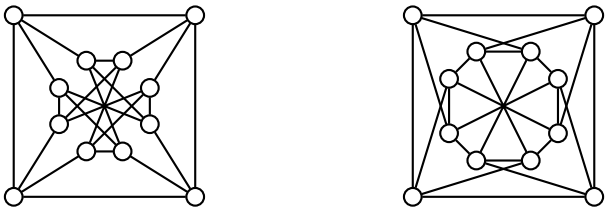
Chapter 1

Introduction



The *graph isomorphism problem* (GI) asks if two graphs are equivalent up to relabelling. The objective of this project is to create graphs that are difficult for practical GI solvers to check for isomorphism.

Over the years there has been great interest in the graph isomorphism problem as it is an NP-Intermediate candidate; although it is known that GI $\in$ NP, no polynomial-time algorithm that solves GI has been found, nor has GI been shown to be NP-Complete. Currently, the best theoretical algorithm has quasi-polynomial time complexity $O(\exp(\log(n)^c))$ [1], where n is the number of nodes in the graph, and c is a constant. If no polynomial-time algorithm exists for GI, then this is sufficient to show that P $\neq$ NP. On the other hand, if GI is shown to be NP-Complete, all problems in NP can be solved in quasi-polynomial time.

Besides theoretical implications, the graph isomorphism problem has practical applications in organic chemistry, hardware synthesis, social networks, and many other network-related areas. Fortunately, GI is relatively easy to solve in most cases: planar graphs, trees, bounded-degree graphs all have polynomial time algorithms, and practical GI solvers such as `nauty` and `Traces` [2] can tackle random graphs in almost linear time.

This discrepancy between theory and practice offers us a challenge — can relatively small graphs that perform badly on GI solvers be constructed? László Babai, a prominent graph theorist, asked a similar question: [1]

> "The question is, 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."

The construction of such graphs could serve as a benchmark for future GI solvers. Moreover, because current state-of-the-art practical solvers employ techniques that are deeply connected to their theoretical counterparts, the analysis of these graphs could also help us find better algorithms for GI and improve the theoretical bound.

Chapter 2

Preparation

This chapter first gives definitions of a graph and its morphisms. It then describes the refinement and individualisation algorithm used by graph isomorphism solvers. The notion of a difficult graph is developed through discussing the theoretical underpinnings and practical optimisations of the refinement-individualisation algorithm. By reviewing the DPLL algorithm used by SAT solvers, we reveal a novel connection between SAT and GI through exhibiting the similarities between the two search processes (original contribution). Linear algebra required for the project is introduced. Finally, it states the starting point of the project, which is used to define the requirements analysis.

2.1 Graphs

A graph G is defined as the pair (V, E), where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. For the purposes of this project, only undirected graphs, where $(v_1, v_2) \in E \iff (v_2, v_1) \in E$, will be considered.

A coloured graph is defined as the pair (G, π), where $\pi : V \to C$ is a colouring function assigning the colour $\pi(i)$ to vertex i. By convention, vertices and colours are labelled numerically from 1 to $|V|$ and 1 to $|C|$ respectively. If π is injective and $|V| = |C|$, then $\pi : V \to V$ is called a discrete colouring and acts as a permutation on the vertices.

2.2 Graph Isomorphism

Given $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ and $V_1 = V_2 = V$, G_1 is isomorphic to G_2 if there exists an isomorphism from G_1 to G_2. A permutation $\sigma : V \to V$ is an isomorphism if it preserves the edge relation:

$$\forall v_i, v_j \in V. (v_i, v_j) \in E_1 \iff (\sigma(v_i), \sigma(v_j)) \in E_2.$$

We would then say that G_1 and G_2 are equivalent up to isomorphism, or $G_1 \cong G_2$. For coloured graphs (G_1, π_1) and (G_2, π_2), both the edge and colour relations are preserved, so a vertex can only be mapped to another vertex of the same colour.

$$\forall v \in V. \forall c \in C \subseteq V. \pi_1(v) = c \iff \pi_2(\sigma(v)) = c.$$

2.3 Graph Automorphism

An automorphism is an isomorphism from a graph G to itself. $\mathrm{Aut}(G)$ is the set of automorphisms of G, and forms a group under composition. $\mathrm{Aut}(G)$ can be thought of as the symmetries of G. A graph that lacks non-trivial automorphisms, any automorphism except the identity permutation $id_V(v) = v$, is *asymmetric*.

2.3.1 Orbits

The orbits of G are defined using an equivalence relation. For $u, v \in V$,

$$u \sim v \triangleq \exists \sigma \in \mathrm{Aut}(G).\, \sigma(u) = v.$$

Since this is an equivalence relation, the orbit relation partitions the vertices into disjoint sets, called orbits. Therefore u and v are in the same orbit if $u \sim v$.

The graph isomorphism problem is computationally equivalent to computing the orbits of a graph as there exist polynomial-time reductions between the two problems [3]. Their high similarity led most practical graph isomorphism solvers to compute the automorphism group of the graph, which is then used to define a *canonical labelling* $C : \mathcal{G} \to \mathcal{G}$ such that $C(G_0) = C(G_1)$ if and only if the two graphs are isomorphic. Checking whether the canonical representation of the graphs are **identical** (not just isomorphic), can be done by directly comparing the adjacency matrices in $O(|V|^2)$ time.

Therefore we can conveniently measure the time required for the solver to find the automorphism group of a graph, instead of the time required to check pairs of similar graphs for isomorphism. In practice, most solvers only output the generators of the automorphism group, as the automorphism group itself may be exponentially large compared to the number of vertices in a graph.

2.4 The Refinement-Individualisation Algorithm

Of the hundreds of approaches tried on graph isomorphism, the strategy of using *colour refinement* and *vertex individualisation* has proven to be the most effective, and is used by most modern graph isomorphism solvers such as `nauty`, `Traces`, `bliss` [4] and `conauto` [5] for automorphism group computation.

This section first informally defines the general refinement and individualisation process to introduce concepts and build an intuition behind the algorithm. Later, the process will be formally redefined in terms of a search tree problem. This will allow me to give justification behind the claim that asymmetric graphs with a high WL dimension (defined in section 2.4.4) are difficult to solve using refinement and individualisation techniques. Drawing an analogy to the SAT algorithm DPLL reinforces the idea that searching for automorphisms in our graphs corresponds to searching for satisfying assignments to our SAT formulas. Finally, we will discuss implementation details and practical differences between graph isomorphism solvers that use the refinement and individualisation algorithm, which will be useful during the evaluation stage for explaining the behaviour of the GI solvers.

2.4.1 Colour Refinement

Warning: In this subsection, the terms "vertices", "nodes", and "cells" have distinct meanings. While vertices retain their usual meaning in a graph, nodes refer to the branch points of the search tree and cells refer to the groupings of vertices with the same colour.

For u and v to be in the same orbit, it is necessary that they have the same number of neighbours. In fact, it is necessary that the neighbours of u have the same number of neighbours as the neighbours of v. If the colouring π captures the long-range adjacencies of the vertices, then $\pi(u) \neq \pi(v)$ is a sufficient condition for $u \nsim v$ allowing the use of colours to disambiguate between vertices. The colouring of a graph partitions its vertices — vertices of the same colour are grouped into a single cell. Given an initial colouring π, colour refinement creates a new colouring π' such that for each colour in π', vertices in the same cell are adjacent to the same number of vertices in other cells. In other words, vertices of the same colour have the same adjacencies in terms of colour. This is called an *equitable colouring*.

Algorithm 1 The colour refinement algorithm.

Require: G is a coloured graph, with $\pi : V \to C$.
Ensure: Returns π', a finer ($\preceq$) equitable colouring for G.

```
1:  procedure VertexRefinement(G, π, C)
2:      π' ← Dictionary()                          // π' is the previous colouring
3:      while π ≠ π' do                            // Repeat until colouring converges
4:          for v in V do                          // Loop invariant: π ⪯ π'
5:              adj ← Dictionary()
6:              for c in C do
7:                  N ← G[v].neighbours
8:                  adj[c] ← len(filter(N, λx : π[x] = c))  // Number of coloured adjacencies
9:                  if adj not in C then
10:                     C.add(adj)
11:                 π'[v] ← C.indexOf(adj)
12:             swap(π, π')
13:     return π'
```

Algorithm 2 The equitable colouring algorithm.

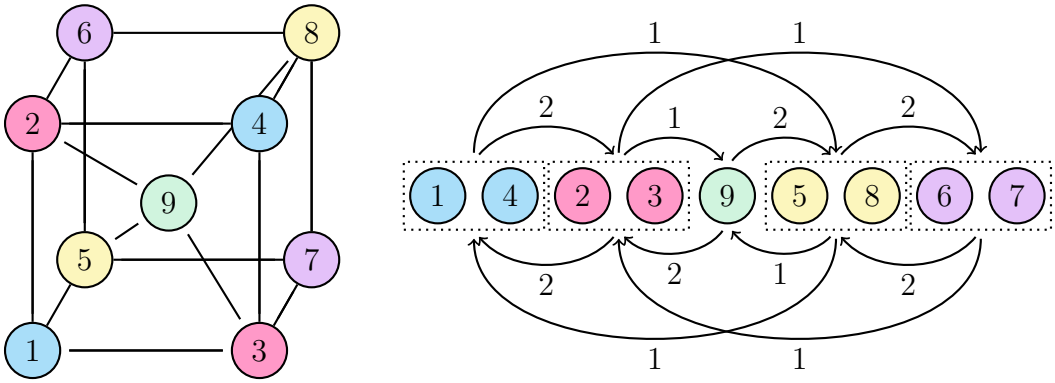
Ensure: Returns $\pi : V \to C$, an initial equitable colouring for graph G.

```
1:  procedure EquitablePartition(G)
2:      C ← OrderedSet()
3:      π ← Dictionary()
4:      for V in V do
5:          n ← len(G[v].neighbours)
6:          if n not in C then
7:              C.add(n)
8:          π[v] ← C.indexOf(n)
9:      return VertexRefinement(G, π, C)
```

Coloured Adjacencies $\mathrm{Adj}_c(v) \triangleq \{(s, v) \in E \mid \pi(s) = c\}$

Equitable Colouring $\forall v_0, v_1.\, \pi(v_0) = \pi(v_1) \implies \forall c \in C.\, |\mathrm{Adj}_c(v_0)| = |\mathrm{Adj}_c(v_1)|$

The cells of a equitable colouring form a weighted directed graph, where the weight $w > 0$ of the directed edge from cell i to cell j is equal to the number of vertices with colour j adjacent to each vertex with colour i. A cell is *non-trivially connected* if it is not connected to all cells nor none of the cells. This graph will be important later when we discuss strategies of cell selection in the search tree.



For all but a vanishing fraction $1/\sqrt[7]{n}$ of graphs of size n, colour refinement produces a graph with vertices of distinct colours, a discrete colouring [6]. However, this is not the case for graphs with a high level of local symmetry. When colouring refinement gets "stuck", the overall refinement-individualisation algorithm individualises a vertex with a non-distinct colour by giving it a new colour, then reattempts colour refinement. This process is guaranteed to terminate after $|V|$ individualisations, but in reality it will take much less than that. Since individualisation assigns a new colour to a vertex in a non-singleton cell, it can be thought of as splitting the cell.

Since a discrete colouring $\pi : \{0, \ldots, |V|\} \to \{0, \ldots, |V|\}$ is a bijective function, it is also a permutation on the vertices. The goal of the algorithm is to search through all permutations of the vertices of the graph to find all permutations that are automorphisms.

2.4.2 Vertex Individualisation

More formally, start with a search tree $\mathcal{T}(G, \pi_0)$ consisting of nodes ν, where G is the graph, π_0 is the initial equitable colouring, and ν is the sequence of vertices individualised. The root node of the tree would be the empty sequence $()$. The colour cells at each node of the search tree are computed by the colour refinement function $R(G, \pi, \nu)$, which is guaranteed to output a *finer* colouring. The notion of "finer" is formally defined as

$$\pi_1 \preceq \pi_0 \triangleq \forall v, w \in V.\, \pi_0(v) < \pi_0(w) \implies \pi_1(v) < \pi_1(w)$$

Note that the strict inequality "$<$" in the definition forces every cell in the new colouring to be a subset of a cell in the original colouring, and that the relation "$\preceq$" is reflexive and transitive. This individualisation-refinement process repeats until all cells are split into singletons.

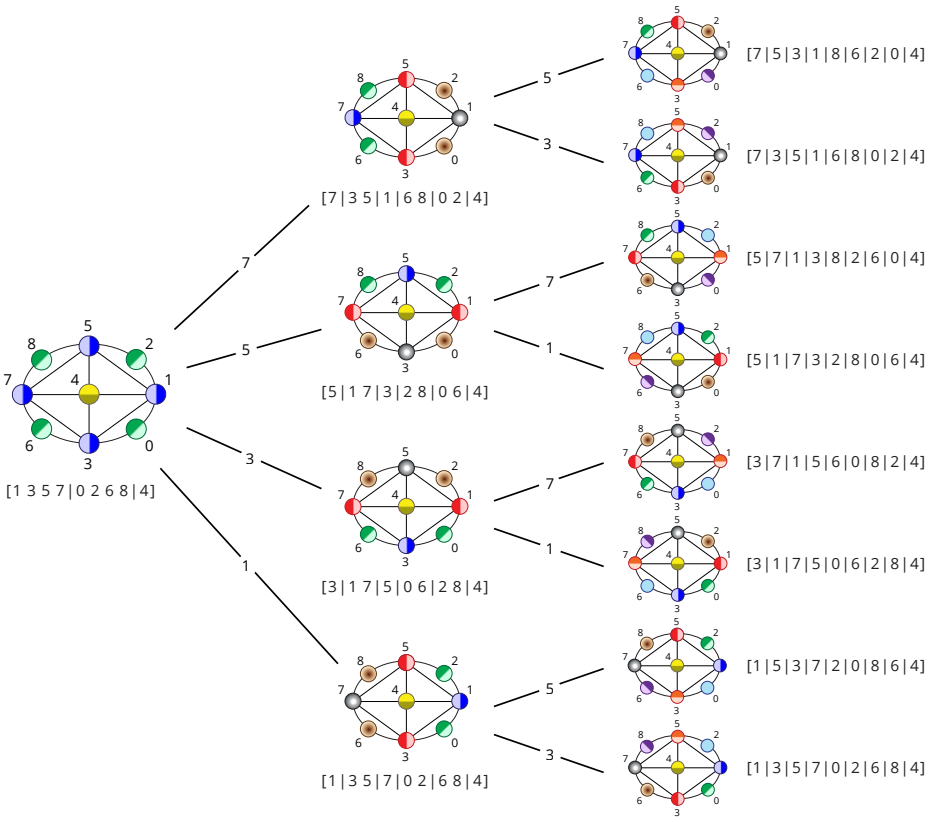


Figure 2.1: A refinement-individualisation search tree (reproduced with permission [2]).

For any node that contains a non-singleton cell, the target cell selector $T(G, \pi, \nu)$ will provide a cell of vertices that can be individualised. The children of that node are

$$\{\nu || w : w \in T(G, \pi_0, \nu)\},$$

where each choice of w leads down a different path of the search tree. At each node colour refinement is performed based on the initial individualisations, the vertices that are precoloured. In practice, the refined colouring would be produced using the colouring of the parent node. The leaves of a tree must be discrete colourings since the target cell selector cannot find any more cells to split, and correspond to a permutation.

The individualisation of a vertex will induce more vertices with unique colours during colour refinement, so not all vertices need to be individualised. Therefore the implementation of the cell selector drastically affects the shape of the search tree.

Finally, with the aid of a node invariant function ϕ, a total ordering is defined on the nodes of the search tree. For brevity, the implementation details of the node invariant function will be omitted. The leaves ν and ν' are equivalent if $\phi(G, \pi_0, \nu) = \phi(G, \pi_0, \nu')$. If this is the case, then there is a unique $\sigma \in \mathrm{Aut}(G, \pi_0)$ such that $\forall i \in |\nu|, \sigma(\nu(i)) = \nu'(i)$, namely:

$$\sigma = R(G, \pi_0, \nu')R(G, \pi_0, \nu)^{-1}.$$

$R(G, \pi_0, \nu)^{-1}$ undoes the original permutation on ν, and $R(G, \pi_0, \nu')$ applies a new one. In fact, σ is an automorphism of G. By finding all pairs of equivalent nodes in the search tree, we will find all automorphisms of the graph.

2.4.3 Canonical Form

A canonical function can be defined using the node invariant function by choosing the greatest leaf of the search tree as a permutation to the canonical form of the graph.

$$C(G, \pi_0) = max\{\phi(G, \pi_0, \nu) : \nu \text{ is a leaf of } \mathcal{T}(G, \pi_0)\}$$

2.4.4 Weisfeiler-Leman Test

The *Weisfeiler-Leman* (WL) test [7] is a generalisation of the vertex refinement algorithm. The exact details of the full Weisfeiler-Leman test are not necessary to understand its significance and will be omitted. The Weisfeiler-Leman test takes all subsets of vertices of size k out of n and assigns them a colour determined by the subgraph formed by the k vertices. k-subgraphs that are isomorphic to each other are assigned the same colour and grouped into a cell. Each of the $\binom{n}{k}$ subgraphs of size k belong to exactly one of $2^{\binom{k}{2}}$ classes of k-subgraphs.

As there are at most $O(n^k)$ colours and $O(n^k)$ iterations that require $O(n^k)$ operations, the k-dimensional Weisfeiler-Leman algorithm has complexity $O(n^{3k})$. The Weisfeiler-Leman dimension of a graph is the smallest k such that the k-dimensional Weisfeiler-Leman test results in the orbit partitioning of the vertices. Intuitively, the WL dimension measures the amount of *local symmetry* in the graph.

There was a lot of excitement around the WL algorithm as bounding the WL dimension of a graph would result in a polynomial time algorithm for the graph isomorphism problem. As an example, it has been established that the WL dimension of planar graphs is at most 3 [8]. Therefore the Weisfeiler-Leman algorithm gives a $O(n^9)$ graph isomorphism algorithm for planar graphs. Unfortunately, Cai, Fürer and Immerman published a paper in 1992 giving a construction of a family of graphs that have unbounded WL Dimension [9]. That paper also introduced the Cai-Fürer-Immerman Gadget, which will be described in the implementation section.

A graph with Weisfeiler-Leman dimension k requires at least $k + 2$ individualisation steps [3]. Thus, the Weisfeiler-Leman dimension serves as a lower bound for the depth of the search tree defined by any refinement-individualisation algorithm used by a GI solver. Therefore it is imperative that the graphs produced by my code have a high Weisfeiler-Leman dimension.

2.5 Refinement Individualisation in Practice

Although `nauty`, `Traces`, `conauto` and `bliss` all use refinement and individualisation techniques, they differ in the implementation of the target cell selector and the order of traversal of the search tree. Furthermore, `nauty` and `Traces` use automorphisms of the graph to prune the search tree.

2.5.1 Search Tree Traversal

Most solvers use depth first search (DFS) to traverse the search tree except `Traces`, which uses breadth first search (BFS). Since it is necessary to reach a leaf node to obtain automorphisms that can be used to prune the search tree, `Traces` first takes "experimental paths" to the leaves of the tree.

2.5.2 Cell Selector Strategy

The choice of cell selector strategy drastically affects the shape of the search tree, and thus on performance. `nauty` has two strategies:

1. The first smallest non-singleton cell.
2. The first maximally non-trivially connected cell.

`Traces` opts for the first largest cell which was split by the individualisation in the parent node of the search tree, with hope of reducing the height of the search tree. `bliss` has multiple cell selector modes, which is listed in the evaluation section (Figure 4.9).

2.5.3 Pruning

A major breakthrough of the graph isomorphism solver `nauty` is the use of presently discovered automorphisms to prune the search tree [2]. This drastically reduces the search space for graphs with a large automorphism group by effectively narrowing the width of the search tree. This feature, along with the lower bound on depth in section 2.4.4, suggests that **asymmetric graphs with a high Weisfeiler-Leman dimension** would perform badly on graph isomorphism solvers that use refinement and individualisation techniques.

2.6 SAT Problem

2.6.1 DPLL

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is used to solve the SAT problem, and is the foundation of most modern SAT solvers. The 3 components of DPLL are case splits, unit propagation and pure literal elimination, which are used to assign truth values to variables. When an assignment is made, conflicting terms and clauses are removed. If an empty clause is reached, then the formula is unsatisfiable; if no clauses remain, then the formula is satisfiable.

Case split: When no unit clauses or pure literals exist, DPLL chooses a variable and assigns true or false to it. To find all solutions, both assignments have to be tried.

$$\{P, Q\}, \{\neg P, Q, R, \neg S\}, \{\neg Q, \neg R\}$$
$$\{P\}, \{\neg P, R, \neg S\}$$
$$P \mapsto \mathbf{?}, Q \mapsto \boldsymbol{f}, R \mapsto \mathbf{?}, S \mapsto \mathbf{?}$$

Unit propagation: If a unit clause $\{P\}$ is found, then we must make P true.

$$\{P\}, \{\neg P, R, \neg S\}$$
$$\{R, \neg S\}$$
$$P \mapsto \boldsymbol{t}, Q \mapsto \boldsymbol{f}, R \mapsto \boldsymbol{?}, S \mapsto \boldsymbol{?}$$

Pure literal assignment: If $\neg R$ does not feature in any clause, then we can make R true.

$$\{R, \neg S\}$$
$$\{\neg S\}$$
$$P \mapsto \boldsymbol{t}, Q \mapsto \boldsymbol{f}, R \mapsto \boldsymbol{t}, S \mapsto \boldsymbol{f}$$

	SAT Problem	GI Problem
Search Goal	Satisfiability	Automorphism
Object	Boolean Formulas	Graphs
Algorithm	DPLL	Refinement-Individualisation
Local Property	k Local Consistency	k Weisfeiler-Leman Dimension
Global Property	Inconsistency	Asymmetry

Figure 2.2: A correspondence between the SAT and GI problems.

2.6.2 Analogy to Vertex Refinement

To better understand the performance of the colour refinement and vertex individualisation algorithm, we draw an analogy to the SAT algorithm DPLL.

DPLL In simple SAT formulas, unit clause propagation and pure literal elimination is sufficient to check the satisfiability of the formula. However, if there are still non-unit clauses, first do a case split on the truth of a literal, then backtrack if the assumption leads to unsatisfiability.

Vertex refinement In simple graphs, colour refinement is sufficient to reach a discrete partition to check a permutation. However, if there are still non-singleton cells, first do a case split on the vertex to colour, then backtrack if no automorphism is found.

In comparison to case splits or individualisations, clause propagations and literal eliminations or colour refinements are relatively cheap, but may not be sufficient to solve the problem. When stuck, we must assign a truth value to a literal or colour to a vertex. If we could non-deterministically choose a truth value or vertex to colour, then we have

a polynomial time algorithm. In reality we must simulate this non-determinism using a heuristic guess and backtrack if the incorrect guess is made.

Since our graph construction is based on Boolean formulas, there will be more parallels to SAT later on. A correspondence between GI and SAT is depicted in Figure 2.2.

2.6.3 Local Consistency

If there are two clauses containing complementary literals, resolution combines the clauses to create a new clause. A more restricted rule is k *bounded clause-width resolution*, where only the creation of clauses no greater than size k are permitted.

$$\frac{\{B, A_1, \dots A_n\} \quad \{\neg B, C_1, \dots C_m\}}{\{A_1, \dots A_n, C_1, \dots C_m\}} \text{ if } n + m \leq k$$

To be k *locally consistent* means the formula cannot be shown to be contradictory by deriving the empty clause through k bounded clause-width resolution, but the overall formula is ultimately unsatisfiable.

In the implementation chapter we describe a graph construction that takes a k locally consistent formula as input and outputs a graph with Weisfeiler-Leman dimension of at least k. Therefore a graph created using a complicated Boolean formula will be difficult to solve as well.

2.7 Linear Algebra

2.7.1 3-XOR

Perhaps due to the Cook-Levin theorem, the satisfiability problem is often phrased in 3-CNF (Conjunctive Normal Form) to allow convenient polynomial-time reductions to other NP-Complete problems. In our case, it is more natural to represent our Boolean formulas in the more restrictive 3-XOR form before reducing them into a graph.

In 3-CNF we have a conjunction of clauses of size 3, where each clause is of the form $(l_0 \vee l_1 \vee l_2)$. The only difference between 3-CNF and 3-XOR is that the logical OR operator $\vee$ is replaced with the exclusive OR operator $\oplus$. Another way to view 3-XOR is as a linear system of equations over $\mathbb{F}_2$, the Galois Field of 2 elements.

$\oplus$	0	1
0	0	1
1	1	0

$\otimes$	0	1
0	0	0
1	0	1

Figure 2.3: Logic table of XOR and AND.

Uniformly sampled 3-XOR formulas are k locally consistent with high probability [3], but can be checked for solutions using techniques in linear algebra. Therefore they are easy to solve using Gaussian elimination but difficult to solve using resolution.

$$\begin{array}{c} \{x_0 \oplus x_1 \oplus x_2\} \\ \{x_1 \oplus \neg x_2 \oplus x_3\} \\ \{\neg x_2 \oplus \neg x_3 \oplus \neg x_0\} \\ \{\neg x_3 \oplus x_0 \oplus \neg x_1\} \end{array} \cong \begin{array}{ll} x_0 \oplus x_1 \oplus x_2 & = 1 \\ x_1 \oplus x_2 \oplus x_3 & = 0 \\ x_2 \oplus x_3 \oplus x_0 & = 0 \\ x_3 \oplus x_0 \oplus x_1 & = 1 \end{array} \cong \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Figure 2.4: A 3-XOR system is equivalent to a matrix equation in $\mathbb{F}_2$

2.7.2 Rank-Nullity Theorem

Let V, W be vector spaces, where V is finite dimensional. Let $T : V \to W$ be a linear transformation. Then $\text{Rank}(T) + \text{Nullity}(T) = \text{Dim}(V)$.

Theorem 1. *If the $m * n$ matrix M is of full rank, where $m \geq n$, the linear system $M\mathbf{x} = \mathbf{0}$ is uniquely satisfied by $\mathbf{x} = \mathbf{0}$.*

Proof. A $m * n$ matrix represents a linear map from a n dimensional subspace to a m dimensional subspace. If the matrix has rank n then by the rank-nullity theorem $\text{Nullity}(M) = 0$. Therefore the set of solutions to the equation $M\mathbf{x} = \mathbf{0}$ form a 0 dimensional space, the point $\mathbf{x} = \mathbf{0}$. □

The rank of a matrix can be determined using Gaussian elimination, which has complexity $O(n^3)$. Moreover, obtaining a upper triangular matrix from Gaussian elimination implies that the matrix is of full rank. Note that Gaussian elimination can be used to solve 3-XOR formulas but not all Boolean formulas in general, so a polynomial time algorithm for SAT continues to elude us.

Algorithm 3 Determining whether an $\mathbb{F}_2$ matrix has full rank.

Require: A $height * width$ $\mathbb{F}_2$ matrix M.

```
1:  procedure GaussianElimination(M, width, height)
2:      for i ← 0 to min(width, height) − 1 do
3:          j ← choose_pivot(M, i)
4:          if j = −1 then return false
5:          swap(M[i], M[j])
6:          for k ← i + 1 to height − 1 do
7:              if M[k][i] = 1 then
8:                  for k ← i to width − 1 do
9:                      M[k][l] = M[k][l] ⊕ M[i][l]
10:     return true
```

2.8 Starting Point

A similar project was attempted in 2017 by Kashif Khan, an ACS student, and its experimental results were published on `https://arxiv.org/abs/1809.08154`. The

project created graphs that are based on the same theoretical sat_cfi construction proposed by my supervisor, but different techniques and heuristics were used to filter out candidate graphs.

My supervisor and I agreed that there was value in redoing the project as its experimental results were not conclusive, and the referees of the paper raised some questions that can only be answered by developing a new project from scratch.

2.9 Requirements Analysis

Performance Since Khan's project was implemented in Python, it is infeasible to generate graphs with extreme parameters due to runtime performance. Developing my construction in a high performance language would allow the creation of unsatisfiable 3-XOR Boolean formulas with a small clause-variable ratio, yielding more difficult graphs.

New Filters A good practical implementation of sat_cfi should include filters which reinforce assertions that make the graph difficult to solve. There is room to design filters that run faster and sieve out more difficult graphs.

More Solvers Khan's project only used the GI solver `Traces` to benchmark the graphs, and most graphs either took negligible time (less than a minute) to solve or the timeout value was reached. This fails to illustrate the exponential growth in solve time as the number of vertices in the graph increased, as proved in the theoretical construction [3].

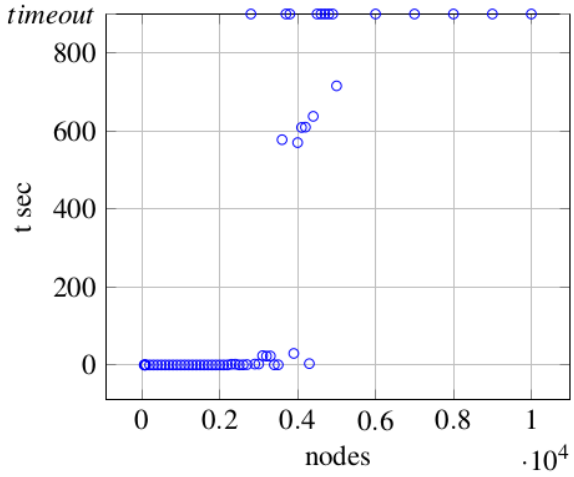


Figure 2.5: Experimental results from the previous project.

Qualitative Explanations By reading the paper by McKay and Piperno [2] on the theory and implementation of `nauty` and `Traces`, I aim to give an explanation as to why `Traces` has this step-like performance characteristic for sat_cfi graphs.

Alternative Constructions During Khan's project, a paper detailing a similar construction was published by Neuen and Schweitzer [10]. To date, there are no graphs generated using the reg_cfi construction publicly available on the internet, nor is there an implementation of the construction. By creating my own implementation of the construction through reading pseudocode and understanding the mathematical proofs from the paper, I could recreate graphs that can be compared against graphs from our construction.

Scientific Method In Khan's project, the solve times recorded were obtained by subtracting the timestamps between the entry and exit of the solve functions. This does not take into account the time lost from context switches and other kernel processes. This systematic error can be ameliorated by only considering the time actually spent on solving the graph and taking repeats across multiple solves. Due to the non-deterministic nature of the construction, it would also be interesting to create multiple graphs using the same parameters to see the intra-parameter variation in solve time.

Implementation	Priority
Implement sat_cfi construction	high
Design and prove correctness of filters	high
Create testing suite for graph generation and evaluation	high
Implement reg_cfi construction	medium
Extend SAT solver to perform bounded clause-width resolution	low
Evaluation	
Evaluate correctness of sat_cfi and reg_cfi construction	high
Investigate distribution of sat_cfi graphs	high
Investigate distribution of reg_cfi graphs	high
Reproduce and explain Khan's results	medium
Investigate clause-variable ratio	medium
Investigate efficacy of heuristic filter	low

Figure 2.6: Tasks to be achieved in the project, in order of priority.

2.10 Summary

Through studying the refinement-individualisation algorithm, we conjecture that asymmetric graphs with high Weisfeiler-Leman dimension will trigger the worst-case behaviour of graph isomorphism solvers. By drawing analogy to DPLL, I discovered a previously unknown connection between the fundamental algorithms used to solve SAT and GI. In the implementation chapter, I describe the sat_cfi construction which converts a difficult SAT formula to check for satisfiability into a difficult graph to check for isomorphism. By looking at the shortcomings of a previous ACS project, I established the targets of my project and conducted requirements analysis.

Chapter 3

Implementation

This chapter consists of three parts. Sections 3.1 and 3.2 describe and justify design decisions made before the start of the project. The sections 3.3 to 3.5 describe the theory and implementation of two graph constructions: the sat_cfi construction proposed by Anuj Dawar [3], and the reg_cfi construction proposed by David Neuen and Pascal Schweitzer [10]. Finally, section 3.6 onwards describe the code developed for testing the correctness of the graphs. Our main focus is the sat_cfi pipeline which consists of a base graph generator, a set of filters and the multipede construction. We will prove that the assertions set by the filters, when combined, result in a pipeline that converts difficult Boolean formulas to check for satisfiability into difficult graphs to compare for isomorphism. My contribution is constructing a new filter with superior performance.

3.1 Language Choice

The project was principally developed in C++ as a faster implementation allows for the creation of larger graphs with more extreme parameters. Bash scripts were necessary to schedule jobs on the Cambridge High Performance Cluster. Bash was also used to interface with command line tools to create a pipeline of filters that ensures the asymmetry and high WL dimension of the graphs. Finally, the experiments and data analysis were conducted in Python. A repository review can be found in Appendix A.4.

3.2 Graph Representation

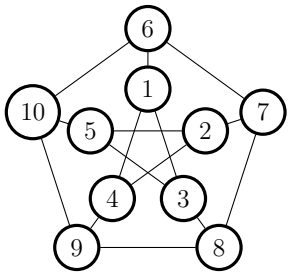
This section addresses an important implementation decision — the data structure used to represent the graphs. Taking into consideration the file formats to be supported and the algorithms to be implemented, the data structure chosen should minimise the time taken to perform required memory access patterns while being as compact as possible.

3.2.1 File Format

DIMACS is a common standard for representing graphs used by `bliss` and `conauto`. By listing edges of the graph, DIMACS is akin to our definition of a graph (V, E).

```
DIMACS:
p edge 10 15    e 3 8
e 1 3           e 4 9
e 1 4           e 5 10
e 1 6           e 6 7
e 2 4           e 7 8
e 2 5           e 8 9
e 2 7           e 9 10
e 3 5           e 10 6
```

```
DREADNAUT:
$=1 n=10 g
1: 3 4 6
2: 4 5 7
3: 5 8
7: 6 8
9: 4 8 10
10:5 6
```



`dreadnaut` is a command line tool which interfaces with `nauty` and `Traces`. The format used by `dreadnaut` is described in its documentation [11]. By storing the graph as an adjacency list, the `dreadnaut` representation is more compact than DIMACS.

3.2.2 Data Structure

In C++ terms, DIMACS resembles `unordered_set<Edge>` and `dreadnaut` resembles a multimap, or `unordered_map<Vertex, unordered_set<Vertex>>`. The latter structure, with $O(1)$ amortised access time, can export both formats quickly, while the former would take much longer as it does not support random access to the edges.

The graph class implemented has a constructor which accepts `vector<Edge>`, but stores the graph internally as `unordered_map<Vertex, unordered_set<Edge>>`. This design decision will give the algorithms to be implemented quick access to adjacency information of the graph, without restricting the data structure used as an intermediate representation of the new graph.

3.3 Multipede Construction

This section describes implementation of the multipede construction, a graph transformation used by the sat_cfi and reg_cfi constructions to create asymmetric graphs with high WL dimension. The gadget used by multipede construction, the Cai-Fürer-Immerman (CFI) gadget X_3, is a graph consisting of 4 inner nodes m_i, and 3 pairs of outer nodes a_i, b_i. Each inner vertex m_i is connected to an even number of a vertices (0 or 2), and an odd number of b vertices (1 or 3). In Figure 3.2, the j^{th} bit in the label of m_i indicates whether m_i is connected to a_j or b_j, so all labels have an even number of "1"s.

$$A = \{a_i \mid 1 \leq i \leq k\} \qquad S = \{s \in \mathcal{P}(A) \mid |s| \text{ is even}\}$$
$$B = \{b_i \mid 1 \leq i \leq k\} \qquad V = A \cup B \cup M$$
$$M = \{m_s \mid s \in S\} \qquad E = \{(m_s, a_j) \mid a_j \in s\} \cup \{(m_s, b_j) \mid a_j \notin s\}$$

Figure 3.1: A mathematical description of $X_d = (V, E)$.

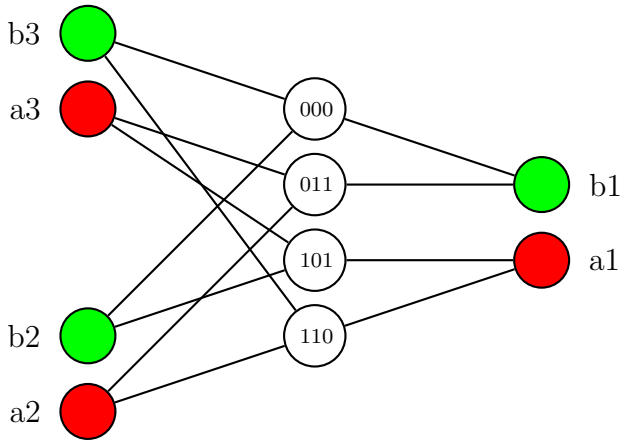


Figure 3.2: The Cai Fürer Immerman Gadget X_3.

By considering the number of adjacencies for each vertex, it can be seen that no automorphism of this gadget graph can map an inner vertex to an outer vertex, or vice versa. In fact, a characteristic feature of the CFI gadget is that all of its automorphisms swap an even number of outer node pairs.

Theorem 2. *If the pairs of outer vertices of the CFI gadget X_3 are fixed by colouring a_i and b_i with colour i, all 2^{3-1} automorphisms on this coloured graph can be uniquely identified by whether a_i and b_i have been swapped.*

Proof. If vertices a_i, b_i are the only vertices with colour i, an isomorphism from a graph to itself must either swap the vertices or leave them fixed to preserve colour. Furthermore, if an odd number of outer vertex pairs are swapped, then the inner vertices would be connected to an odd number of a vertices, and hence cannot be an automorphism. Considering the 4 ways to perform an even number of swaps on the 3 outer vertex pairs, there is a unique permutation on the inner vertices m that effects the overall permutation to be an automorphism. □

These observations allow us to assign some semantic meaning to an automorphism. In the original paper [9], Cai Fürer and Immerman replaced vertices with degree d with X_d to create a family of graphs with unbounded Weisfeiler-Leman Dimension; that is, for every $k > 0$, there exists a graph such that its orbit partition cannot be determined by the k dimensional Weisfeiler-Leman test. The argument is based on the expressiveness of first order logic with k variables, and how the k dimensional WL test does not yield a rich enough language to describe graphs uniquely.

For this project, the shrunken multipede construction is applied on bipartite graphs $(V \cup W, E)$, where each vertex in W is connected to 3 vertices in V. Instead of replacing every vertex with a CFI gadget X_d, vertices in W and V are replaced by the inner and outer vertices of the CFI Gadget X_3 respectively. This optimisation reduces the number of vertices in the final graph while preserving its hardness by eliminating automorphisms.

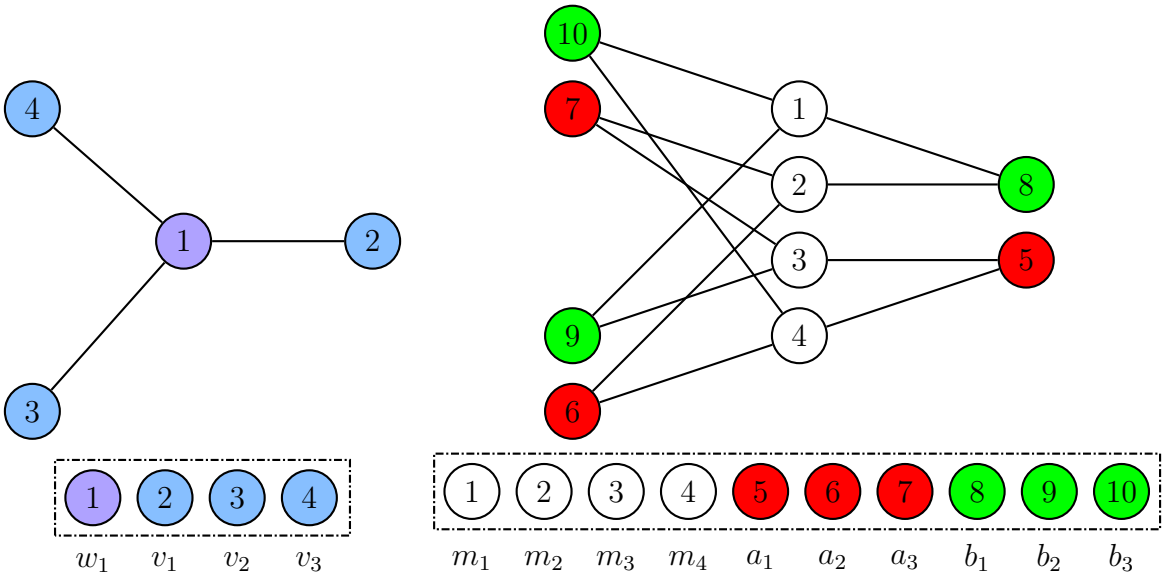


Figure 3.3: The application of the CFI gadget shows the mapping between the vertices.

Internal Layout

I initially considered creating subclasses for the bipartite base and multipede graphs, but decided that it was not necessary as the vertex type can be determined by its index.

By the CFI gadget, each vertex in W is expanded to 4 vertices in M and each vertex in V is expanded to a vertex in each of A and B, so $|M| = 4|W|$ and $|A| = |B| = |V|$. In the base graph, vertices in W were numbered $1, \dots |W|$ and vertices in V were numbered $|W| + 1, \dots |V| + |W|$. In the final multipede graph, inner vertices in M were numbered $1, \dots |M|$, the outer vertices in A and B are labelled $|M| + 1, \dots |M| + |A|$ and $|M| + |A| + 1, \dots |M| + |A| + |B|$ respectively.

The task of implementing the multipede construction is now reduced to finding a numerical mapping of the indices that corresponds to the logical mapping of W to $\mathcal{P}(M)$ and V to $\mathcal{P}(A \cup B)$.

Numerical Mapping	Logical Mapping
$f(w) = \{4(w-1)+i) \mid i = 1,2,3,4\}$	$w_j \mapsto \{m_{j_i} \mid i = 1,2,3,4\}$
$f(v) = \{4\|W\| + v\} \cup \{4\|W\| + \|V\| + v\}$	$v_j \mapsto \{a_j, b_j\}$

The edges need to be added too, according to the CFI gadget. In Figure 3.3, vertex 1 in the base graph is split into outer vertices 1, 2, 3 and 4 in the multipede graph, and vertex 3 in the base graph is split into outer vertices 6 and 9 in the multipede graph.

The sat_cfi and reg_cfi graphs both use the multipede construction and only differ in their choice of base graph. After describing the implementation of multipede construction, we will discuss the generation of these base graphs.

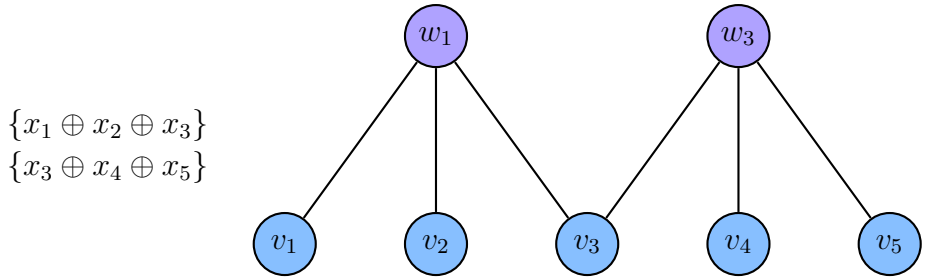


Figure 3.4: Base graph construction of sat_cfi from a 3-XOR Formula.

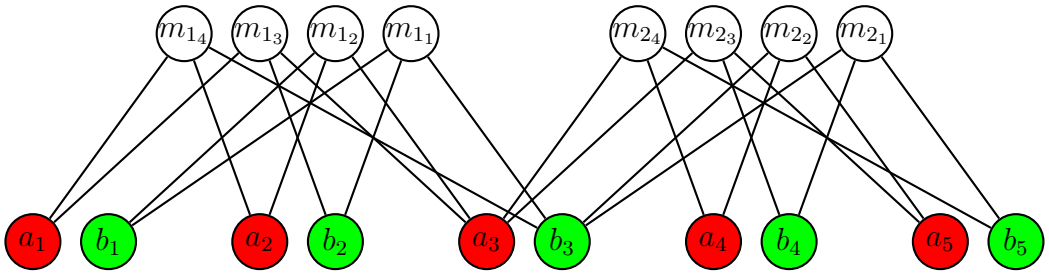


Figure 3.5: Multipede graph constructed from base graph in Figure 3.4.

3.4 The sat_cfi Construction

In the preparation section we discussed how certain Boolean formulas can be written as a conjunction of 3-XOR clauses. Now we will focus on 3-XOR formulas containing only pure variables and unique clauses which can be represented as a bipartite graph.

In the sat_cfi construction, the base graph $B_s(V, W)$ is derived from a given 3-XOR formula with variables V and clauses W as follows: the vertices of $B_s(V, W)$ are $V \cup W$. The edges of $B_s(V, W)$ are $(w, v) \in W \times V$, where v is a variable in the 3-XOR clause w.

Since there are 3 variables in each 3-XOR clause, each clause vertex is connected to 3 variable vertices. Therefore the graph derived from a 3-XOR Boolean formula satisfies preconditions for the multipede construction. The multipede graph of sat_cfi is obtained after applying the multipede construction on to the base graph.

Theorem 3. *If the base graph has no automorphisms, and the 3-XOR system is uniquely satisfiable, then the multipede graph has no automorphisms as well, making it asymmetric.*

Proof. Suppose there exists an automorphism in the multipede graph. Since the base graph does not contain any non-trivial automorphisms, the automorphism must only permute the vertices within each CFI gadget. By Theorem 2, the outer vertices of each CFI gadget must be swapped an even number of times. Similarly, to maintain the truth value of a clause $\{v_i \oplus v_j \oplus v_k\}$ an even number of variables have to be set to false. We define a correspondence between the automorphism and the assignment: if the automorphism swaps vertices a_{v_i} and b_{v_i} then the assignment maps the variable v_i to false. Since the 3-XOR Boolean formula is uniquely satisfiable, the only solution is to assign all variables to true, which corresponds to leaving all the outer vertices of the CFI gadgets unswapped. Therefore the only automorphism of the final graph is $id_V(v) = v$, the identity permutation. □

Theorem 4. *The sat_cfi construction takes a Boolean formula that is k locally consistent to a graph with Weisfeiler-Leman dimension of at least k. [3]*

These two theorems each reveal a connection between the Boolean formulas and the graphs. Firstly, the sat_cfi construction transfers the inconsistency of the formula into global asymmetry of the graph. Secondly, the construction transfers the high local consistency of the formula into high local symmetry in the graph. In other words, The sat_cfi construction transfers the difficulty of the 3-XOR formula to the multipede graph. Just as the k bounded clause-width resolution is unable to derive an empty clause from the formula through a peephole of k variables, the k dimensional Weisfeiler Leman test is unable to find the discrete orbit partition of the graph using a peephole of k vertices.

Therefore a good practical implementation of sat_cfi should aim to find highly locally consistent 3-XOR systems before applying the multipede construction.

In the theoretical construction, an asymmetry gadget is added to each pair of consecutive variable nodes so no automorphism can permute them. In the practical setup the use of the asymmetry gadget may be circumvented by using an asymmetric base graph.

3.4.1 Formula Generation

The sat_cfi construction requires a randomly generated 3-XOR formula to generate the graph. Since the formulas are restricted to have non-negated terms, constructing base graph $B_s(V, W)$ requires $m = |W|$ triples with values from 1 to $n = |V|$. Combinatorial results show that sampling the triples uniformly for a sufficiently large formula will yield an unsatisfiable, locally consistent formula and an asymmetric base graph with high probability, which is sufficient criteria for the final graph to be asymmetric with a high WL Dimension [3].

Great care must be taken in ensuring no bias was introduced during the sampling process. Each triple should be independently sampled before being passed through the filters. However, the sampling process could be modified such that graphs that cannot pass through the filters would not be chosen. First of all, the multipede construction checks that each vertex in W has 3 neighbours, as required by the CFI gadget. To satisfy this condition the variables in a triple are necessarily unique. Moreover, the clauses in a formula also need to be unique — if the two clauses share the same variables, the clause vertices in the graph can be swapped to yield the same graph. This is a non-trivial automorphism.

```
std::unordered_set<Triple> outTriples;
while (outTriples.size() < m) {
    outTriples.insert(generate_random_triple(n));
}
```

Storing the triples in a set ensures that all clauses in the formula will be unique. This optimisation, although simple, is crucial for performance. By the birthday problem, the probability of collision is close to 1/2 when the number of clauses $m \geq \sqrt{\binom{n}{3}} \approx n\sqrt{n}$. By preventing these collisions, we avoid needless restarts of the entire pipeline due to non-trivial automorphisms.

3.4.2 Filters

This section describes the filters implemented for our sat_cfi construction to assert properties of the base graph and the Boolean formula that are sufficient preconditions for the multipede construction to produce an asymmetric graph with high WL dimension.

As discussed in the theoretical construction, a good practical implementation of sat_cfi should aim to find highly locally consistent 3-XOR systems before applying the multipede construction to achieve graphs with high WL dimension. To prevent `nauty` or `Traces` from pruning the search tree, the implementation should also produce graphs without non-trivial automorphisms by using only asymmetric base graphs and uniquely satisfiable 3-XOR formulas.

Asymmetry of base graph

Since a base graph is smaller and has a lower WL dimension compared to its final graph, it can be directly searched for automorphisms by a graph isomorphism solver. I choose `Traces` for the task, passing the graph into `dreadnaut` with a pipe.

An asymmetric graph contains only the identity permutation in its automorphism group $\mathrm{Aut}(G)$, which induces the discrete orbit partition, $u \sim v \iff id_V(u) = u = v$. Therefore, the filter need only search the `stdout` of `dreadnaut` for the strings `grpsize=1` or `orbitsize=[V]` where `[V]` is the number of vertices in the graph, instead of listing all automorphisms of the graph, a potentially IO-intensive operation.

Unique satisfiability of 3-XOR system

It is overkill to use a full-fledged SAT solver to check the unique satisfiability of a 3-XOR system, as its runtime is exponential in the worst case. Rewriting the conjunction of 3-XOR clauses as a matrix equation in $\mathbb{F}_2$, by theorem 1, the unique satisfiability of the linear system can be determined by checking if the matrix has full rank using Gaussian elimination.

Developing my own Gaussian elimination module allows the utilisation of domain-specific optimisations. The first optimisation takes advantage of the fact that the filter is not concerned with computing the rank of the matrix, but only testing if the $m * n$ matrix has full rank $\min(m, n)$. Therefore it is sufficient to check if Gaussian elimination could result in a upper triangular matrix, a diagonal of 1's.

The second optimisation improves space and time consumption by exploiting the field of the linear system. Entries of the matrix can only take the values 0 or 1, and can be represented by the `bool` type. Moreover, representing rows of the matrix with `vector<bool>` potentially saves space. According to the C++ specification [12], the internal representation of `vector<bool>` is implementation defined and permits the storage of `bool`s without padding by coalescing the bits.

Similarly, a third optimisation is possible: Because XOR is equivalent to addition modulo 2, when eliminating non-zero entries in the selected column using the pivot row in a $\mathbb{F}_2$ matrix, it is sufficient to XOR the rows together instead of subtracting a multiple of the pivot row.

This allows a further optimisation: if the compiler does indeed use a packed representation of `vector<bool>`, entries stored in the same word can be XORed in a single instruction.

Instead of writing convoluted code to improve performance, writing straightforward code increases the likelihood that the compiler unwinds the loop and performs the optimisation correctly. This is an instance of the phase ordering problem: if the compiler looks for clever bitwise tricks before loop unwinding, this optimisation will be overlooked.

High local consistency of 3-XOR system

A uniquely satisfiable 3-XOR system containing only pure variables has only one consistent assignment, $v_i \mapsto f$. By excluding the all false solution through the addition of the clause $\bigvee_{i=1}^{n} v_i$ to the 3-XOR formula, the uniquely satisfiable 3-XOR system is reduced to a unsatisfiable system. Note that the converse also holds, as the exclusion of one solution will not make a Boolean formula with multiple solutions unsatisfiable.

A method of checking if a formula is k locally consistent is to modify a SAT solver to perform k bounded clause-width resolution. After discussing with my supervisor who then spoke to two SAT solver experts, we decided this extension of a SAT solver is out of the scope of this Part II project and is substantial enough to be a standalone Part III project.

Instead, a heuristic filter is used — assuming a highly locally consistent formula is harder to solve using normal resolution, the filter compares the runtime between Gaussian elimination and resolution. Since Gaussian elimination has a worst case complexity of $O(n^3)$, its runtime is bounded above regardless the increase in difficulty of the 3-XOR formula. Therefore, if the formula takes much longer to solve using resolution than in Gaussian elimination, we deem it a good candidate for high local consistency.

The open-source SAT solver CryptoMiniSAT, with the option of utilising Gaussian elimination on top of resolution, was well suited for this task. CryptoMiniSAT is a fork of MiniSAT, which accepts Boolean formulas expressed in Conjunctive Normal Form (CNF) in DIMACS format. In order to accept XOR clauses as well, the designer of CryptoMiniSAT extended the DIMACS format. Unfortunately, there is no documentation on the extended format. Instead, there is a post about it on the creator's personal blog. To make sure the XOR format does work, regression testing was performed against the formula in 3-CNF.

The effectiveness of this heuristic will be investigated in the evaluation chapter.

3.4.3 Pipeline

The practical setup used to construct the sat_cfi graphs forms a pipeline. In the beginning a random set of m triples from 1 to n is provided to the base graph constructor, which produces a 3-XOR formula and the base graph. The base graph is checked for asymmetry, while the formula is checked for unsatisfiability and high local consistency. If all 3 criteria are met, then the multipede construction is applied to the base graph to produce the final graph.

$$\{x_0 \oplus x_1 \oplus x_2\} \rightarrow \begin{array}{lcl} (x_0 \wedge x_1 \wedge x_2) & \vee & \\ (x_0 \wedge \neg x_1 \wedge \neg x_2) & \vee & \\ (\neg x_0 \wedge x_1 \wedge \neg x_2) & \vee & \\ (\neg x_0 \wedge \neg x_1 \wedge x_2) & \vee & \\ (x_0 \wedge \neg x_1 \wedge \neg x_2) & \vee & \\ (\neg x_0 \wedge \neg x_1 \wedge x_2) & & \end{array} \rightarrow \begin{array}{c} \{\neg x_0, \neg x_1, \neg x_2\} \\ \{\neg x_0, x_1, x_2\} \\ \{x_0, \neg x_1, x_2\} \\ \{x_0, x_1, \neg x_2\} \\ \{\neg x_0, x_1, x_2\} \\ \{x_0, x_1, \neg x_2\} \end{array}$$

Figure 3.6: Conversion of a single 3-XOR clause into CNF using De Morgan's Law.

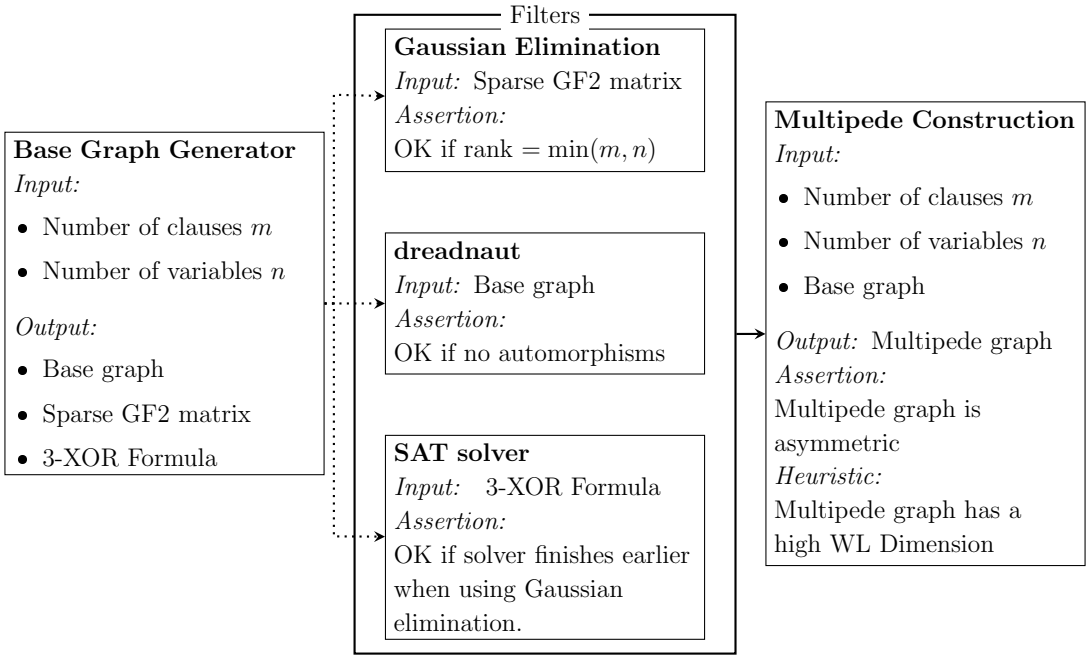


Figure 3.7: The sat_cfi Pipeline

The filters assert properties that, when combined, result in asymmetric graphs with a high WL dimension. Therefore, the graphs produced by this pipeline should on average take longer for a GI solver to solve than the graphs produced without the filters. This hypothesis will be assessed in the evaluation chapter.

Since there are no dependencies between the filters, there is no need to apply the filters in order. Although the initial plan was to use Python to assemble the pipeline, true parallelism in Python is tricky due to its Global Interpreter Lock [13], a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. Instead, the UNIX tool Parallel [14] was used to run the filters simultaneously. If any of the processes launched by Parallel returns a non-zero exit code, then Parallel kills all of its child processes and returns the exit code of the failed process. Therefore, by designing the filters to return a unique non-zero exit code if the graph fails its criterion, the bash script can use the exit code to identify which filters the graph has failed to pass. By short-circuiting, the time required to apply the filters has been reduced from $t_{\text{filter1}} + t_{\text{filter2}} + t_{\text{filter3}}$ to $\max(t_{\text{filter1}}, t_{\text{filter2}}, t_{\text{filter3}})$.

3.5 The reg_cfi Construction

A graph is l-regular if each vertex has l neighbours. It follows from the definition that a monochrome colouring of any regular graph is already equitable. Furthermore, it has been shown experimentally that a high number of individualisations is required to reach a discrete colouring [2]. However, since 3-regular graphs tend to be highly symmetric, their search trees can be pruned using their automorphisms.

The construction devised by Neuen and Schweitzer [10] uses a base graph that represents a random permutation of a 3-regular graph, which is unlikely to be symmetric.

Precondition: G is a 3-regular graph, σ is permutation on the edges of G.

1. Represent each edge as another vertex $e_{(u,v)}$.
2. Insert the edge vertices between each edge by replacing $u - v$ with $u - e_{(u,v)} - v$.
3. Generate a random permutation σ on the edge vertices.
4. Duplicate G and replace $u' - v'$ with $u' - e_{\sigma((u,v))} - v'$.

Since the vertices of G have 3 neighbours, in the base graph $B_r(G, \sigma)$ each of the "vertex" vertices are connected to 3 "edge" vertices. This allows the multipede construction to transform the "vertex" vertices and the "edge" vertices to the inner and outer vertices of the CFI gadget respectively.

The multipede graph of reg_cfi is obtained after applying the multipede construction on to the base graph.

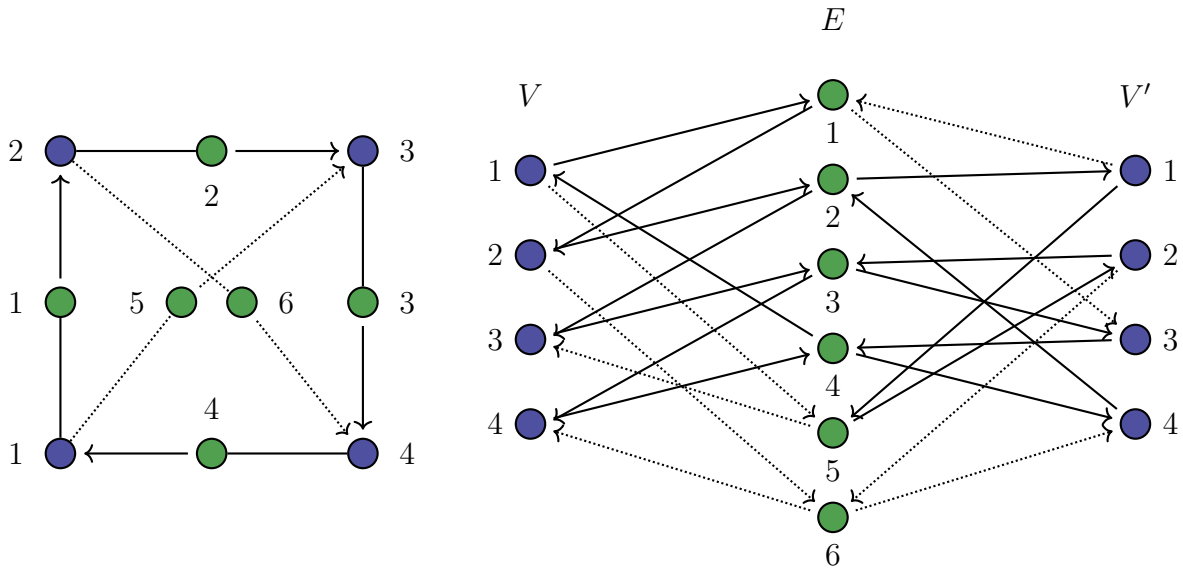


Figure 3.8: Left: 3-regular graph G of size 4 after step 2.
Right: Base graph generation $B_r(G, \sigma)$ using permutation $\sigma = (5\ 3\ 4\ 2\ 1\ 6)$.
The blue nodes are "vertex" vertices and the green nodes are "edge" vertices. Note how the duplicated "vertices" connect to the "edges" in a permutated fashion.

3.5.1 Implementation

For simplicity, the 3-regular graph was generated in a deterministic fashion:

$$
\begin{aligned}
G_n &= (V, E) \\
V &= \{v_i \mid 1 \leq v \leq n\} \\
E &= \{(v_i, v_{i+1}) \mid 1 \leq v < n\} \cup (v_n, v_1) \cup \{(v_i, v_{i+n/2}) \mid 1 \leq v \leq n/2\}
\end{aligned}
$$

For the reg_cfi base graph, it was easier to work with an adjacency list, which is accepted by the base graph constructor. This design decision made the reg_cfi construction straightforward to implement. The complete implementation is available on my `GitHub`.

```
// generate base graph
std::vector<Edge> reg_edges;
for (int i = 1; i < n; i++) {
    reg_edges.push_back({i, i+1});
}
reg_edges.push_back({n, 1});
for (int i = 1; i <= n/2; i++) {
    reg_edges.push_back({i, i+n/2});
}

int e_size = 3*n/2;
// generate permutation
std::vector<int> sigma(e_size);
for (int i = 0; i < e_size; i++) {
    sigma[i] = i;
}
std::random_shuffle(sigma.begin(), sigma.end());

// connect V with E
std::unordered_set<Edge> base_edges;
for (int i = 0; i < e_size; i++) {
    Edge edge = reg_edges[i];
    for (Vertex v : edge.vertices()) {
        base_edges.insert({v.get_id(), i+1 + 2*n});
    }
}
// connect V' with E
for (int i = 0; i < e_size; i++) {
    Edge edge = reg_edges[sigma[i]];
    for (Vertex v : edge.vertices()) {
        base_edges.insert({v.get_id() + n, i+1 + 2*n});
    }
}

Graph base_graph(n + n + e_size, base_edges);
```

Figure 3.9: The base graph construction of reg_cfi.

3.6 Graph Visualisation

A better understanding of the properties of the graph and the identification of bugs could be achieved through the visualisation and manipulation of the graphs. The graph class supports the option of exporting graphs in graphml format so that they can be visualised in Gephi [15], a popular open-source graph analysis and visualisation program. The graphml representation is richer than DIMACS or `dreadnaut` and permits the description of the vertex and edge properties such as the colour, size and text label in XML.

```
<?xml version="1.0" encoding="utf-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi=...>
    <key id="color" for="node" attr.name="color" attr.type="string"/>
    <graph edgedefault="undirected">
        <node id="1"><data key="color">black</data></node>
        ...
        <edge source="17" target="21"/>
    </graph>
</graphml>
```

The regular arrangement of the vertices allows the properties to be dynamically computed. The graphml export method takes a lambda function as argument, which returns a colour based on the label of the vertex. The colour is included as an attribute on the `boost::property_tree` which is later exported to XML.

```
// [=] captures m, n using Call By Value
auto mult_sat_cfi = [=](int i) {
    if (i <= 4*m) {
        return WHITE;
    } else if (i <= 4*m + n) {
        return RED;
    } else { // 4*m + n < i <= 4*m + 2*n
        return GREEN;
    }
}
```

Visualising the generated graphs helped identify logical errors. During the testing stage, hovering on one of the inner nodes of the graph in Gephi revealed that it was connected to an odd number of red nodes, which is impossible for the CFI gadget. Further investigation revealed that the lookup table for the CFI gadget was incorrect.

3.7 Generating Non-Isomorphic Graphs

On the grounds of theoretical equivalence and the practical algorithm used by GI solvers, throughout the project we use GI solvers to search for automorphisms of a single graph instead of comparing pairs of graphs for isomorphism, the purpose of the project. For the sake of completeness, here is a demonstration of how to generate pairs of graphs that are difficult to compare for isomorphism: given a graph generated using sat_cfi or reg_cfi, individualise a pair of outer vertices a_i, b_i by connecting one of them with a new

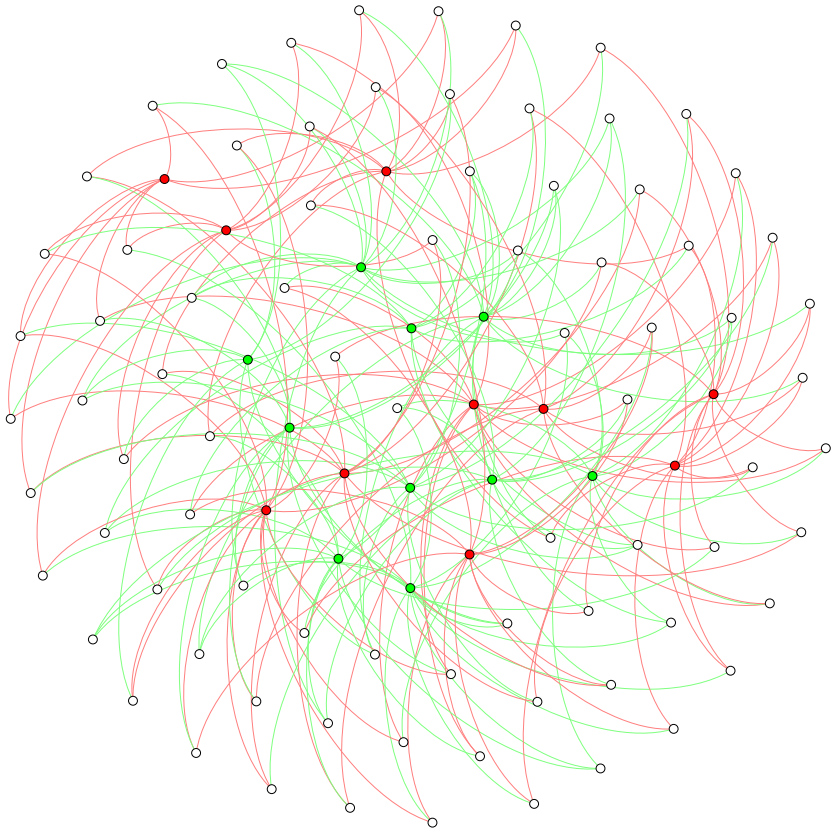


Figure 3.10: A sat_cfi graph with parameters $m = 30, n = 15$, visualised using Gephi. Each inner vertex (white) is connected to an even number of red vertices.

vertex d. Discerning whether d is attached to the a_i vertex or b_i vertex will require the full k dimensional Weisfeiler-Leman test. Against an adversary, select two graphs randomly with replacement to make the probability of the graphs being isomorphic $\frac{1}{2}$. For good measure, the vertices of both graphs are randomly permuted. This procedure can be performed directly using `dreadnaut` (see Appendix A.1).

3.8 Summary

This chapter introduces the multipede construction that transforms a base graph into an asymmetric graph with high WL dimension. It then describes the definition and implementation of the sat_cfi and reg_cfi base graphs. For the sat_cfi construction, I proved sufficient conditions to achieve an asymmetric multipede graph, then implemented a pipeline of filters that asserted these conditions. In particular, my innovative use of Gaussian elimination on the graph's underlying SAT formula improved the complexity of the filter from exponential to cubic. Using the proofs and pseudocode from the original paper, this project contains the world's first publicly available implementation of the reg_cfi construction [10], which serves as a benchmark against the sat_cfi construction [3]. Finally, this chapter demonstrates how to construct pairs of graphs that are hard to check for isomorphism using the sat_cfi and reg_cfi constructions.

Chapter 4

Evaluation

This chapter first describes the testing suite and experimental setup used to generate and evaluate the graphs (sections 4.1 to 4.2). After verifying that our sat_cfi and reg_cfi implementation matches the specification (section 4.3), previously unknown properties of the sat_cfi and reg_cfi distributions are presented (sections 4.4 to 4.5). Afterwards, the relationship between the clause-variable ratio of the SAT formula and the difficulty of the graph is thoroughly investigated, including further statistical analysis on the heuristic filter developed in another previous project that locates highly locally-consistent formulas (section 4.6). Finally, it describes the publication of graphs and experimental results on `arXiv` and `http://pallini.di.uniroma1.it/Graphs.html`, the official library of benchmark graphs (section 4.7).

4.1 Graph Generation

To make the experimental results easily reproducible, I created Python scripts to generate collections of graphs and measured their performance. Those who wish to verify my results need only execute the scripts, available on my `GitHub`, with the same arguments.

When producing graphs in the evaluation chapter, it is more natural to decide on a desirable size of the graph rather than the parameters of the construction. In the sat_cfi construction, instead of specifying the number of clauses m and variables n in the 3-XOR formula used, it is equivalent to specifying the size of the final graph and the ratio between the clauses and variables. As the multipede construction converts the bipartite base graph $(V \cup W, E)$ by expanding each clause vertex in W into 4 vertices and each variable vertex in V to a vertex in A and B each, $Size = 4|W| + 2|V|$. Substitute in $Ratio = \frac{|W|}{|V|}$ to get $|V| = \frac{Size}{Ratio*4+2}$, $|W| = Ratio * |V|$.

In the reg_cfi construction, a 3-regular graph with v vertices has $\frac{3v}{2}$ edges, so the base graph has $2v$ "vertex" vertices in V and $\frac{3v}{2}$ "edge" vertices in E. After applying the multipede construction there should be $4 * 3v + 2 * \frac{3v}{2} = 11v$ vertices in the final graph.

There are some combinatorial constraints on the parameters: for uniqueness, the number of clauses in sat_cfi cannot exceed $\binom{n}{3}$, and the 3-regular graph used in reg_cfi must have an even number of vertices. The input validation for the scripts is conducted using the constraints before the graph generation stage.

Name	Construction	Filters	Graph Size	Clause-Variable Ratio	Quantity
Collection A	sat_cfi	✓	6000	2:1	500
Collection B1	sat_cfi	✓	4400	3:2	500
Collection B2	sat_cfi	✗	4400	3:2	500
Collection C	sat_cfi	✓	100–8000	2:1	185
Collection D	reg_cfi	—	660–4400	—	120
Collection E	sat_cfi	✓	4500	1.5-4.0	55

Figure 4.1: A description of the generated graphs.

Feature	Description
Host	Cambridge HPC
OS	Scientific Linux
CPU	2.6GHz 16-core
Memory	6GB / 12 GB
Storage	40 GB SSD
Filesystem	ZFS

Figure 4.2: A technical description of the testing environment.

4.2 Testing Environment

The experiments were conducted on the Cambridge High Performance Cluster. For consistency, all time-sensitive jobs were submitted to the `skylake` and `skylake-himem` nodes. Each skylake node contains 2 Intel Xeon Gold 6142 CPUs, and each task in a job is assigned 1 CPU with 6GB of RAM. For `skylake-himem` cores, 12GB of RAM is used.

Jobs are submitted using the SLURM Workload Manager. Any changes made to the filesystem will be visible after the job's completion, and the stdout of the job is written to `slurm-[job_id].out`. Each job was allocated a time limit of 8 hours.

In mid-February, jobs on the HPC were severely congested in the queue for over two weeks due to heavy system maintenance, delaying the experiments. Instead of waiting for the experiments to complete, I conducted preliminary experiments on a local machine; their results were used to plan for large scale experiments on the cluster.

4.3 Testing Asymmetry of Graphs

In the implementation chapter we proved that graphs produced by the sat_cfi pipeline will be asymmetric. This was verified experimentally through computing the automorphism groups of over 10,000 graphs using `nauty`. As expected, all the graphs produced with the filters were asymmetric. Although the reg_cfi graphs are only proved to be asymmetric with high probability [10], all graphs generated using our reg_cfi construction were also asymmetric.

4.4 Distribution of sat_cfi Graphs

Compute time is a finite resource. Sampling graphs from sat_cfi's distribution will yield graphs of varying difficulty, including the most difficult graphs. However, sieving out the most difficult graphs by directly running them on `nauty` or `Traces` is extremely time consuming. Applying filters that run faster than the GI solvers will eliminate most candidates without timing them, allowing us to sample more graphs in the same period of time. Figure 4.3 shows the distribution in runtime of sat_cfi graphs generated by our pipeline, including the asymmetry filters. Note the long-tailed distribution. Ideally, the filters should only allow the most difficult graphs to pass through.

Name	Construction	Filters	Size	Ratio	Quantity
Collection A	sat_cfi	✓	6000	2:1	500

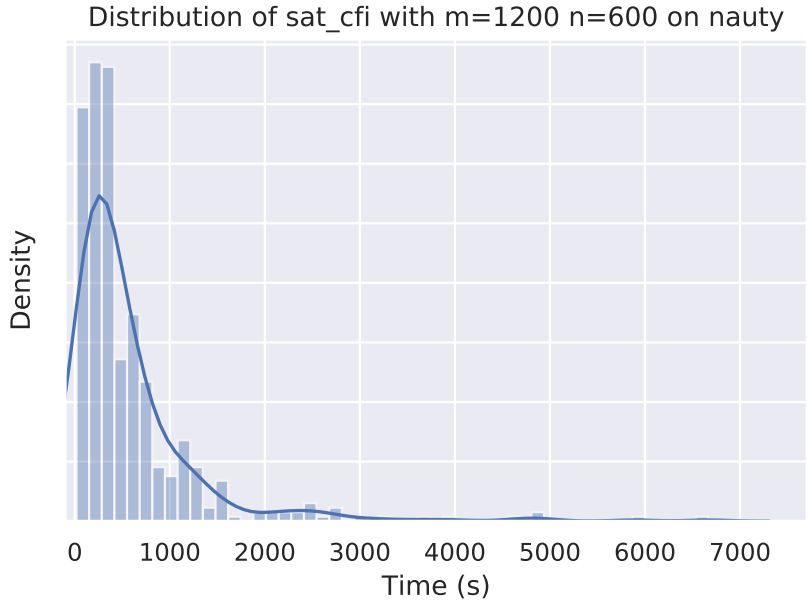


Figure 4.3: The solve time of graphs from collection A. The fitted curve suggests the underlying distribution from which the sat_cfi construction samples the graphs.

To see the extent which the filters affect the posterior distribution, collection B1 was generated with the filters, collection B2 was generated without the filters. A smaller clause-variable ratio was chosen because both theory and experimentation suggest that it yields more difficult graphs (see section 4.6.1). However, candidate graphs are more likely to contain automorphisms and 3-XOR formulas are more likely not to be uniquely satisfiable at this ratio. Therefore the use of filters to eliminate unsuitable base graphs is vital.

At first glance, the distribution in solve times of collections B1 and B2 were nearly identical. However, further investigation revealed that the graphs produced have very different properties: graphs drawn from the sat_cfi distribution with the filters are indeed the slowest graphs at its individualisation depth, but graphs generated without the filter tend to have a greater individualisation depth. This is a trade-off — by insisting the graphs generated to be globally asymmetric, we have sacrificed the opportunity for more

local symmetry within the graph. Indeed, the hardest graphs in Collection B2 were graphs that had a high individualisation depth but only a small amount of automorphisms.

Name	Construction	Filters	Size	Ratio	Quantity
Collection B1	sat_cfi	✓	4400	3:2	500
Collection B2	sat_cfi	✗	4400	3:2	500

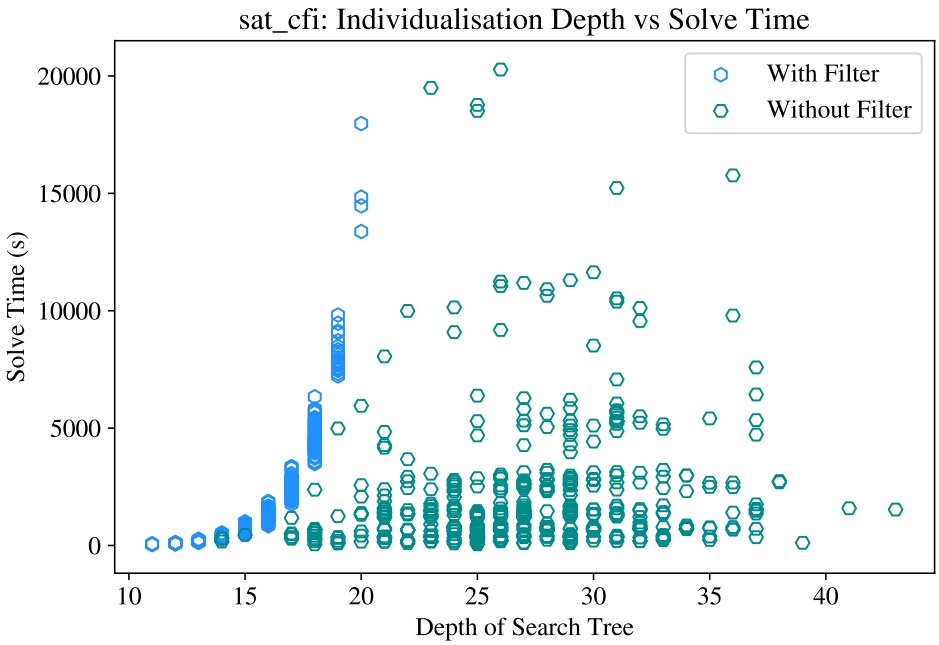


Figure 4.4: sat_cfi graphs produced with the filters have no non-trivial automorphisms, forcing `nauty` to traverse the entire search tree. Therefore solve time grows exponentially to the depth of the search tree.

4.4.1 Variation across Solves

In the requirements section I raised the concern that the solve time for the solvers may vary greatly across repeats. From repeating experiments on Collection A, the variation in solve time is less than 1%. Since the testing environment gives consistent times, there is no need to time the same graph for multiple iterations.

4.4.2 Variation in Size

In the spirit of conducting asymptotic analysis on the performance of graph isomorphism solvers, this experiment measures the solve time of `nauty` across graphs of different sizes. Collection C consists of graphs with 100 to 1000 vertices at an increment of 100 and graphs with 2000 to 8000 vertices at an increment of 250. Due to the non-deterministic nature of the sat_cfi construction, 5 graphs of each size were generated. The linear trend on a logarithmic scale in Figure 4.5 is highly indicative of an exponential growth in runtime. By rearranging the trend line, we can fit an exponential function to the graph on a linear scale.

$$t = 10^{0.00072n} * 10^{-1.92604} \approx 0.0119 * 1.0017^n$$

Name	Construction	Filters	Size	Ratio	Quantity
Collection C	sat_cfi	✓	100-8000	2:1	185

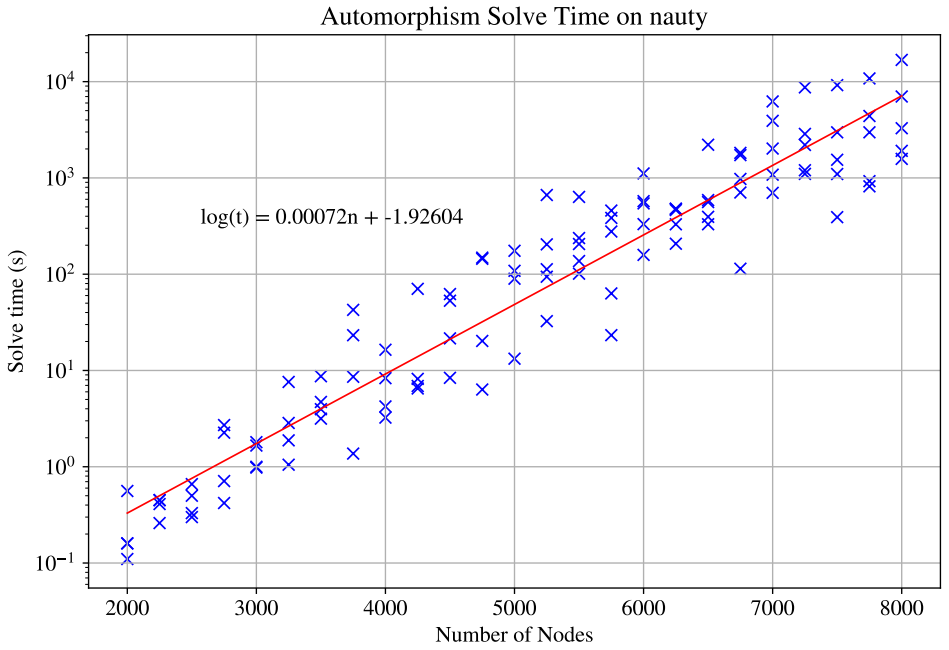


Figure 4.5: Automorphism solve time on `nauty`, using a log scale. As the number of nodes increases, `nauty`'s solve time grows exponentially.

The same experiment was conducted using `Traces` (Figure 4.6), another graph isomorphism solver. A step-like trend was observed — for the graphs in collection B1, `Traces` was able to compute the automorphism group within 10 seconds. However, as the size of the graphs grew, the SLURM Job failed to run to completion. Analysis of the SLURM job's logs indicated that `Traces` had exceeded the memory limit. These results were consistent with Khan's findings in Figure 2.5.

Repeating the experiment on the `skylake-himem` nodes gave better results — most graphs were solved within an hour, with the exception of a few graphs which continued to exhaust the available RAM.

The reason behind `Trace`'s high memory usage boils down to its traversal of the search tree using Breadth-First Search. Due to its size, the search tree is generated dynamically rather than stored. Since `Traces` uses BFS, it has to store all the nodes of the search tree at the current individualisation depth. As each selected cell has at least 2 nodes, there are at least 2^k nodes at individualisation depth k.

Although the height of the search tree is dependent on the cell selector strategy, it is always greater than the WL dimension of the graph (see section 2.4.4). Therefore using a breadth-first strategy inhibits a GI solver's ability of solving graphs beyond a certain WL dimension. Empirically, the limit on `skylake-himem` nodes is 11 individualisation steps.

Figure 4.4 shows a clear correlation between individualisation depth and solve time for filtered graphs. Note that automorphisms found in unfiltered graphs during `nauty`'s search process can be used to prune the search tree.

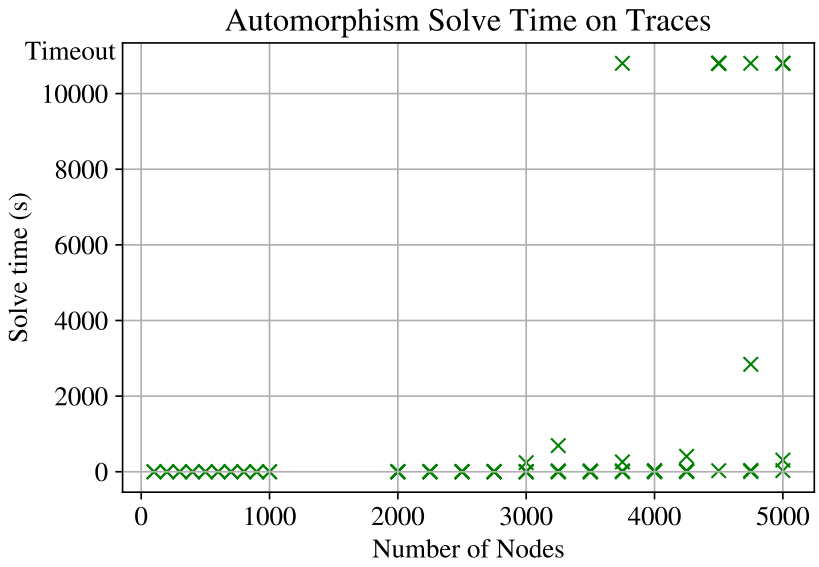


Figure 4.6: Using BFS, `Traces` had insufficient memory to solve large graphs.

4.4.3 Variation across Different Solvers

The same experiment was conducted using alternative graph isomorphism solvers, `conauto` and `bliss`, to measure the automorphism solve time of collection C. Once again, an exponential trend was observed (see Figures 4.7, 4.8).

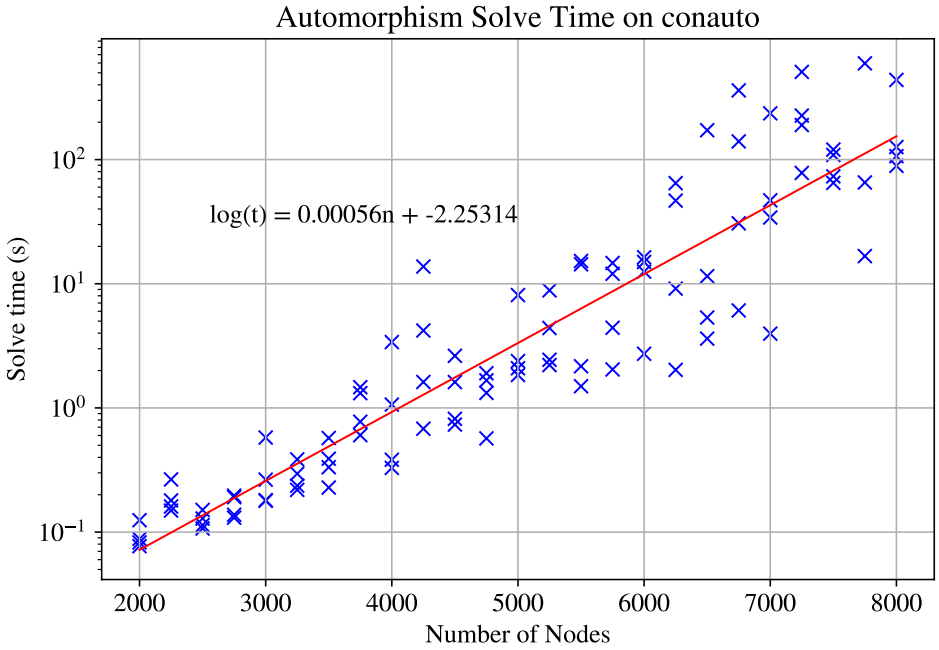


Figure 4.7: Automorphism solve time of sat_cfi graphs on `conauto`.

`bliss` offers a list different cell strategies. With the interest of comparing cell selector strategies, a bash script was used to iterate through all the solvers. The results were a dichotomy — while `bliss` performed reasonably well with the first 3 cell selector modes, it performed much better when using the last 3 strategies; in fact, it is difficult to see

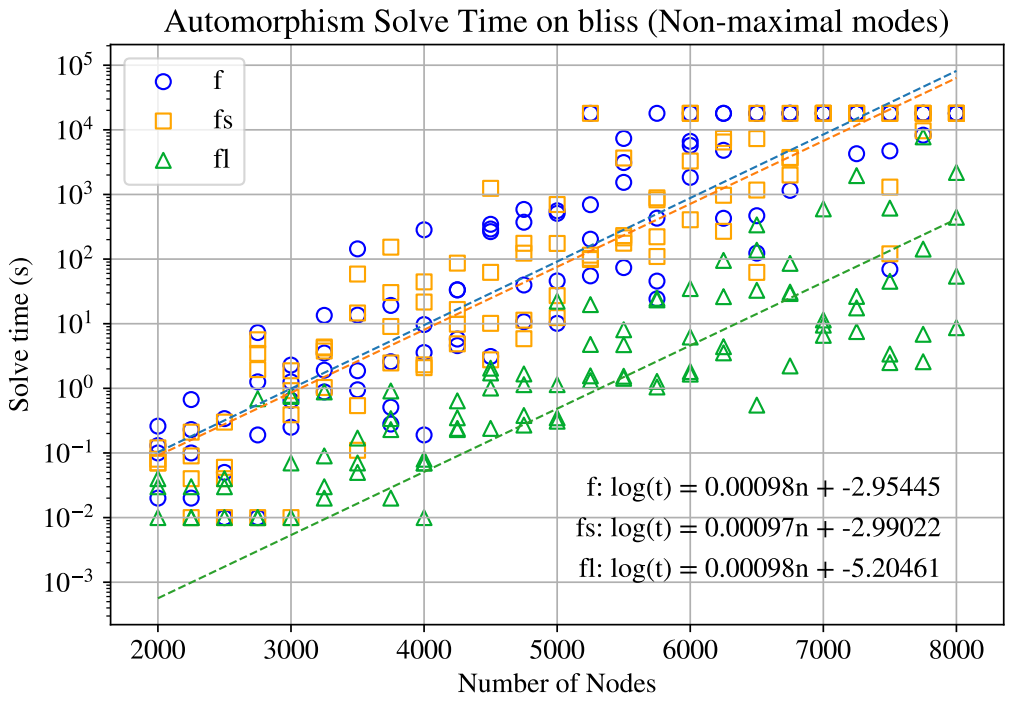


Figure 4.8: Automorphism solve time on `bliss` using different cell selection strategies.

Mode	Description
`f`	First non-singleton cell
`fs`	First smallest non-singleton cell
`fl`	First largest non-singleton cell
`fm`	First maximally non-trivially connected non-singleton cell
`fsm`	First smallest maximally non-trivially connected non-singleton cell (default)
`flm`	First largest maximally non-trivially connected non-singleton cell

Figure 4.9: A list of available cell selection strategies for bliss.

any exponential growth in run time. This does not mean `bliss` can solve the graphs in sub-exponential time; `bliss` could have time complexity $O(\exp(n))$ but a small constant factor. The theoretical lower bound we have on individualisation depth tells us size of the search tree will grow exponentially, regardless of the cell selection strategy. Therefore, if the size of graphs were increased, the runtime should exhibit an exponential trend.

4.5 Comparison with Other Graphs

Random graphs pose little trouble for state-of-the-art graph isomorphism solvers. Random graphs generated with over 10,000 vertices and edge probability $\frac{1}{\sqrt{10000}}$ are solved by `nauty` within a tenth of a second [16]. In contrast, by extrapolating Figure 4.5, a sat_cfi graph of a similar size would take `nauty` over 52 hours to solve.

Combinatorial Objects, such as projective planes and Latin squares, have a rich

Name	Construction	Filters	Size	Ratio	Quantity
Collection D	reg_cfi	—	660–4400	—	120

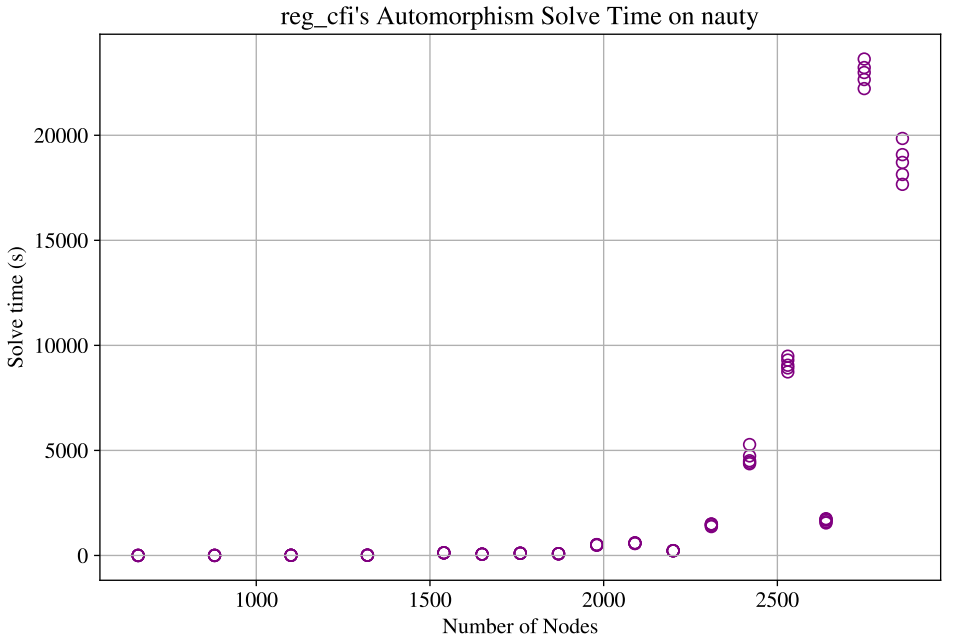


Figure 4.10: Automorphism solve time for reg_cfi in collection D.

mathematical structure, and graphs built on them have a high amount of symmetry and so require many individualisations to reach a discrete colouring. However, the high number of automorphisms can also be used to prune the search tree. For example, the largest Desarguesian plane from the `nauty` distribution, `pg2-49`, has 4902 vertices and 122550 edges. With $1.328752276992 * 10^{14}$ automorphisms, the generators of its automorphism group can be calculated less than 2 seconds.

CFI/Miyazaki Since the writing of Khan's project, there have been other constructions that utilise the Cai-Fürer-Immerman Gadget, notably the Miyazaki graphs. However, they are not asymmetric so their search trees can be pruned. To this date, reg_cfi and sat_cfi are the only CFI type constructions to exhibit exponential growth on `nauty` and `Traces`.

reg_cfi Collection D initially contains reg_cfi graphs with 660 to 4400 vertices, with an increment of 220. Solving graphs with more than 2860 nodes using `nauty` on the HPC timed out after 8 hours. To obtain more points around the knee of the curve, graphs in the range 1650-2750 were generated.

The experimental results (Figure 4.10) show that graphs in reg_cfi are harder than those in sat_cfi. Another observation is that there is very little variation in solve time for graphs of the same size, unlike sat_cfi which has a long-tailed distribution. Finally, the most surprising observation is that the exponential growth in solve time appears to have sinusoidal perturbations. A plausible explanation would be that the solve time depends

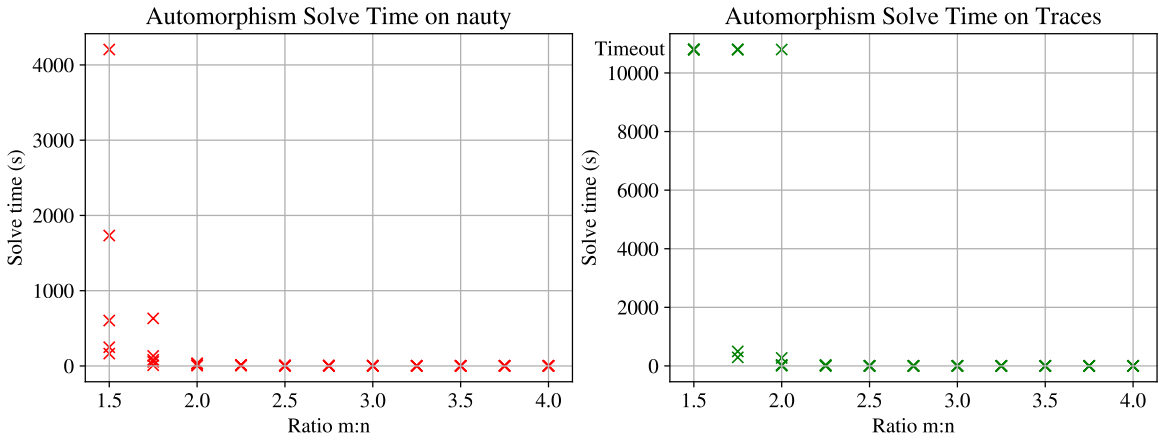


Figure 4.11: A reciprocal relationship between clause-variable ratio and solve time.

on the number-theoretic properties of $\frac{3v}{2}$, the number of edges in the 3-regular graph permuted during the base graph generation.

4.6 Local Consistency of 3-XOR Formulas

4.6.1 Variation in Clause-Variable Ratio

By generating sat_graphs with the same number of nodes but a varying clause-variable ratio $m : n$, the effect of varying the clause-variable ratio on the resulting graph was investigated. Collection E contains graphs of size ≈ 4500, with ratios from 1.50 to 4.00 with a ratio increment of 0.25. Due to the non-deterministic nature of the construction, 5 graphs of each ratio were generated. The generation of graphs itself yielded interesting results; graphs with ratio ≥ 2 only took 1–2 restarts to create, whilst some graphs with ratio 1.5 took over 1000 retries. This is unsurprising from a combinatorial view: it is highly unlikely for a random $n * n$ adjacency matrix to have rank n as this requires all rows to be linearly independent, but quite likely for a $n * (2n)$ matrix since there are many more row vectors available to act as a spanning set of the n-dimensional space.

Figure 4.11 shows that graphs generated with a smaller clause-variable ratio required more time to solve — this is because they require more individualisation steps. Graphs with ratio 1.5 required an average of 15.4 individualisations, whilst graphs with ratio 4 only required 5. The high number of individualisation steps indicates that the graphs have a high WL dimension, since the colour refinement algorithm is analogous to a 1-dimensional Weisfeiler-Leman test. One explanation for this phenomenon is that the small ratio between variables and clauses means that each variable will appear in less clauses, making the formula more locally consistent. However, the Gaussian elimination filter ensures that the Boolean formula is ultimately globally inconsistent.

Note that, similar to the use of filters in section 3.4.2, there is a trade-off between achieving high local consistency and maintaining global inconsistency of the Boolean formula when tweaking the ratio between m and n: a system with too many clauses will have low local consistency, but a system with too few clauses is unlikely to be inconsistent.

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 $m = 600$

Figure 4.12: Khan's experimental justification for the local consistency filter.

4.6.2 Weisfeiler-Leman Dimension Heuristic

This section evaluates the efficacy of using CryptoMiniSAT as a filter for 3-XOR formulas with high local consistency. As discussed in the implementation section, the inspiration comes from the disparity in theoretical complexity of Gaussian elimination and Resolution; for a hard locally consistent 3-XOR formula, resolution should take significantly longer.

In a previous ACS project, Khan provided experimental proof (Figure 4.12) by presenting two 3-XOR formulas; the formula which ran 0.05 seconds slower using resolution resulted in a graph which ran 0.8 seconds slower on `Traces`. With insufficient data, it is difficult to tell whether this time difference is statistically significant.

Taking into account that run times will be affected by context switches, the UNIX tool `time` was used to extract the user mode time, excluding the time the operating system spent in kernel mode, and recorded 20 repeats. For further statistical analysis, the tool `ministat` was used to calculate the minimum, maximum, mean and standard deviation of the datasets. `ministat` also performs the t-test to determine whether the two datasets show any difference at a statistical significance (see Appendix A.2).

This test was first performed on collection E, which shows as the clause-variable ratio decreases, the discrepancy in runtime becomes more noticeable. This is connected to the SAT phase transition problem: at a certain ratio, the number of satisfiable SAT formulas dramatically decrease and formulas near that threshold are more difficult to solve.

The test was also performed on collections B1 and B2, but no correlation between the graph isomorphism solve time and the difference in SAT solve times was found (see Appendix A.3). Therefore I made two conclusions: the formulas with a lower clause variable-ratio are indeed more locally consistent, resulting in a discrepancy in runtime for the SAT solver and more difficult graphs. However, the converse does not hold — the heuristic filter proposed by Khan which filters out formulas with a discrepancy in runtime does not locate 3-XOR formulas that are more locally consistent, nor does it create harder graphs.

4.7 Publishing the Graphs

At the request of my supervisor, I created a public repository containing my graphs and analysis. The analysis is now featured in the revised paper by Dawar and Khan, which refers to the repository.

After conducting the analysis on sat_cfi graphs, I submitted a collection of graphs to Adolfo Piperno, the creator of `Traces`. To make his life easier, the graphs were submitted

in both DIMACS and `dreadnaut` format, using the same naming convention as the other graphs on the website `http://pallini.di.uniroma1.it/Graphs.html`. After reviewing the graphs and performing his own experiments, Professor Piperno published the graphs on his website under the name Dawar-Yeung Graphs, with a link to the revised paper containing my results. During our email exchange, Professor Piperno asked me to give an explanation for the difference in performance of `nauty` and `Traces` on the sat_cfi graphs. Citing his paper [2], I replied that `Traces` uses BFS to traverse the search tree and so runs out of memory.

4.8 Success Criteria

The success criteria of the project, as defined in my project proposal, have all been met.

Criterion 1: I produce code that, by interfacing with a SAT solver, an automorphism finder and a Gaussian elimination library, produces graphs following the protocol defined.
Criterion 2: I have a script that runs these graphs through the isomorphism testers.
Criterion 3: I have produced results for a large number of graphs of varying sizes.

The pipeline described in the implementation section creates graphs using the sat_cfi construction as required by criterion 1. The testing suite described in the evaluation chapter creates and runs the graphs generated by the pipeline, meeting criterion 2. As for criterion 3, I have generated over 500,000 graphs after over 2000 hours on the Cambridge High Performance Cluster, and benchmarked their performance using the solvers `nauty`, `Traces`, `bliss` and `conauto`. Besides varying size, I have investigated the effect clause-variable ratio, individualisation depth, and asymmetry have on runtime.

Additional Extension: If the experimental results show that the solvers run significantly slower than random graphs, then I will submit the graphs to `http://pallini.di.uniroma1.it/Graphs.html`, a library of benchmark graphs.

The graphs produced were submitted and published on the site in February. Professor Piperno, one of the creators of `nauty` and `Traces`, was able to reproduce my results and agreed with my conclusions based on the analysis.

4.9 Meeting the Requirements

Performance The generation of sat_cfi graphs with low clause variable ratio would have been computationally intractable in Python. As an example, the 500 graphs in collection B1 required sieving through over 300,000 candidate base graphs.
New Filters The asymmetry filter, originally implemented using a SAT solver, has now been replaced with Gaussian elimination, which has better performance. Appropriate optimisations were made based on the requirements of the filter. Furthermore, Theorem 3 proves that the filters assert the asymmetry of graphs produced by the pipeline.

More Solvers In Khan's project only `Traces` was used to benchmark the graphs. In this project, `nauty`, `conauto` and `bliss` were used as well. More importantly, the results do demonstrate an exponential growth in solve time as the size of graphs increased; this was not achieved in the previous project.
Qualitative Explanations I was able to reproduce and provide an explanation for Khan's results. My own results are summarised in section 4.10.
Alternative Constructions Based on the theorems and pseudocode in the paper by Neuen and Schweitzer, I have implemented the reg_cfi construction and reproduced their results. Comparing against selected graphs in `Traces`' database, graphs produced by our sat_cfi construction were significantly harder.
Scientific Method Having specified the configuration used to generate the graphs, my results are easily reproducible. The variation in solve time over multiple runs was measured and deemed insignificant. By measuring only the user time, the cycles spent in kernel mode was excluded from my results.

4.10 Summary

This chapter tested the correctness of my sat_cfi and reg_cfi implementations. It then describes the generation and testing of the graphs, meeting the success criteria. Graphs produced using the sat_cfi construction were submitted to `http://pallini.di.uniroma1.it/Graphs.html`, the official graph isomorphism database. From sharing the experimental results with the creator of GI solver `Traces`, I brought to light a weakness in the state-of-the-art solver.

Summary of Results

1. Exponential trend for sat_cfi demonstrated on `nauty`, `conauto` and `bliss`. `Traces` runs out of memory for large graphs because it uses BFS to traverse the search tree. (See Figures 4.5, 4.6, 4.7 and 4.8.)

2. Exponential trend for reg_cfi in [10] was reproduced. Timing characteristics depend heavily on the numeric value of parameter n. (See Figure 4.10.)

3. The combination of the `dreadnaut` and Gaussian elimination filters do guarantee asymmetry in the final graph, but reduces individualisation depth. (See Figure 4.4.)

4. Decreasing the clause-variable ratio increases difficulty in producing asymmetric graphs, but the graphs produced are harder to solve and have a higher individualisation depth. (See Figure 4.11.)

These were previously unknown results prior to the start of the project. Results 1 and 2 were produced in February and included in `https://arxiv.org/abs/1809.08154`. Results 3 and 4 were discovered in early April and provide new directions for future pathological constructions for the graph isomorphism problem.

Chapter 5

Conclusion

The aim of this project is to produce graphs that run slowly on practical graph isomorphism solvers. To understand what constitutes a hard graph, I studied the refinement-individualisation algorithm used by `nauty` and `Traces`. After implementing the sat_cfi construction proposed by my supervisor, I designed a pipeline of filters proved to create asymmetric graphs with a high Weisfeiler-Leman dimension. As part of my evaluation, I was able to demonstrate for the first time an exponential relationship between the size of the sat_cfi graph and the time required to compute its automorphism group. Having demonstrated the equivalence between graph isomorphism and automorphism group computation, I have achieved the aim of my project.

Lessons Learned

This project emphasises the intricate connection between graphs and Boolean formulas, and required viewing the problem in terms of graphs, groups, linear systems and Boolean formulas holistically. Implementing the reg_cfi construction through proofs and pseudocode required a strong command of set theory and combinatorics.

From speaking to leading researchers, to discovering and publishing new results, this project gave me an exciting glimpse into the world of research.

Future Work

Modifying a SAT solver to perform bounded clause-width resolution allows the creation of a filter that determines whether a formula is k-locally consistent, which guarantees the sat_cfi graph to have WL dimension $O(k)$ [3]. This can be a standalone Part III project.

During evaluation I discovered that some graphs generated without the asymmetry filters are actually harder than those with the asymmetry filters. This goes against the current direction of research, which prioritises creating asymmetric graphs over graphs that are locally symmetric. My findings suggest that we should aim to create graphs with a high amount of local symmetry, but a low amount of global symmetry. Such future constructions may yield graphs that are even more difficult, challenging the boundaries of the graph isomorphism problem.

Bibliography

[1] László Babai. "Graph Isomorphism in Quasipolynomial Time". In: *CoRR* abs/1512.03547 (2015). arXiv: 1512.03547. URL: http://arxiv.org/abs/1512.03547.

[2] Brendan D. McKay and Adolfo Piperno. "Practical graph isomorphism, II". In: *Journal of Symbolic Computation* 60 (2014), pp. 94 –112. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2013.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0747717113001193.

[3] Anuj Dawar and Kashif Khan. "Constructing Hard Examples for Graph Isomorphism". In: *CoRR* abs/1809.08154 (2018). arXiv: 1809.08154. URL: http://arxiv.org/abs/1809.08154.

[4] Tommi Junttila. *bliss: A Tool for Computing Automorphism Groups and Canonical Labelings of Graphs*. 2015. URL: http://www.tcs.hut.fi/Software/bliss/ (visited on 05/11/2019).

[5] Luis Núñez Chiroque José Luis López Presa. *Algorithm conauto for Graph Isomorphism Testing and automorphism group computation*. 2013. URL: https://sites.google.com/site/giconauto/ (visited on 05/11/2019).

[6] L. Babai, P. Erdős, and S. Selkow. "Random Graph Isomorphism". In: *SIAM Journal on Computing* 9.3 (1980), pp. 628–635. DOI: 10.1137/0209047. eprint: https://doi.org/10.1137/0209047. URL: https://doi.org/10.1137/0209047.

[7] Anuj Dawar and Bjarki Holm. "Pebble games with algebraic rules". In: *CoRR* abs/1205.0913 (2012). arXiv: 1205.0913. URL: http://arxiv.org/abs/1205.0913.

[8] Sandra Kiefer, Ilia Ponomarenko, and Pascal Schweitzer. "The Weisfeiler-Leman Dimension of Planar Graphs is at most 3". In: *CoRR* abs/1708.07354 (2017). arXiv: 1708.07354. URL: http://arxiv.org/abs/1708.07354.

[9] Jin-Yi Cai, Martin Fürer, and Neil Immerman. "An optimal lower bound on the number of variables for graph identification". In: *Combinatorica* 12.4 (Dec. 1992), pp. 389–410. ISSN: 1439-6912. DOI: 10.1007/BF01305232. URL: https://doi.org/10.1007/BF01305232.

[10] Daniel Neuen and Pascal Schweitzer. "Benchmark Graphs for Practical Graph Isomorphism". In: *CoRR* abs/1705.03686 (2017). arXiv: 1705.03686. URL: http://arxiv.org/abs/1705.03686.

[11] Brendan D. McKay and Adolfo Piperno. ***nauty*** *and* ***Traces*** *User's Guide*. 2012. URL: http://pallini.di.uniroma1.it/Guide.html (visited on 05/03/2019).

[12] cppreference. ***std::vector<bool>***. 2019. URL: https://en.cppreference.com/w/cpp/container/vector_bool (visited on 04/19/2019).

[13] PSF. *Global Interpreter Lock*. 2017. URL: https://wiki.python.org/moin/GlobalInterpreterLock (visited on 04/09/2019).

[14] O. Tange. "GNU Parallel - The Command-Line Power Tool". In: *;login: The USENIX Magazine* 36.1 (2011), pp. 42–47. DOI: http://dx.doi.org/10.5281/zenodo.16303. URL: http://www.gnu.org/s/parallel.

[15] *Gephi*. 2019. URL: https://gephi.org/ (visited on 05/14/2019).

[16] Pallini. *Random Graphs*. 2012. URL: http://pallini.di.uniroma1.it/RandomGraphs.html (visited on 04/23/2019).