$1^2+1^2+3^2+11^2$	1.17	0.38	3.08	SAT
33	$1^2+1^2+7^2+9^2$	0.59	0.26	2.27	SAT
33	$1^2+5^2+5^2+9^2$	1.23	0.43	2.86	SAT
33	$3^2+5^2+7^2+7^2$	0.48	0.22	2.18	SAT
34	$0^2+0^2+6^2+10^2$	1.25	0.31	4.03	SAT
34	$0^2+6^2+6^2+8^2$	0.05	0.03	1.67	SAT
34	$2^2+2^2+8^2+8^2$	0.04	0.02	2.00	SAT
34	$2^2+4^2+4^2+10^2$	0.13	0.09	1.44	SAT

Table 3.5: Timings in seconds for using MapleSAT to search for Williamson sequences of order $25 \leq n \leq 34$ using the divide-and-conquer method, both with and without programmatic filtering enabled. Each case was solved using parallelization with 50 processors.

n	Decomposition	Normal MapleSAT	Programmatic MapleSAT	Programmatic Speedup Factor	Result
35	$1^2+3^2+3^2+11^2$	410.79	11.07	37.11	UNSAT
35	$1^2+3^2+7^2+9^2$	671.05	20.44	32.83	UNSAT
35	$3^2+5^2+5^2+9^2$	311.26	9.15	34.02	UNSAT
36	$0^2+0^2+0^2+12^2$	6.00	6.42	0.93	UNSAT
36	$0^2+4^2+8^2+8^2$	3.45	3.89	0.89	UNSAT
36	$2^2+2^2+6^2+10^2$	0.48	0.14	3.43	SAT
36	$6^2+6^2+6^2+6^2$	0.45	0.06	7.50	SAT
38	$0^2+2^2+2^2+12^2$	0.36	0.08	4.50	SAT
38	$0^2+4^2+6^2+10^2$	0.19	0.03	6.33	SAT
38	$4^2+6^2+6^2+8^2$	0.36	0.07	5.14	SAT
39	$1^2+3^2+5^2+11^2$	301.85	17.48	17.27	UNSAT
39	$1^2+5^2+7^2+9^2$	259.43	16.72	15.52	UNSAT
39	$3^2+7^2+7^2+7^2$	126.16	4.86	25.96	UNSAT
39	$5^2+5^2+5^2+9^2$	30.11	1.89	15.93	SAT
40	$0^2+0^2+4^2+12^2$	5.15	5.14	1.00	SAT
40	$4^2+4^2+8^2+8^2$	6.11	4.46	1.37	SAT
42	$0^2+2^2+8^2+10^2$	4.21	0.16	26.31	SAT
42	$2^2+2^2+4^2+12^2$	3.04	0.16	19.00	SAT
42	$2^2+6^2+8^2+8^2$	9.79	0.20	48.95	SAT
42	$4^2+4^2+6^2+10^2$	11.16	0.15	74.40	SAT
44	$0^2+4^2+4^2+12^2$	3.83	3.42	1.12	UNSAT
44	$2^2+6^2+6^2+10^2$	2.95	0.20	14.75	SAT
45	$1^2+1^2+3^2+13^2$	–	1544.99	–	UNSAT
45	$1^2+3^2+7^2+11^2$	–	1996.79	–	UNSAT
45	$1^2+7^2+7^2+9^2$	–	1448.48	–	UNSAT
45	$3^2+3^2+9^2+9^2$	–	1554.98	–	UNSAT
45	$3^2+5^2+5^2+11^2$	–	1379.05	–	UNSAT
45	$5^2+5^2+7^2+9^2$	–	1093.79	–	SAT

Table 3.6: Timings in seconds for using MapleSAT to search for Williamson sequences of order $35 \leq n \leq 45$ using the divide-and-conquer method, both with and without programmatic filtering enabled. A '$-$' denotes a timeout after 1 hour. Each case was solved using parallelization with 50 processors.

false or both true). Thus, the output of the binary adder must be [false, false] (encoding 0 true inputs) or [true, false] (encoding 2 true inputs). In both cases, the low bit is false; it is not possible for the low bit to be true which is exactly the fact encoded in the UNSAT cores produced by MapleSAT.

Of course, now that we know Lemma 3.4 we could incorporate it into the MathCheck2 generation script by never generating instances whose 2-compressions have a middle entry of zero when n is a multiple of 4. But it is still interesting that MathCheck2 is able to automatically discover Lemma 3.4 and then use that knowledge to make the search more efficient in an automated fashion.

3.6.5 Matching method results

Finally, we present timings for the matching method discussed in Section 3.5 and compare them to timings for running MathCheck2 with all techniques and programmatic filtering enabled. Table 3.8 contains the timings for finding a solution (or proving that none exist) for each given sum-of-squares decomposition both using MathCheck2 and the matching method as described in Section 3.5. When the result is SAT we also provide timings for the matching method to find *all* solutions in the search space; when the result is UNSAT there are no solutions in the search space so in this case there is no advantage to have the matching method stop after finding one solution. The MathCheck2 timings include both the time to generate the SAT instances (which is typically fast) and the time to solve the resulting instances using 50 processors. The matching timings specify the amount of time necessary to perform the matching process (either stopping after one solution or continuing until the space is searched exhaustively), also using 50 processors per sum-of-squares decomposition.

In most sums-of-squares decomposition cases in Table 3.8 we find that MathCheck2 is able to find a solution faster than using the matching method. However, note that since the divide-and-conquer method used by MathCheck2 requires the order to be composite we only give timings for composite orders in Table 3.8; we give the timings for the matching method in prime orders in Table 3.9. The largest prime order that MathCheck2 was able to solve (using just the sums-of-squares technique) was 31, as shown in Table 3.4.

Lastly, let W_n be a complete set of representatives for the set of Williamson sequences of order n. We have provided one possible enumeration of the representatives in W_n where $n \leq 45$ and these results are available online [Bright, 2017]. In Table 3.10 we give the value of $|W_n|$, that is, the number of inequivalent Williamson sequences of order n, for all n up to 45. The counts for odd n up to 59 were published in [Holzmann et al., 2008], and our counts agree with theirs in all cases, providing an independent verification. Additionally,

n	Decomposition	Without UNSAT Cores	With UNSAT Cores	UNSAT Core Speedup Factor	Result
35	$1^2+3^2+3^2+11^2$	11.07	11.37	0.97	UNSAT
35	$1^2+3^2+7^2+9^2$	20.44	21.46	0.95	UNSAT
35	$3^2+5^2+5^2+9^2$	9.15	9.46	0.97	UNSAT
36	$0^2+0^2+0^2+12^2$	6.42	0.11	58.36	UNSAT
36	$0^2+4^2+8^2+8^2$	3.89	0.07	55.57	UNSAT
36	$2^2+2^2+6^2+10^2$	0.14	0.09	1.56	SAT
36	$6^2+6^2+6^2+6^2$	0.06	0.07	0.86	SAT
38	$0^2+2^2+2^2+12^2$	0.08	0.10	0.80	SAT
38	$0^2+4^2+6^2+10^2$	0.03	0.04	0.75	SAT
38	$4^2+6^2+6^2+8^2$	0.07	0.08	0.88	SAT
39	$1^2+3^2+5^2+11^2$	17.48	18.41	0.95	UNSAT
39	$1^2+5^2+7^2+9^2$	16.72	17.65	0.95	UNSAT
39	$3^2+7^2+7^2+7^2$	4.86	5.08	0.96	UNSAT
39	$5^2+5^2+5^2+9^2$	1.89	2.05	0.92	SAT
40	$0^2+0^2+4^2+12^2$	5.14	2.82	1.82	SAT
40	$4^2+4^2+8^2+8^2$	4.46	2.05	2.18	SAT
42	$0^2+2^2+8^2+10^2$	0.16	0.15	1.07	SAT
42	$2^2+2^2+4^2+12^2$	0.16	0.19	0.84	SAT
42	$2^2+6^2+8^2+8^2$	0.20	0.23	0.87	SAT
42	$4^2+4^2+6^2+10^2$	0.15	0.18	0.83	SAT
44	$0^2+4^2+4^2+12^2$	3.42	0.08	42.75	UNSAT
44	$2^2+6^2+6^2+10^2$	0.20	0.23	0.87	SAT
45	$1^2+1^2+3^2+13^2$	1544.99	1520.95	1.02	UNSAT
45	$1^2+3^2+7^2+11^2$	1996.79	2002.40	1.00	UNSAT
45	$1^2+7^2+7^2+9^2$	1448.48	1458.93	0.99	UNSAT
45	$3^2+3^2+9^2+9^2$	1554.98	1640.83	0.95	UNSAT
45	$3^2+5^2+5^2+11^2$	1379.05	1420.56	0.97	UNSAT
45	$5^2+5^2+7^2+9^2$	1093.79	993.50	1.10	SAT

Table 3.7: Timings in seconds for using MapleSAT with programmatic filtering to search for Williamson sequences of order $35 \leq n \leq 45$ using the divide-and-conquer method, with and without using UNSAT cores. Each case was solved using parallelization with 50 processors.

this is the first time that counts for even n have ever been published. Now that counts are available for both even and odd orders an obvious observation which jumps out is that there tends to be many more Williamson sequences up to equivalence of even order than there are for odd order. So far, this phenomenon is unexplained. There does not seem to be any obvious relationship between for example $|W_n|$ and $|W_{n+2}|$ or $|W_{n/2}|$.

n	Decomposition	MathCheck2	Matching (one solution)	Matching (all solutions)	Result
35	$1^2+3^2+3^2+11^2$	11.71	61.99		UNSAT
35	$1^2+3^2+7^2+9^2$	21.84	52.72		UNSAT
35	$3^2+5^2+5^2+9^2$	9.82	50.59		UNSAT
36	$0^2+0^2+0^2+12^2$	3.48	78.15		UNSAT
36	$0^2+4^2+8^2+8^2$	1.61	171.60		UNSAT
36	$2^2+2^2+6^2+10^2$	2.71	0.30	224.20	SAT
36	$6^2+6^2+6^2+6^2$	8.50	0.16	250.74	SAT
38	$0^2+2^2+2^2+12^2$	2.05	2.90	647.23	SAT
38	$0^2+4^2+6^2+10^2$	2.10	19.61	627.13	SAT
38	$4^2+6^2+6^2+8^2$	1.61	11.09	1204.38	SAT
39	$1^2+3^2+5^2+11^2$	18.83	3942.55		UNSAT
39	$1^2+5^2+7^2+9^2$	18.03	2229.72		UNSAT
39	$3^2+7^2+7^2+7^2$	5.48	4764.22		UNSAT
39	$5^2+5^2+5^2+9^2$	2.42	1082.42	2270.15	SAT
40	$0^2+0^2+4^2+12^2$	12.77	9.44	5463.89	SAT
40	$4^2+4^2+8^2+8^2$	27.56	6.05	14441.7	SAT
42	$0^2+2^2+8^2+10^2$	13.79	153.14	45467.6	SAT
42	$2^2+2^2+4^2+12^2$	11.53	154.08	64534.0	SAT
42	$2^2+6^2+8^2+8^2$	13.29	50.81	90023.2	SAT
42	$4^2+4^2+6^2+10^2$	25.35	85.39	74324.0	SAT
44	$0^2+4^2+4^2+12^2$	141.41	184393.0		UNSAT
44	$2^2+6^2+6^2+10^2$	86.13	231.75	205471.0	SAT

Table 3.8: Timings in seconds for using MapleSAT with all techniques enabled and the matching method (both searching for one solution and all solutions) to search for Williamson sequences of order $35 \leq n \leq 44$. Each method used parallelization with 50 processors.

n	Decomposition	Matching (one solution)	Matching (all solutions)	Result
37	$1^2+1^2+5^2+11^2$	171.37		UNSAT
37	$1^2+7^2+7^2+7^2$	143.51		UNSAT
37	$3^2+3^2+3^2+11^2$	111.92	145.35	SAT
37	$3^2+3^2+7^2+9^2$	47.50	178.21	SAT
37	$5^2+5^2+7^2+7^2$	14.45	185.50	SAT
41	$1^2+1^2+9^2+9^2$	3123.38	13223.2	SAT
41	$3^2+3^2+5^2+11^2$	9520.45		UNSAT
41	$3^2+5^2+7^2+9^2$	10890.6		UNSAT
43	$1^2+1^2+1^2+13^2$	48702.2		UNSAT
43	$1^2+1^2+7^2+11^2$	73135.1		UNSAT
43	$1^2+3^2+9^2+9^2$	56508.7		UNSAT
43	$1^2+5^2+5^2+11^2$	40013.0	60679.4	SAT
43	$5^2+7^2+7^2+7^2$	50618.1	74862.8	SAT

Table 3.9: Timings in seconds for using the matching method (both searching for one solution and all solutions) to search for Williamson sequences of prime order $35 \leq n \leq 44$. Each method used parallelization with 50 processors.

n	$\lvert W_n\rvert$	n	$\lvert W_n\rvert$	n	$\lvert W_n\rvert$	n	$\lvert W_n\rvert$	n	$\lvert W_n\rvert$
1	1	10	12	19	6	28	1536	37	4
2	1	11	1	20	412	29	1	38	1284
3	1	12	36	21	7	30	4240	39	1
4	5	13	4	22	432	31	2	40	21504
5	1	14	76	23	1	32	2304	41	1
6	4	15	4	24	768	33	5	42	8904
7	2	16	27	25	10	34	2176	43	2
8	5	17	4	26	484	35	0	44	6048
9	3	18	540	27	6	36	11008	45	1

Table 3.10: The number of inequivalent Williamson sequences which exist in all orders up to 45; here W_n denotes a complete set of inequivalent Williamson sequences of order n.

A proof is a proof.

Jean Chrétien

Chapter 4

Computation of Complex Golay Sequences

As our second case study, we will examine the problem of computing complex Golay sequences and present an algorithm for enumerating all complex Golay sequences of a given order. Additionally, we use this algorithm to enumerate all complex Golay sequences up to order 25 and in Section 4.4 give counts of how many complex Golay sequences exist in each order.

Complex Golay sequences are defined similar to how we defined Williamson sequences in Definition 3.2, except that they are defined in terms of a *nonperiodic* autocorrelation function, they do not necessarily have to be symmetric, their entries are not necessarily real numbers, and the sequences occur in pairs, not quadruples. Despite these differences, some of the applications of Williamson matrices also apply to complex Golay sequences. In particular, complex Golay sequences can be used to construct Hadamard matrices or *complex* Hadamard matrices. A complex Hadamard matrix is a matrix $H \in \{\pm 1, \pm i\}^{n \times n}$ satisfying $HH^* = nI_n$ where H^* denotes the conjugate transpose of H.

The work done in this chapter was done in collaboration with Vijay Ganesh, Albert Heinle, and Ilias Kotsireas.

4.1 Introduction

Complex Golay sequences have been extensively studied in multiple papers over the last 25 years, including [Holzmann and Kharaghani, 1994], [Craigen, 1994], [Craigen et al., 2002],

and [Fiedler, 2013]. They were introduced in order to expand the orders of Hadamard matrices attainable via real Golay sequences (also referred to as Golay pairs). A notion of canonical form for Golay sequences has been introduced [Đoković, 1998] and representatives of the equivalence classes of Golay sequences for all lengths ≤ 40 have been found. More recently, Golay sequences have been classified up to order 100 in [Borwein and Ferguson, 2004] where the authors show that all such pairs can be derived using certain equivalence and composition operations from five primitive Golay pairs.

A positive integer n is called a *complex Golay number* if there exist complex Golay sequences of order n. The fundamental paper [Craigen et al., 2002] contains exhaustive searches for all lengths of complex Golay sequences up to 19, a partial search for orders 20 and 22 and contains a conjecture that 23 is not a complex Golay number. In addition, it is shown that $n = 7$, 9, 14, 15, 17, 19, 21 are not complex Golay numbers. Based on the numerical evidence they have gathered, the authors state four conjectures pertaining to complex Golay sequences and complex Golay numbers, all of which are still open. The fourth of their conjectures states that "every prime divisor of a complex Golay number is a complex Golay number". Given the aforementioned list of known complex Golay numbers, this means that to disprove this conjecture, one would have to construct/find complex Golay sequences of any of the orders 28, 34, 35, 38, 46, ...

Another interesting phenomenon is that all known odd complex Golay numbers besides 1 are prime. Therefore it would be of interest to know whether 33, 35, and 39 are complex Golay numbers or not. Finally, the authors provide theorems on the algebraic structure of complex Golay sequences which connects their structure to polynomial factorization over finite fields. In certain cases these theorems can be used to speed up computational algorithms to search for these rather elusive combinatorial objects, although the present work does not require them.

The following theorem [Craigen et al., 2002, Cor. 6] shows the importance of complex Golay numbers to Hadamard matrices.

Theorem 4.1. *If n is a complex Golay number, then there exists a complex Hadamard matrix of order $2n$ and a Hadamard matrix of order $4n$.*

It is known that complex Golay sequences of order $n = 2^{a+u}3^b5^c11^d13^e$ exist where the variables in the exponents are nonnegative integers which satisfy some linear inequalities. In view of Theorem 4.1, extending the list of known prime complex Golay numbers would enlarge the set of attainable orders of Hadamard matrices constructible via complex Golay sequences.

We present our new algorithm for exhaustively searching for complex Golay sequences of a given order in Section 4.3, following the necessary background covered in Section 4.2. We show that by solving certain Diophantine systems one can derive restrictions on the possible forms that all complex Golay sequences of a given order must satisfy. These restrictions are then used along with a procedure which can generate all permutations of a given form; this allows an exhaustive search to be performed on a space smaller than would be necessary using a naive exhaustive search.

4.2 Background on complex Golay sequences

In this section we present the background necessary to describe our algorithm for enumerating complex Golay sequences. First, we require some preliminary definitions to describe the kind of sequences we will be searching for. We use $\overline{x}$ to denote the complex conjugate of x.

Definition 4.1 (cf. [Kotsireas, 2013a])**.** The *complex aperiodic* (or *complex nonperiodic*) autocorrelation function of a sequence $A = [a_1, \ldots, a_n] \in \mathbb{C}^n$ of length $n \in \mathbb{N}$ is defined as

$$N_A(s) \coloneqq \sum_{k=1}^{n-s} a_k \overline{a_{k+s}}, \qquad s = 0, \ldots, n-1.$$

Definition 4.2 (cf. [Kotsireas, 2013a])**.** Two sequences A and B in $\mathbb{C}^n$ are said to have *constant aperiodic autocorrelation* if there is a constant $c \in \mathbb{C}$ such that

$$N_A(s) + N_B(s) = c, \qquad s = 1, \ldots, n-1.$$

Definition 4.3. A pair of sequences (A, B) with A and B in $\{\pm 1, \pm i\}^n$ are called a *complex Golay sequence pair* if they have zero constant aperiodic autocorrelation, i.e.,

$$N_A(s) + N_B(s) = 0, \qquad s = 1, \ldots, n-1.$$

If such sequences exist for $n \in \mathbb{N}$ we call n a *complex Golay number*.

Note that if A and B are in $\{\pm 1, \pm i\}^n$ then $N_A(0) + N_B(0) = 2n$ by the definition of the complex aperiodic autocorrelation function and the fact that $x\overline{x} = 1$ if x is ± 1 or $\pm i$.

4.2.1 Equivalence operations

There are certain invertible operations which preserve the property of being a complex Golay sequence pair when those operations are applied to sequence pairs (A, B). These are summarized in the following proposition.

Proposition 4.1 (cf. [Craigen et al., 2002], section 4)**.** *Let $([a_1, \ldots, a_n], [b_1, \ldots, b_n])$ be a complex Golay sequence pair. The following are then also complex Golay sequence pairs:*

E1. (Reversal) $([a_n, \ldots, a_1], [b_n, \ldots, b_1])$.

E2. (Conjugate Reverse A) $([\overline{a_n}, \ldots, \overline{a_1}], [b_1, \ldots, b_n])$.

E3. (Swap) $([b_1, \ldots, b_n], [a_1, \ldots, a_n])$.

E4. (Scale A) $([ia_1, \ldots, ia_n], [b_1, \ldots, b_n])$.

E5. (Positional Scaling) $([p_1a_1, \ldots, p_na_n], [p_1b_1, \ldots, p_nb_n])$ *where* $p_k \coloneqq i^k$.

Definition 4.4. We call two complex Golay sequence pairs (A, B) and (A', B') *equivalent* if (A', B') can be obtained from (A, B) using the transformations described in Proposition 4.1.

4.2.2 Useful properties and lemmas

In this subsection we prove some useful properties that complex Golay sequences must satisfy and which will be exploited by our algorithm for enumerating complex Golay sequences. Lemmas 4.1 and 4.2 are previously known results which we restate for convenience but we are not aware of any other work in which Lemma 4.3 appears.

The first lemma provides a relationship that all complex Golay sequences must satisfy. To state it, we require the following definition.

Definition 4.5. The *Hall polynomial* of the sequence $A \coloneqq [a_1, \ldots, a_n]$ is defined to be $h_A(z) \coloneqq a_1 + a_2z + \cdots + a_nz^{n-1} \in \mathbb{C}[z]$.

Lemma 4.1. *Let (A, B) be a complex Golay sequence pair. For every $z \in \mathbb{C}$ with $|z| = 1$, we have*

$$|h_A(z)|^2 + |h_B(z)|^2 = 2n.$$

Proof. Since $|z| = 1$ we can write $z = e^{i\theta}$ for some $0 \leq \theta < 2\pi$. Similar to the fact pointed out in [Kharaghani and Tayfeh-Rezaie, 2005], using Euler's identity one can derive the following expansion:

$$|h_A(z)|^2 = N_A(0) + 2\sum_{j=1}^{n-1}\big(\mathrm{Re}(N_A(j))\cos(\theta j) + \mathrm{Im}(N_A(j))\sin(\theta j)\big).$$

Since A and B form a complex Golay pair, by definition one has that $\mathrm{Re}(N_A(j)+N_B(j)) = 0$ and $\mathrm{Im}(N_A(j) + N_B(j)) = 0$ and then

$$|h_A(z)|^2 + |h_B(z)|^2 = N_A(0) + N_B(0) = 2n. \qquad \square$$

This lemma is highly useful as a condition for filtering sequences which could not possibly be part of a complex Golay sequence pair, as explained in the following corollary.

Corollary 4.1. *Let $A \in \{\pm 1, \pm i\}^n$, $z \in \mathbb{C}$ with $|z| = 1$, and $|h_A(z)|^2 > 2n$. Then A is not a member of a complex Golay sequence pair.*

Proof. Suppose the sequence A was a member of a complex Golay sequence pair whose other member was the sequence B. Since $|h_B(z)|^2 \geq 0$, we must have $|h_A(z)|^2 + |h_B(z)|^2 > 2n$, in contradiction to Lemma 4.1. $\square$

The next lemma is useful to derive conditions on how often each type of entry (i.e., 1, -1, i, $-i$) occurs in a complex Golay sequence pair. It is stated in [Craigen, 1994] using a different notation. We use the notation $\mathrm{resum}(A)$ and $\mathrm{imsum}(A)$ to represent the real and imaginary parts of the sum of the entries of A. For example, if $A \coloneqq [1, i, -i, i]$ then $\mathrm{resum}(A) = \mathrm{imsum}(A) = 1$.

Lemma 4.2. *Let (A, B) be a complex Golay sequence pair. Then*

$$\mathrm{resum}(A)^2 + \mathrm{imsum}(A)^2 + \mathrm{resum}(B)^2 + \mathrm{imsum}(B)^2 = 2n.$$

Proof. Using Lemma 4.1 with $z = 1$ we have

$$|\mathrm{resum}(A) + \mathrm{imsum}(A)i|^2 + |\mathrm{resum}(B) + \mathrm{imsum}(B)i|^2 = 2n.$$

Since $|\mathrm{resum}(X) + \mathrm{imsum}(X)i|^2 = \mathrm{resum}(X)^2 + \mathrm{imsum}(X)^2$ the result follows. $\square$

The next lemma provides some normalization conditions which can be used when searching for complex Golay sequences up to equivalence. We say that a complex Golay sequence is *normalized* if it meets these conditions.

Lemma 4.3. *Let (A, B) be a complex Golay sequence pair. Then (A, B) is equivalent to a complex Golay sequence pair (A', B') which satisfies the conditions*

$$\begin{aligned} 0 \leq \operatorname{resum}(A') &\leq \operatorname{imsum}(A'), \\ 0 \leq \operatorname{resum}(B') &\leq \operatorname{imsum}(B'), \\ \textit{and } \operatorname{resum}(A') &\leq \operatorname{resum}(B'). \end{aligned}$$

Proof. We will transform a given complex Golay sequence pair (A, B) into an equivalent normalized one using the equivalence operations of Proposition 4.1. To start with, let $A' \coloneqq A$ and $B' \coloneqq B$.

First, we ensure that $|\operatorname{resum}(A')| \leq |\operatorname{imsum}(A')|$. If this is not already the case then we apply operation E4 (which has the effect of switching $|\operatorname{resum}(A')|$ and $|\operatorname{imsum}(A')|$) and the updated A' will satisfy this condition.

Next, we ensure that $\operatorname{resum}(A') \geq 0$. If this is not already the case then we apply operation E4 twice (which has the effect of negating each element of A') and the updated A' will satisfy $0 \leq \operatorname{resum}(A') \leq |\operatorname{imsum}(A')|$. If $\operatorname{imsum}(A') \geq 0$ then the first condition is satisfied. If not, then it will be satisfied after applying operation E2 (which negates $\operatorname{imsum}(A')$).

Next, we ensure that the second condition holds. If it not already the case, then we apply operation E3 (switch A' and B'); this will cause the second condition to be satisfied at the cost of causing the first condition to no longer be satisfied. However, we may now repeat the above directions to make the first condition satisfied again; note that these directions do not modify B' so that once we have completed them both the first two conditions will be satisfied.

Lastly, we ensure the final condition $\operatorname{resum}(A') \leq \operatorname{resum}(B')$. If it is not already satisfied then we apply E3 (switch A' and B') and the updated sequence pair will satisfy the condition as required. □

4.2.3 Sum-of-squares decomposition types

A consequence of Lemma 4.2 is that every complex Golay sequence yields a decomposition of $2n$ into a sum of four squares. With the help of a computer algebra system (CAS) one can even enumerate all the ways that $2n$ may be written as a sum of four squares (e.g., using the function `PowersRepresentations` in MATHEMATICA). Furthermore, since it suffices to search for complex Golay sequence pairs up to equivalence, by Lemma 4.3 we can

make assumptions about the form of the decomposition, for example, that the resum and imsum of A and B are non-negative. Thus, it suffices to use a CAS to solve the quadratic Diophantine system

$$r_a^2 + i_a^2 + r_b^2 + i_b^2 = 2n, \qquad 0 \le r_a \le i_a, \qquad 0 \le r_b \le i_b, \qquad r_a \le r_b \tag{4.1}$$

for indeterminants r_a, i_a, r_b, $i_b \in \mathbb{Z}$.

Example 4.1. When $n = 23$ the Diophantine system (4.1) has exactly four solutions with $r_a + i_a \equiv n \pmod 2$ (since $\text{resum}(A) + \text{imsum}(A) \equiv n \pmod 2$ for $A \in \{\pm 1, \pm i\}^n$), as given in the following table:

r_a	i_a	r_b	i_b
0	1	3	6
1	2	4	5
0	3	1	6
1	4	2	5

Let (A, B) be a complex Golay sequence of order n. Furthermore, let u, v, x, and y represent the number of 1s, -1s, is, and $-i$s in A, and let r_a and i_a represent the resum and imsum of A, respectively. We have that

$$u, v, x, y \ge 0, \qquad u - v = r_a, \qquad x - y = i_a, \qquad u + v + x + y = n. \tag{4.2}$$

Given the values of n, r_a, and i_a this is a system of linear Diophantine equations which is to be solved over the non-negative integers. From the last equality we know that $u, v, x, y \le n$ so such a system necessarily has a finite number of solutions.

Example 4.2. When $n = 23$, $r_a = 0$, and $i_a = 1$, the Diophantine system (4.2) has exactly 12 solutions, as given in the following table:

u	v	x	y
0	0	12	11
1	1	11	10
2	2	10	9
3	3	9	8
4	4	8	7
5	5	7	6
6	6	6	5
7	7	5	4
8	8	4	3
9	9	3	2
10	10	2	1
11	11	1	0

The *multinomial coefficient* $\binom{n}{u,v,x,y} = \frac{n!}{u!v!x!y!}$ tells us how many possibilities there are for $A \in \{\pm 1, \pm i\}^n$ with u entries which are 1s, v entries which are -1s, x entries which are is, and y entries which are $-i$s. For example, there are $\frac{23!}{12!11!} = 1{,}352{,}078$ possibilities for A with 12 entries which are is and 13 entries which are $-i$s (i.e., those which correspond to the first row of the table in Example 4.2). Algorithms for explicitly generating all such possibilities for A can be found in [Knuth, 2011].

4.3 Description of our algorithm

First, we fix an order n for which we are interested in generating a list of all inequivalent complex Golay sequence pairs (A, B). Our algorithm finds all solutions r_a, i_a, r_b, i_b of (4.1) and for all pairs (r_a, i_a) then solves the system (4.2). For each solution quadruple (u, v, x, y) we use Algorithm 7.2.1.2L from [Knuth, 2011] to generate all possibilities for A with the appropriate number of 1s, -1s, is, and $-i$s. For each possibility for A we compute $H_k := |h_A(e^{2\pi ik/50})|^2$ for $k = 1, \ldots, 49$. If any value of H_k is strictly larger than $2n$, we immediately discard the sequence A (see Corollary 4.1). If all values of H_k are smaller than $2n$ then we record the sequence A as one which could appear in the first position of a complex Golay sequence pair.

Next, we repeat the above steps except that we solve the system (4.2) for all pairs (r_b, i_b) (replacing r_a with r_b and i_a with i_b), and this time we generate a list of possibilities for B, sequences which could appear in the second position of a complex Golay sequence pair.

Finally, we use the matching technique of [Kotsireas et al., 2009] to compile a list of all complex Golay sequence pairs of a given order. We form the strings

$$\mathrm{Re}(N_A(1)), \mathrm{Im}(N_A(1)), \ldots, \mathrm{Re}(N_A(n-1)), \mathrm{Im}(N_A(n-1))$$

and

$$-\mathrm{Re}(N_B(1)), -\mathrm{Im}(N_B(1)), \ldots, -\mathrm{Re}(N_B(n-1)), -\mathrm{Im}(N_B(n-1))$$

for all possibilities for A and B which were previously generated. We then create two files, those containing the 'A' strings sorted in lexicographic order, and those containing the 'B' strings sorted in lexicographic order. Finally, we perform a linear scan through the files to find all the strings which are common to both files. All matches are guaranteed to produce complex Golay sequences since if the strings derived from sequences A and B matched then $\mathrm{Re}(N_A(s)) + \mathrm{Re}(N_B(s)) = \mathrm{Im}(N_A(s)) + \mathrm{Im}(N_B(s)) = 0$ for $s = 1, \ldots, n-1$. Furthermore, all normalized complex Golay sequences will be among the matches since by construction if (A, B) is a normalized complex Golay sequence then A appears in the first

list of possibilities generated and B appears in the second list of possibilities which were generated.

If one wants the complete list of complex Golay sequence pairs, one can now repeatedly apply the equivalence operations E1–E5 to the list of normalized complex Golay sequence pairs until those operations no longer produce new sequences. Note that normalized complex Golay sequences are not necessarily inequivalent, as the normalization conditions of Lemma 4.3 do not capture all possible equivalences. If there are two sequences which are equivalent to each other this can be checked by applications of the equivalence operations E1–E5 and one of the equivalent sequences can be removed if desired.

4.3.1 Optimizations

There are some further properties which while not essential to the algorithm can be exploited to remove some extraneous computations.

Lemma 4.4. *Let $H_k := |h_A(e^{2\pi ik/50})|^2$ be the quantity which we use in our algorithm's filtering criterion, and let H'_k be the same quantity but computed with respect to A', the reverse of A. Then $H_k = H'_{50-k}$.*

Proof. Let $\theta := 2\pi k/50$. Then $H'_{50-k} = |h_{A'}(e^{-i\theta})|^2$ and as in the decomposition in Lemma 4.1 one has

$$|h_{A'}(e^{-i\theta})|^2 = N_{A'}(0) + 2\sum_{j=1}^{n-1}\big(\mathrm{Re}(N_{A'}(j))\cos(-\theta j) + \mathrm{Im}(N_{A'}(j))\sin(-\theta j)\big).$$

From the definition of the aperiodic autocorrelation function one sees that $N_{A'}(0) = N_A(0)$ and $N_{A'}(s) = \overline{N_A(s)}$ for $s = 1, \ldots, n-1$. Using this with the standard facts that $\cos(-x) = \cos(x)$ and $\sin(-x) = -\sin(x)$ one derives that this expansion is exactly the same as the expansion for $|h_A(e^{i\theta})|^2 = H_k$, as required. □

In light of Lemma 4.4, we do not need to compute the values H_k for both A and its reverse, since the H_k values for the reverse of A will be the exact same as those for A (albeit in reverse order). In other words, A will be discarded by our filtering condition if and only if its reverse is discarded by our filtering condition, so once A has been checked we need not also check its reverse.

To avoid extraneous computations, we only perform the filtering check on one of A and the reverse of A, whichever is lexicographically greater (if A is equal to its reverse it is also

checked). Once the filtering process has been completed we take the list of sequences which passed the filter and add to the list the reverse of each sequence on the list (except for those which are their own reverse).

Of course, we can perform the same optimization when performing the filtering check on the B sequences as well. In this case one can completely discard sequences whose reverses are lexicographically strictly smaller than themselves because of the following lemma.

Lemma 4.5. *The following normalization condition may be added to Lemma 4.3:*

$$B' \geq_{\text{lex}} \text{reverse}(B'). \tag{4.3}$$

Proof. Continuing the proof of Lemma 4.3, if the complex Golay sequence pair (A', B') satisfies (4.3) then we are done. If not, we apply equivalence operation E1 (reversal) to (A', B') so that (4.3) is satisfied. Furthermore, all the normalization conditions of Lemma 4.3 remain satisfied because the operation E1 does not change the resum or imsum of A' or B'. □

Finally, we note that it is possible to optimize the evaluation of the Hall polynomial by reusing previously computed values. If $A \coloneqq [a_1, \ldots, a_n]$ is the sequence which we need to check the filtering condition for, then we want to compute the Hall polynomial evaluation

$$h_A(e^{2\pi ik/50}) = \sum_{j=0}^{n-1} a_{j+1} e^{2\pi ijk/50} \qquad \text{for } k = 1, \ldots, 49.$$

Because of the periodicity $e^{2\pi ijk/50} = e^{2\pi i(jk \bmod 50)/50}$ and the fact $-e^{2\pi ijk/50} = e^{2\pi i(jk+25)/50}$ there are only 100 possible values for the summand in this sum, namely $xe^{2\pi iy/50}$ for $x = 1$ or i and $y = 0, \ldots, 49$. These can be computed once at the start of the algorithm and reused as necessary.

Furthermore, in many cases it is possible to reuse some computations from the Hall polynomial evaluation of the previously checked sequence. The algorithm we used to generate the sequences [Knuth, 2011, §7.2.1.2] generates them in lexicographically increasing order, meaning that consecutively generated sequences often share a large common prefix. If the entries $a_1, \ldots, a_l$ of a sequence are identical to those in the previously generated sequence then the partial sum $\sum_{j=0}^{l-1} a_{j+1} e^{2\pi ijk/50}$ can be reused, assuming it was computed and stored; as the Hall polynomial evaluations are being computed one can remember their partial sums for varying l and k in a table.

4.4 Results

The algorithm described above was implemented in C and run for all orders n up to 25. We find that $n = 23$ and 25 are not complex Golay numbers, i.e., that complex Golay sequences of order 23 and 25 do not exist. This result confirms the conjecture of [Craigen et al., 2002], verifies the results of [Fiedler, 2013], and in addition implies that the next candidate prime complex Golay number is $n = 29$. Our results match the previously computed results of [Craigen et al., 2002] in all cases, but we also provide complete results for orders 20, 22, 23, 24, and 25; these results can be found online [Bright, 2016a].

The computations were performed with all optimizations enabled and on an Intel Xeon CPU running at 3.3GHz under Ubuntu 14.04. The algorithm's run time in hours for orders 20, 21, 22, 23, 24, and 25 was 4, 13, 32, 179, 361, and 3268, respectively. In each case almost all of the time was spent enumerating the permutations of the required forms. Once this enumeration had been completed, the remaining parts of the algorithm could typically be run in only several seconds or minutes.

Table 4.1 contains a summary of how many complex Golay pairs exist for each order up to 25. The second column contains the total number of complex Golay pairs and the third column contains the number of inequivalent complex Golay pairs.

Order	Total Pairs	Inequivalent Pairs
1	16	1
2	64	1
3	128	1
4	512	2
5	512	1
6	2048	3
7	0	0
8	6656	17
9	0	0
10	12,288	20
11	512	1
12	36,864	52
13	512	1
14	0	0
15	0	0
16	106,496	204
17	0	0
18	24,576	24
19	0	0
20	215,040	340
21	0	0
22	8192	12
23	0	0
24	786,432	1056
25	0	0

Table 4.1: A summary of the number of complex Golay pairs which exist in all orders up to 25.

Even if you were right, it'd be
$1 + 1 + 2 + 1$, not $1 + 2 + 1 + 1$.
Wadsworth, *Clue*

Chapter 5

Computation of Minimal Primes

As our final case study, we examine the problem of computing the set of minimal primes in various bases. While this set is known to be finite for every base there are no known bounds on how large the set can be. Thus, this case study is different from both of our previous case studies, whose search spaces were finite when searching for sequences of a given order. This adds an additional complication that the algorithms attempting to solve this problem must address. The naive brute force algorithm of "search the entire space one candidate at a time" will never terminate.

Currently, it is not even known if the problem of determining the minimal primes of a given base is a computable problem or not. Thus, we do not present an algorithm which can be proven to terminate but instead a heuristic algorithm which terminates in many cases of interest, including all bases less than 17 as well as 18, 20, 22, 23, 24, and 30.

The work in this chapter was done in collaboration with Jeffrey Shallit and Raymond Devillers and appeared in the journal *Experimental Mathematics* [Bright et al., 2016a].

5.1 Introduction

Problems about the digits of prime numbers have a long history, and many of them are still unsolved. For example, are there infinitely many primes, all of whose base-10 digits are `1`? Currently, there are only five such "repunits" known [Williams and Dubner, 1986], corresponding to $(10^p - 1)/9$ for $p \in \{2, 19, 23, 317, 1031\}$. It seems likely that four more are given by $p \in \{49081, 86453, 109297, 270343\}$, but this has not yet been rigorously proven.

Another problem on the digits of primes was introduced in the paper [Shallit, 2000]. To describe it, we need some definitions. We say that a string x is a *subword* of a string y, and we write $x \triangleleft y$, if one can strike out zero or more symbols of y to get x. For example, `string` is a subword of `Meistersinger`. (In the literature, this concept is sometimes called a "scattered subword" or "substring" or "subsequence".) A *language* is a set of strings. A string s is *minimal* for L if (a) $s \in L$ and (b) if $x \in L$ and $x \triangleleft s$, then $x = s$. The set of all minimal strings of L is denoted $M(L)$.

In this chapter we describe a heuristic technique for determining $M(L_b)$ in the case where L_b consists of the representations, in base b, of the prime numbers $\{2, 3, 5, \ldots\}$. We obtain a complete characterization of $M(L_b)$ for bases $2 \leq b \leq 16$ and $b = 18$, 20, 22, 23, 24, and 30. For the remaining bases $b = 17$, 19, 21, and $25 \leq b \leq 29$, we obtain results that allow us to "almost" completely characterize this set.

The same technique can also be applied to find minimal sets for subsets of prime numbers. For example, we were able to determine the minimal set for primes of the form $4n + 1$ represented in base 10 (and similarly for those of the form $4n + 3$). This successfully completes the sequences A111055 and A111056 in the *Encyclopedia of Integer Sequences* [OEIS Foundation Inc., 1996], which had been incomplete since their introduction to the encyclopedia in 2005.

5.1.1 Notation

In what follows, if x is a string of symbols over the alphabet $\Sigma_b := \{\mathtt{0}, \mathtt{1}, \ldots, b-1\}$ we let $[x]_b$ denote the evaluation of x in base b (starting with the most significant digit), and $[\epsilon]_b := 0$ where ϵ is the empty string. This is extended to languages as follows: $[L]_b := \{\, [x]_b : x \in L \,\}$. We use the convention that $\mathtt{A} := 10$, $\mathtt{B} := 11$, and so forth, to conveniently represent strings of symbols in base $b > 10$. We let $(x)_b$ be the canonical representation of x in base-b, that is, the representation without leading zeroes. Finally, as usual, for a language L we let $L^n := \underbrace{LL\cdots L}_{n}$ and $L^* := \bigcup_{i \geq 0} L^i$.

5.2 Why minimal sets are interesting

One reason why the minimal set $M(L)$ of a language L is interesting is because it allows us to compute two natural and related languages, defined as follows:

$$\begin{aligned}\mathrm{sub}(L) &\coloneqq \{\, x \in \Sigma^* : \text{there exists } y \in L \text{ such that } x \lhd y \,\};\\ \mathrm{sup}(L) &\coloneqq \{\, x \in \Sigma^* : \text{there exists } y \in L \text{ such that } y \lhd x \,\}.\end{aligned}$$

An amazing fact is that $\mathrm{sub}(L)$ and $\mathrm{sup}(L)$ are always regular. This follows from the following classical theorem due to Higman [Higman, 1952] and Haines [Haines, 1969].

Theorem 5.1. *For every language L, there are only finitely many minimal strings.*

Indeed, we have $\mathrm{sup}(L) = \mathrm{sup}(M(L))$ and $\Sigma^* - \mathrm{sub}(L) = \mathrm{sup}(M(\Sigma^* - \mathrm{sub}(L)))$, and the superword language of a finite language is regular, since

$$\mathrm{sup}\big(\{w_1, \ldots, w_n\}\big) = \bigcup_{i=1}^{n} \Sigma^* w_{i,1} \Sigma^* \cdots \Sigma^* w_{i,|w_i|} \Sigma^*$$

where $w_i = w_{i,1} \cdots w_{i,|w_i|}$ with $w_{i,j} \in \Sigma$.

5.3 Why the problem is hard

Determining $M(L)$ for arbitrary L is in general unsolvable, and can be difficult even when L is relatively simple [Gruber et al., 2007, Gruber et al., 2009].

The following is a "semi-algorithm" that is guaranteed to produce $M(L)$, but it is not so easy to implement:

(1) $M \coloneqq \emptyset$
(2) while $(L \neq \emptyset)$ do
    (3) choose x, a shortest string in L
    (4) $M \coloneqq M \cup \{x\}$
    (5) $L \coloneqq L - \mathrm{sup}(\{x\})$

In practice, for arbitrary L, we cannot feasibly carry out step (5). Instead, we work with L', some regular overapproximation to L, until we can show $L' = \emptyset$ (which implies $L = \emptyset$). In practice, L' is usually chosen to be a finite union of sets of the form $L_1 L_2^* L_3$, where each of L_1, L_2, L_3 is finite. In the case we consider in this chapter, we then have to determine whether such a language contains a prime or not.

However, it is not even known if the following simpler decision problem is recursively solvable:

Problem 5.1. *Given a base b and strings x, y, $z \in \Sigma_b^*$, does there exist a prime number whose base-b expansion is of the form $x \underbrace{yy \cdots y}_{n} z$ for some $n \geq 0$?*

An algorithm to solve this problem, for example, would allow us to decide if there are any additional Fermat primes (of the form $2^{2^n} + 1$) other than the known ones (corresponding to $n = 0$, 1, 2, 3, 4). To see this, take $b := 2$, $x := \mathtt{1}$, $y := \mathtt{0}$, and $z := \mathtt{0}^{16}\mathtt{1}$. Since if $2^n + 1$ is prime then n must be a power of two, a prime of the form $[xy^*z]_b$ must be a new Fermat prime.

Therefore, in practice, we are forced to try to rule out prime representations based on heuristics such as modular techniques and factorizations. This is discussed in the next section.

5.4 Some useful lemmas

It will be necessary for our algorithm to determine if families of the form $[xL^*z]_b$ contain a prime or not. We use two different heuristic strategies to show that such families contain no primes.

In the first strategy, we mimic the well-known technique of "covering congruences" [Choi, 1971], by finding some finite set S of integers $N > 1$ such that every number in a given family is divisible by some element of S. In the second strategy, we attempt to find a difference-of-squares or difference-of-cubes factorization.

5.4.1 The first strategy

We start with the simplest version of the idea: to find an $N > 1$ that divides each element of the family $[xL^*z]_b$. At first glance, this would require checking that N divides xL^nz for

$n = 0, 1, 2, \ldots$. However, the following lemma shows that it is only necessary to check the two cases $n = 0$ and 1. Although divisibility based on digital considerations has a long history (e.g., [Dickson, 1952, Chap. XII]), we could not find these kinds of results in the literature.

Lemma 5.1. *Let x, $z \in \Sigma_b^*$, and let $L \subseteq \Sigma_b^*$. Then N divides all numbers of the form $[xL^*z]_b$ if and only if N divides $[xz]_b$ and all numbers of the form $[xLz]_b$.*

Proof. Let $y = y_1 \cdots y_n \in L^*$, where $y_1, \ldots, y_n \in L$. By telescoping we have

$$[xyz]_b - [xz]_b = \sum_{i=1}^{n} ([xy_i y_{i+1} \cdots y_n z]_b - [xy_{i+1} \cdots y_n z]_b).$$

Cancelling the final $|y_{i+1} \cdots y_n z|$ base-b digits in the summand difference — which are identical — this becomes

$$[xyz]_b = [xz]_b + \sum_{i=1}^{n} b^{|y_{i+1} \cdots y_n z|} ([xy_i]_b - [x]_b).$$

But $b^{|z|}([xy_i]_b - [x]_b) = [xy_i z]_b - [xz]_b$ by adding and subtracting $[z]_b$, so we have

$$[xyz]_b = [xz]_b + \sum_{i=1}^{n} b^{|y_{i+1} \cdots y_n|} ([xy_i z]_b - [xz]_b).$$

Since $N \mid [xz]_b$ and $N \mid [xy_i z]_b$ for each $1 \leq i \leq n$, it follows that $N \mid [xyz]_b$.

The other direction is clear, since $[xz]_b$ and numbers of the form $[xLz]_b$ are both of the form $[xL^*z]_b$. □

In practice, our algorithm employs this lemma with $L \coloneqq \{y_1, \ldots, y_n\} \subseteq \Sigma_b$, and all numbers of the form $[xL^*z]_b$ are shown to be composite with the following corollary.

Corollary 5.1. *If $1 < \gcd([xz]_b, [xy_1 z]_b, \ldots, [xy_n z]_b) < [xz]_b$ then all numbers of the form $[x\{y_1, \ldots, y_n\}^* z]_b$ are composite.*

Proof. By Lemma 5.1, we know that $N \coloneqq \gcd([xz]_b, [xy_1 z]_b, \ldots, [xy_n z]_b) > 1$ divides all numbers of the form $[x\{y_1, \ldots, y_n\}^* z]_b$. By the size condition N is strictly less than each such number, and so is a nontrivial divisor. □

Example 5.1. Since $\gcd(49, 469) = 7$, every number with base-10 representation of the form `46*9` is divisible by 7. Since $1 < 7 < 49$, each such number is composite.

We also generalize this to the following corollary in the case where a single divisor does not divide each number in the family.

Corollary 5.2. *Let* $L \coloneqq \{y_1, y_2, \ldots, y_n\}$. *If*

$$N_0 \coloneqq \gcd\big(\{[xz]_b\} \cup [xL^2z]_b\big)$$

and

$$N_1 \coloneqq \gcd\big([xLz]_b \cup [xL^3z]_b\big)$$

lie strictly between 1 *and* $[xz]_b$, *then all numbers of the form* $[xL^*z]_b$ *are composite.*

Proof. By Lemma 5.1 applied to $[x(L^2)^*z]_b$, we know that N_0 divides all numbers of the form $[xL^*z]_b$ in which an even number of y_i appear. By Lemma 5.1 on $[xy_i(L^2)^*z]_b$ for each $1 \le i \le n$, we know that N_1 divides all numbers of the form $[xL^*z]_b$ for which an odd number of y_i appear. By the size conditions, N_0 and N_1 are nontrivial divisors. □

Example 5.2. Since $\gcd([\texttt{6}]_9, [\texttt{611}]_9) = 2$, every number with base-9 representation of the form $\texttt{61}^*$ of odd length is divisible by 2. Since $\gcd([\texttt{61}]_9, [\texttt{6111}]_9) = 5$, every number with base-9 representation of the form $\texttt{61}^*$ of even length is divisible by 5. Since these numbers lie strictly between 1 and 6, every number with base-9 representation of the form $\texttt{61}^*$ is composite.

We also note that it is simple to generalize Corollary 5.2 to apply to check if there are divisors N_0, N_1, …, N_{k-1} such that N_i divides all numbers of the form $[x\{y_1, \ldots, y_n\}^*z]_b$ in which the number of y_i appearing is congruent to i mod k.

Example 5.3. Let $b \coloneqq 16$. Then 7 divides $[\texttt{8A01}]_b$ and $[\texttt{8A0AAA1}]_b$. Furthermore, 13 divides $[\texttt{8A0A1}]_b$ and $[\texttt{8A0AAAA1}]_b$, and 3 divides $[\texttt{8A0AA1}]_b$ and $[\texttt{8A0AAAAA1}]_b$. Thus all numbers with base-16 representation of the form $\texttt{8A0A}^*\texttt{1}$ are divisible by either 7, 13, or 3, depending on their length mod 3.

A version of Lemma 5.1 which applies to the most general kind of family we need to consider ($x_1L_1^* \cdots x_mL_m^*$, where we allow the case $L_m^* = \emptyset$) is formulated in Lemma 5.2.

Lemma 5.2. *Let* $x_1, \ldots, x_m \in \Sigma_b^*$, *and* $L_1, \ldots, L_m \subseteq \Sigma_b^*$. *Then* N *divides all numbers of the form* $[x_1L_1^*x_2L_2^* \cdots x_mL_m^*]_b$ *if and only if* N *divides* $[x_1 \cdots x_m]_b$ *and all numbers of the form* $[x_1L_1x_2x_3 \cdots x_m]_b, \ldots, [x_1 \cdots x_{m-1}x_mL_m]_b$.

Proof. Say $w \in x_1 L_1^* x_2 L_2^* \cdots x_m L_m^*$; then there exist $y_{i,1}, \ldots, y_{i,n_i} \in L_i$ such that

$$w = x_1 y_{1,1} \cdots y_{1,n_1} x_2 y_{2,1} \cdots y_{2,n_2} \cdots x_m y_{m,1} \cdots y_{m,n_m}$$

for $1 \leq i \leq m$. As in the proof of Lemma 5.1, we have that

$$[w]_b = [x_1 \cdots x_m]_b + \sum_{i=1}^{m} \sum_{j=1}^{n_i} b^{|y_{i,j+1} \cdots y_{m,n_m}|} ([x_1 \cdots x_i y_{i,j} x_{i+1} \cdots x_m]_b - [x_1 \cdots x_m]_b)$$

from which the claim follows. □

As in Lemma 5.1, we typically apply this lemma in the case where each $L_i \subseteq \Sigma_b$ and show that all numbers of the form $[x_1 L_1^* x_2 L_2^* \cdots x_m L_m^*]_b$ have a divisor.

Example 5.4. Take $(L_1, L_2, L_3) \coloneqq (\{\mathtt{0}\}, \{\mathtt{0}\}, \emptyset)$ and $(x_1, x_2, x_3) \coloneqq (\mathtt{9}, \mathtt{8}, \mathtt{1})$. Since 9 divides 981, 9081, and 9801, it follows that 9 divides every number with base-10 representation of the form $\mathtt{90^*80^*1}$.

More generally, if a single divisor doesn't work for every number, Lemma 5.2 can also be applied in the case where all numbers of the form $[x_1 L_1^* \cdots x_i (L_i^2)^* \cdots x_m L_m^*]_b$ have one divisor, and all numbers of the form $[x_1 L_1^* \cdots x_i L_i (L_i^2)^* \cdots x_m L_m^*]_b$ have another divisor.

Example 5.5. Let $b \coloneqq 11$. Since 3 divides each of $[\mathtt{44A1}]_b$, $[\mathtt{44A111}]_b$, $[\mathtt{440A1}]_b$, it follows that every number of the form $[\mathtt{440^*(11)^*1}]_b$ is composite. Since 2 divides each of $[\mathtt{44A11}]_b$, $[\mathtt{44A1111}]_b$, $[\mathtt{440A11}]_b$, we know every number of the form $[\mathtt{440^*(11)^*11}]_b$ is composite. It follows that all numbers of the form $[\mathtt{440^*A1^*1}]_b$ are composite.

Lemma 5.2 can also be applied to the case when all even-length strings under consideration have one divisor, and all the odd-length strings have another divisor. One such case is, for example, if numbers of the form $[x_1 (L_1^2)^* x_2 (L_2^2)^* x_3]_b$ and $[x_1 L_1 (L_1^2)^* x_2 L_2 (L_2^2)^* x_3]_b$ have one divisor, and numbers of the form $[x_1 L_1 (L_1^2)^* x_2 (L_2^2)^* x_3]_b$ and $[x_1 (L_1^2)^* x_2 L_2 (L_2^2)^* x_3]_b$ have another divisor.

Example 5.6. Let $b \coloneqq 9$. Since 2 divides each of $[\mathtt{6}]_b$, $[\mathtt{116}]_b$, $[\mathtt{611}]_b$, $[\mathtt{161}]_b$, $[\mathtt{11161}]_b$, $[\mathtt{16111}]_b$, every odd-length string of the form $\mathtt{1^*61^*}$ is composite. Since 5 divides each of $[\mathtt{16}]_b$, $[\mathtt{1116}]_b$, $[\mathtt{1611}]_b$, $[\mathtt{61}]_b$, $[\mathtt{1161}]_b$, $[\mathtt{6111}]_b$, every even-length string of the form $\mathtt{1^*61^*}$ is composite.

5.4.2 The second strategy

A second way of proving that families of the form xL^*z do not contain a prime is via algebraic factorizations, such as a difference-of-squares factorization.

Lemma 5.3. *Let x, $z \in \Sigma_b^*$, $y \in \Sigma_b$, and let $g \coloneqq \gcd([y]_b, b-1)$, $X \coloneqq ([y]_b + (b-1)[x]_b)/g$, and $Y \coloneqq (b^{|z|}[y]_b - (b-1)[z]_b)/g$. If b, X, and Y are all squares and $\sqrt{b^{|z|}X} - \sqrt{Y} > (b-1)/g$, then all numbers of the form $[xy^*z]_b$ are composite.*

Proof. Evaluating the base-b expansion of xy^nz, we get

$$\begin{aligned}[xy^nz]_b &= b^{|z|+n}[x]_b + b^{|z|}\frac{b^n-1}{b-1}[y]_b + [z]_b \\ &= \frac{b^{|z|+n}X - Y}{(b-1)/g}.\end{aligned}$$

Since b, X, and Y are all squares the numerator factors as a difference of squares. By the size condition both factors are strictly larger than the denominator, and so the factorization is nontrivial. □

Example 5.7. Let $b \coloneqq 16$, $x \coloneqq \texttt{4}$, $y \coloneqq \texttt{4}$, and $z \coloneqq \texttt{1}$. Then $g = 1$, $X = 8^2$, $Y = 7^2$, and

$$[\texttt{44}^n\texttt{1}]_b = \frac{(4^{n+1}\cdot 8 + 7)(4^{n+1}\cdot 8 - 7)}{15}.$$

Since $4 \cdot 8 - 7 > 15$, this factorization is nontrivial and no number of the form $[\texttt{44}^*\texttt{1}]_b$ is prime.

It is also possible to combine Lemma 5.2 with Corollary 5.2 to construct a test which also applies to bases which are not squares.

Corollary 5.3. *Using the same setup as in Lemma 5.2, if $b^{|z|}X$ and Y are squares, $\sqrt{b^{|z|}X} - \sqrt{Y} > (b-1)/g$, and $1 < \gcd([xyz]_b, [xy^3z]_b) < [xz]_b$, then all numbers of the form $[xy^*z]_b$ are composite.*

Proof. Say $n = 2m$ is even. Then from the factorization in Lemma 5.2,

$$[xy^nz]_b = \frac{(b^m\sqrt{b^{|z|}X} + \sqrt{Y})(b^m\sqrt{b^{|z|}X} - \sqrt{Y})}{(b-1)/g}$$

which is nontrivial by the size condition.

Alternatively, if n is odd then as in Corollary 5.2 we have that $\gcd([xyz]_b, [xy^3z]_b)$ divides $[xy^nz]_b$, and by the size condition this divisor is nontrivial. □

Example 5.8. Let $b \coloneqq 17$, $x \coloneqq \mathtt{19}$, $y \coloneqq \mathtt{9}$, and $z \coloneqq \mathtt{9}$. Then $g = 1$, $b^{|z|}X = 85^2$, $Y = 3^2$, and

$$[xy^{2n}z]_b = \frac{(17^n \cdot 85 + 3)(17^n \cdot 85 - 3)}{16}.$$

Since $85 - 3 > 16$ this factorization is nontrivial. Furthermore, all numbers of the form $[xy^{2n+1}z]_b$ are even, so all numbers of the form $[\mathtt{199^*9}]_b$ are composite.

Finally, we present a variant of Lemma 5.3 which applies to a difference-of-cubes factorization.

Lemma 5.4. *Let x, $z \in \Sigma_b^*$, $y \in \Sigma_b$, and let $g \coloneqq \gcd([y]_b, b-1)$, $X \coloneqq ([y]_b + (b-1)[x]_b)/g$, and $Y \coloneqq (b^{|z|}[y]_b - (b-1)[z]_b)/g$. If b, X, and Y are all cubes and $\sqrt[3]{b^{|z|}X} - \sqrt[3]{Y} > (b-1)/g$, then all numbers of the form $[xy^*z]_b$ are composite.*

Proof. As in Lemma 5.3, we have

$$[xy^nz]_b = \frac{\left((b^{|z|+n}X)^{1/3} - Y^{1/3}\right)\left((b^{|z|+n}X)^{2/3} + (b^{|z|+n}XY)^{1/3} + Y^{2/3}\right)}{(b-1)/g}.$$

The second factor is at least as large as the first (except in the single case $b^{|z|+n}X = 1$ and $Y = -1$, which is not possible by construction of X and Y), so by the size condition both factors are strictly larger than the denominator, and the factorization is nontrivial. □

Example 5.9. Let $b \coloneqq 8$, $x \coloneqq \mathtt{1}$, $y \coloneqq \mathtt{0}$, and $z \coloneqq \mathtt{1}$. Then $g = 7$, $X = 1$, $Y = -1$, and

$$[\mathtt{10^n1}]_b = (2^{n+1} + 1)(4^{n+1} - 2^{n+1} + 1).$$

Since $2 - (-1) > 1$, this factorization is nontrivial and no number of the form $[\mathtt{10^*1}]_b$ is prime.

5.5 Our heuristic algorithm

As previously mentioned, in practice to compute $M(L_b)$ one works with an underapproximation M of $M(L_b)$ and an overapproximation L of $L_b - \sup(M)$. One then refines such approximations until $L = \emptyset$ from which it follows that $M = M(L_b)$.

For the initial approximation, note that every minimal prime in base b with at least 4 digits is of the form xY^*z, where $x \in \Sigma_b - \{\mathtt{0}\}$, $z \in \Sigma_b$, and

$$Y \coloneqq \Sigma_b - \{\, y : (p)_b \lhd xyz \text{ for some prime } p \,\}.$$

Making use of this, our algorithm sets M to be the set of base-b representations of the minimal primes with at most 3 digits (which can be found simply by brute force) and L to be $\bigcup_{x,z} xY^*z$, as described above.

All remaining minimal primes are members of L, so to find them we explore the families in L. During this process, each family will be decomposed into possibly multiple other families. For example, a simple way of exploring the family xY^*z where $Y \coloneqq \{y_1, \ldots, y_n\}$ is to decompose it into the families xY^*y_1z, …, xY^*y_nz. If the smallest member (say xy_iz) of any such family happens to be prime, it can be added to M and the family xY^*y_iz removed from consideration. Furthermore, once M has been updated it may be possible to simplify some families in L. In this case, xY^*y_jz (for $j \neq i$) can be simplified to $x(Y - \{y_i\})^*y_jz$ since no minimal prime contains xy_iz as a proper subword.

Another way of decomposing the family xY^*z is possible if one knows that a digit of Y can only occur a certain number of times. For example, if xy_iy_iz has a proper prime subword then the digit y_i can occur at most once in any minimal prime of the form xY^*z, and we can split xY^*z into the two families

$$x(Y - \{y_i\})^*z \qquad \text{and} \qquad x(Y - \{y_i\})^*y_i(Y - \{y_i\})^*z.$$

Lastly, the family xY^*z can be decomposed by considering digits of Y which are mutually incompatible, i.e., they cannot occur simultaneously in a minimal prime. For example, if xy_iy_jz and xy_jy_iz ($i \neq j$) both have proper prime subwords then the digits y_i and y_j cannot occur simultaneously in any minimal prime of the form xY^*z, and we can split xY^*z into the two families

$$x(Y - \{y_i\})^*z \qquad \text{and} \qquad x(Y - \{y_j\})^*z.$$

Sometimes it is not possible to show two digits are mutually incompatible, but it is possible to know that one digit must appear before the other. For example, if xy_iy_jz has a proper prime subword then the digit y_j must appear before y_i in any minimal prime of the form xY^*z, and we can replace xY^*z with the family

$$x(Y - \{y_i\})^*(Y - \{y_j\})^*z.$$

Similarly, if xy_jyy_iz has a proper prime subword then we can split xY^*yY^*z into the two families

$$x(Y - \{y_i\})^*yY^*z \qquad \text{and} \qquad xY^*y(Y - \{y_j\})^*z.$$

We now formulate these arguments for the most general kind of family we need to consider, namely $x_1L_1^* \cdots x_mL_m^*$. For simplicity, we only specify the decompositions as applying to $L_1 \coloneqq \{y_1, \ldots, y_n\}$, but it is straightforward to generalize these decompositions to also apply to L_i for any $1 \leq i \leq m$.

Lemma 5.5. *Every minimal prime of the form $x_1L_1^* \cdots x_mL_m^*$ must also be of the form $x_1x_2L_2^* \cdots x_mL_m^*$ or $x_1y_iL_1^*x_2L_2^* \cdots x_mL_m^*$ for some $1 \leq i \leq n$.*

Proof. Follows from the fact that $L_1^* = \{\epsilon\} \cup \bigcup_{i=1}^{n} y_iL_1^*$. □

Similarly, one can generalize Lemma 5.5 to apply to adding characters to the right of L_1 rather than to the left.

Example 5.10. The family $\texttt{10}\{\texttt{0},\texttt{1}\}^*\texttt{61}^*\texttt{1}$ splits into the families $\texttt{1061}^*\texttt{1}$, $\texttt{10}\{\texttt{0},\texttt{1}\}^*\texttt{061}^*\texttt{1}$, and $\texttt{10}\{\texttt{0},\texttt{1}\}^*\texttt{161}^*\texttt{1}$ by exploring $\{\texttt{0},\texttt{1}\}^*$ on the right.

Lemma 5.6. *If $x_1y_i^kx_2 \cdots x_m$ contains a prime proper subword (for some $k \geq 1$) then every minimal prime of the form $x_1L_1^* \cdots x_mL_m^*$ is of the form*

$$x_1(L_1 - \{y_i\})^*(y_i(L_1 - \{y_i\})^*)^jx_2L_2^* \cdots x_mL_m^*$$

for some $0 \leq j < k$.

Proof. If $w \in x_1L_1^* \cdots x_mL_m^*$ then $w \in x_1yx_2L_2^* \cdots x_mL_m^*$ for some $y \in L_1^*$. If y contains k or more instances of y_i then by assumption it follows that w contains a proper prime subword, and therefore is not a minimal prime. So if w is a minimal prime then y must contain less than k instances of y_i, i.e., y must be of the form $(L_1 - \{y_i\})^*(y_i(L_1 - \{y_i\})^*)^j$ for some $0 \leq j < k$, from which the claim follows. □

Example 5.11. The string $\texttt{661}$ represents a prime in base 9, and is a proper subword of $\texttt{10661}$. It follows that the family $\texttt{10}\{\texttt{0},\texttt{1},\texttt{6}\}^*\texttt{1}$ splits into the families $\texttt{10}\{\texttt{0},\texttt{1}\}^*\texttt{1}$ and $\texttt{10}\{\texttt{0},\texttt{1}\}^*\texttt{6}\{\texttt{0},\texttt{1}\}^*\texttt{1}$ in base 9.

Lemma 5.6 is especially useful when it can be applied with $k = 1$, since in that case the family $x_1L_1^* \cdots x_mL_m^*$ is replaced by a single strictly simpler family, in contrast to the other lemmas we will describe.

Lemma 5.7. *If $x_1y_iy_jx_2 \cdots x_m$ and $x_1y_jy_ix_2 \cdots x_m$ contain prime proper subwords (where $i \neq j$) then every minimal prime of the form $x_1L_1^* \cdots x_mL_m^*$ is of the form*

$$x_1(L_1 - \{y_i\})^*x_2L_2^* \cdots x_mL_m^* \qquad \text{or} \qquad x_1(L_1 - \{y_j\})^*x_2L_2^* \cdots x_mL_m^*.$$

Proof. If $w \in x_1L_1^* \cdots x_mL_m^*$ then $w \in x_1yx_2L_2^* \cdots x_mL_m^*$ for some $y \in L_1^*$. If y contains both y_i and y_j then by assumption it follows that w contains a proper prime subword, and therefore is not a minimal prime. So if w is a minimal prime then y cannot contain y_i and y_j simultaneously, i.e., y must be of the form $(L - \{y_i\})^*$ or $(L - \{y_j\})^*$, from which the claim follows. □

Example 5.12. The string `4611` represents a prime in base 8, and is a proper subword of `446411` and `444611`. It follows that the family $\texttt{44}\{\texttt{4},\texttt{6}\}^*\texttt{11}$ splits into the families $\texttt{444}^*\texttt{11}$ and $\texttt{446}^*\texttt{11}$ in base 8.

Lemma 5.8. *If $x_1y_iy_jx_2\cdots x_m$ contains a prime proper subword (where $i \neq j$) then every minimal prime of the form $x_1L_1^*\cdots x_mL_m^*$ is of the form*

$$x_1(L_1 - \{y_i\})^*(L_1 - \{y_j\})^*x_2L_2^*\cdots x_mL_m^*.$$

Proof. If $w \in x_1L_1^*\cdots x_mL_m^*$ then $w \in x_1yx_2L_2^*\cdots x_mL_m^*$ for some $y \in L_1^*$. If y contains y_i before y_j then by assumption it follows that w contains a proper prime subword, and therefore is not a minimal prime. So if w is a minimal prime then y cannot contain y_i before y_j, i.e., y must be of the form $(L-\{y_i\})^*(L-\{y_j\})^*$, from which the claim follows. □

Example 5.13. The string `10` represents a prime in base 11, and is a proper subword of `90101`. It follows that the family $\texttt{90}\{\texttt{0},\texttt{1},\texttt{9}\}^*\texttt{1}$ splits into the family $\texttt{90}\{\texttt{0},\texttt{9}\}^*\{\texttt{1},\texttt{9}\}^*\texttt{1}$ in base 11.

Lemma 5.9. *If $x_1y_ix_2y_{2,j}x_3\cdots x_m$ contains a prime proper subword (where $y_{2,j} \in L_2$) then every minimal prime of the form $x_1L_1^*\cdots x_mL_m^*$ is of the form*

$$x_1(L_1 - \{y_i\})^*x_2L_2^*\cdots x_mL_m^* \qquad \text{or} \qquad x_1L_1x_2(L_2 - \{y_{2,j}\})^*x_3L_3^*\cdots x_mL_m^*.$$

Proof. If $w \in x_1L_1^*\cdots x_mL_m^*$ then $w \in x_1yx_2y'x_3L_3^*\cdots x_mL_m^*$ for some $y \in L_1^*$ and $y' \in L_2^*$. If y contains y_i and y' contains $y_{2,j}$ then by assumption it follows that w contains a proper prime subword, and therefore is not a minimal prime. So if w is a minimal prime then either y cannot contain y_i or y' cannot contain $y_{2,j}$, i.e., y is of the form $(L_1 - \{y_i\})^*$ and y' is of the form L_2^*, or y is of the form L_1^* and y' is of the form $(L_2 - \{y_{2,j}\})^*$, from which the claim follows. □

Example 5.14. The string `60411` represents a prime in base 8, and is a proper subword of `604101`. It follows that the family $\texttt{60}\{\texttt{0},\texttt{4}\}^*\texttt{10}^*\texttt{1}$ splits into the families $\texttt{600}^*\texttt{10}^*\texttt{1}$ and $\texttt{60}\{\texttt{0},\texttt{4}\}^*\texttt{11}$ in base 8.

We call families of the form xy^*z (where x, $z \in \Sigma_b^*$ and $y \in \Sigma_b$) *simple* families. Our algorithm then proceeds as follows:

1. $M \coloneqq \{\text{minimal primes in base } b \text{ of length} \leq 3\}$
   $L \coloneqq \bigcup_{x,z\in\Sigma_b} xY^*z$, where $x \neq \texttt{0}$ and Y is the set of digits y such that xyz has no subword in M

2. While L contains non-simple families:

   (a) Explore each family of L by applying Lemma 5.5, and update L.

   (b) Examine each family of L:

      i. Let w be the shortest string in the family. If w has a subword in M, then remove the family from L. If w represents a prime, then add w to M and remove the family from L.

      ii. If possible, simplify the family by applying Lemma 5.6 with $k = 1$.

      iii. Using the techniques of Section 5.4, check if the family can be proven to only contain composites, and if so then remove the family from L.

   (c) Apply Lemmas 5.6, 5.7, 5.8, and 5.9 to the families of L as much as possible and update L; after each split examine the new families as in (b).

The process of exploring/examining/splitting a family can be concisely expressed in a tree of decompositions. A sample tree of decompositions, for the family $\mathtt{1}\{\mathtt{0},\mathtt{1},\mathtt{6}\}^*\mathtt{1}$ in base 9, is displayed in Figure 5.1. When applying Lemma 5.5 on a family with only one nonempty L_i we don't show the first decomposition (simply removing L_i^*) since that results in the shortest word in the family, which is always implicitly checked for primality/minimal subwords at each step anyway.

5.5.1 Implementation

An implementation of our algorithm as described above was written in C using the GMP library [Granlund et al., 1991] and the code is open source [Bright, 2014]. An independent implementation using the same ideas was written in C++ using the MIRACL library [Scott, 2003] and the same results were derived (with a less extensive search for primes in the simple families, but with results for bases $b \leq 50$ [Devillers, 2016]).

Note that when exploring the families in line (a), one should not always apply Lemma 5.5 directly as stated by adding characters to the left of L_1; in our implementation we alternate between adding characters to the left and right of $L_{1+k \bmod m}$, where k is the number of times we have previously passed through line (a). Also, the application of Lemma 5.8 tended to initially produce an excess number of cases, so it was useful to only apply it following a certain number of iterations (6 in our implementation). It also led to duplicate families after simplifying, for example families of the form $xy^*y^ny^*z$ for varying n, but these could be detected and removed.

At the conclusion of the algorithm described, L will consist of simple families (of the form xy^*z) which have not yet yielded a prime, but for which there is no obvious reason why there can't be a prime of such a form. In such a case, the only way to proceed is to test the primality of larger and larger numbers of such form and hope a prime is eventually discovered.

The numbers in simple families are of the form $(ab^n + c)/d$ for some fixed a, b, c, d where $d \mid ab^n + c$ for all n. Except in the special case $c = \pm 1$ and $d = 1$, when n is large the known primality tests for such a number are too inefficient to run. In this case one must resort to a probable primality test such as a Miller–Rabin test, unless a divisor of the number can be found. Since we are testing many numbers in an exponential sequence, it is possible to use a sieving process to find divisors rather than using trial division.

To do this, we made use of Geoffrey Reynolds' SRSIEVE software [Reynolds, 2010]. This program uses the baby-step giant-step algorithm to find all primes p which divide $ab^n + c$ where p and n lie in a specified range. Since this program cannot handle the general case $(ab^n + c)/d$ when $d > 1$ we only used it to sieve the sequence $ab^n + c$ for primes $p \nmid d$, and initialized the list of candidates to not include n for which there is some prime $p \mid d$ for which $p \mid (ab^n + c)/d$. The program had to be modified slightly to remove a check which would prevent it from running in the case when a, b, and c were all odd (since then $2 \mid ab^n + c$, but 2 may not divide $(ab^n + c)/d$).

Once the numbers with small divisors had been removed, it remained to test the remaining numbers using a probable primality test. For this we used the software LLR [Penné, 2015] written by Jean Penné. Although undocumented, it is possible to run this program on numbers of the form $(ab^n + c)/d$ when $d \neq 1$, so this program required no modifications. A script was also written which allowed one to run SRSIEVE while LLR was testing the remaining candidates, so that when a divisor was found by SRSIEVE on a number which had not yet been tested by LLR it would be removed from the list of candidates.

In the cases where the elements of $M(L_b)$ could be proven prime rigorously we employed the program PRIMO [Martin, 2011] which is an elliptic curve primality proving implementation written by Marcel Martin.

5.6 Results

A summary of the results of our algorithm is presented in Table 5.1; it was able to completely solve all bases up to 30 except for 17, 19, 21, and those between 25 and 29. The results in

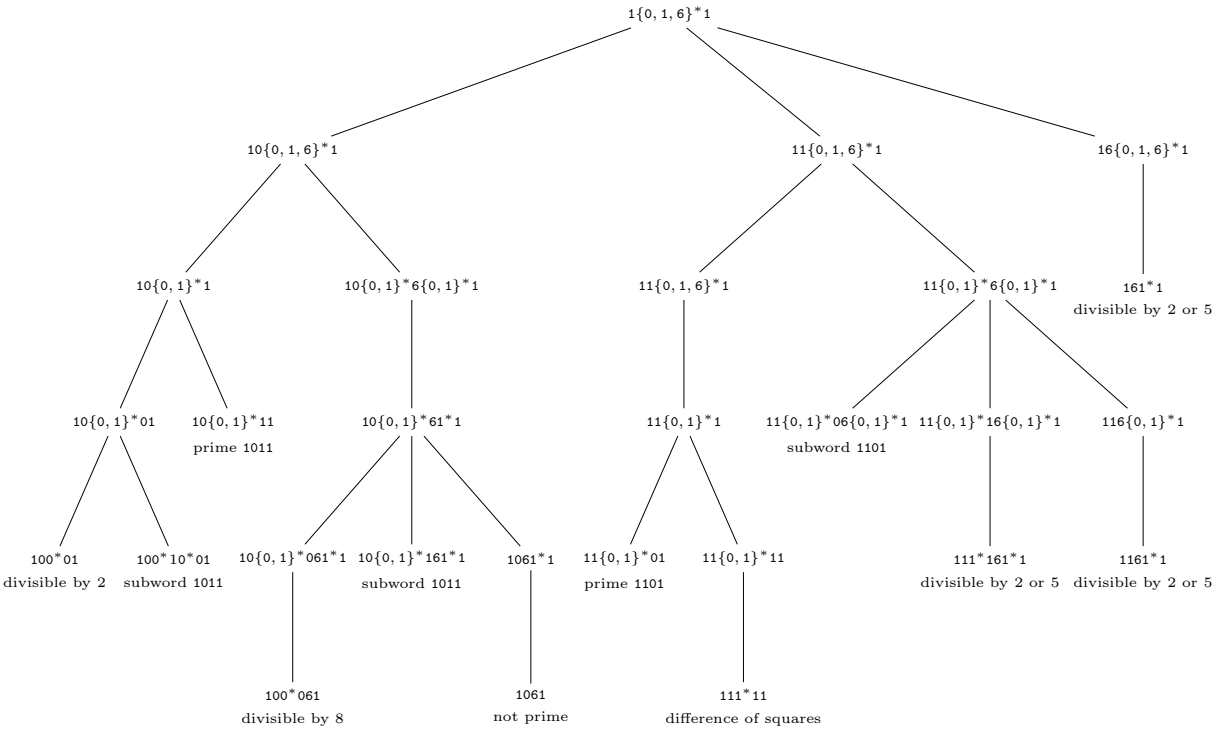

Figure 5.1: Tree of decompositions for $1\{0,1,6\}^*1$ in base 9.

base 29 required some additional strategies, as described in Section 5.7. The full collection of minimal elements is available online [Bright, 2012].

For every solved base, we give the size $|M(L_b)|$ and the "width" $\max_{x \in M(L_b)} |x|$ of the corresponding family. For the unsolved bases, we give a lower bound on the size and width of $M(L_b)$, along with the number of families of the form xy^*z for which no prime member could be found, nor could the family be ruled out as only containing composites. Since such simple families can contain at most one minimal prime, an upper bound on $|M(L_b)|$ is given by the sum of the lower bound for $|M(L_b)|$ and the number of unsolved families. Additionally, we give the height to which the simple families were searched for primes; if there are any more primes in $M(L_b)$ they must have at least this many digits in base b.

The family $\texttt{80}^*\texttt{1}$ in base 23, corresponding to the generalized Proth numbers $8 \cdot 23^n + 1$, was already known to be prime for minimal $n = 119215$ in the process of solving the generalized Sierpiński conjecture in base 23 [Barnes, 2007].

The largest probable prime we found was the number $\texttt{9E}^{800873}$ (expressed as a base 23 string) or $(106 \cdot 23^{800873} - 7)/11$ in standard notation. It contains 1,090,573 decimal digits and at the time of discovery was the tenth largest known probable prime according to Henri and Renaud Lifchitz's ranking [Lifchitz and Lifchitz, 2000].

5.6.1 Unsolved families

There were 37 families for which we were unable to determine if they contain a prime or not. They are explicitly described in Table 5.2 in both their "base b" string representation xy^*z and "algebraic" form $(ab^n + c)/d = [xy^nz]_b$. The numbers a and c are given in their factorized form so as to help spot algebraic factorizations.

5.6.2 Primes of the form $4n + 1$ and $4n + 3$

The heuristic algorithm described in Section 5.5 may also be easily modified to apply to recursive subsets of prime numbers, e.g., those congruent to 1 mod 4 (alternatively, 3 mod 4). The modifications necessary are to the initialization of M in line 1, and to update the check "w represents a minimal prime" to also check that w is in the subset under consideration. When searching for minimal primes of the form $4n + 1$, it was necessary to remove families only containing numbers of the form $4n + 3$, and vice versa. A modified Lemma 5.1 can be used to check this, e.g., $[x(L^0 \cup L)z]_b$ are all congruent to c mod d implies that $[xL^*z]_b$ are all congruent to c mod d.

b	$\lvert M(L_b)\rvert$	$\max_{x\in M(L_b)} \lvert x\rvert$	# unsolved families	searched height
2	2	2	0	–
3	3	3	0	–
4	3	2	0	–
5	8	5	0	–
6	7	5	0	–
7	9	5	0	–
8	15	9	0	–
9	12	4	0	–
10	26	8	0	–
11	152	45	0	–
12	17	8	0	–
13*	228	32,021	0	–
14	240	86	0	–
15	100	107	0	–
16	483	3545	0	–
17*	$\geq$ 1279	$\geq$ 111,334	1	1,000,000
18	50	33	0	–
19*	$\geq$ 3462	$\geq$ 110,986	1	707,000
20	651	449	0	–
21*	$\geq$ 2600	$\geq$ 479,150	1	506,000
22	1242	764	0	–
23*	6021	800,874	0	–
24	306	100	0	–
25*	$\geq$ 17,597	$\geq$ 136,967	12	303,000
26	$\geq$ 5662	$\geq$ 8773	2	486,000
27*	$\geq$ 17,210	$\geq$ 109,006	5	368,000
28*	$\geq$ 5783	$\geq$ 94,538	1	543,000
29*	$\geq$ 57,283	$\geq$ 174,240	14	242,000
30	220	1024	0	–

*Data based on results of probable primality tests.

Table 5.1: Summary of results for the prime numbers for each base b.

Base	Family	Algebraic form
17	F19*	$(5 \cdot 821 \cdot 17^n - 3^2)/16$
19	EE16*	$(2^2 \cdot 13 \cdot 307 \cdot 19^n - 1)/3$
21	G0*FK	$2^4 \cdot 21^{n+2} + 5 \cdot 67$
25	6MF*9	$(1381 \cdot 25^{n+1} - 53)/8$
	CM1*	$(59 \cdot 131 \cdot 25^n - 1)/24$
	EE1*	$(8737 \cdot 25^n - 1)/24$
	E1*E	$(337 \cdot 25^{n+1} + 311)/24$
	EFO*	$2 \cdot 3 \cdot 61 \cdot 25^n - 1$
	F1*F1	$(19^2 \cdot 25^{n+2} + 37 \cdot 227)/24$
	F0*KO	$3 \cdot 5 \cdot 25^{n+2} + 2^2 \cdot 131$
	F0K*O	$(5 \cdot 11 \cdot 41 \cdot 25^{n+1} + 19)/6$
	LOL*8	$(53 \cdot 83 \cdot 25^{n+1} - 3 \cdot 37)/8$
	M1*F1	$(23^2 \cdot 25^{n+2} + 37 \cdot 227)/24$
	M10*8	$19 \cdot 29 \cdot 25^{n+1} + 2^3$
	OL*8	$(199 \cdot 25^{n+1} - 3 \cdot 37)/8$
26	A*6F	$(2 \cdot 26^{n+2} - 7 \cdot 71)/5$
	I*GL	$(2 \cdot 3^2 \cdot 26^{n+2} - 11 \cdot 113)/25$
27	80*9A	$2^3 \cdot 27^{n+2} + 11 \cdot 23$
	999G*	$(101 \cdot 877 \cdot 27^n - 2^3)/13$
	CL*E	$(3^2 \cdot 37 \cdot 27^{n+1} - 7 \cdot 29)/26$
	EI*F8	$(191 \cdot 27^{n+2} - 2^3 \cdot 149)/13$
	F*9FM	$(3 \cdot 5 \cdot 27^{n+3} - 113557)/26$
28	OA*F	$(2 \cdot 7 \cdot 47 \cdot 28^{n+1} + 5^3)/27$
29	1A*	$(19 \cdot 29^n - 5)/14$
	68LO*6	$7 \cdot 757 \cdot 29^{n+1} + 2 \cdot 3$
	AMP*	$(8761 \cdot 29^n - 5^2)/28$
	C*FK	$(3 \cdot 29^{n+2} + 2 \cdot 331)/7$
	F*OPF	$(3 \cdot 5 \cdot 29^{n+3} + 139 \cdot 1583)/28$
	FKI*	$(6379 \cdot 29^n - 3^2)/14$
	F*OP	$(3 \cdot 5 \cdot 29^{n+2} + 7573)/28$
	LPO9*	$(31 \cdot 16607 \cdot 29^n - 3^2)/28$
	OOPS*A	$2 \cdot 10453 \cdot 29^{n+1} - 19$
	PC*	$(2 \cdot 89 \cdot 29^n - 3)/7$
	PPPL*O	$(87103 \cdot 29^{n+1} + 3^2)/4$
	Q*GL	$(13 \cdot 29^{n+2} - 3 \cdot 1381)/14$
	Q*LO	$(13 \cdot 29^{n+2} - 19 \cdot 109)/14$
	RM*G	$(389 \cdot 29^{n+1} - 5 \cdot 19)/14$

Table 5.2: The unsolved families listed in their "base b" and "algebraic" representations.

The lemmas of Section 5.4 can be used without modification, since a family which contains only composites of course does not contain any primes of a special form, either. For simplicity the lemmas of Section 5.5 are stated to apply when the minimal set consists of primes, but actually apply to general minimal sets and also need no modification.

When run on the primes of the form $4n+1$ represented in base 10, our implementation successfully computes the minimal set consisting of 146 elements, the largest of which contains 79 digits. The *Encyclopedia of Integer Sequences* [OEIS Foundation Inc., 1996] contains the elements for this minimal set in entry A111055. Prior to our work, the longest 11 elements were missing from this listing.

When run on the primes of the form $4n+3$ represented in base 10, our implementation successfully computes the minimal set consisting of 113 elements, the largest of which contains 19,153 digits. Because of its size this number is not easily proven prime, but François Morain successfully produced a primality certificate for it [Morain, 2015]. The second-largest number has 50 digits, so the remaining elements are easily proven prime. The *Encyclopedia of Integer Sequences* [OEIS Foundation Inc., 1996] contains the elements for this minimal set in entry A111056. Prior to our work, the longest 10 elements were missing from this listing.

5.7 Some additional strategies

The strategies discussed so far suffice to restrict the possible forms of minimal primes to a finite number of simple families in all bases $2 \le b \le 28$. However, as b increased, in addition to the calculations becoming more costly, it was found to be necessary to use increasingly complicated strategies. We now describe some additional strategies which we found sufficient to solve all non-simple families in base 29.

Lemma 5.10. *If every number of the form $x_1(L_1 - \{y_i\})^* x_2 L_2^* \cdots x_m L_m^*$ is composite, then every minimal prime of the form $x_1 L_1^* \cdots x_m L_m^*$ must also be of the form $x_1 L_1^* y_i L_1^* x_2 L_2^* \cdots x_m L_m^*$.*

Proof. If $w \in x_1 L_1^* \cdots x_m L_m^*$ then $w \in x_1 y x_2 L_2^* \cdots x_m L_m^*$ for some $y \in L_1^*$. If y does not contain a y_i then w is composite by assumption. Therefore if w is a prime then y contains a y_i, i.e., $y \in L_1^* y_i L_1^*$, from which the result follows. (Note that one could improve this result via $y \in (L_1 - \{y_i\})^* y_i L_1^* \cup L_1^* y_i (L_1 - \{y_i\})^*$, but this was found to be unnecessary for our purposes.) □

Example 5.15. The numbers represented by the family $\texttt{F}\{\texttt{0},\texttt{9},\texttt{F}\}^*\texttt{F}$ in base 29 are divisible by 3, so every minimal prime of the form $\texttt{F}\{\texttt{0},\texttt{9},\texttt{F},\texttt{P}\}^*\texttt{F}$ must also be of the form $\texttt{F}\{\texttt{0},\texttt{9},\texttt{F},\texttt{P}\}^*\texttt{P}\{\texttt{0},\texttt{9},\texttt{F},\texttt{P}\}^*\texttt{F}$.

Lemma 5.11. *If $x_1y_iy_jy_ix_2\cdots x_m$ contains a prime proper subword (where $i \neq j$) then every minimal prime of the form $x_1L_1^*\cdots x_mL_m^*$ is of the form*

$$x_1(L_1 - \{y_i\})^*(L_1 - \{y_j\})^*(L_1 - \{y_i\})^*x_2L_2^*\cdots x_mL_m^*.$$

Proof. If $w \in x_1L_1^*\cdots x_mL_m^*$ then $w \in x_1yx_2L_2^*\cdots x_mL_m^*$ for some $y \in L_1^*$. If y contains $y_iy_jy_i$ then by assumption it follows that w contains a proper prime subword, and therefore is not a minimal prime. So if w is a minimal prime then either y does not contain a y_j, or y contains a y_j and all y_is in y either come before or after the y_j. In each case, y is of the form $(L - \{y_i\})^*(L - \{y_j\})^*(L - \{y_i\})^*$, from which the claim follows. □

Example 5.16. The string $\texttt{QLQ}$ represents a prime in base 29, and is a proper subword of $\texttt{LQLQL}$. It follows that the family $\texttt{L}\{\texttt{L},\texttt{Q}\}^*\texttt{L}$ splits into the family $\texttt{LL}^*\texttt{Q}^*\texttt{L}^*\texttt{L}$ in base 29.

This rule may also be generalized to apply to the case when $x_1y_iy_jy_iy_jx_2\cdots x_m$ contains a prime proper subword (where $i \neq j$).

Example 5.17. The string $\texttt{LL9L9LQL}$ represents a prime in base 29. It follows that the family $\texttt{LL}\{\texttt{9},\texttt{L}\}^*\texttt{Q}^*\texttt{QL}$ splits into the family $\texttt{LLL}^*\texttt{9}^*\texttt{L}^*\texttt{9}^*\texttt{Q}^*\texttt{QL}$ in base 29.

In Section 5.4.1 we described a number of strategies for determining if every member of a family has a divisor, but for some families divisors exist which will not be found using those tests. The following lemma can help one discover when this is the case; in particular when every member of a family is divisible by some small prime. For a language L we use the notation $[L]_b \bmod N$ to denote the finite set of residues $\{\, [x]_b \bmod N : x \in L \,\}$.

Lemma 5.12. *If $1 < \gcd(k, N)$ for every $k \in [L]_b \bmod N$ and $p \notin [L]_b$ for every prime p which divides N, then all numbers of the form $[L]_b$ are composite.*

Proof. Let x be an arbitrary member of L. Then $\gcd([x]_b, N) = \gcd([x]_b \bmod N, N) > 1$ by assumption, so $[x]_b$ cannot be prime unless $[x]_b = \gcd([x]_b, N)$. But in that case $[x]_b$ would be a prime which divides N, and so $[x]_b \notin [L]_b$ by the second assumption, in contradiction to $x \in L$. □

To make use of this lemma, we need to be able to compute $[x_1 L_1^* \cdots x_m L_m^*]_b \bmod N$. To do this, we can make use of the following relations:

$$\begin{aligned}
[Lx]_b \bmod N &= (b \cdot ([L]_b \bmod N) + [x]_b) \bmod N \\
[L\{y_1, \ldots, y_n\}]_b \bmod N &= \bigcup_{i=1}^{n} [Ly_i]_b \bmod N \\
[L\{y_1, \ldots, y_n\}^*]_b \bmod N &= \bigcup_{i=0}^{\infty} [L\{y_1, \ldots, y_n\}^i]_b \bmod N
\end{aligned}$$

In the final case, the union may be taken to be finite over $i < l$ where l is chosen such that $\bigcup_{i=0}^{l} [L\{y_1, \ldots, y_n\}^i]_b \bmod N = \bigcup_{i=0}^{l-1} [L\{y_1, \ldots, y_n\}^i]_b \bmod N$. Using these relations, we can compute $[x_1 L_1^* \cdots x_m L_m^*]_b \bmod N$ progressively, working left to right and starting from $[\emptyset]_b \bmod N = \{0\}$. To solve base 29 it was sufficient to use $N = 2 \cdot 3 \cdot 5$.

Example 5.18. Let $b \coloneqq 29$ and $N \coloneqq 30$. Then

$$[\texttt{L1}^*\texttt{61}^*\texttt{LK}^*\texttt{K}]_b \bmod N = \{4, 5, 6, 14, 15, 16, 25\},$$

so if k is in this set then $\gcd(k, N) \in \{2, 5, 6, 15\}$ and $\gcd(k, N) > 1$. Since $2,\ 3,\ 5 \notin [\texttt{L1}^*\texttt{61}^*\texttt{LK}^*\texttt{K}]_b$, by Lemma 5.12 all numbers of the form $\texttt{L1}^*\texttt{61}^*\texttt{LK}^*\texttt{K}$ are composite.

Truth is difficult.

Tim Bray

Chapter 6

Conclusion

In this thesis we have studied some computational methods for certain combinatorial and number theoretic problems. In particular, we have demonstrated the effectiveness of the recently proposed SAT+CAS paradigm of combining tools and methods from the symbolic computation and satisfiability checking communities.

In the course of our work in the SAT+CAS area we applied the programmatic SAT idea of [Ganesh et al., 2012] and provided evidence of its effectiveness in speeding up a SAT solver searching for Williamson matrices. In this context enabling the programmatic functionality consistently made the search finish faster; programmatic timings were often 10 times faster and sometimes over 100 times faster. Additionally, many of the SAT instances which we generated were only solved using programmatic functionality.

6.1 When the SAT+CAS paradigm is likely to be effective

Of course, the SAT+CAS paradigm is not something which can be effortlessly applied to solve problems, and should not be expected to be useful for all types of problems. While completing our work in this area we performed many experiments concerning the case studies in this thesis and compared many different search techniques, encodings, splitting methods, and types of domain-specific knowledge. Figure 6.1 contains an outline of the MathCheck2 system but annotated with the generic techniques that we found useful in our experiments.

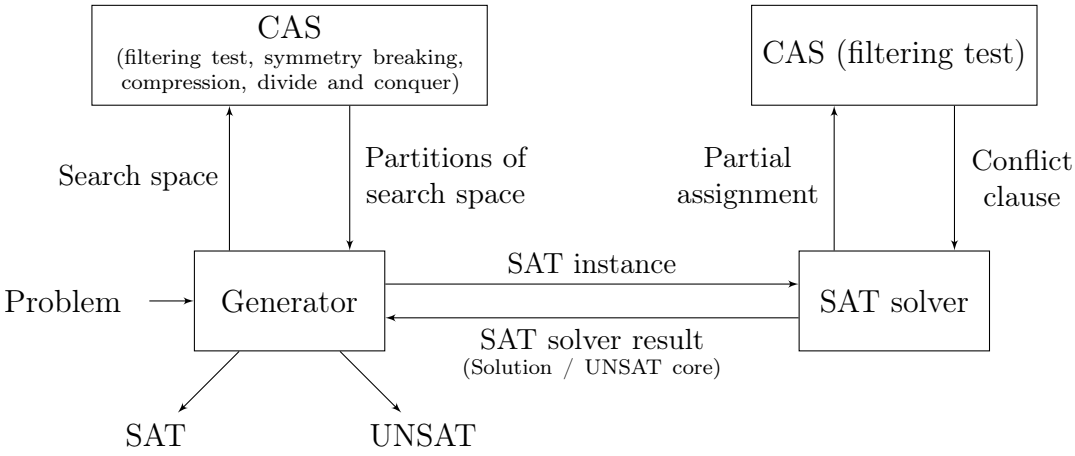

Figure 6.1: Diagram of the MathCheck2 system annotated with the generic techniques used.

Determining which techniques were the most effective was a nontrivial problem in its own right, since the heuristics used in the SAT solver can sometimes get unusually lucky or unlucky, making the solver's running time inconsistent. Because of this, using the SAT solver's runtime on isolated instances was an unreliable way to determine which techniques were the best to use. Instead, we compared the SAT solver's runtime over a class of instances, like the those given in the tables of Section 3.6. One should also keep in mind that some techniques perform well on very small instances but do not scale well. For example, the naive "binomial" encoding of the cardinality constraint which says that exactly half of $2n$ given variables are true contains about $\binom{2n}{n} \sim 4^n/\sqrt{\pi n}$ constraints. This is much too inefficient for large n, though the encoding may perform well for small n. Thus, one cannot only rely on small instances when testing the effectiveness of different techniques.

Our experience running these experiments and seeing which were effective at making the searches run efficiently means that we can offer some guidance about which kinds of problems the SAT+CAS paradigm is likely to be useful for. In particular, we highlight the following properties of problems which makes them good candidates to study using the SAT+CAS paradigm:

1. *There is an efficient encoding of the problem into a Boolean setting.* Since the problem has to be translated into a SAT instance or multiple SAT instances the encoding should ideally be straightforward and easy to compute. Not only does this make the process of generating the SAT instances easier and less error-prone it also means that the SAT solver is executing its search through a domain which is closer to the original

problem. The more convoluted the encoding the less likely the SAT solver will be able to efficiently search the space.

2. *There is some way of dividing the Boolean formula into multiple instances using the knowledge from a CAS.* Of course, a SAT instance can always be split into multiple instances by hard-coding the values of certain variables and then generating instances which cover all possible assignments of those variables. However, this strategy was not an ideal way of splitting the search space. The instances generated in this fashion tended to have wildly unbalanced difficulties, with some very easy instances and some very hard instances, limiting the benefits of using many processors to search the space. Instead, the process of splitting using domain-specific knowledge allows instances which cannot be ruled out *a priori* to not even need to be generated because they encode some part of the search space which can be discarded based on domain-specific knowledge.

3. *The search space can be split into a reasonable number of cases.* One of the disadvantages of using SAT solvers is that it can be difficult to tell how much progress is being made as the search is progressing. In our experience, if the solver has not finished running in 24 hours then it is unlikely to finish at all in a reasonable amount of time. The process of splitting the search space allows one to get a better estimate of the progress being made, assuming the difficulty of the instances isn't extremely unbalanced. As a rule of thumb, splitting the search space into 100 to 10,000 instances seems to work well, at least in the cases we examined. This allowed each instance to complete significantly faster than the original instance would have taken to complete while not having too many calls to the SAT solver. Splitting the search space into too many instances is suboptimal because with a very large number of cases the overhead of repeatedly calling a SAT solver becomes more significant to the total running time.

4. *The SAT solver can learn something about the space as the search is running.* The efficiency of SAT solvers is in part due to the facts that they learn as the search progresses. It can often be difficult for a human to make sense of these facts but they play a vital role to the SAT solver internally and therefore a problem where the SAT solver can take advantage of its ability to learn nontrivial clauses is one in which the SAT+CAS paradigm is well suited for. As an example, we showed in Section 3.6.4 that the SAT solver was able to learn Lemma 3.4 on its own. For more sophisticated lemmas that the SAT solver would be unlikely to learn (because they rely on domain-specific knowledge) it is useful to learn clauses programmatically via the programmatic SAT idea [Ganesh et al., 2012].

5. *There is domain-specific knowledge which can be efficiently given to the SAT solver.* Domain-specific knowledge was found to be critical to solving instances of the problems besides those of the smallest sizes. The instances which were generated using naive encodings were typically only able to be solved for small sizes and past a certain point the instances became too expensive to solve. All significant increases in the size of the problems past that point came from the usage of domain-specific knowledge, showing how valuable it is to the SAT solver. Of course, for the information to be useful to the solver there needs to be an efficient way for the solver to be given the information; it can be encoded directly in the SAT instances or generated on-the-fly using programmatic SAT functionality.

6. *The solutions of the problem lie in spaces which cannot be simply enumerated.* If the search space is highly structured and there exists an efficient algorithm for searching the space which exploits that structure then using this algorithm directly is likely to be more efficient than using a SAT solver. For example, in Chapter 4.3 we use an algorithm for generating permutations whose form allows them to be enumerated with little overhead. A SAT solver can also perform this enumeration, but it is not likely to be able to do it faster than an algorithm specifically designed to do this—unless there are lemmas which allow one to avoid enumerating *all* such permutations.

6.2 Future work

In this thesis the effectiveness of the SAT+CAS paradigm was demonstrated in the case study of computing Williamson matrices (see Chapter 3). Although this is a topic of interest in its own right, the scope of the SAT+CAS idea is much broader than its application to this one case study and the SAT+CAS approach shows potential to be an effective method for solving many other computational problems. On the other hand, we have also pointed out that the approach is not something that can be expected to be helpful if just applied blindly. Currently the SAT+CAS approach requires a lot of experimentation and trial-and-error before finding a combination of encodings, domain-specific knowledge, and splitting methods which perform well together.

Many of the techniques we discussed for computing Williamson matrices apply to related combinatorial problems but will not necessarily apply to problems from other fields. Thus, an important area of future research is to apply the SAT+CAS approach to different problems in different fields. Furthermore, it would be very useful to determine if there are general principles for selecting good encodings, domain-specific knowledge, and

splitting methods which could be applied across multiple domains. Ideally, one would like an automated framework for choosing and using optimal encodings, knowledge, and methods for the problem at hand but we are currently far from such an automated system and more experience will be necessary before we can design such a system. It could also be very useful for the system to make more use of the facts learned when solving the SAT instances to make solving other instances more efficient or to aid the mathematician in the discovery of theorems. Section 3.6.4 contains some preliminary results of this type, allowing us to state and prove Lemma 3.4, but it would be nice for the system to learn even more sophisticated facts.

It would also be interesting to see if the SAT+CAS paradigm could be useful in the other case studies in this thesis. In the case of computing minimal primes it is unclear if the SAT+CAS idea would be useful. It would be unlikely to be useful for searching through the simple families for primes, as this process involved sieving for primality candidates and then testing the candidates for primality using a primality test, which the programs SRSIEVE and LLR have been fined-tuned to do directly. On the other hand, our heuristic algorithm also relies on a search through families of strings the form $x_1 L_1^* \cdots x_m L_m^*$ while using the lemmas of Sections 5.4 and 5.5 to refine the search. One could potentially use a string SMT solver coupled with those lemmas but it remains to be seen if this would be effective in practice.

For our remaining case study of computing complex Golay sequences, some initial experimentation using a SAT solver to perform the search was already performed, but more work remains to be done. In the case where the SAT solver was searching for complex Golay sequences of order n with u entries which were 1, v entries which were -1, x entries which were i, and y entries which were $-i$, the SAT solver would usually enumerate all $\binom{n}{u,v,x,y}$ sequences of that form, i.e., when one sequence was shown to not be a Golay sequence it was not able to use that information to learn something which would also discount other similar sequences.

Because the SAT solver was internally enumerating all sequences of the given form it makes sense to use an algorithm specifically tailored to do that, as we did in Chapter 4. On the other hand, this does not definitively mean the SAT+CAS paradigm is useless for this problem, as there is still the possibility of using additional domain-specific knowledge to make the search run more efficiently. The mathematician Frank Fiedler gives a criterion [Fiedler, 2013] which could potentially be useful in this context. He shows that if $A = [a_1, \ldots, a_n]$ is a complex Golay sequence and $z \in \mathbb{C}$ with $|z| = 1$ then not only must we have $|h_A(z)|^2 \leq 2n$ (see Corollary 4.1), we must even have $|h_{A'}(z)|^2 \leq 2n$ where A' is a subsequence of A whose entries all have indicies in the same equivalence class modulo some

constant. For example, if $n = 23$ then possibilities for A' are

$$[a_1, a_3, a_5, \ldots, a_{23}] \qquad \text{and} \qquad [a_2, a_4, a_6, \ldots, a_{22}].$$

This type of filtering criteria is the sort of domain-specific knowledge which we found useful in our work with SAT+CAS systems. It meets the criteria we outlined in Section 3.4 of being nontrivial, useful, and efficiently computable. Additionally, there is no simple way of modifying permutation enumeration algorithms like Algorithm 7.2.1.2L from [Knuth, 2011] to *only* enumerate permutations which satisfy Fiedler's criterion. The simplest method would be to enumerate all permutations and then filter those which fail the criterion. A SAT solver, on the other hand, if given access to the domain-specific knowledge, can use that knowledge to speed up the search by *not* enumerating permutations which don't satisfy the filtering criterion.

References

[Ábrahám, 2015] Ábrahám, E. (2015). Building bridges between symbolic computation and satisfiability checking. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, pages 1–6, New York. ACM.

[Ábrahám et al., 2016a] Ábrahám, E., Abbott, J., Becker, B., Bigatti, A. M., Brain, M., Buchberger, B., Cimatti, A., Davenport, J. H., England, M., Fontaine, P., Forrest, S., Griggio, A., Kroening, D., Seiler, W. M., and Sturm, T. (2016a). SC^2: Satisfiability Checking meets Symbolic Computation (Project Paper). In *Intelligent Computer Mathematics: 9th International Conference, CICM 2016, Bialystok, Poland, July 25–29, 2016, Proceedings*, pages 28–43, Cham. Springer International Publishing. http://www.sc-square.org/.

[Ábrahám et al., 2016b] Ábrahám, E., Fontaine, P., Sturm, T., and Wang, D. (2016b). Symbolic Computation and Satisfiability Checking (Dagstuhl Seminar 15471). *Dagstuhl Reports*, 5(11):71–89.

[Appel and Haken, 1976] Appel, K. and Haken, W. (1976). Every planar map is four colorable. *Bull. Amer. Math. Soc.*, 82(5):711–712.

[Barnes, 2007] Barnes, G. (2007). Sierpinski conjectures and proofs. http://www.noprimeleftbehind.net/crus/Sierp-conjectures.htm.

[Barras et al., 1997] Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al. (1997). *The* Coq *proof assistant reference manual: Version 6.1*.

[Bauer, 2016] Bauer, M. (2016). A Message from the Scientific Director. https://www.sharcnet.ca/my/about/sd_message.

[Baumert et al., 1962] Baumert, L., Golomb, S. W., and Hall, M. (1962). Discovery of an Hadamard matrix of order 92. *Bull. Amer. Math. Soc.*, 68(3):237–238.

[Biere et al., 2009] Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2009). *Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185*. ios Press.

[Borwein and Ferguson, 2004] Borwein, P. B. and Ferguson, R. A. (2004). A complete description of Golay pairs for lengths up to 100. *Math. Comp.*, 73(246):967–985.

[Bosma et al., 1997] Bosma, W., Cannon, J., and Playoust, C. (1997). The MAGMA algebra system I: The user language. *Journal of Symbolic Computation*, 24(3):235–265.

[Bright, 2012] Bright, C. (2012). Computed data for the MEPN project. github.com/curtisbright/mepn-data/tree/master/data.

[Bright, 2014] Bright, C. (2014). MEPN implementation. github.com/curtisbright/mepn.

[Bright, 2016a] Bright, C. (2016a). Complex Golay Sequences. https://cs.uwaterloo.ca/~cbright/golay/.

[Bright, 2016b] Bright, C. (2016b). MATHCHECK2 source scripts. https://bitbucket.org/cbright/mathcheck2.

[Bright, 2017] Bright, C. (2017). Williamson Sequences. https://cs.uwaterloo.ca/~cbright/williamson/.

[Bright et al., 2016a] Bright, C., Devillers, R., and Shallit, J. (2016a). Minimal elements for the prime numbers. *Experimental Mathematics*, 25(3):321–331.

[Bright et al., 2016b] Bright, C., Ganesh, V., Heinle, A., Kotsireas, I. S., Nejati, S., and Czarnecki, K. (2016b). MATHCHECK2: A SAT+CAS Verifier for Combinatorial Conjectures. In Gerdt, V. P., Koepf, W., Seiler, W. M., and Vorozhtsov, E. V., editors, *Computer Algebra in Scientific Computing - 18th International Workshop, CASC 2016, Bucharest, Romania, September 19-23, 2016, Proceedings*, volume 9890 of *Lecture Notes in Computer Science*, pages 117–133. Springer.

[Char et al., 1986] Char, B. W., Fee, G. J., Geddes, K. O., Gonnet, G. H., and Monagan, M. B. (1986). A tutorial introduction to MAPLE. *Journal of Symbolic Computation*, 2(2):179–200.

[Choi, 1971] Choi, S. (1971). Covering the set of integers by congruence classes of distinct moduli. *Mathematics of Computation*, 25(116):885–895.

[Colbourn and Dinitz, 2007] Colbourn, C. J. and Dinitz, J. H., editors (2007). *Handbook of Combinatorial Designs*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, second edition.

[Cooper, 2013] Cooper, J. (2013). NASA Jet Propulsion Laboratory Blog | Hadamard Matrix. http://www.jpl.nasa.gov/blog/2013/8/hadamard-matrix.

[Craigen, 1994] Craigen, R. (1994). Complex Golay sequences. *J. Combin. Math. Combin. Comput.*, 15:161–169.

[Craigen et al., 2002] Craigen, R., Holzmann, W., and Kharaghani, H. (2002). Complex Golay sequences: Structure and applications. *Discrete Mathematics*, 252(1-3):73–89.

[Davenport, 2016] Davenport, J. (2016). SC^2: Satisfiability Checking meets Symbolic Computation: www.sc-square.org. http://staff.bath.ac.uk/masjhd/Slides/SCSC-ACA16-slides-post.pdf.

[Davis and Jedwab, 1999] Davis, J. A. and Jedwab, J. (1999). Peak-to-mean power control in OFDM, Golay complementary sequences, and Reed-Muller codes. *IEEE Transactions on Information Theory*, 45(7):2397–2417.

[Devillers, 2016] Devillers, R. (2016). Results in searching for minimal primes wrt subword order. github.com/RaymondDevillers/primes.

[Dickson, 1952] Dickson, L. (1952). *History of the Theory of Numbers: Divisibility and primality, Volume 1*. Carnegie institution of Washington. Chelsea.

[Đoković, 1993] Đoković, D. Ž. (1993). Williamson matrices of order $4n$ for $n = 33, 35, 39$. *Discrete Mathematics*, 115(1):267–271.

[Đoković, 1998] Đoković, D. Ž. (1998). Equivalence classes and representatives of Golay sequences. *Discrete Mathematics*, 189(1-3):79–93.

[Đoković and Kotsireas, 2015] Đoković, D. Ž. and Kotsireas, I. S. (2015). Compression of periodic complementary sequences and applications. *Designs, Codes and Cryptography*, 74(2):365–377.

[Eén and Sörensson, 2004] Eén, N. and Sörensson, N. (2004). An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers*, pages 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Fiedler, 2013] Fiedler, F. (2013). Small Golay sequences. *Advances in Mathematics of Communications*, 7(4).

[Frigo and Johnson, 2005] Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231.

[Frigo and Johnson, 2014] Frigo, M. and Johnson, S. G. (2014). FFTW FAQ - Section 3. http://www.fftw.org/faq/section3.html.

[Ganesh et al., 2015] Ganesh, V. et al. (2015). MathCheck. https://sites.google.com/site/uwmathcheck/home.

[Ganesh et al., 2012] Ganesh, V., O'Donnell, C. W., Soos, M., Devadas, S., Rinard, M. C., and Solar-Lezama, A. (2012). Lynx: A programmatic SAT solver for the RNA-folding problem. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 143–156. Springer.

[Gibson and Jedwab, 2011] Gibson, R. G. and Jedwab, J. (2011). Quaternary Golay sequence pairs I: Even length. *Designs, Codes and Cryptography*, 59(1-3):131–146.

[Golay, 1961] Golay, M. (1961). Complementary series. *IRE Transactions on Information Theory*, 7(2):82–87.

[Golay, 1949] Golay, M. J. (1949). Multi-slit spectrometry. *JOSA*, 39(6):437–444.

[Gonthier, 2008] Gonthier, G. (2008). Formal proof—The four-color theorem. *Notices of the AMS*, 55(11):1382–1393.

[Gopalakrishnan, 2006] Gopalakrishnan, G. (2006). *Computation engineering: Applied automata theory and logic*. Springer Science & Business Media.

[Granlund et al., 1991] Granlund, T. et al. (1991). The GNU multiple precision arithmetic library. gmplib.org.

[Gruber et al., 2007] Gruber, H., Holzer, M., and Kutrib, M. (2007). The size of Higman–Haines sets. *Theoretical Computer Science*, 387(2):167–176.

[Gruber et al., 2009] Gruber, H., Holzer, M., and Kutrib, M. (2009). More on the size of Higman–Haines sets: Effective constructions. *Fundamenta Informaticae*, 91(1):105–121.

[Hadamard, 1893] Hadamard, J. (1893). Résolution d'une question relative aux déterminants. *Bull. sci. math*, 17(1):240–246.

[Haines, 1969] Haines, L. H. (1969). On free monoids partially ordered by embedding. *Journal of Combinatorial Theory*, 6(1):94–98.

[Hales et al., 2015] Hales, T., Adams, M., Bauer, G., Dang, D. T., Harrison, J., Hoang, T. L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T. T., et al. (2015). A formal proof of the Kepler conjecture. *arXiv preprint arXiv:1501.02155*.

[Hales, 2005] Hales, T. C. (2005). A proof of the Kepler conjecture. *Annals of mathematics*, 162(3):1065–1185.

[Harrison, 1996] Harrison, J. (1996). HOL Light: A tutorial introduction. In *Formal Methods in Computer-Aided Design*, pages 265–269. Springer.

[Hedayat et al., 1978] Hedayat, A., Wallis, W., et al. (1978). Hadamard matrices and their applications. *The Annals of Statistics*, 6(6):1184–1238.

[Heule et al., 2016] Heule, M. J., Kullmann, O., and Marek, V. W. (2016). Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 228–245. Springer.

[Higham, 1993] Higham, N. J. (1993). The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799.

[Higman, 1952] Higman, G. (1952). Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(1):326–336.

[Holzmann and Kharaghani, 1994] Holzmann, W. H. and Kharaghani, H. (1994). A computer search for complex Golay sequences. *Australas. J. Combin.*, 10:251–258.

[Holzmann et al., 2008] Holzmann, W. H., Kharaghani, H., and Tayfeh-Rezaie, B. (2008). Williamson matrices up to order 59. *Designs, Codes and Cryptography*, 46(3):343–352.

[Kharaghani and Tayfeh-Rezaie, 2005] Kharaghani, H. and Tayfeh-Rezaie, B. (2005). A Hadamard matrix of order 428. *Journal of Combinatorial Designs*, 13(6):435–440.

[Knuth, 2011] Knuth, D. E. (2011). *The Art of Computer Programming. Vol. 4A, Combinatorial Algorithms, Part 1.* Addison-Wesley, Upple Saddle River (N.J.), London, Paris.

[Konev and Lisitsa, 2014] Konev, B. and Lisitsa, A. (2014). A SAT attack on the Erdős discrepancy conjecture. In Sinz, C. and Egly, U., editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *LNCS*, pages 219–226. Springer International Publishing, Cham.

[Kotsireas, 2013a] Kotsireas, I. S. (2013a). Algorithms and Metaheuristics for Combinatorial Matrices. In *Handbook of Combinatorial Optimization*, pages 283–309. Springer.

[Kotsireas, 2013b] Kotsireas, I. S. (2013b). Structured Hadamard conjecture. In Borwein, J., Shparlinski, I., and Zudilin, W., editors, *Number theory and related fields*, volume 43 of *Springer Proc. Math. Stat.*, pages 215–227. Springer, New York. In Memory of Alf van der Poorten.

[Kotsireas et al., 2009] Kotsireas, I. S., Koukouvinos, C., and Seberry, J. (2009). Weighing matrices and string sorting. *Annals of Combinatorics*, 13(3):305–313.

[Koukouvinos and Kounias, 1988] Koukouvinos, C. and Kounias, S. (1988). Hadamard matrices of the Williamson type of order $4 \cdot m$, $m = p \cdot q$ an exhaustive search for $m = 33$. *Discrete mathematics*, 68(1):45–57.

[Lagarias, 2011] Lagarias, J. C. (2011). The Kepler conjecture and its proof. *The Kepler Conjecture*, pages 3–26.

[Li, 2016] Li, C. (2016). Parallel backtracking for 4 way matching. github.com/ian8170/Parallel-backtracking-for-4-way-matching.

[Liang et al., 2016] Liang, J. H., Ganesh, V., Poupart, P., and Czarnecki, K. (2016). Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 3434–3440. AAAI Press.

[Lifchitz and Lifchitz, 2000] Lifchitz, H. and Lifchitz, R. (2000). PRP Records. www.primenumbers.net/prptop/prptop.php.

[Manders and Adleman, 1976] Manders, K. and Adleman, L. (1976). NP-complete decision problems for quadratic polynomials. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 23–29. ACM.

[Marques-Silva et al., 1999] Marques-Silva, J. P., Sakallah, K., et al. (1999). GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521.

[Martin, 2011] Martin, M. (2011). Primo for Linux. www.ellipsa.eu/public/primo/primo.html.

[Morain, 2015] Morain, F. (2015). Quelques nombres premiers prouvés par mes programmes (some primes proven by my programs). www.lix.polytechnique.fr/Labo/Francois.Morain/Primes/myprimes.html.

[Moskewicz et al., 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). CHAFF: Engineering an Efficient SAT Solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM.

[Muller, 1954] Muller, D. E. (1954). Application of Boolean Algebra to Switching Circuit Design and to Error Detection. *Electronic Computers, Transactions of the IRE Professional Group on*, EC-3(3):6–12.

[Nazarathy et al., 1989] Nazarathy, M., Newton, S. A., Giffard, R., Moberly, D., Sischka, F., Trutna, W., and Foster, S. (1989). Real-time long range complementary correlation optical time domain reflectometer. *Journal of Lightwave Technology*, 7(1):24–38.

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). ISABELLE/HOL: *A proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media.

[Nowicki et al., 2003] Nowicki, A., Secomski, W., Litniewski, J., Trots, I., and Lewin, P. (2003). On the application of signal compression using Golay's codes sequences in ultrasound diagnostic. *Archives of Acoustics*, 28(4).

[OEIS Foundation Inc., 1996] OEIS Foundation Inc. (1996). The On-Line Encyclopedia of Integer Sequences. oeis.org.

[Paley, 1933] Paley, R. E. (1933). On Orthogonal Matrices. *J. Math. Phys.*, pages 311–320.

[Penné, 2015] Penné, J. (2015). LLR version 3.8.15. jpenne.free.fr/index2.html.

[Reed, 1954] Reed, I. (1954). A Class of Multiple-Error-Correcting Codes and the Decoding Scheme. *Transactions of the IRE Professional Group on Information Theory*, 4(4):38–49.

[Reynolds, 2010] Reynolds, G. (2010). Sierpinski/Riesel sieve version 0.6.17. sites.google.com/site/geoffreywalterreynolds/programs/srsieve.

[Ricker, 2012] Ricker, D. (2012). *Echo Signal Processing*. The Springer International Series in Engineering and Computer Science. Springer US.

[Riel, 2006] Riel, J. (2006). `nsoks`: A MAPLE script for writing n as a sum of k squares.

[Robertson et al., 1997] Robertson, N., Sanders, D., Seymour, P., and Thomas, R. (1997). The four-colour theorem. *Journal of combinatorial theory, Series B*, 70(1):2–44.

[Scott, 2003] Scott, M. (2003). Multiprecision integer and rational arithmetic cryptographic library. www.certivox.com/miracl.

[Seberry, 1999] Seberry, J. (1999). Library of Williamson Matrices. http://www.uow.edu.au/~jennie/WILLIAMSON/williamson.html.

[Shallit, 2000] Shallit, J. (2000). Minimal primes. *Journal of Recreational Mathematics*, 30(2):113–117.

[Shallit, 2006] Shallit, J. (2006). The Prime Game. http://recursed.blogspot.ca/2006/12/prime-game.html.

[Skolem, 1938] Skolem, T. (1938). *Diophantische gleichungen*. Number v. 5, no. 4 in Ergebnisse der Mathematik und ihrer Grenzgebiete. Chelsea.

[Sloane, 2004] Sloane, N. (2004). Library of Hadamard Matrices. http://neilsloane.com/hadamard/.

[Sylvester, 1867] Sylvester, J. J. (1867). Thoughts on inverse orthogonal matrices, simultaneous signsuccessions, and tessellated pavements in two or more colours, with applications to Newton's rule, ornamental tile-work, and the theory of numbers. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 34(232):461–475.

[Turyn, 1972] Turyn, R. J. (1972). An infinite class of Williamson matrices. *Journal of Combinatorial Theory, Series A*, 12(3):319–321.

[Tymoczko, 1979] Tymoczko, T. (1979). The four-color problem and its philosophical significance. *The Journal of Philosophy*, 76(2):57–83.

[Wallis, 1974] Wallis, J. S. (1974). Williamson matrices of even order. In Holton, D. A., editor, *Combinatorial Mathematics: Proceedings of the Second Australian Conference*, pages 132–142. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Walsh, 1923] Walsh, J. L. (1923). A Closed Set of Normal Orthogonal Functions. *American Journal of Mathematics*, pages 5–24.

[Williams and Dubner, 1986] Williams, H. and Dubner, H. (1986). The primality of R_{1031}. *Mathematics of computation*, pages 703–711.

[Williamson, 1944] Williamson, J. (1944). Hadamard's Determinant Theorem and the Sum of Four Squares. *Duke Math. J*, 11(1):65–81.

[Wolfram, 1999] Wolfram, S. (1999). *The* MATHEMATICA *Book, version 4*. Cambridge University Press.

[Zeilberger, 2015] Zeilberger, D. (2015). The Babylonian vs. the Greek Approaches to Computer Proofs. Presented at the Fields Institute's Workshop on Algebra, Geometry and Proofs in Symbolic Computation. http://www.math.rutgers.edu/~zeilberg/mamarim/mamarimhtml/babylon.html.

[Zulkoski et al., 2017] Zulkoski, E., Bright, C., Heinle, A., Kotsireas, I. S., Czarnecki, K., and Ganesh, V. (2017). Combining SAT solvers with computer algebra systems to verify combinatorial conjectures. *J. Autom. Reasoning*, 58(3):313–339.

[Zulkoski et al., 2015] Zulkoski, E., Ganesh, V., and Czarnecki, K. (2015). MATHCHECK: A Math Assistant via a Combination of Computer Algebra Systems and SAT Solvers. In Felty, A. P. and Middeldorp, A., editors, *Automated Deduction - CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 607–622. Springer International Publishing.