Constructing Hard Examples for Graph Isomorphism

Kashif R. Khan

Supervisor: Prof. Anuj Dawar

Computer Laboratory
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy

Sidney Sussex College

June 2017

Declaration

I Kashif R. Khan of Sidney Sussex College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count:

Signed:

Date:

Acknowledgements

I would like to thank my project supervisor Professor Anuj Dawar for ceaseless support, advice and mentoring throughout this project. I am grateful for his continuous patience and insight.

I would also like to thank my family, friends and the Computer Laboratory faculty for their kindness and guidance.

Abstract

The graph isomorphism problem (GI) has been proved to be solvable in quasi-polynomial time. This theoretical breakthrough does not necessarily describe the complexity of fast GI solvers. Following this discovery, a call was made to produce instances of graphs which execute slowly on these fast GI solving programs, namely Traces. The complexity of such solvers is benchmarked using numerous families of graphs and the construction of such difficult graphs is non-trival. In searching for a difficult family of graphs, there may be interesting results which bring light to the exploration of GI.

We describe and implement a new variety of graphs using concepts in random 3-XOR-formulas on the threshold of satisfiability and multipedes. The rigid construction resists vertex-colour-refinement and automorphism factoring utilised in the backtracking search of Traces resulting in slow execution times.

A program for finding and executing such constructions is provided. In addition to an evaluation consisting of reproducing benchmark tests and comparing execution times. Furthermore, an analysis is provided on searching for these structures. Results showed that for many instances of our constructions, execution times were slow for graphs of equal node size in benchmark tests. Our construction consistently proved slow on Traces on a range of instances. Developing on previous work, graphs were small enough to be executed on Traces. Multiple packages were generated, which ranged in time complexity. Notably, those closest to the threshold of satisfiability showed the greatest execution time.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Determining if two graphs $G = (V,E)$ and $G' = (V',E')$ are identical is equivalent to questioning the existence of a bijective function between vertex sets V and V' which preserves edges. If there is such a function $f : V \to V'$, then we say that the graphs are isomorphic to one another $G \simeq G'$ and that there is an isomorphism f. This is the graph isomorphism problem (GI), and is in essence the question of graph equality and the focus of our report. GI has long existed from the dawn of computer science and discrete mathematics. It is a fundamental property of graphs in graph theory; its computation, however, is not straightforward. Nonetheless, the problem has spawned efficient GI solvers such as Nauty [1], Traces [2], Bliss [3], Conauto [4] among others to solve the problem.

Babai et al. proved that worst-case complexity of the problem was $exp(O\sqrt{n \log n})$ in 1983 by means of a theoretical algorithm [5]. After recent improvements in 2015 by Babai, a professor of mathematics in Chicago University, proved that it is now known to be $exp((log\ n)^{O(1)}))$ through a related problem known as String Isomorphism [6]. From his most recent work, the proof implied that GI is computable in quasi-polynomial time, that is, slower than polynomial-time, yet not as slow as exponential-time. In other words, determining if an isomorphism exists between two graphs is exponential in some logarithmic scale. Since we can easily determine if a bijection is an isomorphism, we know that the problem is in the complexity class NP. However, we cannot classify it is either P or NP-Complete. If $P \neq NP$, then we say that GI is a problem in NP-intermediate; in NP but neither P nor NP-Complete.

Babai concluded his paper by noting that he does provide a worst-case complexity abstractly, but he does not contribute to the complexity of GI solving implementations. These implementations are heuristic algorithms which perform GI solving using search heuristic methods, and not the algorithm which Babai provides. The complexity of such solvers remains unknown, due to their intricate techniques. He asks, "Does there exist an infinite family of pairs of graphs on which these heuristic algorithms fail to perform efficiently? The search for such pairs might turn up interesting families of graphs" [6]. Our work addresses this question, and is of importance in understanding boundaries of computational complexity.

We intend to construct a new type of graph that executes slowly on GI solvers. In doing so, provide a new benchmark in worst-case complexity for implementations that Babai mentions. Since the worst-case complexity cannot be easily determined, GI solvers typically use a variety of graphs, differing in size and shape, to determine their performance. In the work of Piperno et al, Traces was tested using graphs from different families, such as CFI [7], Miyazaki [8], random graphs with differing edge probabilities and more. Our work is to present another difficult case of benchmark graph which will run slower than the ones used in benchmark tests. We use Traces since it outperforms its competitors for most cases of difficult graph classes [2].

The report is structured as follows:

- Chapter 2 provides the background knowledge of the constructions proposed, including basic graph theory and related works providing references and a point of further reading.

- Chapter 3 contains our requirements. This is important in reproducing the work and will help to understand the fundamental choices in implementation. The project will utilise a number of tools, which will result in numerous considerations.

- Chapter 4 encapsulates how the design was realised to aid understanding of the codebase. It expands on the choice of parameters, and what this resulted in executing code. This is important for further implementations to take into consideration.

- Chapter 5 evaluates and analyses results, with emphasis on what parameters entailed.

- Chapter 6 summarises the project, providing a point of departure for further work and refinement.

Chapter 2

Background

In this chapter, preliminary knowledge for understanding our graph is presented. Beginning with Graph Theory and Complexity Theory, this will provide basic a understanding of its form, and the problem we are trying to solve. Additionally, we will highlight key concepts from related works, which will be used in constructing our graph. Furthermore, we will briefly describe the history of related tools that we require in development. We primarily use use Diestel's definitions in Graph Theory and Papadimitriou's in Complexity Theory [9][10].

2.1 Graph Theory

2.1.1 Graphs

A *graph* is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$. Hence, the elements of E are 2-element subsets of V. We shall assume $V \cap E = \emptyset$. The elements of V are the *vertices* (or *nodes*, or *points*) of the graph G, the elements of E are its *edges* (or *lines*).

2.1.2 Morphisms

Given graphs $G = (V, E)$ and $G' = (V', E')$. If there exists a bijection between vertex sets $f : V \to V'$ with $(x, y) \in E \iff (f(x), f(y)) \in E'$ for all $x, y \in V$, then we call G and G' *isomorphic*, denoted $G \simeq G'$. Such a map f is called an *isomorphism*; if $G = G'$, it is called an *automorphism*. The set of all automorphisms forms a group which we call an *automorphism group* $\mathrm{Aut}(G)$. If $\mathrm{Aut}(G)$ is trivial, that is, it contains only the identity mapping, then we say that G is *rigid*.

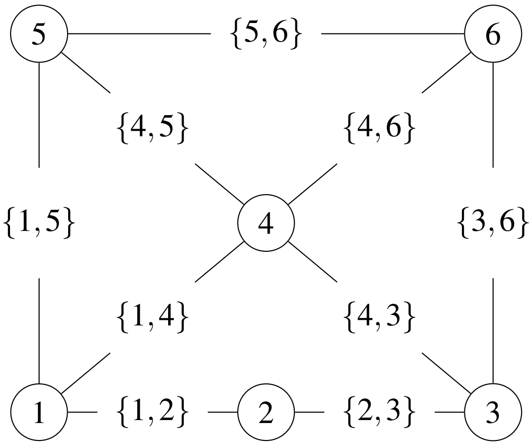

Fig. 2.1 An example graph
The graph on $V = \{1,2,3,4,5,6\}$ with edge set
$E = \{\{1,2\},\{2,3\},\{1,4\},\{4,3\},\{1,5\},\{4,5\},\{4,6\},\{3,6\},\{5,6\}\}$

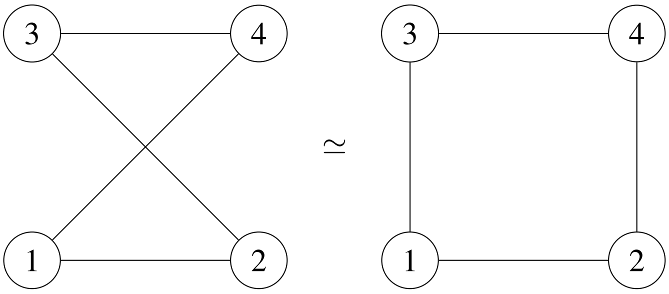

Fig. 2.2 Isomorphic graphs

2.2 Complexity Theory

A *complexity class* is a set of computational problems that can be decided within some bound specified by their performance, that is, whether input to a Turing Machine completes or runs forever. *P* is the complexity class that contains all decision problems that can be solved by a deterministic Turing Machine in polynomial time. The complexity class *NP* is the set of all problems which can be validated to be true in polynomial time, or solvable in polynomial time by a non-deterministic Turing Machine. Problems that are *NP-Hard* are at least as hard as the hardest problems in NP. *NP-Complete* problems are those which contain the hardest problems in NP.

A *quasi-polynomial* time algorithm is one that runs slower than polynomial time, yet not slow as to be exponential time. The worst case running time of a quasi-polynomial time algorithm is $2^{O((\log n)^c)}$ for some fixed $c > 0$.

The graph isomorphism problem (GI) is the decision problem determining whether two graphs are isomorphic. GI has been proven to have a quasi-polynomial time algorithm solution by Babai [6]. The boolean satisfiability problem (SAT) is the decision problem in determining whether there exists a satisfying argument to a boolean formula, which is NP-Complete. The XOR-SAT problem is similar to SAT, but we are limited to using $\oplus$, the exclusive or, when constructing clauses in logical statements. XOR-SAT when viewed as system of linear equations (mod 2) can be solved in polynomial-time by using Gaussian elimination [11].

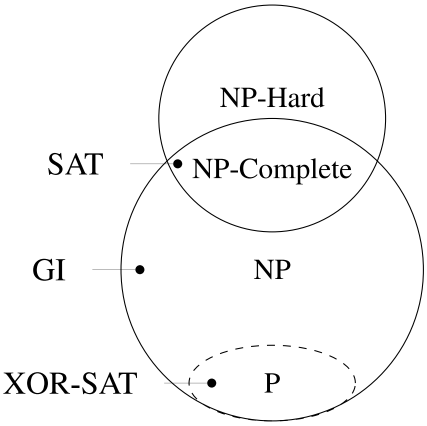

Fig. 2.3 Fundamental complexity classes and computational problems

2.3 XOR-Formulas on the Threshold of Satisfiability

XOR formulas are logical statements that consist of the exclusive $\oplus$ and $\wedge$ connectives; for example $(a \oplus b \oplus c) \wedge (d)$ is an XOR-Formula consisting of two *clauses*, namely $a \oplus b \oplus c$ and d. Each variable has the value of either False or True. If the statement can be satisfied with True or False values, then we say it is *satisfiable*, otherwise way say it is *unsatisfiable*.

XOR-Formulas are also represented in algebraic terms. Using the previous example, $1 = a + b + c(mod2)$ and $1 = d(mod2)$ is its equivalent statement, also known its *algebraic normal form*. Here, values are assigned either 0 or 1, representing False and True respectively. Therefore, we can interchange descriptions when we mention XOR-Formula: when we say logical statement, we also mean algebraic equation mod 2; when we say formula, we also mean set of equations.

Let ϕ denote an XOR-Formula. If ϕ has at most k variables in every clause, then we say it is a k-XOR-Formula; for example, $(a \oplus b \oplus c) \wedge (d)$ is a 3-XOR-Formula. We let n denote the number of variables in a formula ϕ and m the total number of clauses. A k-XORSAT problem is determining the truth of an k-XOR-Formula. Dubois and Mandler proved that if ϕ has an equal number of variable to clauses $n = m$ and $k \geq 3$, then was say it is on the *threshold of satisfiability* [12]. That is to say, if $n > m$, then ϕ will almost surely be satisfiable, and if $n < m$ then ϕ is likely to be $unsatisfiable$. Therefore, $n = m$ is a sharp bound for satisfiability for any given instance of k-XORSAT, where $k \geq 3$. This is confirmed by Pittel and Sorkin [13].

2.3.1 Multipedes

Multipedes are a form of hypergraph, where there exists hyperedges. Such hyperedges have edges which connect to any number of nodes, that is, they can connect 3 or more elements $h = (x, y, z)$. Normally, we would take (x, y) to represent an edge between nodes, but now we allow z to observe that there exists an edge between x and y. We will not describe the mathematical definition of multipedes, as this is covered extensively, and can not be summarised quickly [14]. We can say that they are proven to be rigid due to their *odd* property, but not C^k rigid (See Appendix B).

2.4 Construction

Here we will refer Professor Dawar's notes provided in the Appendix B to describe our construction and non-standard terminology. A 3-XOR-Formula ϕ is *homogenous* if the all equations are equal to 0. A homogeneous system is always satisfied by the all-zero solution, whereby all variables equal 0. If the all-zero is the *only* solution to a 3-XOR-Formula, then we say it is *uniquely satisfiable* (See Fig. 2.4). Our definition of *k-local consistency* is more lengthy, and requires us to describe it in the form of a hypothetical game.

Our construction will be created at random. Recall that n and m represent the number of variables and clauses respectively in a 3-XOR-Formula. From these values, we generate ϕ to be uniquely satisfiable and k-locally consistent. Once we know that it has both these properties, we generate our preliminary graph G_A, which we will check for automorphisms. If G_A has no non-trivial automorphisms, then we construct our final graph G_B. These graphs as described as follows.

G_A has nodes for each variable used and clause. Therefore, if we have n variables and m clause, G_A has $n+m$ nodes. The edges are connections between nodes, such that if a clause contains a literal, then there exists an edge. Every clause as a node, has exactly 3 edges, since every clause has 3 variables. (See Fig. 2.5)

G_B is our final construction, if G_A has no non-trivial automorphisms. For every literal x, we add two nodes x_0 and x_1. For every clause C, we insert four clauses C_1, C_2, C_3, C_4. Since C has 3 variables, there is a total of 4 ways in which we build a homogeneous system. For example, take $0 = a+b+c$, the all-zero solution is possible $a=0, b=0, c=0$; furthermore, since our equation is *mod 2*, $a=0, b=1, c=1$ and $a=1, b=1, c=0$ and $a=1, b=0, c=1$ are also solutions. We will allocate each solution to a clause such that, if x is in the solution C and equals 0, then there is an edge between x_0 and C, and if x is in the solution C and equals 1, then there is an edge between x_1 and C. (See Fig. 2.6). G_B has a total of $2n+4m$ nodes.

$$
\begin{array}{rcrcr}
b+c+e=0 & & (\neg b \oplus \neg c \oplus \neg e)\wedge & & (b \oplus c \oplus e)\wedge \\
b+c+f=0 & & (\neg b \oplus \neg c \oplus \neg f)\wedge & & (b \oplus c \oplus f)\wedge \\
c+d+e=0 & & (\neg c \oplus \neg d \oplus \neg e)\wedge & & (c \oplus d \oplus e)\wedge \\
a+b+d=0 & \Longleftrightarrow & (\neg a \oplus \neg b \oplus \neg d)\wedge & \Longleftrightarrow & (a \oplus b \oplus d)\wedge \\
b+e+f=0 & & (\neg b \oplus \neg e \oplus \neg f)\wedge & & (b \oplus e \oplus f)\wedge \\
b+c+d=0 & & (\neg b \oplus \neg c \oplus \neg d) & & (b \oplus c \oplus d)
\end{array}
$$

Fig. 2.4 Example 3-XOR-Formula ϕ
that is homogeneous and uniquely satisfiable.

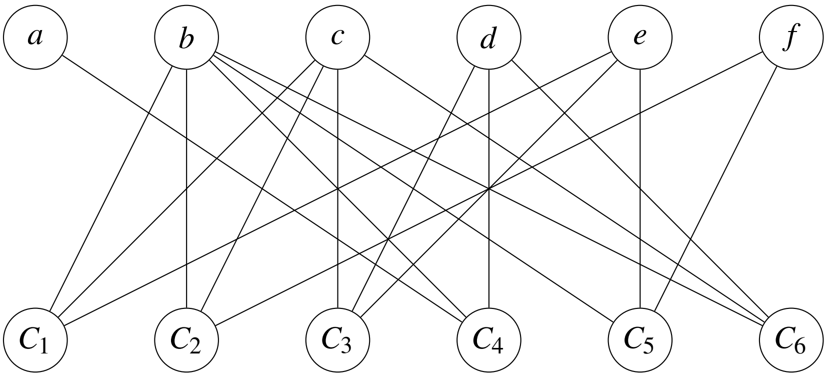

Fig. 2.5 Example preliminary graph G_A on ϕ

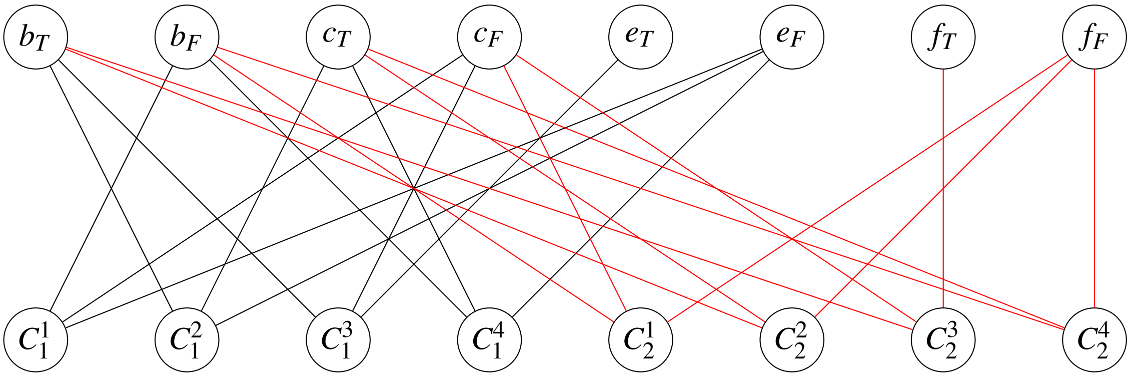

Fig. 2.6 Snippet of example final graph G_B on ϕ.
Only C_1 and C_2 is shown

Professor Dawar's construction creates rigid graphs based off the *odd* property of multipedes. The oddness property is replaced with check for unique satisfiability of an 3-XOR-Formula. Similarly the check for k-local consistency is replaced by identifying whether 3-XOR-Forumlas are solved quicker using Gaussian elimination operating within the algorithm, in comparison to the same formula without Gaussian elimination. In short, the check for oddness is replaced by unique satisfiability, thereby rendering them rigid, and the check for k-local consistency is estimated by comparing Gaussian elimination on and off times.

2.5 Solvers

In order to construct our graphs, we will require multiple checks at different stages of construction. When we generate a random 3-XOR-Formula ϕ, we need to ensure that it is k-locally consistent, uniquely satisfiable and gives rise to a preliminary graph with no automorphisms. Checking for k-local consistency and unique satisfiability requires a SAT solving program to determine these attributes. Whereas a GI solver, namely Traces, is utilised to check for automorphisms.

2.5.1 GI Solvers

The programs Nauty and Traces are two of the fastest GI solvers for most cases of small and large graphs respectively. Nauty was developed in the 1980s by B. Mckay [1], which was subsequently revised and incorporated into Traces [2] in 2013. These programs utilise two key components in their algorithms: vertex colour refinement and factoring of graphs through found automorphisms.

Two main procedures provided by the Nauty and Traces are Automorphism Group ($Aut(G)$) and Canonical Labelling ($Canon(G)$) of graphs. Our work is concerned only with the $Aut(G)$ function, as graphs which execute slowly on $Aut(G)$ tend execute slowly on Canon(G) [2]. These procedures are used to benchmark the performance of their algorithm, since the exact complexity of these programs is unknown. Additionally, since $Aut(G)$ shows us whether there are any automorphisms of a graph G, we use this feature in validating our preliminary graph G_A, as well as determining the performance of our final graph G_B.

The heuristic algorithm behind Nauty and Traces to determine graph isomorphism is a backtracking search which is based off two key elements: vertex colour refinement (also known as a 1-dimensional Weisfeiler-Leman test) and factoring graphs through found automorphisms [2]. Our construction, being both rigid and resistant to Weisfeiler-Leman tests and its high dimensional variants, should be difficult for this algorithm.

2.5.2 SAT Solvers

The algorithms behind SAT solvers typically use Gaussian Elimination in solving a system of linear equations. In some cases, this feature can be disabled, and some implementations do not use it all. By toggling Gaussian Elimination on and off, and recording the times, one can make a good guess at whether ϕ is k-locally consistent. Randomly generated systems which are solved significantly faster with Gaussian Elimination on versus off are ones of interest. The prediction is that systems with greater differences in execution speeds hints at k-local consistency of a system.

2.6 Previous Attempt

Recently, a study searched for the family of graphs Babai describes. E. Arrighi, a Cambridge intern guided by Prof. A. Dawar, the project supervisor, found that multipedes are theoretically complex for Traces. Arrighi concluded that the smallest examples of suitable probabilistically generated multipedes, containing significantly high probability of non-trivial automorphisms, would be too large to generate, so rendering them too large for experimentation on Traces. Instead, our approach utilised the same concepts from such hypergraphs, but constructs them in a smaller fashion.

Chapter 3

Requirements

The design of our program hinges on finding suitable constructions that match our criteria. Theoretically, two primary requirements must be met. Firstly, to find a set of equations (clauses), each of which contains three literals (variables), from a pool containing n elements. m is the number of equations within the formula. Hence, we must randomly generate a formula ϕ, which has n literals and m equations, for specified values. ϕ must be both k-locally consistent and homogeneous. Secondly, to ensure that the set of equations ϕ must give rise to a construction, as defined in the Background segment, which has no non-trivial automorphisms. Thus, we will have constructed a graph that encapsulated CFI graphs and multipedes, proving resistant to the backtracking search of Traces.

In this portion, we will consider what further requirements must be met, including defining our system conceptually, permitting us to take a reasoned step toward implementation. Furthermore, this allows the reader to reproduce our system without requiring much technical detail, permitting similar systems of different programming languages and so on.

3.1 Solvers

Prior to defining system requirements, we must first consider the specification of tools we intend to invoke. Two programs are of importance. One is the GI solver, which we have already identified as Traces, and the other is a SAT solver, of which there are many implementations. To this end, we are able to use SAT solving competitions as a starting point.

3.1.1 Cryptominisat

The international SAT solving competition is an annual meet where competitors pit their programs in a race to solve the SAT and UNSAT problem [15]. Here lies a choice of open-source software to implement, all of which follow the same DIMACS specification of usage [16]. Many are regularly maintained and well documented.

Following the pre-implementation decision of using Python as a programming language of choice, we select Cryptominisat as our solver. Although this package is primarily written in C, it provides compact Python libraries to import and is well maintained on GitHub. Additionally, for reasons other than being awarded fastest solver in its track, we will describe how Cryptominisat can also be of great use in checking for *k*-local consistency and unique satisfiability.

Input

In order to comprehend how homogeneous systems are determined efficiently, we will first consider the DIMACS format previously mentioned. (For an elaborate explanation, see the DIMACS website. For our purposes, we will consider only special cases [16].)

Input to the SAT solvers takes the form of text files. An example is shown below.

```
p cnf 4 5
1 3 4 0
2 -3 4 0
1 2 -4 0
1 -2 -3 0
-1 2 3 0
```

Fig. 3.1 Example 3-XOR-Formula in DIMACS

The first line defines the number of literals and the number of clauses in the cnf formula, 4 and 5 respectively. This is the equivalent of n and m as used in ϕ. The following lines are cnf clauses. Each clause contains three literals, and is demarcated with the 0 value. Thus, line 2 denotes $(1 \vee 3 \vee 4)$, followed by line 3 stating $\wedge(2 \vee \neg 3 \vee 4)$ and so on. (Equivalently, $(a \vee c \vee d) \wedge (b \vee \neg c \vee d)$). We can translate this syntax into more common mathematical system of equations and CNF formula:

Evidently, there is a satisfying value for these example: $a = 0, b = 1, c = 0, d = 1$ or $a = False, b = True, c = False, d = True$.

Similarly, we have the following example input for an XOR-Formula, although CNF and XOR can be readily inferred from one another without the need to change syntax:

$$\begin{aligned} a+c+d&=1\\ b-c+d&=1\\ a+b-d&=1\\ a-b-c&=1\\ -a+b+c&=1 \end{aligned} \qquad \iff \qquad \begin{aligned} (a\lor c\lor d)\land\\ (b\lor\neg c\lor d)\land\\ (a\lor b\lor\neg d)\land\\ (a\lor\neg b\lor\neg c)\land\\ (\neg a\lor b\lor c) \end{aligned}$$

```
p cnf 6 6
x2 3 5 0
x2 3 6 0
x3 4 5 0
x1 2 4 0
x2 5 6 0
x2 3 4 0
```

$$\iff \qquad \begin{aligned} b+c+e&=0\\ b+c+f&=0\\ c+d+e&=0\\ a+b+d&=0\\ b+e+f&=0\\ b+c+d&=0 \end{aligned} \qquad \iff \qquad \begin{aligned} False(b\oplus c\oplus e)\land\\ False(b\oplus c\oplus f)\land\\ False(c\oplus d\oplus e)\land\\ False(a\oplus b\oplus d)\land\\ False(b\oplus e\oplus f)\land\\ False(b\oplus c\oplus d)\land \end{aligned}$$

The notable difference in the syntax of the input is that we now prepend x at the beginning of each line to demarcate an XOR clause. Note that this is also a homogeneous system that is uniquely satisfiable and gives rise to our desired construction G_B.

This distinction is important as we have more than one method of validating unique satisfiability, which will ultimately effect the running time and space complexity of our algorithm. Specifically in searching for uniquely satisfiable set of clauses, that are on the threshold of satisfiability. As systems become larger, or closer to the threshold, the running time is expected to increase exponentially.

Unique satisfiability

Given Cryptominisat allows input in the form of mixed CNF and XOR clauses, we have to solutions to check for uniquely satisfiable and homogeneous solutions. We use uniquely satisfiable systems of equations as a replacement for the *odd* property of multipedes, rendering them rigid.

Method A: Insert an additional XOR clause which contains a single literal. Then check for satisfiability. If for any of these literals there is a satisfying argument, then there is another solution to the system of equations which is not the all-zero solution. This is the only option if we are enforced to use strictly XOR clauses. However, Cryptominisat provides us the functionality of mixing XOR and CNF clauses.

Method B: Insert an additional OR clause which contains all literals. Then check for satisfiability. If this statement is TRUE, then at least one literal must be TRUE for the statement to hold. Thus, there is another solution other than the all-zero solution.

Both solutions have their caveats. The first solution would require n repetitions to check for satisfiability. As these systems become large, validating a single system of equations may take a long period of time, but adds only small amount of complexity to the problem. Whereas the second solution would require that the XOR solver permit a combination of XOR systems and OR systems of different sizes, with one clause containing n variables. This can be problematic as n becomes large complexity wise, but requires only one check. Depending on how the SAT solver manages, both options are available to test.

k-local consistency

The validation of k-local consistency of a system of equations is much more difficult than a simple check of a logical statement.

One way would be to locate uniquely satisfiable homogeneous systems which execute faster with Gaussian elimination working in the SAT solving algorithm, in contrast to the same system executing on a solver without this feature. Thus, we could compare a systems execution time on a SAT solver with Gaussian elimination to one without. Two separate algorithms would be required to make this comparison. Furthermore, Gaussian elimination is not central to these efficient algorithms, as we will see later. However, Cryptominisat does provide an option to build without this feature. Hence, rather than utilise two separate programs, we are able to build the same program twice with different algorithms. Thus eliminating the need to facilitate two separate packages, which will save implementation time.

In checking k-local consistency using this feature, we would invoke Cryptominisat twice, once with "Gauss-On" and then with "Gauss-Off", on a uniquely satisfiable 3-XOR-Formula. We would take systems which execute faster with Gauss-On versus Gauss-Off. Those systems which have a larger discrepancy in times are deemed as more likely to be k-locally consistent, and would result in slower execution times as a construction. These are the systems which we are looking for.

3.1.2 Traces

Traces is our GI solver of choice. It has its own specifications of input, namely the DIMACS and *Dreadnaut* standards. The graph constructions available on the Traces website are of these varieties, which are provided to reproduce benchmark tests and experiment with the suite [17]. We will use a number of these packages available to compare the performance of our construction. We intend to reproduce the benchmarks, and go further by involving more packages and extending the ones available, that is, creating larger instances of random graphs. Since we intend to execute numerous packages, we must implement an efficient way of executing them. Our goal is to run a family of graphs sequentially, and record execution times. Our construction will be packaged similarly to these graphs.

Dreadnaut

Executing graphs on Traces comes in a variety of implementations, however, we will be concerned with the well-documented and simple interface that is Dreadnaut. Dreadnaut is a tool that permits us to invoke both Nauty and Traces, with a variety of parameters. It is a command line tool provided in the Nauty/Traces suite which opens a shell for subroutines. Thus, we are able to invoke this tool by means of system class from our Python implementation. The major design issue is that we must use the *.dre* (Dreadnaut) filetype in passing stored graphs as input, and not the generic DIMACS format. Therefore, in outputting our construction, we must adhere to this syntax.

Benchmarks

In reproducing benchmark tests, there is the issue of fairly testing graphs against one another. Hence, a suitable environment for executing constructions is required. For this issue, we consider using a machine that is uninterrupted by system processes. Traces does provide a means of accurately measuring execution times, however, this requires multiple executions of the same graph instance, which becomes costly as we increase the size of graphs. In addition, to attain sub-second timings, many repetitions must be executed to attain these values. This is problematic even for small graphs, where sub-second differences matter. Therefore, a makeshift method of recording times is presented, using the times of system calls. (Both Traces and system call times are captured in testing, which showed minor differences in times.) All packages are therefore timed in this manner on an external machine whose sole purpose is to conduct experiments. Moreover, many requirements produced in the benchmarks tended to run for very long periods of time, and had to be bounded by some timeout value. An appropriate value is chosen at the time of implementation.

3.2 Software Requirements

Now that we have defined the requirements of the major tools utilized, we will describe the requirements of our implementation. These are more general issues, rather than design choices enforced by external software. Our product must meet these specified areas, in addition to the issues raised by tools.

3.2.1 Data Management

We will be handling files in storing and executing our data, as well as the graphs provided by Traces. Hence, we will need to program our system to manipulate files, such as stored XOR-Formulas and constructions., which will be used as input to implemented tools. A database could be utilised, although, in our case, we will use the (Linux) filesystem as storage. This will allow us to transfer files as required, rather than having to persist in some datastore. It is important to ensure these files are stored, since locating difficult constructions will take time. The stored files will be provided along with our own package, which will be available as an open-source repository which also houses installation information [18].

3.2.2 Reproducibility

Ensuring that our results and findings are easily verifiable and reproducible, we provide a GitHub repository where all our work is publicly available. We intend to properly document our codebase, including instructions on how to install our package.

As a requirement, we should be able to reproduce finding and constructing systems, execute graphs and benchmarks using Traces, and any other important discoveries we make. The constructions we find will be tend to the .dre specification, and will appear similar to one of the benchmark graph packages provided by Traces.

3.2.3 Resources

Our construction must be small enough to execute on Traces, yet be easy to find given the right parameters. Ideally, we would like our system to be located much faster than the execution time on Traces, so that we can build a database of constructions quickly. We therefore have two resource requirements: space and time, on our tools and system hardware.

Traces is prone to exceed its allocated memory and crash or invoke thrashing of loading from secondary memory. Hence, we must keep this in mind when setting up our test server. Instances at given size will inevitably be too large for a system to handle in memory.

The outcome of executing systems on our SAT solver was unknown, however, we intended to implement a timeout similar to the one utilised in Traces benchmarks. This can be problematic as we will require multiple runs to locate k-locally consistent systems, so some upper bound must be put in place for an efficient search. Enough memory and a time bound must be provided for both of our tools, which will be manipulating our constructions. Both in searching for constructions and executing constructions, we expect to reach execution times within the scale of hours. Hence, one can expect some very difficult instances or a group of instances to take many days to execute. Moreover, it is unknown how it will take to find suitable systems at large values of m and n. We expect a sparse amount along the threshold of satisfiability. Hence, searching, validating and executing graphs and systems will take a long period of time. This is why a process must remain uninterrupted for many days on our test server.

Chapter 4

Design and Implementation

Moving onward from our requirements, we take the first step in producing our final product. Prior to development however is the setup of our working environment, which is of importance if we are to describe the issues and choices in development, preventing or influencing capabilities. We leave much of the directory structure and information on where results are stored and how they can be reproduced in our GitHub repository [18].

4.1 Development Environment

Our development setup is shown in Table 4.1. Evidently, we have enough hardware capabilities to meet the baseline requirements of our tools. However, this still led to issues described in later segments, including out of memory errors and erroneous execution times. Although, we did not experience all errors on our test server.

Table 4.1 Development Setup

Feature	Description
Operating System	Ubuntu 16.10 64-bit
Programming Language	Python 2.7
Programming Language Tools	numpy, timeit, matplotlib
Command Line Tools	dreadnaut, nauty/Traces
Versioning	Github (Public Repo)
RAM	8GB DDR3
Disk	500GB ATA SanDisk Ultra II SSD
CPU	8 x Intel Core i7-3612QM CPU @ 2.10GHz

4.2 Architecture

Our system setup separates development and experiment stages of our program. We first created our package on a laptop computer, which would regularly update experiments, data and code through SSH, FTP and GitHub respectively. Once experiments were completed, we would save these results to our repository and then generate plots. This was to esnure that the python process in action would remain undisturbed by any other system process. Both systems used a form of Linux, and the experiment server was hoster on Digital Ocean, which allowed us to change system resources as required. This is important to note as we frequently had to do this.

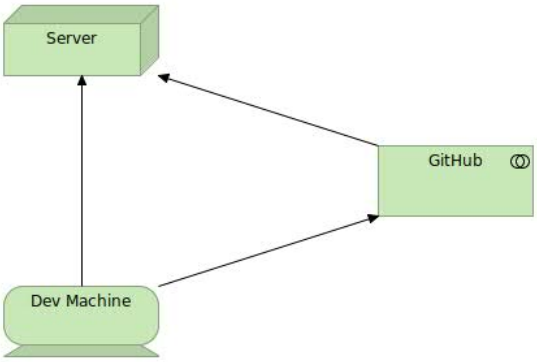

Fig. 4.1 System Architecture

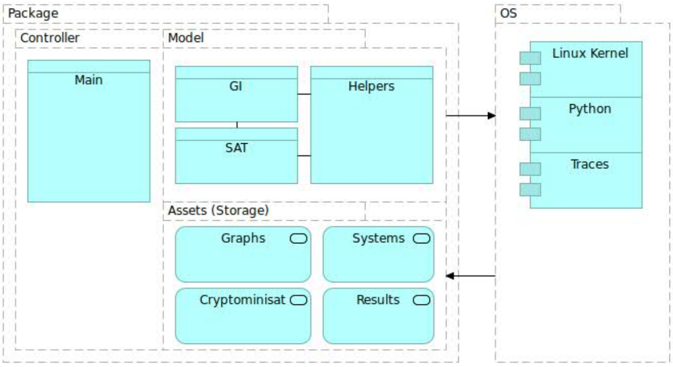

Fig. 4.2 Software Architecture

4.3 Algorithms

We two major steps in our program: locating our desired systems, and then converting them to our construction. With the added stages of separating graphs into packages which varied in complexity, and then executing these graphs to find execution times, we have a total of four key steps. The following diagram depicts our workflow:

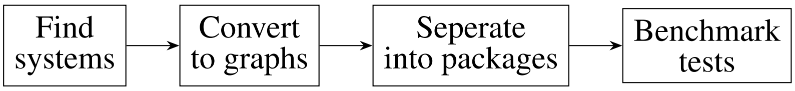

Fig. 4.3 Typical workflow

Each stage has its own form of elaborate checks and logic. Here we will describe each part does, how it is integrated into our architecture and notable pieces of logic. Note that this is the most basic usage of our software, and additional features and options are mentioned later. We will only describe the first two parts here, and the second parts in the Evaluation, as the following steps were a result of findings.

4.3.1 Timing

Times were monitored using pythons Timeit library. If a timeout was specified, and exception would be thrown and handled appropriately. Runs which required the uses of system calls, such as dreadnaut and Traces, would be called using the Process Handler. One would record the time before and after the call was made. Granted this does not give the very accurate times, but was satisfactory enough to gauge the general idea. We assumed that making the system call would only add sub second times to overall execution, which results in no large changes in the execution of large graphs, which executed for multiple minutes.

4.3.2 Systems Search

Given values max_n and max_m, the maximum number of clauses and literals respectively, we search for systems in the following manner (see algorithm for details): let $n = m = 4$; iterate over the set $N = \{n, n + step, .., max\_n\}$; for each $n \in N$, iterate over the set $M = \{m, m + step, .., max\_m\}$; for these values of n and m, generate a random system of equations from a pool of n literals, that is $\{1, 2, .., n\}$ with m clauses; validate this system for unique satisfiability and k-local consistency; if it is valid, then store this system. See Algorithm 1.

There are, of course, many checks not mentioned, however, we take this as the simplest form of generating a range of systems, over the Cartesian product of N and M, where $n \leq m$. Thereby ranging from the line of satisfiability, $n = m$, to some $n = max_n$ and $m = max_m$

ALGORITHM 1: Fundamental Search: Finding uniquely satisfiable and k-locally consistent systems

```
Input: Integers $min\_n$, $min\_m$, $max\_n$, $max\_m$, $step$, $max\_tries$.
Output: A set of uniquely satisfiable and k-locally consistent XOR-Formulas.
$systems = \{\}$;
repeat
    $tries = 0$;
    repeat
        if $tries == max\_tries$ then
            $tries = 0$;
            $min\_m += step$;
            skip this iteration;
        end
        $system$ = Generate random system($min\_n, min\_m$);
        if system is uniquely satisfiable and k-locally consistent(system) then
            $systems$ add $system$;
            $tries = 0$ ;
            $min\_m += step$;
        else
            $tries += 1$;
        end
    until $min\_m == max\_m$;
    $min\_n += step$;
until $min\_n == max\_n$;
return $systems$
```

4.3.3 Constructing Graphs

In transforming eligible systems into our final construction, we must first ensure that the graphical representation of our XOR-Formula contains no automorphisms. This is determined by Traces, and is fairly easy to compute. Recall in the background segment we label the preliminary and final graphs G_A and G_B respectively. See Algorithm 2.

G_A is trivial to construct into its graphical representation, whereas G_B requires only slightly more logic. We used numpy matrices to construct adjacency matrices from a system, which worked sufficiently well for large systems.

ALGORITHM 2: Generate random system

```
Input: Integers $n$, $m$.
Output: A set of random 3 literal clauses
$pool = \{1,2,..,n\}$;
$system = \{\}$;
$tries = 3$;
$index = 0$;
repeat
    if $tries == 0$ then
        return $False$;
    end
    $clause$ = random 3 literals from $pool$;
    if $clause \in system$ then
        $tries- = 1$;
    else
        $system$ add $clause$;
        $index+ = 1$;
    end
until $index == m$;
return $systems$
```

4.4 Parameter Settings

During implementation, many constants and variables were assigned values in response to issues. These hard coded values are of importance when we discuss results. For the meantime, we will define them here so that they can easily be identified. See the following table for descriptions.

Parameter	Description
Step	Searching for systems requires some iterator over n and m. This increments m by some value.
Limit	We may want to find this number of systems from a given m. Useful if we want to find the smallest ratios.
Tries	Generating random systems has two problems: 1, we include the same clause twice from a pool of literals. 2, The system may not match our criteria Therefore, we try this many times before moving on.
Timeout	dreadnaut, nauty/Traces
Upper bound and lower bound	Suppose we wanted to search along the threshold of satisfiability. we would set this value to 1. This describe the upper and lower bound of n = u * m, for some u.
Strongly *k*	A system is strongly *k* if there is a large difference in Gauss-On versus Gauss-Off. Some systems have a larger discrepancy than others of the same size. We put aside systems with the largest discrepancy during our Advanced Search.

Table 4.2 Search parameters

4.5 Development Issues

Faults are inevitable in ever undertaking, and ours are reported in this segment.

We noticed during execution strange behaviour from Traces and the operating system. This is notable since these issues may effect our report of results, although, these issues may be uncommon and spurious.

4.5.1 Python OOM

Python processed tended to crash after short periods of time on the experiment server. This required swap space to be allocated as RAM was not satisfactory. Attempts were made without having to include swap space, the caveats being access to secondary memory should be averted. However, larger implementations which included a 2 CPU machines with 2gb RAM still was not satisfactory. Therefore, swap space must be included on small installations.

4.5.2 Traces OOM and Indefinite Execution

Traces was expected to halt when instances became too large to process. Luckily, our construction was below this threshold for small instances that mattered. However, at node sizes roughly at 6000, Traces began to fall short at seemingly random times. Occasionally, Traces would run one graph for an indefinite period of time (which had to be manually cancelled), and for another run, ran for a few hours. Given that we had the instances we needed, this issue was not explored, and may have only been an operating system issue, since 20gb swap space had to be allocated to permit for larger instances.

For some instances, even at the relatively small end of our graphs, which consisted of a manageable number of nodes, say 4000, Traces would crash unexpectedly after multiple hours. The details of these instances were not recorded, however for relatively small values of our graphs, graphs which executed for numerous hours, say n=500, Traces would fail. It is unknown whether this due to hardware insufficiencies. Therefore, for instances which timeout, where no timeout is defined - this describes a system error of some sort. One such example is the tnn(2) graph, whose similar graph tnn(1), would run for multiple days, causing strange CPU behaviour. There is a picture of this issue displayed in appendix. The cause is unknown, a snapshot of the development server is also placed in the appendix to reproduce problems. Hence, some instances which timed out, may have in fact fell under this bug. It is quite possible that instances became too complex at a point, which required more swap space than allocated. However, this is only a speculation.

4.6 Additional Features

Given that there is more than one way to program our logic, multiple strategies were used to generate our desired construction. Here we define additional methods utilised and features that extended beyond our requirements and design. They may be of importance in future applications.

4.6.1 Alternative XOR-Formula Generation

Our algorithm to generate desired systems picks three random literals and adds these to a set of clauses. However, for some values of n and m, a system cannot be located. We know that there exists a suitable system for these values, but cant be selected at random due to the large permutation of n. For small values of n and m, we believed that all combinations could be held in memory and systematically tested, rather than being generated at random. Hence, a recursive search is provided to find all combinations to be tested. However, in implementation, this could not be used even for small values of n and m due to running times. Nonetheless, this feature may be of importance when time is not an issue, and a set of desired systems are required for given values.

4.6.2 Updating Slow Systems

Since we determine k-local consistency by checking if one system executes faster with Gauss-On versus Gauss-Off, it can be problematic to determine if a system is suitably slow. Small systems execute very quickly, and differ into two runs by only sub second values. These sub second difference could be mistaken for fluctuations in system performance, rather actual operating time (since we are measuring time not by process execution time, but start and end times). Here we have two solutions to mitigate this error.

The first solution is to use some parameter, say *minimal_difference*, to provide a lower bound on the value required for a k-locally consistent system. However, this may be too large for small systems to register or too small to attain difficult instances. The second option is to solve this problem by repeating runs. We save those systems which have the largest difference in a separate directory, and replace these systems when comparing differences. If a newly found system is slower, that is it has a larger positive difference than the stored one, then we replace the stored one. This way, by using many iterations, we can generate a set of systems which have the largest differences to date. We are able to filter out those instances with exceptionally slow times. These graphs will be the most difficult found, and thus, the ideal type to construct to use in benchmarking. We prove this experimentally.

4.6.3 Advanced Search

Considering that we must check if a suitable system gives rise to a graph with no non-trivial automorphism, we can combine this check when searching for systems. In addition to updating strongly k-systems, we have the tools to search for the ideal graph.

If we convert the system to a graph and check for automorphisms on the fly, that is, whilst searching and validating systems, we eliminate the need to build up a database of suitable systems up to the point of automorphisms, at the cost of running time. There are parameters defined enabling us to use this extended validation check.

4.6.4 Optimised Search

Numerous parameters are available to speed up searching. Here we briefly describe how they can be used.

The upper bound and lower bound variables search for a given region of $nandm$. For example, if we wanted to search for systems along the line of satisfiability, we would set both values to 1. This would ensure we get the most difficult instances possible, but results in longer execution time due to failed tries. Moreover, we can check along $2n = m$ or between $n = m$ and $2n = m$, and so forth. This becomes clear in the following Chapter.

Similar to the upper bound and lower bound variables, the limit variable permits us to search for a given number of systems for a given n. That is, move up n by step if we have found $limit$ systems. This is important when we consider the next paragraph.

Efficient search variable checks that we solve the following optimisation problem. If we begin by searching along the line n=m, for large n, and no such system was found up until n=2m, then it is unlikely that the following iteration $n = n + step$ will find a system between n=m and n=2m. Thus, efficient search remembers the last good m value. Thereby eliminating the need to make redundant checks. This is useful, for suppose we set limit=1, lower-bound = upper-bound = 1, tries = 1000, $update\_strongly\_k = true$, then we could efficiently find systems close the the threshold of satisfiability. A number of these example searches are provided in the appendix.

4.6.5 Extending Traces Benchmarks

To permit us to see how large we could potentially make out construction, numerous random graphs provided by Traces were extended. Graphs with edge probability 1/2, 1/10 and sqrt(n) were constructed using programs provided by the nauty/Traces suite: genrang (to build) and showg (to convert .g6 to .dre). Instances ranged from 5 to 30,000 nodes, and were executed successfully.

Chapter 5

Evaluation

We evaluate our system in two parts: evaluation of program performance, and then how our construction did. Once defined, we will elaborate on results on various findings. Firstly, we will define our tested and the performance of benchmark graphs provided by Traces. For results which were not crucial to this project, see our GitHub repository [18].

5.1 Testing Environment

Our experiments were conducted on a cloud computer server to ensure long-running processes, such as searching and executing families of graphs, remained undisturbed. We often had to resize and make adjustments to system resources, as computing with our instances became difficult, as well as some benchmark graphs. We had the following setup.

Table 5.1 Test Environment

Feature	Description
Host	DigitalOcean
Operating System	Ubuntu 16.10 64-bit
Memory	2GB
Disk	20GB SSD
CPU	2 CPUs
Swap space	4GB
Programming Language	Python 2.7
Command Line Tools	dreadnaut, nauty/Traces

Note that we had to allocate 4GB swap space to tackle large instances to accommodate both Python and Traces memory requirements. A good tutorial is here:
https://www.digitalocean.com/community/tutorials/how-to-add-swap-on-ubuntu-14-04

5.2 Benchmark Tests

In gaining insight of the performance of our graphs, we first describe what results were reproduced. In total, we tested a suite of 33 graphs for the Automorphism Group function of Traces, more than what was necessary to determine our performance. We will now describe a few interesting packages worth mentioning; graphs we extended, slow graphs and interesting graphs. The graphs here mentioned are of most importance, additional benchmarks are provided in the appendix. For additional information about these graphs, refer to the Traces website. All graphs had a timeout value of three hours.

Random Graphs

Knowing that Traces may run out of memory, and in part, the most intuitive graph we can compare to are random graphs with differing edge probabilities. Our construction is also generated at random, but with a much more sophisticated algorithm. So random graphs provided a starting point for comparison and testing the capabilities of Traces.

The random graphs provided by Traces differed in edge probabilities: 1/2, 1/10 and sqrt(n), where n is the number of vertices. that is to say, given a number of nodes n, add edges with the probabilities to each pair of nodes with edge probabilities above. The sizes of these graphs however, was somewhat small, ranging up to 5000 nodes. In practice, these executed the Automorphism Group test of Traces quickly. We then made it our business to extend up to 30,000 nodes to see what would happen.

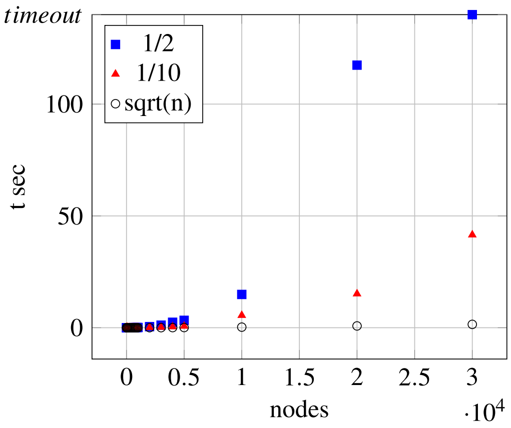

Fig. 5.1 Extended random graphs up to 30,000 nodes

Cai-Furer-Immerman Graphs

Another notable graph family are the Cai-Furer-Immerman graphs, also known as CFI. These constructions are resistant to the 1-dimensional Weisfeller-Lehman tests and variants, but are prone to being factored by automorphisms in Traces. See Figure 5.2.

Other Graphs

Other notable graphs which we include here are graphs which were amongst the slowest executing. These include Hadamard (had), Disjoint union of triparite graphs (tnn) and projective planes of order 25 (pp). Note that on the *x* axis of pp graphs we select only the graphs which have 1302 nodes. thus, the *x* axis only represents the label of the graph.

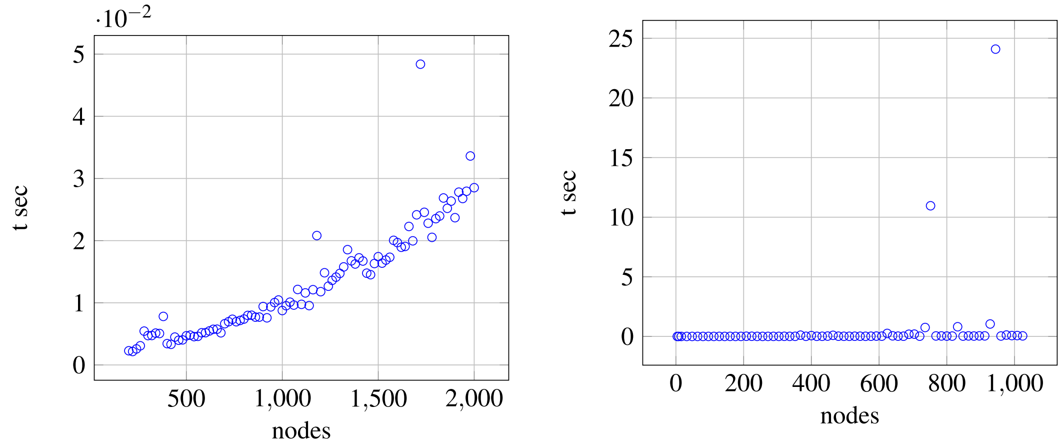

Fig. 5.2 Left: Cai-Furer-Immerman graphs (cfi). Right: Hadamard (had).

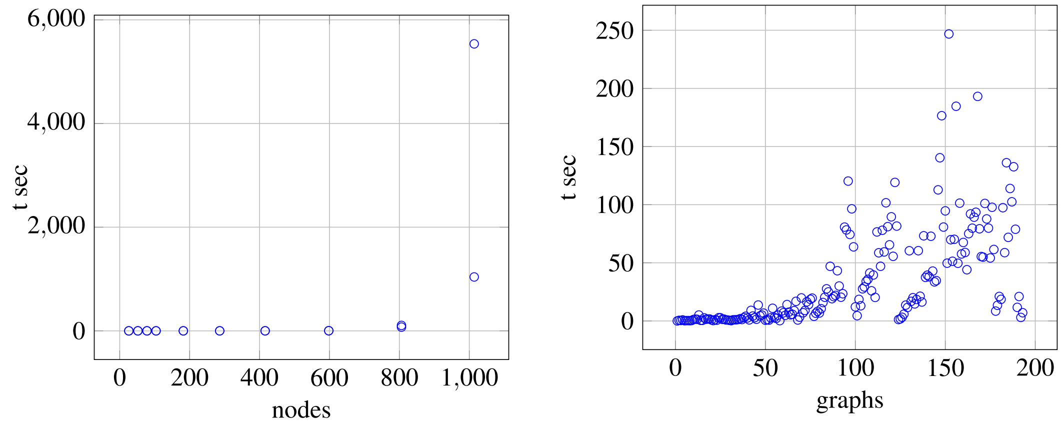

Fig. 5.3 Left: Disjoint union of tripartite graphs (tnn).
Right: Projective planes of order 25 containing 1302 nodes (pp).

5.3 Results

Here we present the performance of our construction for varying node sizes.

5.3.1 Packages

Since we range our systems in complexity and along different rations, we package our graphs into three: $n = m$, $2n = m$ and smallest ratios found. Given that we stop finding systems for a reasonable number of tries at 139 for $n = m$, we continue searching the closest we can, without having to ramp up the try value. Hence, for the most part, tries were left at 10, and we performed and Advanced Search with multiple iterations (>30), updating k-locally consistent systems and checking for automorphisms on the fly. We then executed some logic to separate these packages along the lines defined above. Of course, we could have had more lines, say $3n = m$ and $xn = m$, where $1 < x < 2$. But our results proved satisfactory to show distinctive times. We did provide more packages, but we will focus on these for now. Figure 5.4 best describes what we mean by various lines and instances found for n and m. Note that these instances are not necessarily k-locally consistent.

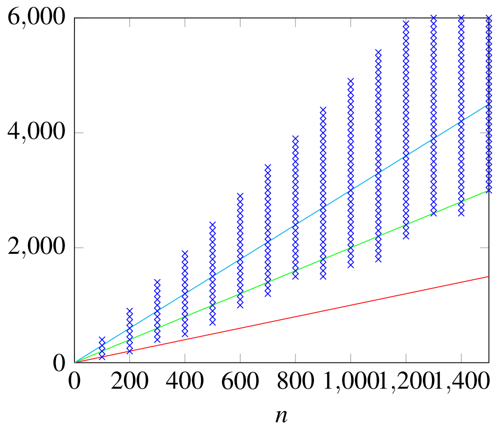

Fig. 5.4 Uniquely satisfiable instances found.
Red: $n = m$; Green: $2n = m$; Cyan:$3n = m$; Crosses: Uniquely satisfiable instances found

5.3.2 Performance

Here we present the performance of varying packages. Figure 5.5 displays the timings of the three packages created. Figure 5.6 illustrates our most difficult package *con_sml* versus existing benchmarks. Results showed that we consistently produced graphs which executed for longer than three hours, however, since many of the packages did not execute for longer than the specified timeout, we could not compare graphs of equal sizes for most graphs which timed out.

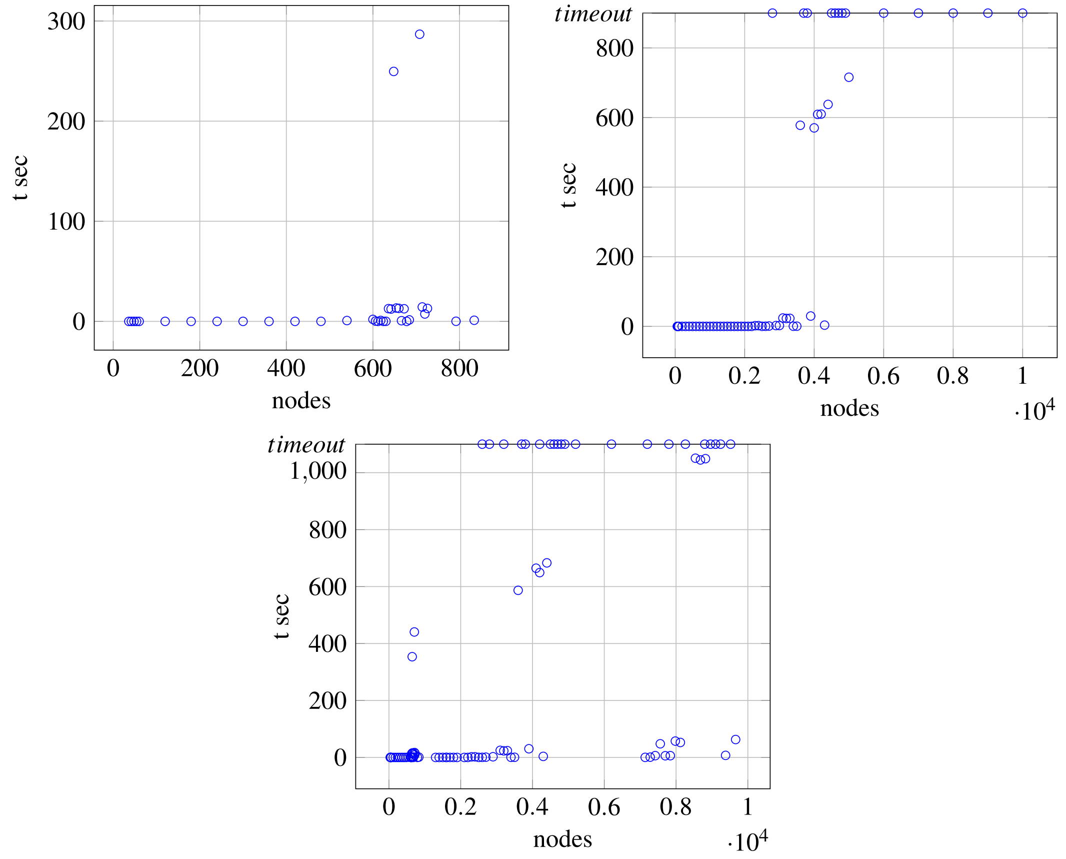

Fig. 5.5 Left: n=m (con_n).
Right: n=2m (con_2n).
Center: smallest ratios found (con_sml)

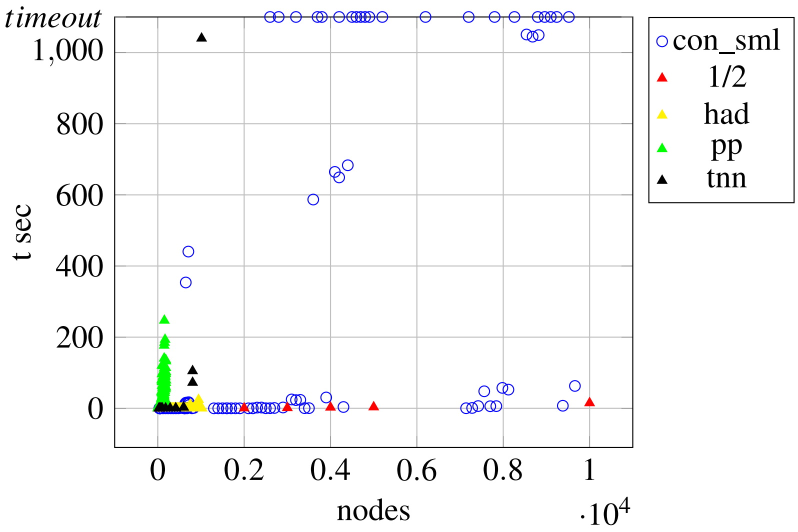

Fig. 5.6 con_sml (slowest running graphs) versus benchmarks

5.4 Discussion

5.4.1 Searching

Randomly generating our construction given n and m permits us to try and generate many instances of the combinations of n and m. So, for any value of n and m, we are able to search for instances that lie along the $2n = m$ line. This is better described in the following plot below:

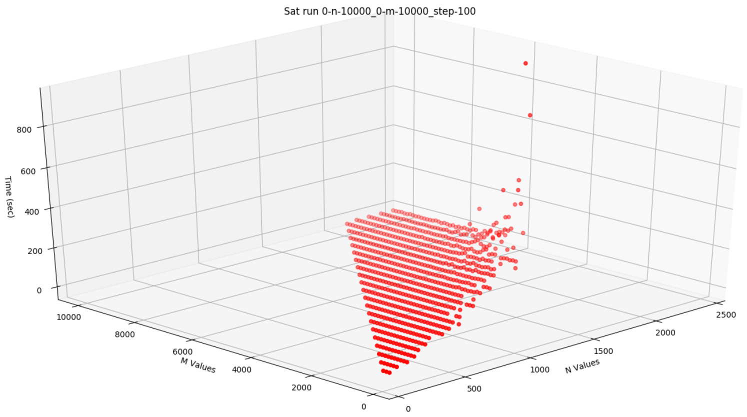

Fig. 5.7 Time taken to search for uniquely satisfiable instances

(Note that many graphical representations have rendered values illegible. To this end, we will describe some verbally, but retain enlarged images in our package. Such results were not saved as values, but kept as images after long running experiments.)

The x axis represents variables n, y represents clauses m and the z axis represents time taken to locate in seconds. This figure was produced after recording the time taken to find a uniquely satisfiable system of equations for combinations of n and m. Hence, what we mean by the $2n = m$ line, are those instances which satisfy that equation (e.g. $n = 10$ and $m = 20$). This is an important distinction ot make, as we see that instances along certain lines differ in time taken to find, namely, those that are rightmost, that is, close to the threshold of satisfiability $n = m$ (the left is bounded by $3n = m$). Figure 5.7 shows the cumulative time taken to locate each instance, which means each point includes failures and regenerating and testing instances for unique satisfiability (not necessarily k-locally consistent). It does not describe the validation time by Cryptominisat for a single instance, but does hint at this. Figure 5.8 illustrates that we have to try much harder near the line of satisfiability to generate uniquely satisfiable systems, shown by the increase in time to the right. We see that for a single instance to be located, it took more than a hour to find a suitable instances near the $n = 2000$ and $m = 6000$ mark. One point to make, is that the number of tries stays the same. The empty space above lines $n = m$ and below the rightmost plots are instances which were not located using 10 attempts. Hence, it is not only becomes more unlikely to find instances, it also becomes more timely to validate them.

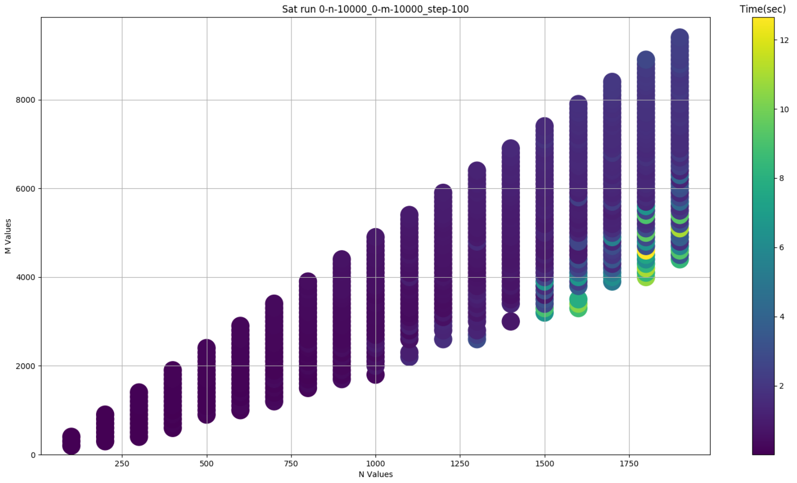

Fig. 5.8 Plot of various systems found.
Note that instances close to the $n = m$ line take far greater time in validation.

5.4.2 Validation

Ensuring we had our desired constructions meant we had to make three checks: k-local consistency and unique satisfiability by Cryptominisat, and automorphisms on our preliminary graph using Traces. (Checking for automorphisms is left to Section 5.5.)

Unique satisfiability

Aforementioned, the time taken to check for unique satisfiability increases along the threshold of satisfiability with respect to the size of the systems. Figure 5.9 shows this.

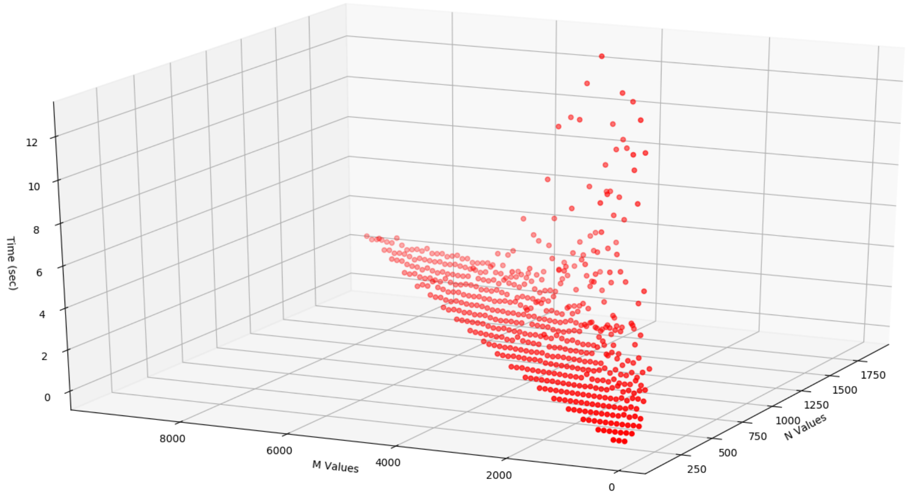

Fig. 5.9 Time taken to validate unique satisfiability using Gauss Off

We note that Cryptominisat works faster with Gauss-Off feature versus Gauss-On. This is Counter-intuitive, since XOR-SAT is known to have a P-Time algorithm using Gaussian elimination. Evidently, validation of unique satisfiability systems, which are along the threshold, appear to take a vastly greater time than those further away.

k-local consistency

Our makeshift k-local consistency check found systems which were significantly quicker using Gauss-On versus Gauss-Off. In our Advanced Search, where we updated our store of slow running systems, that is, with the greatest difference in Gauss-On and Gauss-Off times. The following figure shows systems with negative plots are pseudo k-locally consistent. See Figure 5.10 below.

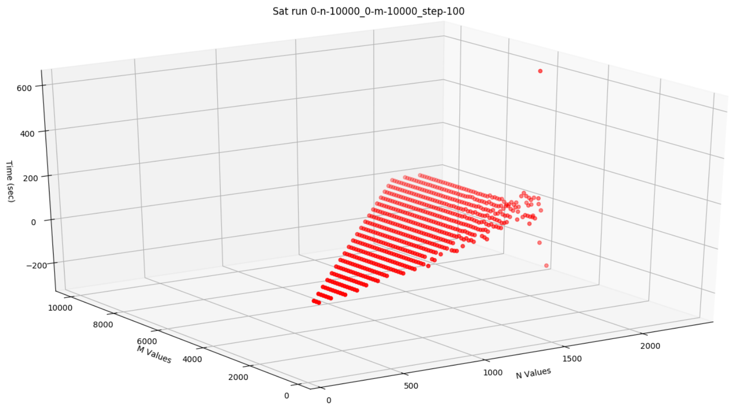

Fig. 5.10 Time taken to validate unique satisfiability Gauss On - Gauss Off.

Notice that is more difficult to detect k-local consistency the further we move away from the threshold $n = m$. The difference in running times are fairly similar away from $n = m$, but erratic close to $n = m$. Especially as we increase the size of the systems. To which the boundary also moves further away from the line.

Strengthening

As we will prove later, the larger difference between a Gauss-On and Gauss-Off execution time, where Gauss-On is faster for a given uniquely satisfiable 3-XOR-Formula ϕ, the slower the final graph G_B will be after transforming ϕ to G_B. Recall that in our Advanced Search we provide the facility to save the slowest found ϕ. By repeating system searches, and continuously updating the slowest found ϕ, we filter out very slow systems from the pool of random uniquely satisfiable systems. Hence, repeating a system search 30 times for specific instances, say $n = 100$ and $m = 100$, we can hope to find a strong instance which is slow for that value of n and m. However, it is critical to note that for these values, systems must be readily available and not scarce.

5.4.3 Varying Tries

As we search for instances close to the line of satisfiability $n = m$, our desired construction becomes much more difficult to locate. Recall that the tries parameter in our systems search determines how many times to generate ϕ before quitting.

We noticed that as n and m became larger, if we fixed the number of tries, then it would become more difficult to find instances close to the line of satisfiability. That is to say, as n and m increased, there would be a point where finding $2n = m$ would require more random generations to produce a desired instance, similarly with $3n = m$ and so on. This is better illustrated, but not accurately shown, in Figure 5.11 below. The figure depicts that as values of n and m increase, instances along certain lines become scarce. This is important to note, since if we scarcely find instances at certain points, requiring a greater number of tries to locate, we are less likely to filter out a k-locally consistent system through repeating runs.

One notable test that can be reproduced is searching along $n = m$, and setting the number of attempts to a large 100,000 attempts to generate a system. One would find that desired instances are hardly available above the $n = m = 100$ mark. We know such instances exist, but it becomes increasingly difficult to find an instance for say $n = m = 150$. The largest ϕ we found for this ratio of n and m is 139, but this was not necessarily k-locally consistent as we generated this in 1 run: this is due to scarcely seeing graphs between 105 and 139, that is, many instances exceeded the 100,000 try value. Those below 100 were easy to find (10 tries), and strong systems were found after many iterations, but the same cannot be said for $n > 105$, although, this would be ideal since it would produce slow running graphs.

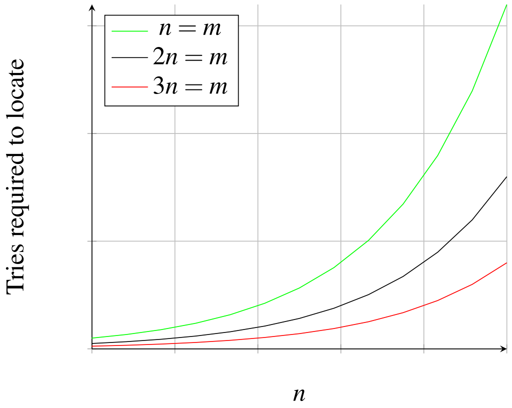

Fig. 5.11 Illustration of instances becoming scarce

5.4.4 Varying Step Size

In our packages we provide systems and graphs which increase in size by some step factor, as defined in Table **??**. If we vary the number of steps, we alter the amount of systems produces by our systems search, and ultimately how many graphs are generated. For some packages, for example *con_n*, we have a step size of 1 between $6 < n < 10$, which increases to 10 at $10 < n < 100$, but then decreases to 1 between $100 < n < 150$. Recall that as instances become larger, they are begin to be scarcely available, which is prominent in the $n = m$ case.

5.4.5 Bounds

In searching and constructing our graphs, we hit multiple bounds on what we could find. Such bounds are purely time constraints. For example, we decided to stop searching systems above $n = 3000$, since our final graph G_B would undoubtedly take longer than three hours to determine $Aut(G)$. A graph of $n = 3000$ would result in G_B having at least $18,000$ nodes. Figure 5.6 shows it was not necessary to generate graphs at this scale. Nonetheless, uniquely satisfiable instances were found at this range. Moreover, the time taken to validate such a system became large; Figure 5.7 shows that it would take longer than 800s to find and validate such systems. We do not supplement uniquely satisfiable systems above $n = 2500$ and do not provide graphs above $n = 1000$ for these reasons.

5.4.6 Timings

Given that our construction is a graph which we test for automorphism that corresponds to a uniquely satisfiable and k-locally consistent 3-XOR-Formula, and that such a construction is tested on two types of solver, we provide a table of figures to see the overall performance. It should be emphasised that these times are based on the time taken to execute system calls initiated by Python, rather than actual CPU time for processes. Since Cryptominisat does not have such an inbuilt feature to show these values, we provide a subset of times for Traces $Aut(G_B)$ and Cryptominisat $uniquely\_satisfiable(\phi)$ using Gauss-Off. To save space, we refer to Appendix A. In total, Table 5.2 tells us the percentage of instances which timed out in our provided packages.

Package	Largest Graph $n : m$	Graphs	% of Timeouts
con_sml	1000:1700	95	21%
con_n	139:139	35	0%
con_2n	1000:2000	59	22%

Table 5.2 Percentage of graphs which took longer than three hours to execute on Traces

5.5 Preliminary Graph Automorphism Test

Since we want our graphs with no non-trivial automorphisms, we must validate that a given system when translated into our preliminary representation has this attribute. See Figure 5.12

We found that testing graph G_A was a fairly fast check, and executed at most after a few seconds. This led to the integration of this check into our advanced search as a feature. The following figure shows run times of some preliminary graphs for automorphisms versus the corresponding final construction. We see the significance and drastic increase in execution times once we apply the multipede transformation.

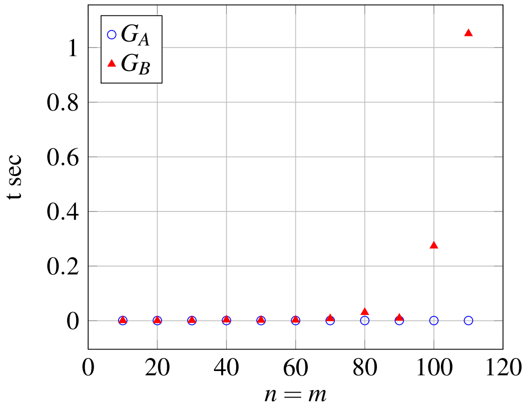

Fig. 5.12 Graph G_A versus G_B for $n = m$

5.6 k-local Consistency Experimental Proof

Here we provide experimental proof that using gauss elimination on versus off did indeed make a difference in times. We take graphs of two identical sizes, $n = 300$ and $m = 600$. For one system, we generated a uniquely satisfiable set of clauses without checking our stored version, which is the other. We convert both systems into graphs and find that one system is substantially slower than the other. We see that a system with a large discrepancy in time gives rise to a drastically slower system. See Table 5.3.

Instance 300:600 ($n : m$)	Gauss On	Gauss Off	Traces time
Random uniquely satisfiable system	0.0078	0.0077	0.18
Likely k-locally consistent	0.0074	0.0129	1.04

Table 5.3 Comparing two graphs. Both are $2n = m$ with $n = 300$ and $m = 600$

Chapter 6

Conclusion

In summary, we have presented and produced a family of graphs which execute slowly on the fast graph isomorphism solver Traces. We provided the capability to generate difficult graphs at random, satisfying rigidity and resistance the vertex colour refinement algorithm underlying the solving program. We utilised a SAT solver, and implemented concepts from combinatorics, computational complexity and graph theory. Our package of graphs consistently took longer than three hours to execute for many instances. We successfully utilised Gaussian elimination and unique satisfiability of 3-XOR-Formulas as good guess of odd and k-locally consistent instances. Furthermore, we executed our searching algorithm for many thousands of iterations to incrementally increase the complexity of graphs. In addition, we provided experimental proof of the importance of k-local consistency when generating difficult instances. Including benchmark tests with existing graph families.

Previously, the best known randomly generated graphs were those with differing edge probability. Here we provide open source code to randomly generate difficult graphs to further our understanding of the graph isomorphism problem.

6.1 Further Work

During the time of this project, another analysis into benchmark graph isomorphism solvers was produced. The work in question is of Neuen and Schweitzer, which utilised similar ideas of multipedes to generate difficult graphs, however the approach differed in that shrunken multipedes were used, in contrast to our method of randomly generating such graphs from an 3-XOR-Formula [19]. During the final stages of experiments, we selected a similar timeout value to this paper (three hours). We must clarify that this paper arose during our final stages, and influenced our work only by means of determining a timeout value. This was done to test our graph against the similar construction. Given that experiments were already underway, we point to this work to compare our own. Since different techniques were used to generate similar graphs. That is, using a form of reduction to make graphs more difficult. A point of departure would be to include the newly created package and test against this.

In this work, there is potential to find stronger and more difficult instances than the ones we have found. By using multiple experiments to build up a database of slow graphs, the ones we have provided in packages are assuredly not the slowest existing. To this end, a point of departure would be continuously run our searching algorithm, make improvements to the efficiency in locating and incorporate better techniques to find systems. The backbone of finding difficult graphs for the GI problem has been presented. With further work to this algorithm, we can undoubtedly come a step closer to understanding the graph isomorphism problem and its limits.

References

[1] Brendan D McKay et al. Practical graph isomorphism. 1981.

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

[3] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 135–149. SIAM, 2007.

[4] José Luis López-Presa, Antonio Fernández Anta, and Luis Núñez Chiroque. Conauto-2.0: Fast isomorphism testing and automorphism group computation. *arXiv preprint arXiv:1108.1060*, 2011.

[5] László Babai, William M Kantor, and Eugene M Luks. Computational complexity and the classification of finite simple groups. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 162–171. IEEE, 1983.

[6] László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 684–697. ACM, 2016.

[7] Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

[8] Takunari Miyazaki. The complexity of mckay's canonical labeling algorithm. In *Groups and Computation II*, volume 28, pages 239–256. Aer. Math. Soc.: Providence, RI, 1997.

[9] Reinhard Diestel. *Graph theory*. Springer, 2017.

[10] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[11] C. Moore and S. Mertens. *The Nature of Computation*. OUP Oxford, 2011.

[12] Olivier Dubois and Jacques Mandler. The 3-xorsat threshold. *Comptes Rendus Mathematique*, 335(11):963–966, 2002.

[13] Boris Pittel and Gregory B Sorkin. The satisfiability threshold for k-xorsat. *Combinatorics, Probability and Computing*, 25(02):236–268, 2016.

[14] Yuri Gurevich and Saharon Shelah. On finite rigid structures. *The Journal of Symbolic Logic*, 61(02):549–562, 1996.

[15] 2017. [online] http://www.satcompetition.org/.

[16] 2017. [online] http://dimacs.rutgers.edu/.

[17] Brendan Piperno. Nauty traces – home, 2017. [online] http://pallini.di.uniroma1.it/.

[18] 2017. [online] https://github.com/quasipolynomial/mphil.

[19] Daniel Neuen and Pascal Schweitzer. Benchmark graphs for practical graph isomorphism. *arXiv preprint arXiv:1705.03686*, 2017.

Appendix A

Timings

$n : m$	n	Cryptominisat	Traces (via Python)
n=m	10	0.019	0.001
	20	0.016	0.001
	30	0.014	0.002
	40	0.015	0.002
	50	0.014	0.003
	60	0.012	0.003
	70	0.013	0.003
	80	0.015	0.038
	90	0.017	0.822
	100	0.024	1.991
	101	0.02	0.287
	102	0.017	0.014
	103	0.014	0.919
	104	0.016	0.014
	105	0.015	0.08
	106	0.013	12.722
	107	0.014	12.326
	108	0.014	249.625
	109	0.016	13.336
	110	0.015	13.012
	111	0.017	0.672
	112	0.019	12.6
	113	0.015	0.016
	114	0.016	1.469
	118	0.016	286.805
	119	0.014	14.362
	120	0.017	7.39
	121	0.017	12.912
	132	0.015	0.104
	139	0.028	1.109

Table A.1 Results for n=m

$n : m$	n	Cryptominisat	Traces (via Python)
2n=m	10	0.01	0.001
	20	0.01	0.001
	30	0.011	0.001
	40	0.011	0.002
	50	0.012	0.003
	60	0.013	0.002
	70	0.014	0.002
	80	0.013	0.004
	90	0.012	0.004
	100	0.017	0.002
	200	0.021	0.053
	300	0.03	2.486
	400	0.059	570.172
	500	0.044	715.733
	600	0.07	Timeout
	700	0.078	Timeout
	800	0.102	Timeout
	900	0.101	Timeout
	1000	0.158	Timeout

Table A.2 Results for 2n=m

Appendix B

Supplementary Material

Constructing Hard Examples for Graph Isomorphism

Anuj Dawar

1 XOR Formulas

Fix a countable set $\mathcal{X}$ of Boolean variables. We use capital letters $X, Y, \ldots$ to range over this set. A 3-XOR-formula is a finite set of clauses, where each clause contains exactly 3 literals, each of which is either a variable X or a negated variable $\overline{X}$.

We say that a 3-XOR-formula φ is satisfiable if there is an assignment $T : \mathcal{X} \to \{0,1\}$ of truth values to the variables $\mathcal{X}$ such that in each clause of φ, an *even* number of literals is made true.

Given a 3-XOR-formula φ, we can construct a system of linear equations over the two-element field $\mathbb{F}_2$. That is, for each clause C of φ we construct the equation $x + y + z = c$ where x, y, z are the variables occurring in the literals of C and c is 1 if an odd number of them appear negated and 0 otherwise. It is easily verified that this system of equations has a solution if, and only if, φ is satisfiable. Note that two distinct clauses may give rise to the same equation. Say that two clauses are *equivalent* if they give rise to the same equation.

2 k-local Consistency

We are interested in 3-XOR-formulas that are unsatisfiable but k-locally consistent, for suitable integer k. For our purposes, we define k-local consistency by means of the following pebble game, played by two players called Spoiler and Verifier. The game is played on a 3-XOR-formula φ with k pebbles $p_1, \ldots, p_k$. At each move Spoiler chooses a pebble p_i (either one that is already in play, or a fresh one) and places it on a variable X appearing in φ. In response, Duplicator has to choose a value from $\{0,1\}$ for the variable X. If, as a result, there is a clause C such that all literals in C have pebbles on them and the assignment of values to them given by Duplicator results in C being unsatisfied, then Spoiler has won the game. Otherwise the game can continue. If Duplicator has a strategy to play the game forever without losing, we say that φ is k-locally consistent.

It is known (see for instance [1]) that k-local consistency has a close relationship with definability in the logic $\exists L^k_{\infty\omega}$—the existential positive fragment of the infinitary logic with k variables. In particular, the class of non k-locally consistent formulas is definable in this logic, while the class of unsatisfiable formulas is not. There are, for each k, unsatisfiable formulas that are k-locally consistent.

3 Random XOR Formula

For fixed positive integers m, n we write $F(m, n)$ for the set of all 3-XOR-formulas over the variables $X_1, \ldots, X_n$ containing exactly m inequivalent clauses. We also write $\mathcal{F}(m, n)$ for the

uniform probability distribution over $F(m,n)$. It is known that, for large enough values of m and n, with $m > n$, a random formula drawn from this distribution is unsatisfiable (see [7]).
Question 1: How large does n have to be to make the probability of unsatisfiability greater than $\frac{1}{2}$, say?
Experiment: Construct for some large values of n, random 3-XOR-formulas and run them through a SAT solver to check for satisfiability. Use this to build up a database of random unsatisfiable formulas.

Also, for any k there is a constant c_k such that, for sufficiently large m and n with $m > c_k n$, a random formula drawn from $\mathcal{F}(m,n)$ is k-locally consistent. This is proved for 3CNF-formulas in [1], but should also follow for 3-XOR-formulas from results in that paper and [3, 2].
Question 2: What is the value of c_k?
Question 3: How large does n have to be to make the probability of k-local consistency greater than $\frac{1}{2}$, say?

4 Homogeneous Systems of Equations

Thinking of a 3-XOR-formula as a system of equations, we say that it is *homogeneous* if the right hand side of each equation is 0. Note that a homogeneous system of equations is always satisfied by the constant 0 assignment. Say that a homogeneous system of equations is *uniquely satisfiable* if this is the only satisfying assignment to its variables.

Define $H(m,n)$ to be the set of all homogenous systems of equations with m clauses and n variables, and $\mathcal{H}(m,n)$ for the uniform probability distribution over this set.
Question 4: Is it the case that for some constant Δ, and sufficiently large m and n with $m > \Delta n$, a random system from $\mathcal{H}(m,n)$ is uniquely satisfiable? If so, what is the value of Δ? How large does n have to be to make the probability large enough?

Note that every homogeneous system is k-locally-consistent, because it is satisfiable. For the construction, we actually require a property stronger than k-local-consistency. Say that a formula φ in $F(m,n)$ is *strongly k-locally consistent* if for any variable X in $X_1,\dots,X_n$ the formula $\varphi[X \leftarrow 1]$ obtained by fixing the value of the variable X to be 1 is k-locally-consistent.

It should be provable that, again, for any k there is a constant s_k such that, for sufficiently large m and n with $m > s_k n$, a random formula drawn from $\mathcal{F}(m,n)$ is strongly k-locally consistent. And the same should follow for $\mathcal{H}(m,n)$.
Question 5: What are the values of the parameters s_k and what are sufficiently large values of n for this to work?

We are especially interested in homogeneous systems that are uniquely satisfiable and strongly k-locally consistent.

5 Graph Construction

We give a construction of graphs from formulas that is inspired by those of Cai et al. [4] and the multipedes of [5].

For any 3-XOR-formula φ, we define a graph G_φ by the following construction. If φ has m inequivalent clauses and n variables, G_φ has a total of $4m + 2n + 3(n-1)$ vertices.

For each clause C of φ, let $C_1 = C$ and let C_2, C_3, C_4 be the three clauses equivalent to C obtained by negating exactly two of the literals of C. We then have a vertex in G_φ for each of these clauses. Also, for each variable X in φ, we have two vertices X^0 and X^1. In addition, for each i with $1 \leq i < n$ we have three vertices i_l, i_r, i_s.

The edges are described as follows. For each clause C, if the literal X occurs in C, we have an edge from C to X^1 and if the literal $\overline{X}$ occurs in C, we have an edge from C to X^0. There is an edge between X^0 and X^1. For each i we also have the edges: (i_l, i_r), (i_r, i_s) and (i_l, X_i^0), (i_l, X_i^1), (i_r, X_{i+1}^0) and (i_r, X_{i+1}^1).

Recall that we say that a graph G is *rigid* if it has no non-trivial automorphisms.

Proposition 1. *If φ is homogeneous, then it is uniquely satisfiable if, and only if, G_φ is rigid.*

Proof. Letα be any automorphism of G_φ. Note that every clause vertex C has degree 3. Every variable vertex X^0 or X^1 has degree at least 4. Every vertex i_s has degree 1. Thus, each of the following sets is fixed (set-wise) by α:

- the set $S = \{i_s \mid 1 \leq i < n\}$: this is the set of vertices of degree 1;
- the set $R = \{i_r \mid 1 \leq i < n\}$: this is the set of vertices adjacent to a vertex in S;
- the set of clause vertices $\mathcal{C}$: this is the set of vertices of degree 3 that are not within distance 2 of a vertex in S;
- the set of variable vertices $\mathcal{X}$: this is the set of neighbours of $\mathcal{C}$; and
- the set $L = \{i_l \mid 1 \leq i < n\}$: this is everything else.

Indeed, we can say more. Each of the sets S, L and R is fixed *pointwise* by α. If this were not so, there would be some i, j with $i < j$ such that $\alpha(i_s) = j_s$ (since the set S is fixed). Then, $\alpha(i_r) = j_r$ (since these are the sole nieghbours), $\alpha(\{X_i^0, X_i^1\}) = \{X_j^0, X_j^1\}$ (since these are the only neighbours in $\mathcal{X}$ of i_r and j_r respectively), and so $\alpha((i+1)_l) = (j+1)_l$ and $\alpha((i+1)_r) = (j+1)_r$. Proceeding by induction, we have for all k $\alpha((i+k)_r) = (j+k)_r$. Taking k large enough so that $j + k > n$, we get a contradiction.

It also follows that, for each variable X, $\alpha(\{X^0, X^1\}) = \{X^0, X^1\}$. That is, α either fixes each of the two vertices or it interchanges them. Note that if α fixes all the variable vertices, then it is the identity everywhere, since no two vertices in $\mathcal{C}$ have the same neighbours in $\mathcal{X}$. Let T be the assignment that maps X to 0 if α is the identity on $\{X^0, X^1\}$ and 1 otherwise. We now check that T satisfies φ.

In the other direction, suppose there is a truth assignment T that satisfies φ, we can show that the map on $\mathcal{X}$ that exchanges the vertices X^0 and X^1 just in case $T(X) = 1$ and is the identity everywhere else on $\mathcal{X}$ can be extended to an automorphism of G_φ. □

Note and Extension: The role of the vertices i_l, i_r, i_s is just to eliminate automorphisms based on a permutation of the variables. Perhaps we don't need them if we can prove that a random formula does not admit such automorphisms anyway.

6 C^k-rigidity

C^k is the fragment of first-order logic where we allow counting quantifiers: $\exists^i x\varphi$ means that there exist at least i distinct elements x satisfying φ, but each formula has at most k distinct variables.

For a graph G and a vertex $v \in V(G)$, the C^k-type of (G, v) is the collection of all C^k formulas $\varphi(x)$ in one free variable such that $G \models \varphi[v]$. We write $u \equiv^k v$ to indicate that (G, u) and (G, v) have the same C^k-type. This equivalence relation is characterized by the bijection game of Hella [6]. We say that a graph is C^k-rigid if no two vertices have the same C^k-type. The Hella bijection game can be used to establish the following.

Proposition 2. *If φ is strongly k-locally consistent, then for any variable X occuring in it, $X^0 \equiv^k X^1$ in G_φ.*

It is also known that equivalence in C^{k+1} is the same as indistinguishability in the k-dimensional Weisfeiler-Leman algorithm for graph isomorphism. This is a significant generalization of the method of vertex refinement.

Hence, if φ is a homogeneous system of equations that is strongly k-locally consistent, then G_φ is rigid but not C^k-rigid. Such graphs should be hard for Traces. Moreover, choosing φ at random from the distribution $H(m, n)$ for suitably large values of m and n should ensure that φ has both these properties with high probability.

References

[1] A. Atserias. On sufficient conditions for unsatisfiability of random formulas. *J. ACM*, 51:281–311, 2004.

[2] A. Atserias and V. Dalmau. A combinatorial characterization of resolution width. *J. Comput. Syst. Sci.*, 74:323–334, 2008.

[3] E. Ben-Sasson and A. Wigderson. Short proofs are narrow - resolution made simple. *J. ACM*, 48:149–169, 2001.

[4] J-Y. Cai, M. Fürer, and N. Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

[5] Y. Gurevich and S. Shelah. On rigid structures. *Journal of Symbolic Logic*, 61:549–562, 1996.

[6] L. Hella. Logical hierarchies in PTIME. *Information and Computation*, 129:1–19, 1996.

[7] B. Pittel and G. B. Sorkin. The satisfiability threshold for k-xorsat. *Combinatorics, Probability & Computing*, 25:236–268, 2016.