© 2021 Vaibhav Karve

GRAPHICAL STRUCTURE OF UNSATISFIABLE BOOLEAN FORMULAE

BY

VAIBHAV KARVE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Mathematics
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Nathan Dunfield, Chair
Associate Professor Anil N. Hirani, Director of Research
Professor Yuliy Baryshnikov
Assistant Professor Anush Tserunyan

Abstract

The presented research is an introduction and analysis of a novel graph decision problem called GRAPHSAT. Using the tools of topology and graph theory, this new variant builds upon the classical logic and computer science problem of boolean satisfiability (kSAT). kSAT asks if there exists a truth assignment that satisfies a given boolean formula. Our variant deals with multi-hypergraphs instead of boolean formulae and uses truth assignments on vertices instead of variables. This graph-theoretic picture helps us explore and exploit patterns in unsatisfiable instances of kSAT, which in turn helps us identify minimal obstruction sets to graph satisfiability.

Historically, kSAT (for $k \geq 3$) was the first problem that was proven to be NP-complete, independently by Cook [1] and Levin [2], making it central to the study of algorithms and computational complexity. We shed new light on kSAT by analyzing GRAPHSAT.

We demonstrate that 2GRAPHSAT is in complexity class P and has a finite obstruction set containing four simple graphs. Further, our exploration of 3GRAPHSAT gives rise to the local graph rewriting theorem, which leverages the fact that taking a union over all possible vertex-assignments preserves the satisfiability status of a graph. Using this theorem, we generate a list of graph reduction rules and an incomplete list of obstructions to satisfiability of looped-multi-hypergraphs.

A part of this research, especially the search for unsatisfiable instances of GRAPHSAT, was carried out using computational tools. Hence, some results are aided by a Python package specifically written to carry out computations on multi-hypergraph instances and implement the local rewriting algorithm. These computational steps are included in the thesis in the form of code blocks to give a glimpse of the back-end.

To my family

Acknowledgments

I wish to thank my advisor and mentor, Anil Hirani, for his patience, support, and guidance throughout this journey, for his willingness to let me do things my way, for his research ideas, wisdom, and his mathematical expertise, and for helping me through every small and large decision involved in this PhD with kindness and compassion. I thank him for making me feel comfortable and heard throughout the process. I feel lucky to have found the perfect advisor.

I am grateful to my thesis committee members Nathan Dunfield, Anush Tserunyan and Yuliy Baryshnikov for regular discussions on the subject, for giving me their valuable time, and for giving me ideas whenever I seemed to run out of them. I am grateful to Yuliy also for suggesting the problem of studying satisfiability from the viewpoint of the underlying structure of the sentences and for early discussions on the subject.

I am grateful to the University of Illinois Campus Research Board for funding part of this study via the Arnold O. Beckman Award in 2019. I am grateful to the Bourgin family for funding part of this study via the David G. Bourgin Mathematics Fellowship in Fall 2020. I am grateful to the Department of Mathematics, University of Illinois at Urbana-Champaign for its support, and for funding part of this study via the Gene H. Golub Summer Research Fellowship in Summer 2019. I would also like to thank Patryk Szuta for setting me up on the `mem.math.illinois` computer cluster, which enabled large GRAPHSAT computations that would have been otherwise impossible on my personal machine.

While carrying out this research, a major part of my day-to-day also involved teaching. My experience as a TA would not have been the same without the support of Jennifer McNeilly and the awesome way in which she runs the Merit Program for Emerging Scholars. She helped give meaning to the time I spent as a teacher.

I want to thank Richard Sowers and Philipp Hieronymi for leading IGL projects that gave me research problems I could work on that were removed from my PhD thesis. This kept me busy whenever I was stuck on GRAPHSAT, helped me diversify my skill set, and helped me be employable after graduation.

I am grateful to Richard Laugesen for always keeping his door open. His advice, encouragement and foresight has been critical at multiple junctures in my PhD. I will always be grateful for Rick's advise on reaching out to Anil as a potential thesis advisor. I have had no need to second-guess that advise.

I am grateful to my friends Derrek, Simone, Emily, Ravi, Ciaran, Sungwoo, Rishabh, Neelotpal, Madhura, Vivek, Mohit, and Akshita, for continuing to be my friends even though I kept using my PhD progress as an excuse to not talk to them often enough. I will need to find a new excuse now.

I would like to acknowledge the role of Emacs and especially org-mode, which I used for writing this thesis and for keeping my research organized; TEX and LATEX for making this research more readable; Python for enabling lengthy computations and for giving me constant programming joy; and coffee for helping me function during all of the above. I would like to denounce the SARS-Cov-2 virus for making everything so much harder.

I wish to thank my Alma Mater, the Indian Institute of Science Education and Research – Kolkata. It was the place where I started my mathematical journey in 2010. IISER-K not only taught me the math, it also taught me what kind of mathematician I want to be. In particular, I am grateful to Saugata Bandyopadhyay (IISER-K) and Pranay Goel (IISER-P) for helping me start my first research projects and for teaching me how to deal with the mundane parts of day-to-day research.

This doctoral thesis is a culmination of six years of my efforts and this effort would have been impossible without the love and support of my family. This PhD was not a one-person job, it took an entire family to finish it. I wish to thank my siblings Shreya and Shekhar for keeping me grounded and for listening to me complain and bicker about anything and everything. I am grateful to my parents Madhavi, Suresh, Anupama and Sanjay, for their unwavering faith in my abilities – often their faith far exceeded my actual abilities.

I wish to thank my beautiful and loving partner Sukanya for her support and more importantly her tolerance, for helping me stay confident, sane and

productive, for taking care of me, and for not letting me give up during the difficult times. Several of the mathematical ideas in this text were born from discussions with her, and she understands GRAPHSAT almost as well as me. She has sacrificed her time and well-being to make this thesis possible, for which I will be eternally grateful.

Table of Contents

Chapter 1

Introduction

The presented research is an introduction and analysis of a novel graph decision problem called GRAPHSAT. Using the tools of topology and graph theory, this new variant builds upon the classical logic and computer science problem of boolean satisfiability (kSAT). kSAT asks if there exists a truth assignment that satisfies a given boolean formula. Our variant deals with multi-hypergraphs instead of boolean formulae and uses truth assignments on vertices instead of variables. This graph-theoretic picture helps us explore and exploit patterns in unsatisfiable instances of kSAT, which in turn helps us identify minimal obstruction sets to graph satisfiability (see Figure 6.1 and Appendix).

Historically, kSAT (for $k \geq 3$) was the first problem that was proven to be NP-complete, independently by Cook [1] and Levin [2], making it central to the study of algorithms and computational complexity. We shed new light on kSAT by analyzing GRAPHSAT.

A part of this research, especially the search for unsatisfiable instances of GRAPHSAT, was carried out using computational tools. Hence, a part of the results are in the form of a Python package specifically written to handle multi-hypergraph instances and local rewriting of graphs. These computational steps are included in the thesis in the form of code blocks to give a glimpse of the back-end.

1.1 Overview

Following paragraphs give an overview of this thesis. This chapter also includes an overview for the non-mathematician in §1.2.

Chapter 2 introduces definitions and notations used in this thesis. It introduces some type-theoretic notation that streamlines the mathematical ex-

position. It also includes definitions for the two halves of GRAPHSAT, i.e. boolean formulae and graphs.

Chapter 3 is a discussion of GRAPHSAT restricted to multi-graphs and 2Cnfs. The key result in this chapter is a proof that graph homeomorphisms preserve satisfiability statuses. It also includes a complete set of minimal unsatisfiable simple graphs.

Chapter 4 describes the mathematical underpinnings for our analysis of 3GraphSAT. We state and prove the local graph rewriting theorem, that enables rewriting of graphs at a vertex, while preserving its satisfiability status.

Chapter 5 equips us with the computational tools necessary for carrying out local rewriting in practice on multi-hypergraphs. This is done by introducing the `graphsat` Python package, its constituent modules, and the functions within them.

Chapter 6 outlines computational results surrounding 3GRAPHSAT. It includes graph reduction rules, a selection of unsatisfiable multi-hypergraphs, and computations involving triangulations and infinite graphs, followed by a brief section on computational logistics.

Chapter 7 provides a conclusion to this study by summarizing key results as well as future directions and conjectures.

1.2 Overview for the non-mathematician

Presented here is the culmination of four years of research (2017–2021), on a problem we have dubbed *GRAPHSAT* – a novel question that we created and then attempted to answer using tools already known to mathematics.

GRAPHSAT is a combination of two parts – graphs and SAT. Graph is the name mathematicians use when talking about networks. When you read "graph", just imagine a bunch of points connected by some lines. The position of the points do not matter, neither do the lengths of the lines. For example, the following two graphs are indistinguishable as far as a mathematician is concerned –

The positions, lengths, shapes are merely extra details. We strip those details away and focus on connections between points. There is a whole wide subject that studies graphs and their various properties dating back to 1736 CE. Note: this is an old subject by scientific standards, but a young subject by mathematical ones. Graph theory is particularly useful for describing structures, exploring patterns, and letting us focus on the essential connections in any problem – say a problem like SAT.

SAT is shorthand for a problem with the much longer, formal name of *Boolean Satisfiability* and a history dating back to the late 1960s. SAT falls under the purview of computer science and is the problem that underlies technologies such as artificial intelligence, automated planning and scheduling, algorithms, cryptography, cyber-security, etc.

An over-simplified way of looking at SAT is as a series of "variables" and "conditions". The variables are just x, y, z, etc., which can only take values of True or False. The conditions look like –

x AND y must be True, OR y but NOT z can be False.

The conditions (or constraints) are always made of ANDs, ORs, and NOTs. SAT asks if such a system of variables and conditions has a solution i.e. can we find True/False values that satisfy the conditions? This question crops up everywhere in day-to-day software and algorithms applications.

SAT might seem like an easy question on the surface – an obvious solution is to simply try out all the True/False values we can think of for x, y and z. However, the trouble is that this trial-and-error-search-all-possibilities strategy is really inefficient when the number of variables involved routinely runs into millions and the conditions one needs to check runs into billions. This is exactly what makes SAT interesting. Researchers have spent the last few decades searching for better and faster strategies to either get to a satisfying solution, or declare the problem unsatisfiable.

The success in trying to find a quick solution to SAT has been mixed. The success has been marred by two unsolved issues –

- we do not have the most efficient algorithm possible yet, and
- we are still trying to understand why an efficient algorithm is so hard to find i.e. what makes SAT so hard.

The study presented in this thesis attempts to shed new light on SAT using graph theory. We take a SAT problem and turn it into a graph. Then, we apply all the tools in the arsenal of graph theory to study SAT problems and find patterns in SAT problems that have a solution (*satisfiable*) and SAT problems that don't (*unsatisfiable*). This combination results in GRAPHSAT.

Unlike the other sciences, mathematicians traditionally do not require laboratories or expensive research equipment for their studies. Math is traditionally done with pen-and-paper. However, this research was carried out in a different style. Throughout the past four years, I frequently encountered calculations that were hard to accomplish manually either because they were too time-consuming, or because the calculations were too human-error-prone. This prompted the authoring of a computer-program (a package written in the Python programming language) called `graphsat`. The package lets me delegate large calculations to a computer, making this thesis a mix of traditional pen-and-paper proofs and non-traditional computer-aided calculations.

Chapter 2 details all the definitions needed for stating and solving the GRAPHSAT problem (we mathematicians like being hyper-precise). Chapter 3 details a simpler flavor of GRAPHSAT called 2GRAPHSAT – these results were found in 2018 and published in 2020. Chapter 4 contains our key mathematical results and proofs, making it the densest chapter in this thesis. Chapter 5 is all about the `graphsat` Python package and how we translated mathematics into computer code. A basic understanding of programming languages is required for reading this chapter. Chapter 6 outlines all the calculations we carried out and the findings we made. If this study can be thought of as a census of graphs that are pertinent to GRAPHSAT, then this chapter can be considered a listing of all the candidates that this census has yielded. Lastly, our conclusions are summarized in Chapter 7.

Did we end up solving GRAPHSAT? The honest answer is No. We learned a lot about SAT and GRAPHSAT through this study, but we now have a vast number of unanswered questions that merit further investigation. On the way, we proved and published some results, found some interesting and surprising patterns in GRAPHSAT, authored a Python package, crashed several computers multiple times by over-committing to certain calculations, and completed a PhD. Our hope is that this thesis will serve as a starting point to further studies of GRAPHSAT (and SAT) in the future.

Chapter 2

Definitions and notation

We start by inductively defining boolean formulae in conjunctive normal form in §2.2 and graphs (in fact multi-hypergraphs) in §2.3. We introduce a way to view graphs as sets of Cnfs in §2.3.1. After translating Cnfs to graphs, we also define a notion of satisfiability for graphs in §2.3.2. We also define notation and conventions aimed at making the connection between Cnfs and graphs more intuitive. These are summarized in Table 2.2.

2.1 Type theory annotations

Throughout this chapter, we add various annotations to our terms in order to aid the reader in parsing and understanding the mathematics presented herein. These annotations are inspired from the field of type theory and can be thought of as representing the "category" or the "type" of a term.

For example, we write $c : \text{Clause} = x_1 \vee x_2$ to mean c is of type Clause and is equal to $x_1 \vee x_2$. The type annotations help with clarity without changing the mathematical content. For this reason, we use them wherever they can add to the exposition and avoid them wherever they might be unnecessarily verbose.

We will use a fixed list of types in this chapter, collected in Table 2.1. Full definitions for each type can be found in the sections that follow. Let V denote an arbitrary type, used to parameterize the other types. Elements of V will be called variables. For example, we define the type $\text{Cnf}\, V$ to be the type of all Cnfs on the variable set V. In practice, we will avoid mentioning V explicitly and simply write type judgments like $(x : \text{Cnf})$ instead of $(x : \text{Cnf}\, V)$.

In Table 2.1, we use the notation $A \equiv B$ to be mean that the two types have exactly the same terms. We write $A \sqsubset B$ to mean that A is a subtype

of B. This indicates that there are some extra restrictions placed on A. For example, we have Clause $\sqsubset$ Set Literal because every clause can also be viewed as a set of literals. However, not every set of literals is a clause because we require that clauses be nonempty. We also use $\oplus$ and $\times$ to denote the disjoint-sum and Cartesian-product of types respectively.

Table 2.1: Summary of all the types defined and used in this chapter.

Type	Relation to other types	Description
Variable	Variable $\equiv V$	variables; an alias of V
Literal V	Literal $\equiv V \oplus V \oplus$ Bool	literals of variable type V
Clause V	Clause $\sqsubset$ Set Literal	clauses of variable type V
Cnf V	Cnf $\sqsubset$ Set Clause	Cnfs of variable type V
Assignment V	Assignment $\sqsubset$ Set Literal	assignment of variable type V
Vertex	Vertex $\equiv V$	vertices; an alias of V
Edge V	Edge $\sqsubset$ Set Vertex	hyperedges of vertex type V
Graph V	Graph $\sqsubset$ Multiset Edge	multi-hypergraphs of vertex type V
Bool		the type of boolean values (true and false)
$\mathbb{N}$		the type of natural numbers
Set V		homogeneous sets of elements of type V
Multiset V	Multiset $V \equiv \mathbb{N} \times$ Set V	homogeneous multi-sets of elements of type V

2.2 Boolean formulae

We inductively define variables, literals, clauses, Cnfs and assignments. We then define satisfiability and equi-satisfiability for Cnfs.

2.2.1 Variables

We fix an arbitrary countable set and call its elements *variables*. Variables will be denoted by x_1, x_2, y_1, y_2, etc. In order to save ourselves the trouble of having to write all the x_i's we will use x_i and i interchangeably. We denote the type of variables as Variable and thus can write 2 : Variable to mean that 2 is a variable.

2.2.2 Literals and Booleans

A *literal* is either a variable, denoted by the same symbol as the variable; or its negation, denoted by $\neg i$ or $\overline{i}$ or `-i` (especially when it is written as part of computer code). We also declare that there are two additional literals called true (denoted by $\top$ or `<Bool: TRUE>`) and false (denoted by $\bot$ or `<Bool: FALSE>`). We denote the type of literals as Literal and can thus write $\overline{2}$: Literal to mean that $\overline{2}$ is a literal.

The Literal type is thus composed of two copies of V and one copy of Bool (the type of true and false). We can therefore write Literal $\equiv V \oplus V \oplus$ Bool.

2.2.3 Clauses

A *clause* is a disjunction of one or more literals. For example, $1 \vee \overline{2}$ is a clause. A clause made of a single literal i is also denoted by i. Moreover, we can leave the disjunctions as implicit when writing a clause, e.g. $x_1 \vee \overline{x_2} \vee x_3$ as $1\overline{2}3$. We denote the type of clauses as Clause. Thus, the following are all valid examples of clauses

$$(\top : \text{Clause}) \qquad (\bot : \text{Clause}) \qquad (1 : \text{Clause}) \qquad (1\overline{2} : \text{Clause})$$

```
(<Bool: TRUE>: Clause)     ((1, -2): Clause)
```

2.2.4 Conjunctive normal form (Cnf)

A boolean formula is said to be in *conjunctive normal form* (Cnf) if it is a conjunction of one or more clauses. Instead of saying a formula is in conjunctive normal form, we say it is a Cnf. Thus, Cnf refers to both the property of being in this form and to the type of all such formulae. We can write a Cnf x as,

$$x : \text{Cnf} = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4),$$

where x_i : Variable. This can be expressed more briefly as

$$x : \text{Cnf} = (1\overline{2}) \wedge (12\overline{3}) \wedge (\overline{1}4),$$

We can further drop the parentheses, and replace the conjunctions with a comma to produce

$$x : \text{Cnf} = 1\overline{2}, 12\overline{3}, \overline{1}4$$

To avoid ambiguity in these expressions, we stipulate that the disjunctions will always bind "tighter" than the conjunctions if no parentheses are used. For example, $1\overline{2}, 3$ is to be interpreted as $(x_1 \vee \neg x_2) \wedge x_3$ and not as $x_1 \vee (\neg x_2 \wedge x_3)$.

Since we dropped parentheses, a Cnf containing a single clause has the same representation as the clause itself. Thus, the following are all valid ways of writing a Cnf

$$(\top : \text{Cnf}) \qquad (\bot : \text{Cnf}) \qquad (1 : \text{Cnf}) \qquad (1\overline{2} : \text{Cnf}) \qquad (1\overline{2}, 2\overline{3} : \text{Cnf})$$

```
(<Bool: TRUE>: Cnf)     ((1, -2)(2, -3): Cnf)     (((1, -2), (2,
                                                    -3)): Cnf)
```

2.2.5 Assignments

A *truth assignment* (or simply an *assignment*) is a set of literals with the additional condition that a literal and its negation cannot both belong to an assignment. We denote the type of assignments by Assignment. For example,

$$a_1 : \text{Assignment} = \{1, \overline{2}, 3\}$$

is a valid assignment but,

$$a_2 : \text{Set Literal} = \{1, \overline{2}, \overline{1}\}$$

is not, hence its type is written merely as a set of literals but not as an assignment. We are interested in assignments only with regards with the way they "act" upon Cnfs. Fixing a : Assignment, we define this "action"

with the help of the following functions parameterized by a:

$$
\begin{aligned}
f_{a,\text{Literal}} : \text{Literal} &\to \text{Literal} \\
l &\mapsto \top, \quad \forall\, (l : \text{Literal}) \in a; \\
\overline{l} &\mapsto \bot, \quad \forall\, (l : \text{Literal}) \in a; \\
l &\mapsto l, \quad \forall\, (l : \text{Literal}) \notin a.
\end{aligned}
$$

$$
\begin{aligned}
f_{a,\text{Clause}} : \text{Clause} &\to \text{Clause} \\
\bigvee_i l_i &\mapsto \bigvee_i f_{a,\text{Literal}}(l_i), \quad \forall\, l_i : \text{Literal}.
\end{aligned}
$$

$$
\begin{aligned}
f_{a,\text{Cnf}} : \text{Cnf} &\to \text{Cnf} \\
\bigwedge_i c_i &\mapsto \bigwedge_i f_{a,\text{Clause}}(c_i), \quad \forall\, c_i : \text{Clause}.
\end{aligned}
$$

Thus, $f_{a,\text{Cnf}}$ replaces all occurrences of a literal with $\top$ and its negation with $\bot$ while keeping everything else unchanged in a Cnf. We write $x[a]$ instead of $f_{a,\text{Cnf}}(x)$, $c[a]$ instead of $f_{a,\text{Clause}}(c)$, and $l[a]$ for $f_{a,\text{Literal}}(l)$. The overloaded notation will always be disambiguated based on the types of x, c, l etc. For example,

$$
\begin{aligned}
&\left(1\overline{2}, 12\overline{3}, \overline{1}4 : \text{Cnf}\right)\left[\{1, \overline{2}\} : \text{Assignment}\right] \\
&\quad = \left(\top\overline{2}, \top 2\overline{3}, \bot 4\right)\left[\{\overline{2}\}\right] && \text{(setting 1 to } \top\text{)} \\
&\quad = \left(\top\top, \top\bot\overline{3}, \bot 4\right) && \text{(setting 2 to } \bot\text{)} \\
&\quad = (\top, \top, 4) \\
&\quad = 4 : \text{Cnf}
\end{aligned}
$$

We simplify this notation further by writing $x[1, \overline{2}]$ instead of $x[\{1, \overline{2}\}]$.

2.2.6 Satisfiability of Cnfs

A Cnf x is *satisfiable* if there exists a truth assignment a such that $x[a] = \top$. Otherwise, x is *unsatisfiable*.

Two Cnfs x and y are *equi-satisfiable* if either they are both satisfiable or they are both unsatisfiable. Equi-satisfiability is an equivalence relation and

we denote it by $x \sim y$. Using this notation, we can write the phrase "x is satisfiable" simply as $x \sim \top$ and "x is unsatisfiable" simply as $x \sim \bot$.

For example, $12, 2\overline{3} \sim \top$ because $12, 2\overline{3}[2] = \top$. Similarly, $12, 1\overline{2}, \overline{1}2, \overline{12} \sim \bot$ because there is no assignment that satisfies it.

We also introduce a map for Cnf-satisfiability denoted $\sigma : \text{Cnf} \longrightarrow \text{Bool}$, which maps satisfiable Cnfs to True and unsatisfiable Cnfs to False.

We now prove a lemma that shows that assigning to a literal does not change the satisfiability status of a Cnf.

Lemma 1. *Let x be a Cnf and l be a literal. Assigning l in x is equisatisfiable to introducing a new clause with the literal l, i.e. $x[l] \sim x, l$.*

Proof. If x, l is satisfiable, then there is an assignment a such that $x, l[a] = \top$. Since $x, l[a] = x[a], l[a]$, we must have $l \in a$. We can therefore write a as a union of its constituent parts, $a = \{l\} \cup a'$, for some assignment a'. Since $x[a] = \top$, we have $x[a'][l] = \top$ and therefore, $x[l] \sim \top$. Thus we can conclude that $x[l]$ is also satisfiable.

Conversely, if $x[l]$ is satisfiable, then there is an assignment a such that $x[l][a] = \top$. We also note that the assignment a does not contain l nor $\bar{l}$. We can therefore conclude that $x[a \cup \{l\}] = \top$. Therefore, $(x, l)[a \cup \{l\}] = \top$ and hence x, l is satisfiable. □

2.3 Graphs

Let V be a countable set. Elements of the set V will be called *vertices*. We will omit any mention of a specific V and simply denote elements of V using boldface numerals $\mathbf{1}, \mathbf{2}, \ldots$: Vertex.

An *edge* on V is a nonempty set of vertices. We omit the surrounding braces while denoting edges and simply write the vertices in a contiguous fashion. For example, **12**, **234**, and **1** are all edges. We refer to edges of size one, two and three as *loops*, *simple edges* and *hyperedges* respectively.

A *graph* on V is a nonempty multiset of edges on V. In literature, these are typically referred to as *multi-hypergraphs*, but we simply call them graphs. We omit the surrounding braces and separate the edges by boldface-commas. For example,

$$\boldsymbol{g} : \text{Graph} = \mathbf{12, 123, 14}.$$

If an edge repeats in a graph, we denote its multiplicity as a superscript. We can therefore write

$$\boldsymbol{g'} : \text{Graph} = \mathbf{12, 23^4, 4^2}.$$

2.3.1 Graphs as sets of Cnfs

In this section we define graphs inductively starting with vertices and then building up through edges (looped, simple and hyper) and then multi-edges. The standard term for these graph objects would be "looped-multi-hypergraphs".

We then define a novel way of interpreting a graph as a set of Cnfs that can "live on that graph". This translation of graphs into sets of Cnfs is lies at the heart of our attempt to turn boolean satisfiability instances into graph theory problems. Consequently, we make this translation as explicit as possible and provide enough details so that the interested reader may generate a translation "algorithm" from our definitions to readily turn any graph into a set of Cnfs.

We define the functions f_1, f_2, f_3, f_4 as follows —

$$\begin{aligned} f_1 : \text{Vertex} &\to \text{Set Literal} \\ \boldsymbol{v} &\mapsto \{v, \overline{v}\} \end{aligned}$$

$$\begin{aligned} f_2 : \text{Edge} &\to \text{Set Clause} \\ \boldsymbol{v_1 v_2 \cdots v_k} &\mapsto \{l_1 \vee l_2 \vee \cdots \vee l_k \mid (l_i : \text{Literal}) \in f_1(\boldsymbol{v_i})\} \end{aligned}$$

$$\begin{aligned} f_3 : \mathbb{N} \times \text{Edge} &\to \text{Set Cnf} \\ (n, \boldsymbol{e}) &\mapsto \{c_1 \wedge \cdots \wedge c_n \mid (c_i : \text{Clause}) \in f_2(\boldsymbol{e}) \text{ and } c_i \neq c_j\} \end{aligned}$$

$$\begin{aligned} f_4 : \text{Graph} &\to \text{Set Cnf} \\ \boldsymbol{e_1}^{n_1}, \ldots, \boldsymbol{e_k}^{n_k} &\mapsto \{x_1 \wedge \cdots \wedge x_k \mid (x_i : \text{Cnf}) \in f_3(n_i, \boldsymbol{e_i})\} \end{aligned}$$

The image of a graph under f_4 gives us the set of all Cnfs that "live on that graph". We use this set often enough that we will simply omit writing f_1, f_2, f_3, and f_4 and we will conflate the graph with its set of Cnfs. For

example, the following graph is also a set.

$$\begin{aligned}\boldsymbol{g} &= \mathbf{12, 13} : \text{Graph} \\ &= \left\{(12, 13),\ (12, 1\overline{3}),\ \ldots,\ (\overline{12}, \overline{13})\right\} : \text{Set Cnf}\end{aligned}$$

We further define two special sets of Cnfs, which we denote by boldface true and false symbols —

$$\top : \text{Set Cnf} = \{(\top : \text{Cnf})\}$$

$$\bot : \text{Set Cnf} = \{(\bot : \text{Cnf})\}$$

Note that both of these are sets of Cnfs that are not graphs (due to the requirement that a graph be a nonempty multiset of edges). Similarly, the empty set is also a valid term of type Set Cnf but is not a valid term of type Graph.

The set of Cnfs living on a graph can sometimes be empty. For example, the following graphs are all empty

$$\mathbf{1}^3 : \text{Graph} = \emptyset : \text{Set Cnf} \qquad \mathbf{12}^5 : \text{Graph} = \emptyset : \text{Set Cnf}$$

$$\mathbf{123}^9 : \text{Graph} = \emptyset : \text{Set Cnf} \qquad \mathbf{12}^5\mathbf{, 23} : \text{Graph} = \emptyset : \text{Set Cnf}$$

These sets are empty because their image under f_3 is empty. In general, a graph is empty if and only if it has an edge of size n with multiplicity more than 2^n.

2.3.2 Graph satisfiability

A set of Cnfs $\boldsymbol{g}$ is *totally satisfiable* if it is nonempty and if every Cnf in it is satisfiable. Otherwise, it is *unsatisfiable*. We denote by $\boldsymbol{\gamma} : \text{Set Cnf} \to \text{Bool}$, the map that sends a Graph to $\top$ if it is totally satisfiable, and to $\bot$ otherwise.

The following are equivalent (as a direct consequence of the definitions):

- $(\boldsymbol{g} : \text{Set Cnf})$ is totally satisfiable.
- $\boldsymbol{\gamma}(\boldsymbol{g}) = \top$.
- $\boldsymbol{g} \neq \emptyset$ and $\forall (x : \text{Cnf}) \in \boldsymbol{g},\ x \sim \top$.

- $\boldsymbol{g} \neq \emptyset$ and $\forall(x : \text{Cnf}) \in \boldsymbol{g},\ \exists(a : \text{Assignment}),\ x[a] \sim \top$.

The following are also equivalent:

- $(\boldsymbol{g} : \text{Set Cnf})$ is unsatisfiable.
- $\boldsymbol{\gamma}(\boldsymbol{g}) = \bot$.
- $\boldsymbol{g} = \emptyset$ or $\exists(x : \text{Cnf}) \in \boldsymbol{g},\ x \sim \bot$.
- $\boldsymbol{g} = \emptyset$ or $\exists(x : \text{Cnf}) \in \boldsymbol{g},\ \forall(a : \text{Assignment}),\ x[a] \sim \bot$.

Lemma 2. *For any set of Cnfs* $\boldsymbol{g_1}$ *and* $\boldsymbol{g_2}$*, we have* $\boldsymbol{\gamma}(\boldsymbol{g_1} \cup \boldsymbol{g_2}) = \boldsymbol{\gamma}(\boldsymbol{g_1}) \wedge \boldsymbol{\gamma}(\boldsymbol{g_2})$.

Proof. We have that $\boldsymbol{\gamma}(\boldsymbol{g_1} \cup \boldsymbol{g_2}) = \top$ if and only if $\boldsymbol{g_1} \cup \boldsymbol{g_2}$ is totally satisfiable. This in turn is true if and only if every Cnf in $\boldsymbol{g_1}$ is satisfiable and every Cnf in $\boldsymbol{g_2}$ is satisfiable. This in turn is true if and only if $\boldsymbol{\gamma}(\boldsymbol{g_1})$ and $\boldsymbol{\gamma}(\boldsymbol{g_2})$ are both equal to $\top$. □

2.3.3 Graph equi-satisfiability

Two sets of Cnfs $\boldsymbol{g_1}$ and $\boldsymbol{g_2}$ are *equi-satisfiable* if they are both totally satisfiable or are both unsatisfiable. We write this as $\boldsymbol{g_1} \sim \boldsymbol{g_2}$. We say that $\boldsymbol{g_1}$ *equi-implies* $\boldsymbol{g_2}$ by the $\bot$-criterion (denoted $\boldsymbol{g_1} \overset{\bot}{\Longrightarrow} \boldsymbol{g_2}$) if,

$$\forall(x_1 : \text{Cnf}) \in \boldsymbol{g_1},\ \exists(x_2 : \text{Cnf}) \in \boldsymbol{g_2},\ x_1 \sim \bot \implies x_2 \sim \bot.$$

If both $\boldsymbol{g_1} \overset{\bot}{\Longrightarrow} \boldsymbol{g_2}$ and $\boldsymbol{g_2} \overset{\bot}{\Longrightarrow} \boldsymbol{g_1}$, then we denote this more compactly as $\boldsymbol{g_1} \overset{\bot}{\Longleftrightarrow} \boldsymbol{g_2}$. We say that $\boldsymbol{g_1}$ *equi-implies* $\boldsymbol{g_2}$ by the $A\bot$-criterion (denoted $\boldsymbol{g_1} \overset{A\bot}{\Longrightarrow} \boldsymbol{g_2}$) if,

$$\forall(x_1 : \text{Cnf}) \in \boldsymbol{g_1},\ \exists(x_2 : \text{Cnf}) \in \boldsymbol{g_2},\ \forall(a : \text{Assignment}),$$

$$x_1[a] = \bot \implies x_2[a] = \bot.$$

If both $\boldsymbol{g_1} \overset{A\bot}{\Longrightarrow} \boldsymbol{g_2}$ and $\boldsymbol{g_2} \overset{A\bot}{\Longrightarrow} \boldsymbol{g_1}$, then we denote this more compactly as $\boldsymbol{g_1} \overset{A\bot}{\Longleftrightarrow} \boldsymbol{g_2}$.

Lemma 3. *For any set of Cnfs* $\boldsymbol{g_1}$ *and* $\boldsymbol{g_2}$*,*

$$\boldsymbol{g_1} \sim \boldsymbol{g_2} \quad \textit{iff and only if} \quad \boldsymbol{\gamma}(\boldsymbol{g_1}) = \boldsymbol{\gamma}(\boldsymbol{g_2}).$$

Proof. This follows directly from the definition of $\boldsymbol{\gamma}$. □

We now prove a lemma that is useful when establishing that two given graphs are equi-satisfiable.

Lemma 4. *Let* $\boldsymbol{g_1}$ *and* $\boldsymbol{g_2}$ *be sets of Cnfs. Then,* $\boldsymbol{g_1} \sim \boldsymbol{g_2}$ *if and only if* $\boldsymbol{g_1} \overset{\perp}{\Longleftrightarrow} \boldsymbol{g_2}$*. In other words, we can restrict our attention to only finding unsatisfiable Cnfs in both sets in order to prove their equi-satisfiability.*

Proof. Suppose that $\boldsymbol{g_1} \sim \boldsymbol{g_2}$. If both sets are totally satisfiable, then $\boldsymbol{g_1} \overset{\perp}{\Longleftrightarrow} \boldsymbol{g_2}$ is vacuously true. If both sets are unsatisfiable, then for every unsatisfiable Cnf in $\boldsymbol{g_1}$, we choose an unsatisfiable Cnf in $\boldsymbol{g_2}$. Similarly, for every unsatisfiable Cnf in $\boldsymbol{g_2}$, we can choose a corresponding unsatisfiable Cnf in $\boldsymbol{g_1}$, thus satisfying the requirements of the $\perp$-criterion of equi-implication.

Conversely, if $\boldsymbol{g_1} \overset{\perp}{\Longleftrightarrow} \boldsymbol{g_2}$, then given an unsatisfiable Cnf in $\boldsymbol{g_1}$ we can obtain an unsatisfiable Cnf in $\boldsymbol{g_2}$. Thus, either both sets are totally satisfiable or both are unsatisfiable. □

The procedure for proving equi-satisfiability of sets of Cnfs (or graphs in particular) can be further simplified using the following lemma.

2.3.4 Disjunction and conjunction of graphs

We define *disjunction* for sets of Cnfs to be a binary operation f_1 as

$$f_1 : \text{Set Cnf} \times \text{Set Cnf} \to \text{Set Cnf}$$

$$(s_1, s_2) \mapsto \{f_2(x_1 \vee x_2) \mid (x_1 : \text{Cnf}) \in s_1,\ (x_2 : \text{Cnf}) \in s_2\},$$

where, f_2 : Boolean-Formula $\to$ Cnf, is a map that converts boolean formulae into Cnfs by repeatedly using distributivity of disjunction over conjunction. We note that this is only one of several possible "implementations" of f_2. We choose this implementation because it has the advantage of not introducing

any new variables. A disadvantage of this approach is that this conversion to Cnfs can lead to an explosion in the number of resulting clauses. However, we

2.3.5 Consequence of graph disjunction

Lemma 6. *Let* $\boldsymbol{g_1}$ *and* $\boldsymbol{g_2}$ *be graph. Then,* $\boldsymbol{g_1} \vee \boldsymbol{g_2} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_1}$*, and similarly,* $\boldsymbol{g_1} \vee \boldsymbol{g_2} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_2}$*.*

Proof. Let $(x : \text{Cnf}) \in \boldsymbol{g_1} \vee \boldsymbol{g_2}$. We can factor x as $x = x_1 \vee x_2$ for some $(x_1 : \text{Cnf})$ and $(x_2 : \text{Cnf})$, such that $x_1 \in \boldsymbol{g_1}$ and $x_2 \in \boldsymbol{g_2}$. We choose $y = x_1$. Then, any assignment that falsifies x falsifies both x_1 and x_2. In particular, any such assignment also falsifies y. □

The converse of the above lemma is also true.

Lemma 7. *Let* $\boldsymbol{g_1}$ *and* $\boldsymbol{g_2}$ *be sets of Cnfs. If both* $\boldsymbol{g_1}$ *and* $\boldsymbol{g_2}$ *are unsatisfiable, then we can conclude that* $\boldsymbol{g_1} \vee \boldsymbol{g_2}$ *is also unsatisfiable.*

Proof. Let $(x_1 : \text{Cnf}) \in \boldsymbol{g_1}$ and $(x_2 : \text{Cnf}) \in \boldsymbol{g_2}$ be unsatisfiable Cnfs. Any assignment that satisfies $x_1 \vee x_2$ must also satisfy either x_1 or x_2 or both. Hence, we can conclude that no assignment satisfies $x_1 \vee x_2$. Since $x_1 \vee x_2 \in \boldsymbol{g_1} \vee \boldsymbol{g_2}$, we can conclude that $\boldsymbol{g_1} \vee \boldsymbol{g_2}$ is unsatisfiable. □

In the next lemmas, we show that if we take our arrows to be equi-implications under the $A\perp$ criterion, then graph disjunction obeys the universal property of products, while union of Cnf sets possesses the universal property of sums.

Lemma 8. *Let* $\boldsymbol{g_1}$*,* $\boldsymbol{g_2}$ *and* $\boldsymbol{g}$ *be sets of Cnfs.*

1. *If* $\boldsymbol{g} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_1}$ *and* $\boldsymbol{g} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_2}$*, then* $\boldsymbol{g} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_1} \vee \boldsymbol{g_2}$*.*

2. *If* $\boldsymbol{g_1} \overset{A\perp}{\Longrightarrow} \boldsymbol{g}$ *and* $\boldsymbol{g_2} \overset{A\perp}{\Longrightarrow} \boldsymbol{g}$*, then* $\boldsymbol{g_1} \cup \boldsymbol{g_2} \overset{A\perp}{\Longrightarrow} \boldsymbol{g}$*.*

Proof. For 1., we note that for every $(x : \text{Cnf}) \in \boldsymbol{g}$, there exist Cnfs $x_1 \in \boldsymbol{g_1}$ and $x_2 \in \boldsymbol{g_2}$ such that any assignment that falsifies x also falsifies both x_1 and x_2. Thus, any assignment that falsifies x also falsifies $x_1 \vee x_2$.

For 2., we note that for every $(x_1 : Cnf) \in \boldsymbol{g_1}$, there exists a Cnf $x \in \boldsymbol{g}$ such that any assignment that falsifies x_1 also falsifies x. This covers every Cnf in $\boldsymbol{g_1} \cup \boldsymbol{g_2}$ coming from $\boldsymbol{g_1}$. A similar argument hold for every Cnf coming from $\boldsymbol{g_2}$. This proves that $\boldsymbol{g_1} \cup \boldsymbol{g_2} \overset{A\perp}{\Longrightarrow} \boldsymbol{g}$. □

Lemma 9. *Let* $\boldsymbol{g}$*,* $\boldsymbol{g_1}$ *and* $\boldsymbol{g_2}$ *be sets of Cnfs. If either one of* $\boldsymbol{g} \wedge \boldsymbol{g_1}$ *or* $\boldsymbol{g} \wedge \boldsymbol{g_2}$ *is totally satisfiable, then, so is* $\boldsymbol{g} \wedge (\boldsymbol{g_1} \vee \boldsymbol{g_2})$*.*

Proof. Suppose $\boldsymbol{g} \wedge (\boldsymbol{g_1} \vee \boldsymbol{g_2})$ is unsatisfiable. Then, there exist Cnfs $x \in \boldsymbol{g}$, $x_1 \in \boldsymbol{g_1}$, and $x_2 \in \boldsymbol{g_2}$ such that $x \wedge (x_1 \vee x_2)$ is unsatisfiable. Then, both $x \wedge x_1 \sim \bot$ and $x \wedge x_2 \sim \bot$. In other words, both $\boldsymbol{g} \wedge \boldsymbol{g_1}$ and $\boldsymbol{g} \wedge \boldsymbol{g_2}$ are unsatisfiable. □

2.3.6 Assignments on graphs

We now extend the notion of assignments to sets of Cnfs. For a clause or Cnf, we defined assignments at a literal in §2.2.5. We now define assignments for a graph at a vertex. We note that this is a completely new notion that does not exists in graph theory and can be defined here only because of the connection we have established between graphs and Cnfs in the previous sections.

Let $\boldsymbol{v}$ be a vertex, and $\boldsymbol{g}$ be a set of Cnfs. We define,

$$\boldsymbol{g}[\boldsymbol{v}] : \text{Set Cnf} = \{x[v] \vee x[\overline{v}] \mid (x : \text{Cnf}) \in \boldsymbol{g},\ (v : \text{Literal}) \in \boldsymbol{v}\}$$

We note here that despite what the notation might suggest, $x[v]$ is in general not an element of $\boldsymbol{g}[\boldsymbol{v}]$. If $x \in \boldsymbol{g}$, then $x[v]$ is in fact an element of the graph $\text{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge \text{link}(\boldsymbol{g}, \boldsymbol{v})$. The definitions of sphere and link can be found in §2.3.7.

Next, we prove that assignments on graphs do not alter their satisfiability status. This is useful because post-assignment the graphs (or sets of Cnfs) always result in sets with Cnfs having one fewer variable, while not altering their satisfiability status

Lemma 10. *Let $\boldsymbol{g}$ be a set of Cnfs and let $\boldsymbol{v}$ be a vertex. Then, $\boldsymbol{g}[\boldsymbol{v}] \sim \boldsymbol{g}$.*

Proof. The proof follows by expanding the definition of $\boldsymbol{g}[\boldsymbol{v}]$.

$$\begin{aligned}
\boldsymbol{g}[\boldsymbol{v}] &= \{x[v] \vee x[\overline{v}] \mid (x : \text{Cnf}) \in \boldsymbol{g},\ (v : \text{Literal}) \in \boldsymbol{v}\} \\
&\sim \{x \wedge v \vee x \wedge \overline{v} \mid (x : \text{Cnf}) \in \boldsymbol{g},\ (v : \text{Literal}) \in \boldsymbol{v}\} \\
&= \{x \wedge (v \vee \overline{v}) \mid (x : \text{Cnf}) \in \boldsymbol{g},\ (v : \text{Literal}) \in \boldsymbol{v}\} \\
&= \{x \mid (x : \text{Cnf}) \in \boldsymbol{g}\} \\
&= \boldsymbol{g}
\end{aligned}$$

□

2.3.7 Parts of a graph

We now define some parts of graphs that will be useful for stating several subsequent lemmas. Let $\boldsymbol{g}$ be a graph and let $\boldsymbol{v}$ be a vertex. The *sphere* of $\boldsymbol{g}$ at $\boldsymbol{v}$ is the set of all edges not containing $\boldsymbol{v}$.

$$\text{sphere} : \text{Graph} \times \text{Vertex} \to \text{Graph}$$

$$(\boldsymbol{g}, \boldsymbol{v}) \mapsto \{\boldsymbol{e} : \text{Edge} \mid \boldsymbol{e} \in \boldsymbol{g} \text{ and } \boldsymbol{v} \notin \boldsymbol{e}\}$$

The *star* of $\boldsymbol{g}$ at $\boldsymbol{v}$ is the set of all edges containing $\boldsymbol{v}$.

$$\text{star} : \text{Graph} \times \text{Vertex} \to \text{Graph}$$

$$(\boldsymbol{g}, \boldsymbol{v}) \mapsto \{\boldsymbol{e} : \text{Edge} \mid \boldsymbol{e} \in \boldsymbol{g} \text{ and } \boldsymbol{v} \in \boldsymbol{e}\}$$

The *link* of $\boldsymbol{g}$ at $\boldsymbol{v}$ is the following set of edges, with $(-)$ denoting the usual set difference operation,

$$\text{link} : \text{Graph} \times \text{Vertex} \to \text{Graph}$$

$$(\boldsymbol{g}, \boldsymbol{v}) \mapsto \{(\boldsymbol{e} - \{\boldsymbol{v}\}) : \text{Edge} \mid \boldsymbol{e} \in \mathit{star}(\boldsymbol{g}, \boldsymbol{v})\}\,.$$

A graph $\boldsymbol{g}$ is a *subgraph* of a graph $\boldsymbol{h}$ if every edge of $\boldsymbol{g}$ (counting duplicates as distinct) is also an edge of $\boldsymbol{h}$. We denote this partial order on graphs by $\boldsymbol{g} \leq \boldsymbol{h}$. An edge $\boldsymbol{e}$ is a *face* of an edge $\boldsymbol{f}$ if every vertex in $\boldsymbol{e}$ is also in $\boldsymbol{f}$. A graph $\boldsymbol{g}$ is a *sublink* of a graph $\boldsymbol{h}$ if both graph have the same number of edges and if **every** edge of $\boldsymbol{g}$ (counting duplicates as distinct) is a face of a distinct edge in $\boldsymbol{h}$ (counting duplicates as distinct). We denote this partial order on graphs by $\boldsymbol{g} \ll \boldsymbol{h}$.

We now prove some lemmas outlining the relation between subgraphs, sublinks and satisfiability.

Lemma 11. *Let* $\boldsymbol{g_1}$ *and* $\boldsymbol{g_2}$ *be graphs such that* $\boldsymbol{g_1} \leq \boldsymbol{g_2}$*. Then,* $\boldsymbol{g_1} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_2}$*.*

Proof. We can write $\boldsymbol{g_2} = \boldsymbol{g_1} \wedge \boldsymbol{g}$ for some $(\boldsymbol{g} : \text{Graph})$. Let $(x_1 : \text{Cnf}) \in \boldsymbol{g_1}$. Let $(x : \text{Cnf}) \in \boldsymbol{g}$ be an arbitrary Cnf in $\boldsymbol{g}$. If $(a : \text{Assignment})$ is such that $x_1[a] = \bot$, then we have $(x_1 \wedge x)[a] = x_1[a] \wedge x = \bot$. □

Lemma 12. *Let* $\boldsymbol{g_1}$ *and* $\boldsymbol{g_2}$ *be sets of Cnfs such that* $\boldsymbol{g_1} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_2}$*. Then,*

$\forall(\boldsymbol{g} : \mathit{Graph})$,

$$\boldsymbol{g} \wedge \boldsymbol{g_1} \cup \boldsymbol{g} \wedge \boldsymbol{g_2} \sim \boldsymbol{g} \wedge \boldsymbol{g_2}.$$

Proof. It suffices to prove $\boldsymbol{g_1} \cup \boldsymbol{g_2} \overset{A\perp}{\Longleftrightarrow} \boldsymbol{g_2}$. To prove $\boldsymbol{g_1} \cup \boldsymbol{g_2} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_2}$, we select for every $(x_1 : \mathrm{Cnf}) \in \boldsymbol{g_1}$, a Cnf $x_2 \in \boldsymbol{g_2}$ from the proof of $\boldsymbol{g_1} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_2}$. The reverse side is trivially true. □

Lemma 13. *Let $\boldsymbol{e}$ and $\boldsymbol{f}$ be edges such that $\boldsymbol{e}$ is a face of $\boldsymbol{f}$. Then, $\boldsymbol{f} \overset{A\perp}{\Longrightarrow} \boldsymbol{e}$.*

Proof. Let $(c_f : \mathrm{Clause}) \in \boldsymbol{f}$ be arbitrary. We can write $c_f = c_e \vee c$ for some $(c_e : \mathrm{Clause}) \in \boldsymbol{e}$ and some $(c : \mathrm{Clause}) \in \boldsymbol{f} \setminus \boldsymbol{e}$. Then, any assignment a that falsifies c_f necessarily falsifies c_e, hence proving the result. □

Lemma 14. *Let $\boldsymbol{g_1}$ and $\boldsymbol{g_2}$ be graphs such that $\boldsymbol{g_1} \ll \boldsymbol{g_2}$. Then, $\boldsymbol{g_2} \overset{A\perp}{\Longrightarrow} \boldsymbol{g_1}$.*

Proof. This follows from the previous lemma, applying the result one face at a time. □

2.4 Summary

All operators defined in the previous sections are summarized in Table 2.2. These operators are written in increasing order of binding-tightness. The order of binding-tightness can be used to disambiguate expressions when multiple operators are used at the same time.

Table 2.2: Summary of all the operators.

Operator	Context	Meaning	Remarks
Invisible glue	between literals	boolean disjunction	binds tighter than all other operators
,	between clauses or Cnfs	boolean conjunction	also written as $\wedge$
$\overline{x_1}$	acts on literals	unary negation on literals	also written as $\neg x_1$
$x[a]$	action of assignment on Cnf	$f_{a,\mathrm{Cnf}}(x)$ in §2.2.5	
$\sim$	between two Cnfs	equi-satisfiable Cnfs	equivalence relation
Invisible glue	between vertices	adjacency of vertices	
$\boldsymbol{e}^n$	superscript for a (hyper)edge	edge-multiplicity	
,	between edges, or graphs	graph union	also called the adjacency of edges
$\vee$	between two sets of Cnfs	disjunction in §2.3.4	
$\wedge$	between two sets of Cnfs	conjunction in §2.3.4	
$\boldsymbol{g}[\boldsymbol{v}]$	action of vertex on a set of Cnfs	assignment in §2.3.6	
$\leq$	between two graphs	subgraph relation	
$\lll$	between two graphs	sublink relation	
$\sim$	between two sets of Cnfs	equi-satisfiable graphs/sets	binds looser than all other operators

Chapter 3

2GraphSAT

In this chapter, we focus our attention on 2GRAPHSAT, i.e. the following decision problem –

- **Instance:** Given a specific multi-graph $\boldsymbol{g}$.
- **Question:** Is every 2Cnf x such that $x \in \boldsymbol{g}$ satisfiable.

We use additional notation described below –

1. We denote the class of all looped-multi-graphs with edge size less than or equal to 2 by MGraph and we refer to these graphs as multi graphs (meaning "not hyper" graphs). Thus, MGraph $\sqsubset$ Graph.
2. We denote the class of all Cnfs with clause size less than or equal to $(k : \mathbb{N})$ by kCnf. Thus, the set of Cnfs corresponding to any multi-graph is in fact a set of 2Cnfs.

The corresponding decision problem at the level of Cnfs is well-known and well-studied. It is called 2SAT and is known to be in complexity class P. Instead of algorithmic issues our aim in this chapter is to study the structure of unsatisfiable instances of 2GRAPHSAT.

For completeness, we first briefly summarize the relevant fundamental algorithmic results for 2SAT. The 2SAT problem is in P. An $\mathcal{O}(n^4)$ algorithm was given by Krom [3] and a linear-time algorithm by Even, Itai and Shamir [4] and Aspvall, Plass and Tarjan [5]. All solutions of a given 2SAT instance can be listed efficiently using an algorithm by Feder [6].

In contrast to the algorithmic tractability of 2SAT, the SAT problem in general is NP-complete as was shown by Cook and by Levin independently [1], [2]. As part of the proof of the NP-completeness of SAT, they also proved that every logical sentence can be rewritten as a Cnf while changing its length by no more than a constant factor. Schaefer's dichotomy theorem states

necessary and sufficient conditions under which a finite set S of relations over the Boolean domain yields polynomial-time or NP-complete problems when the relations of S are used to constrain some of the propositional variables [7]. Thus [7] gives a necessary and sufficient condition for SAT-type problems to be in P vs. NP.

GRAPHSAT transfers notions of satisfiability onto graphs. An early connection between satisfiability and graphs was in the proof of NP-completeness of various graph problems, such as the clique decision problem and the vertex cover problem, by Karp [8]. One of the linear-time algorithms for 2-SAT [5] mentioned above also related graphs to satisfiability in its use of strongly-connected graph components as a tool for deciding satisfiability.

We explore the structures of unsatisfiable 2Cnfs by relating 2Cnfs to multi-graphs and examining which multi-graphs can support unsatisfiable sentences. Given a 2Cnf, an associated multi-graph can be created (as described more formally in 2.3.1) by identifying the variables as vertices and each clause as an edge. Since multiple clauses may involve the same two variables, the graph will in general have multi-edges. In §3.2 we prove theorems about the set of unsatisfiable multigraphs showing that this set is closed under graph homeomorphism. Theorem 11 shows that a graph can support an unsatisfiable Cnf if one of its subgraph can. Theorem 2 shows that if a multi-graph can support an unsatisfiable Cnfs, then the multi-graphs obtained by edge-contractions at edges not contained in triangles can. §3.4 is about connectivity properties of graphs that we need to prove the main result. The main result of this chapter is Theorem 3 in which we give a complete characterization of simple graphs that can support unsatisfiable sentences. This is given in the form of a finite set of obstructions to supporting only satisfiable sentences. In §3.5 we discuss how our approach differs from the application of finite obstructions (forbidden minors) theory developed in the Robertson-Seymour graph minor theorem published in a series of papers starting with [9] and ending with [10].

3.1 Simple Cnfs suffice

In this section, we show that every multi-graph is equisatisfiable to a simple graph. Thus when studying satisfiability of multi-graphs, we only need

consider those that are simple.

First, we make the observation that for any variable a, there are at most two length 1 clauses, namely a and $\overline{a}$. Such clauses involving a single variable a (or its negation) will be referred to as (a)-*clauses*. For every pair of variables a, b, there are at most four length 2 clauses, namely ab, $a\overline{b}$, $\overline{a}b$ and $\overline{a}\overline{b}$. Such clauses involving both a and b (or their negations) will be referred to as ab-*clauses*.

Lemma 15. *Let $\boldsymbol{a}$ be a vertex. Let $\boldsymbol{g}$ be a multi-graph.*

1. *The double-loop graph $\boldsymbol{a}^2$ is unsatisfiable*
2. *$\boldsymbol{g} \wedge \boldsymbol{a} \sim \text{sphere}(\boldsymbol{g}, \boldsymbol{a}) \wedge \text{link}(\boldsymbol{g}, \boldsymbol{a})$.*
3. *If $\boldsymbol{g}$ does not have any edges incident on $\boldsymbol{a}$, then $\boldsymbol{g} \wedge \boldsymbol{a} \sim \boldsymbol{g}$.*

Proof. 1 follows from the fact that $a \wedge a \in \boldsymbol{a}^2$ and is an unsatisfiable Cnf. For 2, we note that that a generic element of $\boldsymbol{g} \wedge \boldsymbol{a}$ is of the form $x \wedge a$, for $x \in \boldsymbol{g}$. Since $x \wedge a \sim x[a]$, and since the $x[a] \in \text{sphere}(\boldsymbol{g}, \boldsymbol{a}) \wedge \text{link}(\boldsymbol{g}, \boldsymbol{a})$, the result follows. 3 follows from 2, since under the given hypothesis, the sphere of $\boldsymbol{g}$ at $\boldsymbol{a}$ is $\boldsymbol{g}$ itself and the link is empty. □

Lemma 16. *In the following statements, let $\boldsymbol{a}$ and $\boldsymbol{b}$ be vertices, let $\boldsymbol{g}$ be a multi-graph, let x be a Cnf and let a, b, c and d be literals.*

1. *The quadruple-edge multigraph $(\boldsymbol{ab})^4$ is unsatisfiable.*
2. *$x \wedge (a \vee b) \wedge (a \vee \overline{b}) \wedge (\overline{a} \vee b) \sim x[a, b]$.*
3. *$x \wedge (a \vee b) \wedge (a \vee \overline{b}) \sim x[a]$.*
4. *$x \wedge (a \vee b) \wedge (\overline{a} \vee b) \sim x[b]$.*
5. *$x \wedge (a \vee b) \wedge (\overline{a} \vee \overline{b})$ is equisatisfiable with the Cnf obtained by replacing every occurrence of the literal b in x with $\overline{a}$ (and $\overline{b}$ with a).*

Proof. 1 follows from the fact that the set of Cnfs corresponding to $\boldsymbol{ab}^4$ has a single unsatisfiable element.

For 2, we start with the notion that for any Cnf y and literals a and b, we can state that y is satisfiable if and only if some assignment of a and

b on it is. Thus, we can write that y is satisfiable if and only if the set $\{y[a,b],\ y[a,\overline{b}],\ y[\overline{a},b],\ y[\overline{a},\overline{b}]\}$ is totally satisfiable as a set of Cnfs.

Using this result with $y = x \wedge (a \vee b) \wedge (a \vee \overline{b}) \wedge (\overline{a} \vee b)$ tells us that $y[\overline{b}] = x \wedge a \wedge \overline{a} = \bot$. Thus, both $y[a,\overline{b}]$ and $y[\overline{a},\overline{b}]$ are unsatisfiable.

Similarly, $y[\overline{a}]$ and therefore $y[\overline{a},b]$ are unsatisfiable. Thus y is satisfiable if and only if $\{y[a,b]\}$ is totally satisfiable. We can restate this as $y \sim y[a,b]$. Thus 2 follows.

For 3, let $y = x \wedge (a \vee b) \wedge (a \vee \overline{b})$. Since $(a \vee b) \wedge (a \vee \overline{b}) \to a$, any assignment A that satisfies y must contain the literal a (i.e. it must set a to true). Thus, A must also satisfy $x[a]$. Conversely, consider an assignment A' that satisfies $x[a]$. The assignment $A' \cup \{a\}$ satisfies y. We conclude that $y \sim x[a]$.

Let $y = x \wedge (a \vee b) \wedge (\overline{a} \vee b)$. Since $(a \vee b) \wedge (\overline{a} \vee b) \to b$, by a similar reasoning to the previous case, we conclude that $y \sim x[b]$.

Lastly, let $y = x \wedge (a \vee b) \wedge (\overline{a} \vee \overline{b})$. Any assignment A that satisfies y must map a and b to opposite booleans in order to satisfy the two (a,b)-clauses. Thus, such a truth assignment must also satisfy the Cnf resulting from replacing b with $\overline{a}$ in x. Conversely, consider an assignment A' that satisfies the Cnf obtained by replacing b by $\overline{a}$ in x. If $a \in A'$, add $\overline{b}$ to A', else add b to A'. Then the resultant assignment must satisfy y. Thus, we have proven 5. □

From Lemmas 15 and 16, we conclude that any Cnf whose corresponding multi-graph might contain a multi-loop, or a multi-edge can be replaced by a smaller, equisatisfiable Cnf not containing the multi-loops and multi-edges. Thus we can safely restrict our attention to only Cnfs whose corresponding multi-graphs are simple.

3.2 Graph homeomorphisms preserve satisfiability status

Edge-contraction at an edge $\boldsymbol{ab}$ of a graph results in a graph in which the vertices $\boldsymbol{a}$ and $\boldsymbol{b}$ are merged into a single new vertex $\boldsymbol{c}$ and the all edges incident on $\boldsymbol{a}$ or $\boldsymbol{b}$ are now made incident on $\boldsymbol{c}$ instead. If the original graph contains edges $\boldsymbol{ad}$ and $\boldsymbol{bd}$ already, then edge-contraction at $\boldsymbol{ab}$ will result in multi-edges connection $\boldsymbol{d}$ to the new vertex $\boldsymbol{c}$. We take care to preserve

these multiplicities and not replace them or simplify them to a single edge (as some graph theory textbooks do when defining edge-contraction). We will show that edge-contractions of unsatisfiable graphs are unsatisfiable.

A *subdivision of an edge* $\boldsymbol{ab}$ in a graph yields a graph containing one new vertex $\boldsymbol{c}$, and with an edge set replacing $\boldsymbol{ab}$ by two new edges $\boldsymbol{ac}$ and $\boldsymbol{bc}$. A *subdivision of a graph* $\boldsymbol{g}$ is a graph resulting from the subdivision of edges in $\boldsymbol{g}$. Two graphs are *homeomorphic* if they are subdivisions of the same graph. We will show that if two graphs are homeomorphic, then either both are totally satisfiable or both are unsatisfiable.

A graph $\boldsymbol{g}$ is the *topological minor* of a graph $\boldsymbol{h}$ if a subdivision of $\boldsymbol{g}$ is isomorphic to a subgraph of $\boldsymbol{h}$. We will produce a complete list of topological minors that appear as obstructions to graph satisfiability. Furthermore, it is possible to embed multi-graphs into $\mathbb{R}^3$ and to allow them to inherit the subspace topology of $\mathbb{R}^3$. Once embedded, homeomorphic graphs are also homeomorphic in the topological sense and topological graph minors are simply topological subspaces. We will use this to produce a complete list of minimal unsatisfiable topological subspaces.

Theorem 1. *If multi-graphs* $\boldsymbol{g}$ *and* $\boldsymbol{h}$ *are homeomorphic, then* $\boldsymbol{g} \sim \boldsymbol{h}$.

Proof. Since $\boldsymbol{g}$ and $\boldsymbol{h}$ are homeomorphic graphs, there exists a graph $\boldsymbol{k}$ such that both $\boldsymbol{g}$ and $\boldsymbol{h}$ are subdivisions of $\boldsymbol{k}$. It suffices to prove that $\boldsymbol{k} \sim \bot$ if and only if $\boldsymbol{g} \sim \bot$. In fact, it is enough to prove that $\boldsymbol{k} \sim \bot$ if and only if $\boldsymbol{g'} \sim \bot$, where $\boldsymbol{g'}$ is a graph obtained via a single subdivision at an arbitrary edge $\boldsymbol{ab}$ in $\boldsymbol{k}$. We denote by $\boldsymbol{c}$ the new vertex in $\boldsymbol{g'}$ created by the subdivision.

Suppose that $\boldsymbol{k} \sim \bot$. Then, there exists an unsatisfiable 2Cnf $x_k \in \boldsymbol{k}$. We can write without loss of generality, $x_k = x \wedge (a \vee b)$. Note that if x_k is not in this form, that is, if the (a, b)-clause is not positive in either a or b, then we simply interchange a with $\overline{a}$ and/or b with $\overline{b}$ in order to bring it into the desired form. We then define a simple 2Cnf $x_{g'}$ supported on $\boldsymbol{g'}$ as $x_{g'} = x \wedge (a \vee c) \wedge (\overline{c} \vee b)$. We note that this is indeed a Cnf in the set $\boldsymbol{g'}$. We prove that $x_{g'}$ is unsatisfiable by showing that from any assignment that satisfies $x_{g'}$, we can obtain an assignment satisfying x_k, leading to a contradiction.

Let A be an assignment that satisfies $x_{g'}$. Either c or $\overline{c}$ is an element of A.

If $c \in A$, then it implies

$$\top \sim x_{g'}[A] \sim x_{g'}[c] = x \wedge b \sim x[b] = x_k[b] \sim x_k \wedge b.$$

Thus, x_k is satisfiable, leading to a contradiction.

Similarly, if $\overline{c} \in A$, then it implies

$$\top \sim x_{g'}[A] \sim x_{g'}[\overline{c}] = x \wedge a \sim x[a] = x_k[a] \sim x_k \wedge a.$$

Thus, x_k is satisfiable, again leading to a contradiction. We can therefore conclude that $x_{g'} \sim \bot$ and that $\boldsymbol{g' \sim \bot}$.

Conversely, suppose that $\boldsymbol{g' \sim \bot}$. Then, there exists an unsatisfiable 2Cnf $x_{g'}$, supported on $\boldsymbol{g'}$. By exchanging a with $\overline{a}$, and/or b with $\overline{b}$ and/or c with $\overline{c}$, we can assume without loss of generality that either $x_{g'}$ has the form

$$x_{g'} = x \wedge (a \vee c) \wedge (c \vee b) \qquad \text{or} \qquad x_{g'} = x \wedge (a \vee c) \wedge (\overline{c} \vee b).$$

In the first case, since $x_{g'} \sim \bot$, we must also have $x_{g'}[c] \sim \bot$. Thus $x = x_{g'}[c] \sim \bot$, and therefore the 2Cnf $x \wedge (a \vee b)$ is an unsatisfiable element of $\boldsymbol{k}$. In the other case, since $x_{g'} \sim \bot$, the 2Cnf obtained by replacing all occurrences of c in $x_{g'}$ with b must also be unsatisfiable. Thus, $x \wedge (a \vee b)$ is unsatisfiable, and therefore $\boldsymbol{k \sim \bot}$. □

In view of the theorem we just proved, it is natural to attempt an enumeration of all unsatisfiable simple graphs up to homeomorphism. However, we state here (without proof) that there are infinitely-many mutually non-homeomorphic unsatisfiable graphs. Given below is such an incomplete infinite list of unsatisfiable graphs, none of which are homeomorphic to each other.

We recall here the fact derived from Lemma 11, that if a graph is unsatisfiable, then so are all of its super-graphs. We note that the converse of this result is not true, that is, there exist totally satisfiable graphs $\boldsymbol{g}$ and unsatisfiable graphs $\boldsymbol{h}$ such that $\boldsymbol{g}$ is a subgraph of $\boldsymbol{h}$. For example, the triangle graph is totally satisfiable, (as shown in Lemma 17) but  is not (as

shown in the proof of Lemma 24).

Corollary 2. *Let graph* $\boldsymbol{g}$ *be a topological minor of a graph* $\boldsymbol{h}$*. Then,* $\boldsymbol{g} \overset{\perp}{\Longrightarrow} \boldsymbol{h}$*.*

Proof. By definition of topological minors, some subdivision $\boldsymbol{g'}$ of $\boldsymbol{g}$ is isomorphic to a subgraph of $\boldsymbol{h}$. By Theorem 1, if $\boldsymbol{g} \sim \perp$ then $\boldsymbol{g'} \sim \perp$. By Lemma 11, if $\boldsymbol{g'} \sim \perp$ then $\boldsymbol{h} \sim \perp$. Thus, $\boldsymbol{g} \overset{\perp}{\Longrightarrow} \boldsymbol{h}$. □

We note that the converse of Corollary 2 is not true for the same reason that the converse of Lemma 11 is not. Corollary 2 motivates the following definition — a graph $\boldsymbol{g}$ is a *minimal unsatisfiability graph* if both of the following conditions hold —

1. $\boldsymbol{g}$ is untotally satisfiable, and
2. every proper topological minor $\boldsymbol{g'}$ of $\boldsymbol{g}$ is totally satisfiable.

Furthermore, a set M of simple graphs graphs is *complete* if every simple, unsatisfiable graph has a subgraph that is homeomorphic to some element of M. One can deduce, starting from Corollary 2, not only that such a complete set of minimal unsatisfiability graphs exists but also that it must be unique. After proving a result about the relation between *edge-contraction* and unsatisfiable graphs, the remainder of this chapter is dedicated to finding this unique complete set of minimal unsatisfiable simple graphs.

Theorem 2. *Let* $\boldsymbol{g}$ *and* $\boldsymbol{h}$ *be graphs such that* $\boldsymbol{g}$ *can be obtained via a series of edge-contractions at edges of* $\boldsymbol{h}$*. Then,* $\boldsymbol{h} \overset{\perp}{\Longrightarrow} \boldsymbol{g}$*.*

Proof. If is enough to prove the theorem for the case when $\boldsymbol{g}$ can be obtained from $\boldsymbol{h}$ via a single edge-contraction, say at the edge $\boldsymbol{uv}$ of $\boldsymbol{h}$. We label the new vertex in $\boldsymbol{g}$ formed by the merger of $\boldsymbol{u}$ and $\boldsymbol{v}$ by $\boldsymbol{w}$.

Suppose that $\boldsymbol{h} \sim \perp$. Then, there exists an unsatisfiable 2Cnf x_h in the set $\boldsymbol{h}$. We can write without loss of generality,

$$x_h = x \wedge \left(\bigwedge_{a \in A} a \vee u \right) \wedge \left(\bigwedge_{b \in B} b \vee \overline{u} \right) \wedge \left(\bigwedge_{c \in C} c \vee v \right) \wedge \left(\bigwedge_{d \in D} d \vee \overline{v} \right) \wedge (u \vee v),$$

where x is either trivially true or is a simple 2Cnf, and A, B, C and D are sets of literals such that $A, B, C, D, \neg A, \neg B, \neg C$ and $\neg D$ are pairwise-disjoint. If x_h is not already in the desired form, then it can be modified as detailed below.

1. If the (u, v)-clause in x_h is negative in u, then exchange u with $\overline{u}$.
2. If the (u, v)-clause in x_h is negative in v, then exchange v with $\overline{v}$.
3. For every $X \in \{A, B\}$ and for every $x \in X$, if the (x, u)-clause in x_h is negative in x, then exchange x with $\overline{x}$.
4. For every $X \in \{C, D\}$ and for every $x \in X$, if the (x, v)-clause in x_h is negative in x, then exchange x with $\overline{x}$.

We then choose

$$x_g = x \wedge \left(\bigwedge_{a \in A \cup D} a \vee w \right) \wedge \left(\bigwedge_{b \in B \cup C} b \vee \overline{w} \right),$$

and note that x_g is a simple 2Cnf supported on $\boldsymbol{g}$. We prove that x_g is unsatisfiable by showing that from any assignment that satisfies x_g, we can obtain an assignment satisfying x_h, leading to a contradiction.

Given an assignment for x_g, we can extend it to an assignment for x_h by replacing all occurrences of u by w, and all occurrences of v by $\overline{w}$. The resultant 2Cnf is then equal to

$$x \wedge \left(\bigwedge_{a \in A \cup D} a \vee w \right) \wedge \left(\bigwedge_{b \in B \cup C} b \vee \overline{w} \right) = x_g,$$

is satisfiable. This contradicts the unsatisfiability of x_h. We conclude that x_g is unsatisfiable, and hence that $\boldsymbol{g}$ is unsatisfiable. □

We note that the converse of this theorem is not true, that is, there exist a graph $\boldsymbol{g}$ is unsatisfiable that can be obtained via edge-contractions of a totally satisfiable graph $\boldsymbol{h}$. For example, consider the graphs $\boldsymbol{g} =$  and $\boldsymbol{h} =$  . The graph $\boldsymbol{g}$ is unsatisfiable (as shown in Lemma 24) and can be obtained from an edge-contraction while $\boldsymbol{h}$ is totally satisfiable (implied by Theorem 2 and Lemma 18 when combined with the fact that $\boldsymbol{h}$ can be reduced to $\boldsymbol{K_4 - e}$ via other edge-contractions).

3.3 Totally satisfiable graph families

Lemma 17. *Let $\boldsymbol{C_3}$ denote the triangle graph. $\boldsymbol{C_3}$ is unsatisfiable.*

Proof. We write $\boldsymbol{C_3}$ as $\boldsymbol{ab \wedge ac \wedge bc}$. Without loss of generality, every 2Cnf in $\boldsymbol{C_3}$ can be written either in the form

$$x = (a \vee b) \wedge (a \vee c) \wedge (b, c)\text{-clause} \qquad \text{or} \qquad x = (a \vee b) \wedge (\overline{a} \vee c) \wedge (b, c)\text{-clause}.$$

If x is not originally in this form, we modify it by interchanging a with $\overline{a}$ and/or b with $\overline{b}$ and/or c with $\overline{c}$ till it is. In the first case, setting a to true yields a single (b, c)-clause. This clause can be satisfied by making the appropriate assignment for either b or c. In the second case, by setting both a and c to true, we get $x[a, c] = (b, c)\text{-clause}[c]$.

The literal b can then be set to an appropriate boolean value in order to satisfy the (b, c)-clause. Thus, every 2Cnf in $\boldsymbol{C_3}$ totally satisfiable. □

Corollary 3. *Let $\boldsymbol{C_n}$ denote the cycle graph on n vertices. The graph $\boldsymbol{C_n}$ is unsatisfiable for every $n \geq 3$.*

Proof. The graph $\boldsymbol{C_n}$ is homeomorphic to $\boldsymbol{C_3}$ for every $n \geq 3$. Theorem 1 and Lemma 17 therefore imply the result. □

Lemma 18. *Let $\boldsymbol{K_4}$ denote the complete graph on four vertices. Let $\boldsymbol{K_4 - e}$ denote the graph obtained by deleting a single edge $\boldsymbol{e}$ from $\boldsymbol{K_4}$. The graph $\boldsymbol{K_4 - e}$ is unsatisfiable.*

Proof. We enumerate the vertex set of $\boldsymbol{K_4}$ as $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}\}$. Let $\boldsymbol{e}$ be the edge $\boldsymbol{cd}$. Every 2Cnf x supported on $\boldsymbol{K_4 - e}$ can be written either in the form

$$\begin{aligned} x &= (a \vee b) \wedge (a \vee c) \wedge (a \vee d) \wedge (b, c)\text{-clause} \wedge (b, d)\text{-clause}, \quad \text{or} \\ x &= (a \vee b) \wedge (a \vee c) \wedge (\overline{a} \vee d) \wedge (b, c)\text{-clause} \wedge (b, d)\text{-clause}. \end{aligned}$$

If x is not already in the desired form, then we can interchange each variable with its negation till it is.

In the first case, by setting a to true, we obtain

$$x[a] = (b, c)\text{-clause} \wedge (b, d)\text{-clause}.$$

This can be satisfied by making appropriate assignments for c and d so that they satisfy each of the clauses. In the second case, we can set a to true to obtain

$$x[a] = d \wedge (b, c)\text{-clause} \wedge (b, d)\text{-clause}.$$

This resulting Cnf can be satisfied by setting d to true, by choosing an assignment for b that would satisfy the (b, d)-clause, and by then choosing an assignment for c that would satisfy the (b, c)-clause. We conclude that every 2Cnf in the set $\boldsymbol{K_4} - e$ is totally satisfiable. □

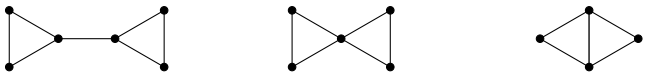
Lemma 19. *Tree graphs are totally satisfiable.*

Proof. Every tree graph can be reduced via edge-contractions to a single-vertex graph. A single-vertex graph is clearly totally satisfiable. The result thus follows from Theorem 2. □

3.4 Structure of graphs with two or three independent cycles

In this section we prove lemmas about the structure of graphs that have either two or three independent cycles. These structural results are needed for proving the results in §3.4.1, including the main result of this chapter.

Lemma 20. *Every connected graph having exactly two copies of $\boldsymbol{C_3}$ as subgraphs has one of the following three graphs as a topological minor –*



p-configuration *v-configuration* *e-configuration*
(Bow-tie graph) *(Butterfly graph)* $(\boldsymbol{K_4 - e})$

Remark 1. *For convenience, we have labeled the three graphs as the p-configuration (for path), the v-configuration (for vertex-adjacency) and the e-configuration (for edge-adjacency) respectively.*

Proof. We construct this list of topological minors from the bottom up. Two copies of $\boldsymbol{C_3}$ can be put together to create a connected graph in exactly three ways –

1. either the two cycles share zero vertices, or

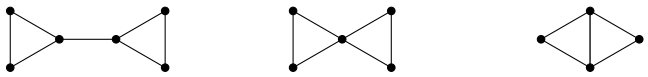
2. they share exactly one vertex, or

3. they share exactly two vertices (that is, they share an edge).

In the first case, since the graph is still supposed to be connected, we claim that the two copies of $\boldsymbol{C_3}$ must be connected by one or more paths. Such a graph will always have the two copies connected by a single path as a subgraph. Hence, the graph will also always have the p-configuration as a topological minor.

In the second case, the graph will always have the v-configuration as a subgraph and therefore, also as a topological minor. Similarly, in the third case, the graph will always have the e-configuration as a subgraph and therefore, also as as topological minor. □

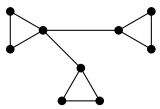
Lemma 21. *Every connected graph having exactly two or more independent cycles has one of the following three graphs as a topological minor.*



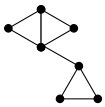
p-configuration *(Bow-tie graph)* | *v-configuration* *(Butterfly graph)* | *e-configuration* $(\boldsymbol{K_4 - e})$

Proof. Let $\boldsymbol{g}$ be a connected graph with two or more independent cycles. If any two cycles share one or more edges, then $\boldsymbol{K_4 - e}$ is a topological minor of $\boldsymbol{g}$. If no pair of cycles share any edges, but at least one pair shares a vertex, then the v-configuration is a topological minor of $\boldsymbol{g}$. If every pair of cycles share neither any edges nor any vertices, then using the connectedness of $\boldsymbol{g}$, we infer that there must be a path connecting vertices in every pair of cycles. Thus, in this case, the p-configuration is a topological minor of $\boldsymbol{g}$. □

Lemma 22. *Every connected graph having three or more copies of* $\boldsymbol{C_3}$ *as subgraphs has one of the following fifteen graphs is a topological minor.*

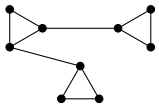


ppp-config. #1

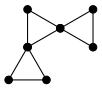


ppe-config. #2

vve-config. #1

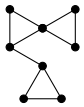


ppp-config. #2

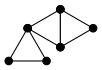


pvv-configuration

vve-config. #2

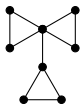


ppv-config. #1



pve-configuration

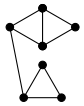
vee-configuration



ppv-config. #2

vvv-config. #1

eee-config. #1



ppe-config. #1

vvv-config. #2

eee-config. #2

Remark 2. *Labels for each of the fifteen graphs (like ppp-config.* #1*) are explained as part of the proof. The three copies of* $\boldsymbol{C_3}$ *that we use for this labeling have been colored in the fifteen graphs for the sake of easy visual identification and do not represent filled-in hyperedges.*

Proof. We arrive at the list of topological minor by constructing them from the bottom up. From the proof of Lemma 20, we know that every pair of $\boldsymbol{C_3}$ can be joined in exactly one of three ways and that these three ways are inequivalent when considering the topological minor relation. Graphs with three or more copies of $\boldsymbol{C_3}$ will have at least $\binom{3}{2=3}$ distinct pairs of $\boldsymbol{C_3}$.

To enumerate all possible pairwise-joints of these three $\boldsymbol{C_3}$, we form all three-letter words formed using the letters $\{p, v, e\}$, with repetition allowed, but order being irrelevant. For example, the word *ppe* would correspond to the family of graphs where the two pairs of $\boldsymbol{C_3}$ are joined via the p-configuration, while the third pair of $\boldsymbol{C_3}$ is joined via the e-configuration. We also note that each word corresponds to a family of graphs and there could be multiple non-isomorphic configurations in each such family. We analyze each case separately below –

ppp-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in a *p*-configuration, the third copy of $\boldsymbol{C_3}$ can be added in two different non-isomorphic ways (such that the first and third, as well as the second and third are joined via the *p*-configuration). Thus, we obtain only two configurations in this case – *ppp*-config. #1 and *ppp*-config. #2.

ppv-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in a *v*-configuration, the third copy of $\boldsymbol{C_3}$ can be added in two non-isomorphic ways. Thus, we obtain only two configurations in this case, namely *ppv*-config. #1 and *ppv*-config. #2.

ppe-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in an *e*-configuration, the third copy of $\boldsymbol{C_3}$ can be added in two non-isomorphic ways. Thus, we obtain only two configurations in this case, namely *ppe*-config. #1 and *ppe*-config. #2.

pvv-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in a *v*-configuration, the third copy of $\boldsymbol{C_3}$ can be added in only one way. Thus, we obtain only one configurations in this case, namely the *pvv*-configuration.

pve-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in an *e*-configuration, the third copy of $\boldsymbol{C_3}$ can added in only one way. Thus, we obtain only one configurations in this case, namely the *pve*-configuration.

pee-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in an *e*-configuration, the third copy of $\boldsymbol{C_3}$ cannot be added in a way that it is shares an edge with the first and is connected by a path to the second. Thus, we do not obtain configurations in this case.

vvv-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in an *v*-configuration, the third copy of $\boldsymbol{C_3}$ can be added in two non-isomorphic ways. Thus, we obtain only two configurations in this case, namely *vvv*-config. #1 and *vvv*-config. #2.

vve-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in an *e*-configuration, the third copy of $\boldsymbol{C_3}$ can be added in two non-isomorphic ways. Thus, we obtain only two configurations in this case, namely *vve*-config. #1 and *vve*-config. #2.

vee-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in an *e*-configuration, the third copy of $\boldsymbol{C_3}$ can added in only one way. Thus, we obtain only one configurations in this case, namely the *vee*-configuration.

eee-**configuration.** After joining the first pair of $\boldsymbol{C_3}$ in an *e*-configuration, the third copy of $\boldsymbol{C_3}$ can be added in two non-isomorphic ways. Thus, we obtain only two configurations in this case, namely *eee*-config. #1 and *eee*-config. #2.

□

Lemma 23. *Every connected graph having three or more independent cycles has one of the following four graphs as a topological minor.*

v-configuration (*Butterfly graph*) | *p-configuration* (*Bow-tie graph*) | *eee-config.* #1 ($\boldsymbol{K_4}$) | *eee-config.* #2 ($\boldsymbol{K_{1,1,3}}$)

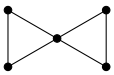
Proof. It is enough to show that every graph listed in Lemma 22 has one of the four graphs listed above as a topological minor. The *p*-configuration is a subgraph of both the *ppp*-configurations, both the *ppv*-configurations, both *ppe*-configurations, the *pvv*-configuration, and the *pve*-configuration. Of the remaining configurations, both the *vvv*-configurations, both the *vve*-configurations, and the *vee*-configuration contain the *v*-configuration as a subgraph.

The first *eee*-configuration is isomorphic to $\boldsymbol{K_4}$ while the second is isomorphic to $\boldsymbol{K_{1,1,3}}$. Hence, all 15 configurations have at least one of the four graphs listed above as a topological minor. □

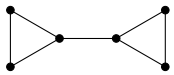
3.4.1 The complete set of minimal unsatisfiable simple graphs

We note that a graph $\boldsymbol{g}$ is unsatisfiable if and only if some connected component of $\boldsymbol{g}$ is. We proceed to make some more observations about graphs that are unsatisfiable.

Lemma 24. *The following four graphs are minimal unsatisfiability graphs –*



v-configuration
(Butterfly graph)



p-configuration
(Bow-tie graph)

eee-config. #1
($\boldsymbol{K_4}$)

eee-config. #2
($\boldsymbol{K_{1,1,3}}$)

Proof. Consider the following unsatisfiable 2Cnfs.

$$x_1 = 12 \wedge \overline{1}3 \wedge \overline{2}3 \wedge \overline{3}4 \wedge \overline{3}5 \wedge \overline{45}$$
$$x_2 = 12 \wedge \overline{1}3 \wedge \overline{2}3 \wedge \overline{3}4 \wedge \overline{4}5 \wedge \overline{4}6 \wedge \overline{56}$$
$$x_3 = 12 \wedge 13 \wedge \overline{1}4 \wedge \overline{23} \wedge 2\overline{4} \wedge 3\overline{4}$$
$$x_4 = 12 \wedge \overline{1}4 \wedge 23 \wedge \overline{2}4 \wedge \overline{2}5 \wedge \overline{34} \wedge \overline{45}$$

Their associated graphs are the four graphs listed above in order. Since these 2Cnfs are unsatisfiable, so are their graphs. To prove that they are minimal unsatisfiability graphs, we need to show that all of the proper topological minors of these graphs are totally satisfiable. Since the four listed graphs are not subdivisions of other graphs, it is enough to show that every proper subgraph of these four graphs is totally satisfiable.

Every proper subgraph of the v-configuration is either a subgraph of  or  . Both of these graphs can be reduced to $\boldsymbol{C_3}$ via edge-contractions. In Lemma 17 we proved that $\boldsymbol{C_3}$ is totally satisfiable, and hence by Theorem 2 we conclude that both  and  are totally satisfiable. Therefore, by Theorem 11, all the subgraphs of v-configuration are totally satisfiable.

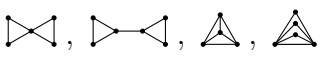
Every proper subgraph of the p-configuration is either a subgraph of  ,  or  . The first two of these graphs can be reduced to proper subgraphs of the v-configuration via edge-contractions. Since we proved that all proper subgraphs of the v-configuration are totally satisfiable, by Theorem 2 we conclude that both  and  are totally satisfiable. Lastly, the graph  is unsatisfiable if and only if at least one of its connected components is. From Lemma 17, we conclude that  is totally satisfiable. Theorem 11 implies that all proper subgraphs of the p-configuration are totally satisfiable.

In Lemma 18, we proved that $\boldsymbol{K_4 - e}$ is totally satisfiable. Since every proper subgraph of $\boldsymbol{K_4}$ is also a subgraph of $\boldsymbol{K_4 - e}$, using Theorem 11 we conclude that all proper subgraphs of $\boldsymbol{K_4}$ are totally satisfiable.

Every proper subgraph of $\boldsymbol{K_{1,1,3}}$ is either a subgraph of\;  or $\boldsymbol{K_{2,3}}$, both of which can be reduced to $\boldsymbol{K_4 - e}$ via edge-contractions. Since $\boldsymbol{K_4 - e}$ is totally satisfiable, by Theorem 2 we conclude that all the proper subgraphs of $\boldsymbol{K_{1,1,3}}$ are totally satisfiable. □

Corollary 4. *Every connected graph having three or more independent cycles is unsatisfiable.*

Proof. From Lemma 23 we know that every connected graph having three or more independent cycles has one of the four graphs listed in that lemma as a topological minor. Since we proved in Lemma 24 that all four of these graphs are unsatisfiable, the result follows from Corollary 2. □

Theorem 3. *The set*  *is the complete set of minimal unsatisfiable graphs.*

Proof. We denote the set by M. Since we proved in Lemma 24 that the elements of M are minimal unsatisfiability graphs, it suffices to show that a graph $\boldsymbol{g}$ is unsatisfiable if and only if $\boldsymbol{g}$ has some element of M as a topological minor. Corollary 2 implies the "if" part of this statement. We now show that every unsatisfiable graph $\boldsymbol{g}$ has some element of M as a topological minor. Throughout the remainder of the proof we suppose that $\boldsymbol{g}$ is unsatisfiable.

If $\boldsymbol{g}$ is connected and has three or more independent cycles, then the result follows from Lemma 23. If $\boldsymbol{g}$ is connected and has exactly two independent cycles, then by Lemma 21, either $\boldsymbol{g}$ has an element of M as a topological minor or $\boldsymbol{g}$ has $\boldsymbol{K_4 - e}$ as a topological minor. For the latter case, we note that a graph with exactly two independent cycles having $\boldsymbol{K_4 - e}$ as a topological minor must in fact be a graph with two cycles sharing one or more edges along with zero or more leaf edges (edges incident on vertices of degree one). Such a graph can be reduced to $\boldsymbol{K_4 - e}$ via edge-contractions. Using Theorem 2, we conclude that $\boldsymbol{K_4 - e}$ is totally satisfiable. However, this contradicts the result proved in Lemma 18. We therefore conclude that any connected graph $\boldsymbol{g}$ having exactly two independent cycles must have some element of M must as a topological minor.

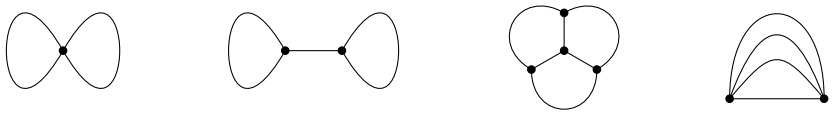
If $\boldsymbol{g}$ is connected and has exactly one cycle, then $\boldsymbol{g}$ can be reduced to $\boldsymbol{C_3}$ by edge-contractions. We know from Lemma 17 that $\boldsymbol{C_3}$ is totally satisfiable, and thus Theorem 2 implies that $\boldsymbol{g}$ is totally satisfiable, leading to a contradiction. Thus $\boldsymbol{g}$ cannot have exactly one cycle. If $\boldsymbol{g}$ is connected and

has no cycles, then $\boldsymbol{g}$ is a tree graph. We know from Lemma 19 that tree graphs are totally satisfiable, which leads to a contradiction. Thus $\boldsymbol{g}$ cannot have zero cycles.

Finally, suppose $\boldsymbol{g}$ is unsatisfiable is not connected. Then, some connected component $\boldsymbol{g'}$ of $\boldsymbol{g}$ is unsatisfiable. By our previous argument, we know that $\boldsymbol{g'}$ has some element of M. as a topological minor. This element of M must therefore also be a topological minor of $\boldsymbol{g}$. We have showed that any unsatisfiable graph connected or not, has an element of M as a topological minor. □

Using the correspondence between graph and their topological embedding into $\mathbb{R}^3$, we state a result (without proof) that is equivalent to the statement of Theorem 3.

Remark 3. *A graph $\boldsymbol{g}$ is unsatisfiable if and only if $\boldsymbol{g}$ when embedded into $\mathbb{R}^3$ has a subspace is homeomorphic as a topological space to one of the following cell complexes.*



3.5 Conclusion to our study of 2GraphSAT

The results of this chapter can all be found in our paper [11].

Given a set of graphs A, we can use the notation $\text{Forb}^*(A)$ to denote the set of all graphs not containing any element from A as a topological minor. In other words, this is the set obtained by forbidding elements of A as topological minors. Using this definition, we can summarize the results proved in this chapter more compactly as –

1. Set of all totally satisfiable simple graphs
   = $\text{Forb}^*(\boldsymbol{K_4}$, **butterfly graph**, **bow-tie graph**, $\boldsymbol{K_{1,1,3}})$.

2. Set of all totally satisfiable looped-multi-graphs = Language of 2GRAPH-SAT decision problem
   = $\text{Forb}^*(\boldsymbol{K_4}$, **double-loop graph**, **dumbbell graph**, $\boldsymbol{ab}^4)$.

3. There is a P-time algorithm for 2GRAPHSAT– though implementing this algorithm is not a practical thing to do.

3.5.1 Connection to the Robertson-Seymour graph minor theorem

In our main result, Theorem 3, we showed that given a simple graph, all 2Cnfs in its corresponding set are satisfiable if and only if four specific graphs are forbidden as topological minors of the original graph. Thus it is natural to ask if the forbidden minors theory of Robertson-Seymour [9]–[10] applies to our problem.

The Robertson-Seymour graph Theorem (RST) states that if a graph-family $\mathcal{F}$ is minor-closed, meaning that every minor of a graph in $\mathcal{F}$ is also in $\mathcal{F}$, then there are at most finitely-many forbidden minors of $\mathcal{F}$ – meaning a graph belongs to $\mathcal{F}$ if and only if it does not contain as a minor any of these forbidden graphs. In our case, the graph family of interest $\mathcal{F}$ is the set of totally satisfiable graphs. If totally satisfiable graphs are shown to be minor-closed then by RST we will be able to find a finite number of forbidden minors. However, totally satisfiable graphs are only closed under topological minoring and not under minoring. Hence RST does not apply and a finite list of topological minor obstructions is not guaranteed. Fortunately, our result implies that there are indeed only finitely-many obstructions.

3.5.2 Computational complexity of 2GraphSAT

There are interesting implications of our result on the computational complexity of 2SAT and 2GRAPHSAT. Since 2SAT is known to be in complexity class P, it would at first glance appear that we have made the problem worse: one can infer from our main theorem that the satisfiability of a given 2Cnf can be established by looking for four specific topological minors in its associated graphs. If these four topological minors are absent, then the 2Cnf must be satisfiable. If they are present, then the 2Cnf may or may not be satisfiable. This may appear to be a setback since the GRAPH MINOR decision problem is known to be NP-complete, implying that the GRAPH TOPOLOGICAL MINOR problem will also be NP-complete.

However, on closer inspection, one realizes that the decision problem of determining if a fixed graph is present as a topological minor can actually be resolved in polynomial time as a consequence of the graph minor theorem. Since our set of forbidden topological minors is finite we can therefore guar-

antee that the task of searching for them can be accomplished in polynomial time.

Thus, we have done "no worse" than the original 2SAT problem as both can still be solved in polynomial time. We emphasize however that as we have stated earlier, our goal in this chapter is the characterization of unsatisfiable 2Cnfs and graphs as opposed to questions of algorithmic efficiency for solving the satisfiability problem for 2Cnfs for which, in any case, efficient algorithms already exist.

Chapter 4

Local rewriting in graphs

We start by lifting any restrictions placed in the previous chapter on the Cnfs and graphs we were studying. A Cnf can now have clauses of any length, and a graph in general refers to a looped-multi-hypergraph. We take a brief look at 3GRAPHSAT in §4.1 and explore the need for graph local rewriting in §4.2. We then explain how graph rewriting works in §4.3 before stating and proving the main result of this chapter – the local graph rewriting theorem in §4.4. The close this chapter a discussion on some consequences of this theorem and an implementation of local rewriting in code (as a part of `graphsat`).

4.1 A brief look at 3GraphSAT

3GRAPHSAT is the graph decision problem that asks if a given looped-multi-hypergraph (also called a MHGraph in the next chapter) is totally satisfiable.

- **Instance**: Given a specific looped-multi-hypergraph $\boldsymbol{g}$.
- **Question**: Is every 3Cnf x such that $x \in \boldsymbol{g}$ satisfiable.

In Chapter 3 we enumerated a complete list of minimal unsatisfiable graphs and a proof that 2GRAPHSAT is in complexity class P. We might hope for a similar complete list for 3GRAPHSAT, and perhaps a different complexity class for 3GRAPHSAT. Unfortunately, this is not the case. The 3GRAPHSAT problem is more complicated for three main reasons –

1. The minimality of the 2GRAPHSAT list hinged on homeomorphisms (i.e. edge-subdivisions) preserving graph satisfiability. There is no single analogue of homeomorphisms, edge-subdivisions, and topological minoring in the case of hypergraphs.

2. Bruteforce checking of the satisfiability status of a graph is not a sustainable option for 3GRAPHSAT owing to the large number of Cnfs supported by a typical looped-multi-hypergraph.

3. In studying 2GRAPHSAT, we argued (in Lemma 16) that we can always reduce higher multiplicity edges down to 1. This was because a multiplicity 4 edge is always unsatisfiable, a multiplicity 3 edge forces an assignment on its vertices, and a multiplicity 2 edge forces an equivalence on its vertices. Such a complete result does not exist for hyperedges. The best we can do is to say that a multiplicity 8 hyperedge is always unsatisfiable, and a multiplicity 7 hyperedge forces assignments on its vertices.

These factors conspire to make 3GRAPHSAT a harder problem to solve, but also yield richer structures and relations between various multi-hypergraphs and their satisfiability statuses.

4.2 The need for local rewriting

We are looking for hypergraph analogue(s) of edge-subdivisions/edge-smoothing in order to define the corresponding notion of minimality in 3GRAPHSAT. This leads us naturally into local rewriting, i.e. operations where change the graph at a single vertex and its neighborhood. The idea is to find changes or rewrites that leave the satisfiability of a graph unchanged. For example, edge-subdivision can be though of as the rewrite $\boldsymbol{g} \wedge \boldsymbol{ab} \rightsquigarrow \boldsymbol{g} \wedge \boldsymbol{ac} \wedge \boldsymbol{bc}$, while the inverse operation of edge smoothing can be thought of as $\boldsymbol{g} \wedge \boldsymbol{ac} \wedge \boldsymbol{bc} \rightsquigarrow \boldsymbol{ab}$. Using local rewriting, we try to generalize this rule as well as the proof that it leave the satisfiability of a graph unchanged.

4.3 What is graph rewriting?

Graph rewriting concerns the technique of creating a new graph out of an original graph algorithmically. Formally, a graph rewriting system usually consists of a set of graph rewrite rules of the form $\boldsymbol{g_L} \rightsquigarrow \boldsymbol{g_R}$. The idea is that to apply the rule to a graph $\boldsymbol{g}$, we search it for the presence of $\boldsymbol{g_L}$ and replace the part that matches with $\boldsymbol{g}$ while leaving the rest unchanged $\boldsymbol{g_R}$.

Such rewrite rules can come in two forms, depending on the selection criterion for $\boldsymbol{g_L}$ –

1. **Local rules** – when a graph is rewritten at a particular vertex. In this case, all edges not adjacent to the vertex remain unaffected by the rewrite.

2. **Global rules** – when a graph is rewritten by searching for specific subgraphs that are isomorphic to $\boldsymbol{g_L}$.

We will concern ourselves with local rules in this chapter, while global rules will be handled by §6.2.

For local rules, we focus on the extended notion of "making assignments" detailed in §2.3.6. For a Cnf x, we assign at a literal l and denote it as $x[l]$. For a set of Cnfs $\boldsymbol{g}$, we assign at a vertex $\boldsymbol{v}$ and denote it as $\boldsymbol{g}[\boldsymbol{v}]$.

Literal assignments on Cnfs have two key properties that make them useful –

1. If x is a Cnf and l is a literal that is in the set of literals of x, then $x[l]$ is guaranteed to be smaller than x – it either has fewer clauses, or it has the same number of clauses but with those clauses having fewer literals in them.

2. x is satisfiable if and only if either one of $x[v]$ or $x[\overline{v}]$ are satisfiable.

Vertex assignments on sets of Cnfs also have similar properties –

1. If $\boldsymbol{g}$ is a set of Cnfs and $\boldsymbol{v}$ is a vertex in the vertexset of $\boldsymbol{g}$, then $\boldsymbol{g}[\boldsymbol{v}]$ will always have Cnfs that are smaller than the Cnfs in $\boldsymbol{g}$.

2. $\boldsymbol{g}[\boldsymbol{v}]$ is totally satisfiable if and only if $\boldsymbol{g}$ is totally satisfiable.

In the next section, we present an alternate expression for computing $\boldsymbol{g}[\boldsymbol{v}]$ that is easier to write when performing calculations, easier to code when programming it into a computer, and is a form that is used for proving several global graph rewrite rules.

We call this result the local rewriting theorem. The essence of this result is that even though satisfiability (both boolean and graph) is a global problem i.e. it is affected by the full structure of the Cnf, it can also be broken down into a series of local assignments in the case of Cnfs and a series of local

rewrites in the case of graphs. Given a graph, we can decompose it at one of its vertices of by computing this alternate expression in terms of the sphere and link of the graph without needing to step down to the level of Cnfs.

4.4 The local rewriting theorem

Theorem 4. *Let $\boldsymbol{g}$ be graph. We call $\boldsymbol{g}[\boldsymbol{v}]$ the local decomposition of $\boldsymbol{g}$ at vertex $\boldsymbol{v}$. This local decomposition is given by —*

$$\boldsymbol{g}[\boldsymbol{v}] \;=\; \text{sphere}(\boldsymbol{g},\boldsymbol{v}) \wedge \text{star}(\boldsymbol{g},\boldsymbol{v})[\boldsymbol{v}] \;=\; \bigcup_{\substack{\boldsymbol{g_i}\ \boldsymbol{h_i} : \textit{Graph} \\ \boldsymbol{g_i} \wedge \boldsymbol{h_i} = \text{link}(\boldsymbol{g},\boldsymbol{v})}} \text{sphere}(\boldsymbol{g},\boldsymbol{v}) \wedge (\boldsymbol{g_i} \vee \boldsymbol{h_i}) \tag{4.1}$$

Proof. First, we note that $\boldsymbol{g} = \text{sphere}(\boldsymbol{g},\boldsymbol{v}) \wedge \text{star}(\boldsymbol{g},\boldsymbol{v})$. Said differently, the sphere and the star together form a partition of the edgeset of a graph. Since we know that $\boldsymbol{g}[\boldsymbol{v}] \sim \boldsymbol{g}$, it suffices to show that $\text{star}(\boldsymbol{g},\boldsymbol{v})[\boldsymbol{v}] = \bigcup_{\substack{\boldsymbol{g_i}\ \boldsymbol{h_i} : \text{Graph} \\ \boldsymbol{g_i} \wedge \boldsymbol{h_i} = \text{link}(\boldsymbol{g},\boldsymbol{v})}} (\boldsymbol{g_i} \vee \boldsymbol{h_i})$.

Suppose $(x : \text{Cnf}) \in \text{star}(\boldsymbol{g},\boldsymbol{v})$. We note that there are four types of clauses in x —

- clauses that contain the literal v.
- clauses that contain the literal $\overline{v}$.
- clauses that contain neither v, nor $\overline{v}$ – actually this case is not possible since $x \in \text{star}(\boldsymbol{g},\boldsymbol{v})$.
- clauses that contain both v and $\overline{v}$ – actually this case is not possible since the edges corresponding to x can be incident on the vertex $\boldsymbol{v}$ only once.

Thus, we can partition x into x_1 containing clauses that contain v, and x_2 containing clauses that contain $\overline{v}$. Thus, $x[v] \vee x[\overline{v}] = (\top \wedge x_2[v]) \vee (x_1[\overline{v}] \wedge \top) = x_2[v] \vee x_1[\overline{v}]$. We note that $x_2[v]$ and $x_1[\overline{v}]$ belong to graphs $\boldsymbol{g_i}$ and $\boldsymbol{h_i}$ respectively for some i such that $\boldsymbol{g_i} \wedge \boldsymbol{h_i} = \text{link}(\boldsymbol{g},\boldsymbol{v})$. Thus, we have shown that $\text{star}(\boldsymbol{g},\boldsymbol{v})[\boldsymbol{v}] \subseteq \bigcup_{\substack{\boldsymbol{g_i}\ \boldsymbol{h_i} : \text{Graph} \\ \boldsymbol{g_i} \wedge \boldsymbol{h_i} = \text{link}(\boldsymbol{g},\boldsymbol{v})}} (\boldsymbol{g_i} \vee \boldsymbol{h_i})$.

Conversely, suppose $(y : \mathrm{Cnf}) \in \boldsymbol{g_i} \vee \boldsymbol{h_i}$ for some i, such that $\boldsymbol{g_i} \wedge \boldsymbol{h_i} = \mathrm{link}(\boldsymbol{g}, \boldsymbol{v})$. Thus, we can factor y as $y = y_1 \vee y_2$, for some Cnfs y_1 and y_2 such that $y_1 \in \boldsymbol{g_i}$ and $y_2 \in \boldsymbol{h_i}$. Consider then the Cnf y' given by $y' = (y_1 \vee v) \wedge (y_2 \vee \overline{v})$. Firstly, we observe that $y = y'[v] \vee y'[\overline{v}]$. Secondly, we note that $y' \in \mathrm{star}(\boldsymbol{g}, \boldsymbol{v})$ since the effect of disjuncting with v is to extend each clause in y_1 and y_2 by a literal in $\boldsymbol{v}$. Thus, we can write $y \in \mathrm{star}(\boldsymbol{g}, \boldsymbol{v})[\boldsymbol{v}]$. We have now shown that $\mathrm{star}(\boldsymbol{g}, \boldsymbol{v})[\boldsymbol{v}] \supseteq \bigcup_{\substack{\boldsymbol{g_i}\ \boldsymbol{h_i} : \mathrm{Graph} \\ \boldsymbol{g_i} \wedge \boldsymbol{h_i} = \mathrm{link}(\boldsymbol{g}, \boldsymbol{v})}} (\boldsymbol{g_i} \vee \boldsymbol{h_i})$. □

Corollary 5. *Let $\boldsymbol{g}$ be a Graph and let $\boldsymbol{v}$ be a vertex of $\boldsymbol{g}$. If $\boldsymbol{g}$ is unsatisfiable, then there exists a 2-partition $(\boldsymbol{h_1}, \boldsymbol{h_2})$ of $\mathrm{link}(\boldsymbol{g}, \boldsymbol{v})$ such that both $\mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge \boldsymbol{h_1}$ and $\mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge \boldsymbol{h_2}$ are unsatisfiable.*

Proof. Using the map $\boldsymbol{\gamma}$ defined in §2.3.2 on Equation 4.1 we can write,

$$
\begin{aligned}
&\boldsymbol{\gamma}(\boldsymbol{g}) \\
&= \boldsymbol{\gamma}(\boldsymbol{g}[\boldsymbol{v}]) \\
&= \boldsymbol{\gamma}\left(\bigcup_{\substack{\boldsymbol{h_i}\ \boldsymbol{h_j} : \mathrm{Graph} \\ \boldsymbol{h_i} \wedge \boldsymbol{h_j} = \mathrm{link}(\boldsymbol{g}, \boldsymbol{v})}} \mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge (\boldsymbol{h_i} \vee \boldsymbol{h_j}) \right), && \text{(Lemma 3)} \\
&= \bigwedge_{\substack{\boldsymbol{h_i}\ \boldsymbol{h_j} : \mathrm{Graph} \\ \boldsymbol{h_i} \wedge \boldsymbol{h_j} = \mathrm{link}(\boldsymbol{g}, \boldsymbol{v})}} \boldsymbol{\gamma}\Big(\mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge (\boldsymbol{h_i} \vee \boldsymbol{h_j})\Big) && \text{(Lemma 2)} \\
&\longleftarrow \bigwedge_{\substack{\boldsymbol{h_i}\ \boldsymbol{h_j} : \mathrm{Graph} \\ \boldsymbol{h_i} \wedge \boldsymbol{h_j} = \mathrm{link}(\boldsymbol{g}, \boldsymbol{v})}} \boldsymbol{\gamma}\Big(\mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge \boldsymbol{h_i}\Big) \ \vee\ \boldsymbol{\gamma}\Big(\mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge \boldsymbol{h_j}\Big) && \text{(Lemma 9)}
\end{aligned}
$$

If $\boldsymbol{g}$ is unsatisfiable, then we can infer the existence of some 2-partition $(\boldsymbol{h_1}, \boldsymbol{h_2})$ of $\mathrm{link}(\boldsymbol{g}, \boldsymbol{v})$ such that both $\mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge \boldsymbol{h_1}$ and $\mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge \boldsymbol{h_2}$ are unsatisfiable. □

Corollary 6. *Let $\boldsymbol{g}$ be a Graph and let $\boldsymbol{v}$ be a vertex of $\boldsymbol{g}$. If either one of $\mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge \boldsymbol{h_1}$ or $\mathrm{sphere}(\boldsymbol{g}, \boldsymbol{v}) \wedge \boldsymbol{h_2}$ is totally satisfiable for every 2-partition $(\boldsymbol{h_1}, \boldsymbol{h_2})$ of $\mathrm{link}(\boldsymbol{g}, \boldsymbol{v})$, then $\boldsymbol{g}$ itself is totally satisfiable.*

Proof. This is the contra-positive of Corollary 5. □

4.5 Consequences of local graph rewriting

Given a graph $\boldsymbol{g}$, local graph rewriting gives us a tool to break its satisfiability problem into a series of partial satisfiability graphs. The trouble with this result is that the rewrite is a uni-directional implication and not a bi-directional equality. As a result, it is possible to prove that a graph is totally satisfiable, using this method, but not that it is unsatisfiable.

This short-coming can however be mitigated somewhat by our choice of vertex $\boldsymbol{v}$ at which we choose to rewrite. We will prefer using a vertex of low degree to as to ensure a smaller size of $\text{link}(\boldsymbol{g}, \boldsymbol{v})$. Picking a vertex of degree d results in a link of size d. The number of nonempty partitions of the link (we only care about nonempty partitions) because the empty partition terms do not affect satisfiability) are given by $2^{d-1} - 1$. Seeing as d appears in the exponent of this count, we choose a vertex of lowest possible degree in d.

The other major use of local graph rewriting is that it can be applied recursively, first to $\boldsymbol{g}$ at a a vertex $\boldsymbol{v_1}$, then to the term in the formula $\text{sphere}(\boldsymbol{g}, \boldsymbol{v_1}) \wedge \text{link}(\boldsymbol{h_1}, \boldsymbol{v_1})$, and then to a vertex $\boldsymbol{v_2}$ of this resultant term, and so on .... This results in ever smaller graphs, thus giving us an algorithm for proving that a graph $\boldsymbol{g}$ is totally satisfiable by deferring the calculation to smaller parts of the graph.

We use local rewriting to prove several different graph reduction rules in §6.2. We also enumerate some standard graph disjunctions in §6.1.

4.6 Implementation of local graph rewriting in code

We will introduce the `graphsat` Python package in the next chapter and provide details on its various modules and functions. We mention it here because we will use some of its functions in §6.2 to construct graphs (i.e. terms of type `mhgraph.MHGraph`) and compute their disjunction (using the function `graph_or`).

Below we include the docstring of the function, informing us what exactly the function does, followed by its implementation as a code-snippet. The implementation uses other functions defined in the package like `mhgraph.sphr`, `operations.graph_or`, and `compute_all_two_partitions_of_link`. We will not detail each of these subsidiary functions here. We leave it instead to the

interested reader to look at `graphsat`'s source code for more details.

```
(python3.9) (graph_rewrite.py) <<local-rewrite-docstring>>
"""Locally rewrite at ``vertex`` assuming that the graph is only partially known.

This function only affects edges incident on ``vertex``, assuming that
``mhg`` only represents a part of the full graph.

The result is a dictionary of Cnfs grouped by their MHGraphs.

"""
```

```
(python3.9) (graph_rewrite.py) <<local_rewrite>>
def local_rewrite(mhg: mhgraph.MHGraph, vertex: mhgraph.Vertex) -> dict[Any, Any]:
    <<local-rewrite-docstring>>

    # Compute the sphere of mhg at vertex
    sphr: mhgraph.MHGraph = mhgraph.sphr(mhg, vertex)

    # All 2-partitions of link of mhg at vertex.
    two_partitions: Iterator[tuple[list[mhgraph.MHGraph], list[mhgraph.MHGraph]]]
    two_partitions = compute_all_two_partitions_of_link(mhg, vertex)

    # Initialize resultant Cnf set.
    resultant_cnfs: set[cnf.Cnf] = set()

    # Loop over all 2-partitions of the link.
    for hyp1, hyp2 in two_partitions:
        # Disjunction of the parts.
        hyp1_or_hyp2: set[cnf.Cnf] = op.graph_or(hyp1, hyp2)
        # Conjunction with the sphere
        sphr_hyp: set[cnf.Cnf] = op.graph_and(sphr, hyp1_or_hyp2)
        # Add result to set of resultant Cnfs.
        resultant_cnfs |= sphr_hyp

    # Group Cnfs by the Graphs to which they belong.
    return graph_collapse.create_grouping(set(resultant_cnfs))
```

Chapter 5

`graphsat` Python package

We introduce a Python package called `graphsat`[1] which recognizes clauses, Cnfs, graphs, hypergraphs, and multi-hypergraphs. The package implements local graph-rewriting, graph-satchecking, calculation of graph disjunctions, as well as checking of new reduction rules.

This library is written in Python version 3.9, and is available [12] under the GNU GPLv3.0 open license. It is set to be released on the Python Packaging Index (PyPI) as an open-source scientific package written in the literate programming style. I specifically chose to write this package as a literate program, despite the verbosity of this style, with the goal to create reproducible computational research and ensure that all the computations in this thesis are repeatable and the code is reusable.

Currently, graphsat implements the following algorithms –

- For formulae in conjunctive normal forms (Cnfs), it implements variables, literals, clauses, Boolean formulae, and truth-assignments. It includes an API for reading, parsing and defining new instances.
- For graph theory, the package includes graphs with self-loops, edge-multiplicities, hyperedges, and multi-hyperedges. It includes an API for reading, parsing and defining new instances.
- For satisfiability of Cnfs and graphs, it contains a bruteforce algorithm, an implementation that uses the open-source sat-solver PySAT [13], and an implementation using the MiniSAT solver [14].
- Additionally, for graph theory, the library also implements vertex maps, vertex degree, homeomorphisms, homomorphisms, subgraphs, and isomorphisms. This allows us to encode local rewriting rules as well as parallelized grid-based searching for forbidden structures.

[1] GitHub link: `https://github.com/vaibhavkarve/graphsat`

- Finally, `graphsat` has a tree-based recursive reduction algorithm that uses known local-rewrite rules as well as algorithms for checking satisfiability invariance of proposed reduction rules.

`graphsat` has been written in the functional-programming style whose principles can be summarized as —

- Avoid classes as much as possible. Prefer defining functions instead.
- Write small functions and then compose/map/filter them to create more complex functions (using the `functools` library).
- Use lazy evaluation strategy whenever possible (using the `itertools` library).
- Add type hints wherever possible (checked using the `mypy` static type-checker).
- Add unit-tests for each function (checked using the `pytest` framework).

5.1 Overview of the package

The package consists of several different modules.

1. Modules that act only on Cnfs –

`cnf.py`	Constructors and functions for sentences in conjunctive normal form (Cnf).
`cnf_simplify.py`	Functions for simplifying Cnfs, particularly $(a \vee b \vee c) \wedge (a \vee b \vee \overline{c}) \rightsquigarrow (a \vee b)$.
`prop.py`	Functions for propositional calculus – conjunction, disjunction and negation.

2. Modules that act only on graphs –

`graph.py`	Constructors and functions for simple graphs.
`mhgraph.py`	Constructors and functions for Loopless-Multi-Hyper-Graphs
`morphism.py`	Constructors and functions for Graph and MH-Graph morphisms.

3. Modules concerning SAT and GRAPHSAT–

`sat.py`	Functions for sat-checking Cnfs, Graphs, MH-Graphs.
`sxpr.py`	Functions for working with s-expressions.
`operations.py`	Functions for working with graph-satisfiability and various graph parts.

4. Modules that implement and compute local graph rewriting, rule reduction etc.

`graph_collapse.py`	Functions for collapsing a set of Cnfs into compact graphs representation.
`graph_rewrite.py`	An implementation of the Local graph rewriting algorithm.

5. Finally, the test suite for each module is located in the `test/` folder. This includes the following files –

`test_cnf.py`	`test_cnf_sumplify.py`	`test_graph.py`
`test_morphism.py`	`test_graph_collapse.py`	`test_prop.py`
`test_operations.py`	`test_graph_rewrite.py`	`test_sxpr.py`
`test_sat.py`	`test_mhgraph.py`	

In the next section, we lay out a description and implementation of the `cnf.py` module (and its corresponding test module `test_cnf.py`) in the literate programming style. A similar description could be written for every one of the modules in `graphsat`, though this would make this thesis quite long, unwieldy and dry.

5.2 Introduction to `cnf.py`

This module defines Cnf (boolean formulas in conjunctive normal form) as a python class and creates an API of functions for defining Cnfs and computing with them. We define variables, literals, clauses, Cnfs and assignements in §5.3 and functions that manipulate them in §§5.4–5.6.

The end-goal of this module is to equip us with a Cnf-calculator using python syntax. We illustrate this using the following example. Consider the following statement — assigning the variable x_2 to true and x_3 to false in $(x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee x_5)$ gives us $x_1 \vee x_4$.

We can verify this statement by carrying out the following calculation.

$$\begin{aligned}(1\bar{2}4, 1\bar{3}5)\left[2, \bar{3}\right] &= (1\bot 4, 1\bar{3}5)\left[\bar{3}\right] \\ &= (14, 1\bar{3}5)\left[\bar{3}\right] \\ &= 14, 1\top 5 \\ &= 14, \top \\ &= 14.\end{aligned}$$

The same calcultion can also be carried out in python using the functions defined in this module. The output is included below the codeblock.

```
(python3.9) (scratch) <<example>>
from cnf import assign, Assignment, cnf, Cnf, TRUE, FALSE

x : Cnf = cnf([[1, -2, 4], [1, -3, 5]])
a : Assignment = {2: TRUE, 3: FALSE}

return assign(x, a)
```

Output:

```
Cnf({Clause({1, 4})})
```

All codeblocks in this chapter have been formatted in a manner similar to the one shown above. The top-margin of the codeblock starts by indicating the programming language used in the block (`python3.9` in the example). Shell scripts for example will have `sh` in that position instead.

The second of three slots represents the file of which this codeblock is a part. For example, a codeblock that has `cnf.py` in this slot will be a part of the `cnf.py` file. This helps with persistence of variables and definitions defined the file – codeblocks in `cnf.py` can refer to other variables from different codeblocks that are also part of `cnf.py`. We use the name `scratch` to represent a throw-away file. In the example above, the `scratch` label means that this was a one-time calculation that we do not wish to store in a file.

The third label in the top-margin of the codeblock represents a unique identifier of the codeblock (`<<example>>` in the example above). This identifier is used to keep track of the codeblock we are referring to as well as to refer to it inside other codeblocks (via the process called *tangling* in the literate programming terminology).

5.2.1 Definitions

We start by summarizing the relevant definitions in top-down order. Details for each definition and accompanying implementations can be found in the relevant subsections.

- A Cnf is a conjunction of one or more Clauses. It is implemented as a subclass of frozensets of Clauses in §5.3.5.
- A Clause is a disjunction of one or more Lits. It is implemented as a subclass of frozensets of Lits in §2.2.3.
- A Lit is either a Variable, the negation of a Variable, or a Bool. These are usually called *Literals* in CS literature but this sounds too close to `typing.Literal` which is already defined in python and refers to a different, unrelated object. It is implemented as a subtype of python's built-in data type `int` in §5.3.2.
- A Variable is any element from a fixed countably infinite set of symbols that can be assigned the value of Bool in a boolean expression. We implement it as a subtype of `int`, with the additional restriction that only positive `int` values are allowed. The implementation is explained in §2.2.1.
- A Bool is a set of two reserved the symbols called `TRUE` and `FALSE` (not to be confused with `bool`, `True` and `False`). Justification of the need for defining such a type distinct from `bool` can be found in §5.3.3. That section also details the implementation of Bool as a subclass of Lits.

5.2.2 Overview of `cnf.py`

(python3.9) (cnf.py) <<cnf.py-full>>

```
#!/usr/bin/env python3.9
"""Constructors and functions for sentences in conjunctive normal form (Cnf)."""

# Imports
# =======
<<imports>>

# Classes and Types
# =================
<<Variable>>
<<Lit>>
<<Bool>>
<<TRUE_and_FALSE>>  # Instances of Bool, needed to define Clause and Cnf
<<Clause>>
<<Cnf>>
<<Assignment>>

# Constructor Functions
# =====================
<<variable>>
<<lit>>
<<clause>>
<<cnf>>

# Helpful Constants
# =================
<<constants>>  # not documented, for internal use only

# Basic Functions
# ===============
<<neg>>
<<absolute_value>>
<<lits>>

# Functions for Simplification
# ============================
<<tautologically_reduce_clause>>
<<tautologically_reduce_cnf>>

# Functions for Assignment
# ========================
```

```
<<assign_variable_in_lit>>
<<assign_variable_in_clause>>
<<assign_variable_in_cnf>>
<<assign>>

if __name__ == '__main__':
    <<standalone_run_commands>>
```

5.2.3 Imports and dependencies

We start by importing the required packages. We will be using the `typing` and `collections.abc` libraries to add type-hints throughout our code. This acts as a kind of documentation while also catching type-errors when we use it in conjunction with an external static type-checker like `mypy`.

(python3.9) (cnf.py) <<imports>>

```
from collections.abc import Set, Callable, Collection, Iterator, Mapping
import functools as ft
from typing import final, Final, Mapping, NewType, Union

from loguru import logger
```

5.3 Types and their constructor functions

We now implement each of the definitions given in §5.2.1 as types. For each type we will also define a unique constructor function.

5.3.1 Variables

A *Variable* is any element from a fixed countably infinite set of symbols that can be assigned the value of Bool in a boolean expression. We use python's built-in `int` type as our countably infinite set. Using `typing.NewType`, we can declare Variable to be a subtype of `int`. This means that any `int` n can be cast into a term of type Variable as `Variable(n)`. In type-theoretic notation, we can write the relation between Variable and `int` as Variable $\sqsubset$ `int`.

```
(python3.9) (cnf.py) <<Variable>>
Variable = NewType('Variable', int)
Variable.__doc__ = """`Variable` is a subtype of `int`."""
```

In fact, we only want the positive integers to denote Variables. However, this restriction cannot be encoded when defining the type itself and so we also write a constructor function `variable` to check for positivity.

```
(python3.9) (cnf.py) <<variable>>
def variable(positive_int: int) -> Variable:
    <<variable-docstring>>
    if positive_int <= 0:
        raise ValueError('Variable must be a positive integer.')
    return Variable(positive_int)
```

We note that this function is injective and idempotent.

```
(python3.9) (test/test_cnf.py) <<test_variable>>
def test_variable():
    assert variable(1) == 1
    assert variable(11) == 11
    assert variable(variable(2)) == variable(2)  # Test for idempotence
    pytest.raises(ValueError, variable, 0)
```

5.3.2 Lits

A *Lit* is either a Variable, its negation, or a Bool. We have not yet defined `Bool`, but for now we can simply assume that it is a special type of `int` (justification for this can be found in §5.3.3). Since both `Variable` and `Bool` are subtypes of `int`, we can define `Lit` in a similar. The only caveat to this is that we want `Lit` to act as a subscriptable argument later, allowing us to define `Clause` as a being a subtype/subclass of frozen sets of `Lit`. To make `Lit` subscriptable, we define it as a **subclass** of `int` instead of a **subtype**.

```
(python3.9) (cnf.py) <<Lit>>
class Lit(int):
    """`Lit` is a subclass of `int`. It has no other special methods."""
```

Next, we define a constructor function for `Lit`, called `lit`. On `Bools`, we want the constructor function to act as the identity function, while on an integer argument, it should check that the integer is nonzero (thus ensuring

that it is either a `Variable` or its negation). To implement such a piecewise definition we use `functools.singledispatch`. This allows us to pick the correct variant of `lit` based on the type of its first (and only) argument.

```
(python3.9) (cnf.py) <<lit>>
@ft.singledispatch
def lit(int_or_bool: Union[int, Bool]) -> Lit:
    <<lit-docstring>>
    raise TypeError('Lit must be either Bool or int.')

@lit.register
<<lit_bool>>

@lit.register
<<lit_int>>
```

where we have the following functions that are single-dispatched based on the type of the first argument.

```
(python3.9) (cnf.py) <<lit_bool>>
def lit_bool(arg: Bool) -> Lit:
    """Return as is because Bool is already a subtype of Lit."""
    return arg
```

```
(python3.9) (cnf.py) <<lit_int>>
def lit_int(arg: int) -> Lit:
    """Cast to Lit."""
    if arg != 0:
        return Lit(arg)
    raise ValueError('Lit must be a nonzero integer.')
```

As mentioned in its docstring below, `lit` is an idempotent function. To ensure this, we write the following set of tests. (While we have not yet defined `Bool.TRUE` and `Bool.FALSE`, their meaning in the following assert statements should be obvious from context).

```
(python3.9) (cnf.py) <<lit-docstring>>
"""Constructor-function for Lit type.

By definition, a `Lit` is in the set ℤ \ {0} ∪ {`TRUE`, `FALSE`}.
This function is idempotent.
"""
```

```
(python3.9) (test/test_cnf.py) <<test_lit>>
def test_lit():
    assert lit(1) == 1
    assert lit(-1) == -1
    assert lit(11) ==  11
    assert lit(TRUE) == TRUE
    assert lit(FALSE) == FALSE
    assert lit(lit(2)) == lit(2)          # Test for idempotence.
    assert lit(lit(TRUE)) == lit(TRUE)    # Test for idempotence.
    assert lit(lit(FALSE)) == lit(FALSE)  # Test for idempotence.

    pytest.raises(ValueError, lit, 0)
```

5.3.3 Bools

Bool refers to two reserved symbols – `TRUE` and `FALSE`. Note that python already has a `bool` type (with elements `True` and `False`). However, `bool` is actually implemented as a subclass of the `int` type. This has some unintended consequences when considering data structures that contain both integers and elements due to the fact that `True` and `False` are just aliases for the integers `1` and `0` respectively. This means that python will always simplify the set `{1, 2, 3, True}` to `{1, 2, 3}` since it treats `True` as a duplicate of `1`. This is a big problem and the primary reason why we define Bools to be distinct from `bool`.

We want Bool to be a type that is a subclass of `int` and not just a subtype. In fact, since all Bools are Lits, we also want Bool to be a subtype of Lit. Thus we define Bool as shown below with special methods `__str__`, `__hash__`, and `__eq__`. Additionally, we define the `__repr__` method to be identical to the `__str__` method. All other methods of Bool are inherited from the `int` parent type.

```
(python3.9) (cnf.py) <<Bool>>
@final
class Bool(Lit):
    """`Bool` is a subclass of `Lit`.

    It overrides the ``__str__``, ``__repr__``, ``__hash__`` and ``__eq__``
    methods inherited from :obj:`int` (and from Lit).
    """
    <<Bool-str>>
```

```
    __repr__ = __str__
    <<Bool-hash>>
    <<Bool-eq>>
```

We now give detailed definitions for each of Bool's special methods. The `__str__` method allows us to change the value displayed when we print the a Bool. Even though python will internally store Bools as `0` and `1`, when asked, it will still print the values as `<Bool: TRUE>` and `<Bool: FALSE>` respectively. Any other numeric value raises a `ValueError`.

```
(python3.9) (cnf.py) <<Bool-str>>
def __str__(self) -> str:
    """Bool(0) and Bool(1) are treated as constant values labeled FALSE and TRUE."""
    if self.__int__() == 0:
        return '<Bool: FALSE>'
    if self.__int__() == 1:
        return '<Bool: TRUE>'
    raise ValueError('In-valid Bool value encountered.')
```

To prevent python from simplifying `{1, TRUE}` to `{1}`, we modify its hash value. We set the hash of `TRUE` to `hash(str(TRUE))`, which simplifies to 4422589586725646474. Similarly, the hash of `FALSE` is set to 6211323488567046769.

```
(python3.9) (cnf.py) <<Bool-hash>>
def __hash__(self) -> int:
    """Ensure that ``hash(Bool(n))`` doesn't clash with ``hash(n)``."""
    return hash(str(self))
```

Moreover, we redefine the `__eq__` method to now use hash values as a means to establish equality. This makes `Bool(n)` unequal to `n`.

```
(python3.9) (cnf.py) <<Bool-eq>>
def __eq__(self, other: object) -> bool:
    """Make ``Bool(n)`` unequal to ``n``."""
    return hash(self) == hash(other)
```

Finally, having defined the class of Bool, we now define exactly two instances of Bool. In type-theoretic language, we can say that Bool is an enumerated type (since we enumerate all its instances explicitly). Furthermore, to prevent the user from accidentally (or maliciously) redefining the values of `TRUE` and `FALSE`, we use typing.Final to disable overriding and re-assigning

of these constants. We also mark the class definition of Bool itself with the @typing.final decorator in order to prevent this class from being sub-classed by the user.

```
(python3.9) (cnf.py) <<TRUE_and_FALSE>>
#: ``TRUE = Bool(1)``, a final instance of Bool.
TRUE: Final = Bool(1)
#: ``FALSE = Bool(0)``, a final instance of Bool.
FALSE: Final = Bool(0)
```

Lastly, to check that the Bool values indeed possess all the desired properties, we add unit-tests.

```
(python3.9) (test/test_cnf.py) <<test_Bool>>
def test_Bool():
    assert TRUE == TRUE               # check for consistency
    assert TRUE not in [1, 2, 3]      # check that eq is working
    assert TRUE in {1, 2, 3, TRUE}    # check that eq and hash are working
    assert isinstance(TRUE, Bool)     # check that python recognizes the class
    assert not isinstance(TRUE, bool) # check that Bool and bool are district
```

5.3.4 Clauses

Next, we wish to define a *Clause* to be a container of Lits. The choice of container is driven by three properties we desire in Clauses –

1. We do not care for repetitions in the clause. This is in keeping with the identity $x \vee x = x$.

2. We also do not care about the ordering of elements in a clause. This is in keeping with the fact that disjunction is commutative.

3. Looking ahead, we will want to put clauses inside other containers to form Cnf. Thus it is desirable that clauses be hashable (and therefore immutable) objects.

These properties suggest that we should have our containers be frozensets (python's unordered, immutable, hashable sets). We define Clause as a subclass instead of just a subtype, allowing us to also override its `__str__` method to offer more human-readable printouts.

```
(python3.9) (cnf.py) <<Clause>>
class Clause(frozenset[Lit]):  # pylint: disable=too-few-public-methods
    """`Clause` is a subclass of `frozenset[Lit]`."""
    def __str__(self) -> str:
        """Pretty print a Clause after sorting its contents."""
        sorted_clause: list[Lit] = sorted(self, key=absolute_value)
        return '(' + ','.join(map(str, sorted_clause)) + ')'
```

Having defined a class, we next define a constructor function `clause` for said class. The function ensures that a Clause is always a nonempty collection of Lits. The function is also guaranteed to be idempotent.

```
(python3.9) (cnf.py) <<clause>>
def clause(lit_collection: Collection[int]) -> Clause:
    <<clause-docstring>>
    if not lit_collection:
        raise ValueError(f'Encountered empty input {list(lit_collection)}.')
    return Clause(frozenset(map(lit, lit_collection)))
```

Finally, we add in the doc-string describing the function we have just defined and we add some unit-tests to for the function as well.

```
(python3.9) (cnf.py) <<clause-docstring>>
"""Constructor-function for Clause type.

By definition, a `Clause` is a nonempty frozenset of Lits. This function is idempotent.

Args:
   lit_collection (:obj:`Collection[int]`): a nonempty collection of ints.

Return:
   Check that each element in the collection satisfies axioms for being a Lit and
      then cast to Clause.

Raises:
   ValueError: if ``lit_collection`` is an empty collection.

"""
```

```
(python3.9) (test/test_cnf.py) <<test_clause>>
def test_clause():
    assert clause([1, 2, -3]) == {1, 2, -3}    # check for correct type
    assert clause([1, -1, 2]) == {1, -1, 2}    # +ve and -ve Lits treated as distinct
    assert clause([TRUE]) == {TRUE}            # TRUE can be part of a Clause
```

```
    assert clause([FALSE]) == {FALSE}          # FALSE can be part of a Clause
    assert clause([1, TRUE]) == {1, TRUE}      # TRUE ≠ 1 in a Clause
    assert clause([-1, FALSE]) == {-1, FALSE}  # FALSE ≠ -1 in a Clause
    assert clause([1, TRUE, FALSE]) == {1, TRUE, FALSE}  # TRUE ≠ FALSE

    # Tests for idempotence
    assert clause(clause([1, 2, -3])) == clause([1, 2, -3])
    assert clause(clause([TRUE])) == clause([TRUE])
    assert clause(clause([FALSE])) == clause([FALSE])

    pytest.raises(ValueError, clause, [])
```

5.3.5 Cnfs

We define a *Cnf* to be a container of Clauses. The choice of container is driven by three properties we desire in Cnfs –

1. We do not care for repetitions in the Cnf. This is in keeping with the identity $x \wedge x = x$.
2. We also do not care about the ordering of elements in a Cnf. This is in keeping with the fact that conjunction is commutative.
3. Looking ahead, we will want to put Cnfs inside other containers when performing calculations. Thus it is desirable that Cnfs be hashable (and therefore immutable) objects.

These properties are exactly the ones we desired for Clauses, so we similarly select frozensets as our containers. We define Cnf as a subclass instead of just a subtype, allowing us to also override its `__str__` method to offer more human-readable printouts.

(python3.9) (cnf.py) <<Cnf>>

```
class Cnf(frozenset[Clause]):  # pylint: disable=too-few-public-methods
    """`Cnf` is a subclass of `frozenset[Clause]`."""

    def __str__(self) -> str:
        """Pretty print a Cnf after sorting its sorted clause tuples."""
        sorted_cnf: list[Clause]
        sorted_cnf = sorted(self, key=lambda clause_: sum([lit < 0 for lit in clause_]))
        sorted_cnf = sorted(sorted_cnf, key=len)
```

```
        cnf_tuple: Iterator[str] = map(str, map(clause, sorted_cnf))
        return ''.join(cnf_tuple)
```

We define a constructor function `cnf` for this class. This function ensures that a Cnf is always a nonempty collection of Clauses. This constructor function is also guaranteed to be idempotent.

```
(python3.9) (cnf.py) <<cnf>>
def cnf(clause_collection: Collection[Collection[int]]) -> Cnf:
    <<cnf-docstring>>
    if not clause_collection:
        raise ValueError(f'Encountered empty input {list(clause_collection)}.')
    return Cnf(frozenset(map(clause, clause_collection)))
```

Finally, we add in a docstring and some unit tests for the constructor function.

```
(python3.9) (cnf.py) <<cnf-docstring>>
"""Constructor-function for Cnf type.

By definition, a `Cnf` is a nonempty frozenset of Clauses. This func is idempotent.

Args:
   clause_collection (:obj:`Collection[Collection[int]]`): a nonempty collection
      (list, tuple, set, frozenset) of nonempty collections of integers or Bools.

Return:
   Check that each element in the collection satisfies axioms for being a Clause
      and then cast to Cnf.

Raises:
   ValueError: if ``clause_collection`` is an empty collection.
"""
```

```
(python3.9) (test/test_cnf.py) <<test_cnf>>
def test_cnf():
    fs = frozenset  # a temporary alias for frozenset

    # Generic example use-case
    assert cnf([[1, 2, -3], [-4, 5]]) == {fs([1, 2, -3]), fs([-4, 5])}

    # Test for removing repetitions
```

```
    assert cnf([[1, 1, -1], [1, -1]]) == {fs([1, -1])}

    # Cnf with TRUE and FALSE inside a Clause
    assert cnf([[1, 2, TRUE], [3, FALSE]]) == {fs([1, 2, TRUE]), fs([3, FALSE])}

    # Cnf with a Bool-only Clause
    assert cnf([[1, 2, 3], [FALSE], [TRUE]]) \
        == {fs([1, 2, 3]), fs([FALSE]), fs([TRUE])}
    assert cnf([[TRUE], [TRUE, TRUE]]) == {fs([TRUE])}

    # Single-Lit-single-Clause Cnf
    assert cnf([[1]]) == {fs([1])}
    assert cnf([[-1]]) == {fs([-1])}

    # Test for idempotence.
    assert cnf(cnf([[1, 2, 3], [-4, 5]])) == cnf([[1, 2, 3], [-4, 5]])

    pytest.raises(ValueError, cnf, [])
    pytest.raises(ValueError, cnf, [[]])
```

5.3.6 Assignments

A *truth-assignment* is a map that assigns true/false values to variables and literals. The functions that implement these assignments will be explained in §5.6, but here we specify the type for assignments.

We make a distinction between *full assignments*, which assign all (countably-many) variables to true/false, and *partial assignments*, which assign only some finite subset of variables to true/false. The type of full assignments is best encoded as a function of type $\mathbb{N} \to$ `Bool`, while the partial assignments are best encoded as a dictionary of `Variable` keys and `Bool` values.

For computations, we will be more concerned with partial assignments, so we denote their type by `Assignment` and define it to be a type alias for Mapping from Variables to Bools called `Assignment`. Note that collections.abc.Mappings are the abstract base class for python's `dict` container type.

(python3.9) (cnf.py) <<Assignment>>
```
Assignment = Mapping[Variable, Bool]  # defines a type alias
```

5.3.7 Helpful constants

We introduce some constants for ease of writing functions in the sections that follow. However, it should be noted that these constants are intended for internal use only, and are not to be called by a user of this module. To enforce this restriction, we name these constants starting with an underscore to prevent them from being imported in other modules. Further, we wrap their types in the `typing.Final` type construct to prevent these values from being re-assigned or overridden.

(python3.9) (cnf.py) <<constants>>

```
_TRUE_CLAUSE  : Final[Clause] = clause([TRUE])
_FALSE_CLAUSE : Final[Clause] = clause([FALSE])
_TRUE_CNF     : Final[Cnf]    = cnf([_TRUE_CLAUSE])
_FALSE_CNF    : Final[Cnf]    = cnf([_FALSE_CLAUSE])
```

5.4 Basic functions

We define some basic functions for manipulating terms of one type into another –

- In §5.4.1, we will define a function that negates Lits. This is a function that ties a Lit l to its negation $\neg l$.
- In §5.4.2, we will define a function that computes the absolute value of a Lit by returning its underlying Variable (but cast as a Lit). This is the function that ties the Lits $\{l, \neg l\}$ to their underlying Variable, denoted $|l|$.
- In §5.4.3, we will define a function that returns a set of all Lits in a Cnf. This function will allows us to iterate over, or assign to, every Lit in the Cnf.

5.4.1 Negation of literals

The negation function sends a literal l to $\neg l$. For Bools, the negation sends `TRUE` to `FALSE`, and `FALSE` to `TRUE`.

One important property of the negation function is that it is an involution since $\neg(\neg l) = l$, for any literal l. It should also be noted that our negation function has a severely restricted scope compared to the mathematical operation of $(\neg)$, since our negation function can only act on Lits and not on Clauses or Cnfs. The reason for this is that the negation of a Clause is not in general a Clause, and the negation of a Cnf is not in general a Cnf.

```
(python3.9) (cnf.py) <<neg>>
def neg(literal: Lit) -> Lit:
    <<neg-docstring>>
    if literal == TRUE:
        return FALSE
    if literal == FALSE:
        return TRUE
    return lit(-literal)
```

Finally, we add a docstring and some unit tests for the negation function.

```
(python3.9) (cnf.py) <<neg-docstring>>
"""Negate a Lit.

This function is an involution.

Args:
   literal (:obj:`Lit`): a Lit formed from a nonzero integer or from a Bool.

Return:
   Return the Lit cast from the negative of ``literal``. If ``literal`` is of
      type Bool, then return ``TRUE`` for ``FALSE``, ``FALSE`` for ``TRUE``.

"""
```

```
(python3.9) (test/test_cnf.py) <<test_neg>>
def test_neg():
    assert neg(lit(1))  == lit(-1)
    assert neg(lit(-1)) == lit(1)
    assert neg(lit(23)) == lit(-23)
    assert neg(TRUE)    == FALSE
    assert neg(FALSE)   == TRUE

    # Test for involution.
    assert neg(neg(lit(1)))  == lit(1)
    assert neg(neg(lit(-1))) == lit(-1)
```

```
    pytest.raises(ValueError, neg, 0)
```

5.4.2 Absolute value of literals

The *absolute value* of a literal l is denoted $|l|$ and is defined as the literal formed by the underlying variable. For Bools, the absolute value of a Bool is itself.

```
(python3.9) (cnf.py) <<absolute_value>>
def absolute_value(literal: Lit) -> Lit:
    <<absolute_value-docstring>>
    if isinstance(literal, Bool):
        return literal
    return lit(abs(literal))
```

We then add a docstring and some unit tests for the function.

```
(python3.9) (cnf.py) <<absolute_value-docstring>>
"""Unnegated form of a Lit.

This function is idempotent.

Args:
   literal (:obj:`Lit`): a Lit formed from a nonzero integer.

Return:
   Check that ``literal`` is not of type Bool and then return the absolute value
      of ``literal``. If it is of type Bool, then return ``literal`` as is.
"""
```

```
(python3.9) (test/test_cnf.py) <<test_absolute_value>>
def test_absolute_value():
    assert absolute_value(lit(1)) == lit(1)
    assert absolute_value(lit(-1)) == lit(1)

    # Test for idempotence.
    assert absolute_value(absolute_value(lit(1))) == absolute_value(lit(1))
    assert absolute_value(absolute_value(lit(-1))) == absolute_value(lit(-1))

    # Test for Bools
    assert absolute_value(TRUE) == TRUE
```

```
    assert absolute_value(FALSE) == FALSE

    pytest.raises(ValueError, absolute_value, 0)
```

5.4.3 Literals in a Cnf

This function returns the set of Lits in a given Cnf. We wish to keep these sets immutable, and hence we return frozensets instead of regular sets. It is computed simply by combining all the clauses and putting the Lits thus gathered into a frozenset.

```
(python3.9) (cnf.py) <<lits>>
def lits(cnf_instance: Cnf) -> frozenset[Lit]:
    <<lits-docstring>>
    return frozenset.union(*cnf_instance)
```

Lastly, we add in a docstring and a unit test for this function.

```
(python3.9) (cnf.py) <<lits-docstring>>
"""Return frozenset of all Lits that appear in a Cnf.

Args:
   cnf_instance (:obj:`Cnf`)

Return:
   A frozenset of all lits that appear in a Cnf.
"""
```

```
(python3.9) (test/test_cnf.py) <<test_lits>>
def test_lits():
    assert lits(cnf([[1, -2],[3, TRUE], [FALSE]])) \
        == frozenset({1, -2, 3, TRUE, FALSE})
```

5.5 Functions for simplification

We define some functions for reducing/simplifying clauses and Cnfs –

- In §5.5.1 we define a function for applying a set of standard tautologies for simplifying a Clause.

- In §5.5.2 we define a function for applying a set of standard tautologies for simplifying a Cnf.

Both these functions come in use especially after making assignments in Cnfs. Selectively assigning some Lits to true/false results in Clauses and Cnfs that contain Bools and can be simplified. The simplification functions defined in this section help us do just that.

5.5.1 Tautologically reduce clauses

We can reduce a Clause using the following tautologies –

$$\top \vee c = \top, \quad \bot \vee c = c, \quad c \vee \neg c = \top,$$

where c is a Clause. These are tautologies that feature disjunctions – which is what bind all the Lits together in a Clause.

We check for these tautologies on a case-by-case basis. It should be noted that the order in which the cases are checked is important for the correct behavior of this function.

Furthermore, the input type of this function is abstracted to `Set[Lit]`, of which `set[Lit]` and `frozenset[Lit]` are subtypes. This allows us to use both sets and frozensets as valid inputs to this function.

Checking of the first two tautologies is straightforward. We describe here the implementation of how the third tautology ($c \vee \neg c = \top$) is checked. Given a Lit-set $\{l_i \mid i \in I\}$, for some finite indexing set I, we check that $\{\neg l_i \mid i \in I\} \cap \{l_i \mid i \in I\} = \emptyset$. If yes, then we return a Clause formed from $\{l_i \mid i \in I\}$. If not, then we return the Clause $\top$.

We also ensure that this reduction (and our implementation of it in the form of this function) are idempotent.

```
(python3.9) (cnf.py) <<tautologically_reduce_clause>>
def tautologically_reduce_clause(lit_set: Set[Lit]) -> Clause:
    <<tautologically_reduce_clause-docstring>>
    if TRUE in lit_set:
        return _TRUE_CLAUSE
    if lit_set == {FALSE}:
        return _FALSE_CLAUSE
    if FALSE in lit_set:
        lit_set -= _FALSE_CLAUSE
```

```
    if not set(map(neg, lit_set)).isdisjoint(lit_set):
        return _TRUE_CLAUSE
    return clause(lit_set)
```

Having implemented the function, we now include a docstring and some unit tests for it.

```
(python3.9) (cnf.py) <<tautologically_reduce_clause-docstring>>
r"""Reduce a Clause using various tautologies.

The order in which these reductions are performed is important. This function is
idempotent.

Tautologies affecting Clauses:
   (⊤ ∨ c = ⊤)  (⊥ = ⊥)  (⊥ ∨ c = c)  (c ∨ ¬c = ⊤),
   where `x` is a Clause, `⊤` represents ``TRUE``, `⊥` represents ``FALSE``, and
   `∨` is disjunction.

Args:
   lit_set (:obj:`Set[Lit]`): an abstract set (a set or a frozenset) of Lits.

Return:
   The Clause formed by performing all the above-mentioned tautological reductions.
"""
```

```
(python3.9) (test/test_cnf.py) <<test_tautologically_reduce_clause>>
def test_tautologically_reduce_clause():
    trc = tautologically_reduce_clause  # local alias for long function name
    assert trc(clause([lit(1), TRUE]))    == clause([TRUE])
    assert trc(clause([FALSE]))           == clause([FALSE])
    assert trc(clause([lit(1), FALSE]))   == clause([lit(1)])
    assert trc(clause([lit(1), lit(-1)])) == clause([TRUE])

    # Test for idempotence
    _clause = clause([lit(1), lit(-2), lit(3), lit(3)])
    assert trc(trc(_clause)) == trc(_clause)

    pytest.raises(ValueError, trc, set())
```

5.5.2 Tautologically reduce Cnfs

We can reduce a Cnf using the following tautologies –

$$x \wedge \bot = \bot, \quad \top \wedge x = x,$$

where x is a Cnf. These are tautologies that feature conjunctions – which is what bind all the Clauses together in a Cnf.

We check for these tautologies on a case-by-cases basis. It should be noted that the order in which the cases are checked is important for the correct behavior of this function.

The input type is abstracted to `Set[Set[Lit]]`. This means the input can be one of the following types –

1. a frozenset of frozenset of Lits – this is the preferred type and all other inputs will be converted to this type by the function.
2. a frozenset of set of Lits – this form is not possible because a frozenset will only accept entries of hashable type.
3. a set of frozenset of Lits
4. a set of set of Lits – this form is not possible because a set will only accept entries hashable type.

Given an input of the right type, the function first reduces all its clauses using the `tautologically_reduce_clause` function. In the case when the clause set contains the TRUE Clause, we remove the TRUE clause and call our function recursively on the reduced clause-set. This recursive call is guaranteed to terminate because we always apply it to a clause set of smaller length each time.

Lastly, we note that this function is idempotent.

```
(python3.9) (cnf.py) <<tautologically_reduce_cnf>>
def tautologically_reduce_cnf(clause_set: Set[Set[Lit]]) -> Cnf:
    <<tautologically_reduce_cnf-docstring>>
    clause_set_reduced: set[Clause]
    clause_set_reduced = set(map(tautologically_reduce_clause, clause_set))

    if _FALSE_CLAUSE in clause_set_reduced:
        return _FALSE_CNF
```

```
    if clause_set_reduced == _TRUE_CNF:
        return _TRUE_CNF
    if _TRUE_CLAUSE in clause_set_reduced:
        return tautologically_reduce_cnf(clause_set_reduced - _TRUE_CNF)
    return cnf(clause_set_reduced)
```

Below, we specify the docstring and some unit tests for this function.

(python3.9) (cnf.py) <<tautologically_reduce_cnf-docstring>>

```
r"""Reduce a Cnf using various tautologies.

The order in which these reductions are performed is important. This
function is idempotent. This is a recursive function that is guaranteed to
terminate.

Tautologies affecting Cnfs:
   (x ∧ ⊥ = ⊥)  (⊤ = ⊤)  (⊤ ∧ x = x),
   where `x` is a Cnf, `⊤` represents ``TRUE``, `⊥` represents ``FALSE``, and `∧`
   is conjunction.

Args:
   clause_set (:obj:`Set[Set[Lit]]`): an abstract set (set or frozenset) of
   abstract sets of Lits.

Return:
   The Cnf formed by first reducing all the clauses tautologically and then
   performing all the above-mentioned tautological reductions on the Cnf
   itself.
"""
```

(python3.9) (test/test_cnf.py) <<test_tautologically_reduce_cnf>>

```
def test_tautologically_reduce_cnf():
    trc = tautologically_reduce_cnf  # local alias for long function name
    # cases where Clause reductions appear within Cnf reductions
    assert trc(cnf([[1, TRUE], [1, 2]]))  == cnf([[1, 2]])
    assert trc(cnf([[FALSE], [1, 2]]))    == cnf([[FALSE]])
    assert trc(cnf([[1, FALSE], [1, 2]])) == cnf([[1], [1, 2]])
    assert trc(cnf([[1, -1], [1, 2]]))    == cnf([[1, 2]])

    # cases where we might have two simultaneous clause reductions
    assert trc(cnf([[1, TRUE], [FALSE]]))    == cnf([[FALSE]])
    assert trc(cnf([[1, TRUE], [1, FALSE]])) == cnf([[1]])
    assert trc(cnf([[1, TRUE], [1, -1]]))    == cnf([[TRUE]])
```

```
assert trc(cnf([[FALSE], [1, FALSE]]))   == cnf([[FALSE]])
assert trc(cnf([[FALSE], [1, -1]]))      == cnf([[FALSE]])
assert trc(cnf([[1, FALSE], [1, -1]]))   == cnf([[1]])

# cases where we might have a cnf-related tautology
assert trc(cnf([[1], [FALSE]])) == cnf([[FALSE]])
assert trc(cnf([[TRUE]]))       == cnf([[TRUE]])
assert trc(cnf([[1], [TRUE]]))  == cnf([[1]])

# Test for idempotence.
_cnf = cnf([[lit(1), lit(2)], [lit(-2)]])
assert trc(trc(_cnf)) == trc(_cnf)

pytest.raises(ValueError, trc, set())
pytest.raises(ValueError, trc, frozenset())
```

5.6 Functions for assignment

We define functions for applying truth-assignments to Lits, Clauses, and Cnfs.

- In §5.6.1 we define a function that assigns a given truth value to a variable in a Lit.
- In §5.6.2 we define a function that assigns a given truth value to a variable in a Clause.
- In §5.6.3 we define a function that assigns a given truth value to a variable in a Cnf.
- In §5.6.4 we define a function that applies an Assignment to Cnf.

We refer to the `<<example>>` code-block in §5.2 for a demonstration of the usage of the `assign` function defined in §5.6.4.

Throughout this section, we will denote the assignment of a to an object x by the notation $x[a]$.

5.6.1 Assign to variable in a literal

For a literal l and a variable v, this function encodes the result that the value $l[v]$, i.e. the value obtained by setting v to $\top$ is,

- $\top$, if $l = v$,
- $\bot$, if $l = \neg v$,
- l, otherwise.

This function is idempotent because $l[v][v] = l[v]$, for every literal l and every variable v.

```
_____________ (python3.9) (cnf.py) <<assign_variable_in_lit>> _____________
def assign_variable_in_lit(literal: Lit, variable_: Variable, boolean: Bool) -> Lit:
    <<assign_variable_in_lit-docstring>>
    if literal == variable_:
        return boolean
    if neg(literal) == variable_:
        return neg(boolean)
    return literal
```

We complete the definition by adding in a docstring and some unit-tests for the function.

```
_________ (python3.9) (cnf.py) <<assign_variable_in_lit-docstring>> _________
"""Assign Bool value to a Variable if present in Lit.

Replace all instances of ``variable_`` and its negation with ``boolean``
and its negation respectively. Leave all else unchanged. This function is
idempotent.

Args:
   literal (:obj:`Lit`)
   variable_ (:obj:`Variable`)
   boolean (:obj:`Bool`): either ``TRUE`` or ``FALSE``.

Return:
   Lit formed by assigning ``variable_`` to ``boolean`` in ``literal``.

"""
```

```
———— (python3.9) (test/test_cnf.py) <<test_assign_variable_in_lit>> ————
def test_assign_variable_in_lit():
    avil = assign_variable_in_lit  # local alias for long function name

    assert avil(1, 1, TRUE)     == TRUE
    assert avil(1, 1, FALSE)    == FALSE
    assert avil(-1, 1, TRUE)    == FALSE
    assert avil(-1, 1, FALSE)   == TRUE
    assert avil(1, 2, TRUE)     == 1
    assert avil(TRUE, 1, TRUE)  == TRUE
    assert avil(FALSE, 1, TRUE) == FALSE

    # Test for idempotence
    assert avil(avil(1, 1, TRUE), 1, TRUE)  == avil(1, 1, TRUE)
    assert avil(avil(-1, 1, TRUE), 1, TRUE) == avil(-1, 1, TRUE)
```

5.6.2 Assign to variable in a clause

Next, we define a function that assigns variables to Bools wherever the variable occurs in a clause. The function carries out the following computation –

$$\begin{aligned} c[v] &= (l_1 \vee l_2 \vee \cdots \vee l_n)[v], \\ &= l_1[v] \vee l_2[v] \vee \cdots \vee l_n[v], \end{aligned}$$

where c is a clause, v is a variable, and each l_i is a literal.

In the implementation of the function, we use `functools.partial` to define `assign_variable` to be the partial function `assign_variable_in_lit(__, variable_, boolean)`. This partial function takes a single Lit as an input while freezing the other arguments in place.

We note also that this function is idempotent in the first argument. Meaning, for a fixed variable and Bool value, applying this function twice to the same Clause is the same as applying it only once.

```
———————— (python3.9) (cnf.py) <<assign_variable_in_clause>> ————————
def assign_variable_in_clause(lit_set: Set[Lit], variable_: Variable, boolean: Bool) \
        -> Clause:
    <<assign_variable_in_clause-docstring>>
    assign_variable: Callable[[Lit], Lit]
    assign_variable = ft.partial(assign_variable_in_lit, variable_=variable_,
```

```
                                boolean=boolean)
    mapped_lits: set[Lit]
    mapped_lits = set(map(assign_variable, lit_set))

    return tautologically_reduce_clause(mapped_lits)
```

We now add in a docstring and some unit-tests for the function we just defined.

```
(python3.9) (cnf.py) <<assign_variable_in_clause-docstring>>
"""Assign Bool value to a Variable if present in Clause.

Replace all instances of ``variable_`` and its negation in ``lit_set`` with
``boolean`` and its negation respectively. Leave all else unchanged.
Perform tautological reductions on the Clause before returning results.
This function is idempotent.

Args:
   lit_set (:obj:`Set[Lit]`): an abstract set (set or frozenset) of Lits.
   variable_ (:obj:`Variable`): a variable instance
   boolean (:obj:`Bool`): either ``TRUE`` or ``FALSE``.

Return:
   Tautologically-reduced Clause formed by assigning ``variable_`` to ``boolean`` in
      ``lit_set``.
"""
```

```
(python3.9) (test/test_cnf.py) <<test_assign_variable_in_clause>>
def test_assign_variable_in_clause():
    avic = assign_variable_in_clause  # local alias for long function name

    assert avic(clause([1, -2]), 1, TRUE)      == {TRUE}
    assert avic(clause([1, -2]), 1, FALSE)     == {-2}
    assert avic(clause([1, -2, -1]), 1, TRUE)  == {TRUE}
    assert avic(clause([1, -2, -1]), 1, FALSE) == {TRUE}
    assert avic(clause([1, -2]), 2, TRUE)      == {1}
    assert avic(clause([1, -2]), 2, FALSE)     == {TRUE}
    assert avic(clause([1, -2, -1]), 2, TRUE)  == {TRUE}
    assert avic(clause([1, -2, -1]), 2, FALSE) == {TRUE}

    # Test for idempotence
    _clause: Clause = clause([lit(1), lit(-2), lit(-1)])
    _var: Variable = variable(2)
```

```
    assert avic(avic(_clause, _var, FALSE), _var, FALSE) == avic(_clause, _var, FALSE)

    pytest.raises(ValueError, assign_variable_in_clause, [], 1, TRUE)
```

5.6.3 Assign to variable in a Cnf

We define a function that assigns variables to Bools whenever the variable occurs in a Cnf. This function carries out the following computation –

$$\begin{aligned} x[v] &= (c_1 \wedge c_2 \wedge \cdots \vee c_n)[v], \\ &= c_1[v] \wedge c_2[v] \wedge \cdots \wedge c_n[v], \end{aligned}$$

where x is a Cnf, v is a variable, and each c_i is a clause.

In the implementation of the function, we use `functools.partial` to define `assign_variable` to be the partial function `assign_variable_in_clause(__, variable_, boolean)`. This partial function takes a single Clause as an input while freezing the other arguments in place.

We note also that this function is idempotent in the first argument. Meaning, for a fixed variable and Bool value, applying this function twice to the same Cnf is the same as applying it only once.

```
(python3.9) (cnf.py) <<assign_variable_in_cnf>>
def assign_variable_in_cnf(clause_set: Set[Set[Lit]], variable_: Variable,
                           boolean: Bool) -> Cnf:
    <<assign_variable_in_cnf-docstring>>
    assign_variable: Callable[[Clause], Clause]
    assign_variable = ft.partial(assign_variable_in_clause,
                                 variable_=variable_,
                                 boolean=boolean)

    mapped_clauses: set[Clause]
    mapped_clauses = set(map(assign_variable, clause_set))

    return tautologically_reduce_cnf(mapped_clauses)
```

We now add in a docstring and some unit-tests for the function.

```
(python3.9) (cnf.py) <<assign_variable_in_cnf-docstring>>
"""Assign Bool value to a Variable if present in Cnf.
```

```
Replace all instances of ``variable_`` and its negation in ``clause_set``
with ``boolean`` and its negation respectively. Leave all else unchanged.
Perform tautological reductions on the Cnf before returning results. This
function is idempotent.

Args:
   clause_set (:obj:`Set[Set[Lit]]`): an abstract set (set or frozenset) of
      abstract sets of Lits.
   variable_ (:obj:`Variable`)
   boolean (:obj:`Bool`): either ``TRUE`` or ``FALSE``.

Return:
   Tautologically-reduced Cnf formed by assigning ``variable_`` to ``boolean`` in
      ``clause_set``.
"""
```

```
_________ (python3.9) (test/test_cnf.py) <<test_assign_variable_in_cnf>> _________
def test_assign_variable_in_cnf():
    avic = assign_variable_in_cnf  # local alias for long function name

    assert avic(cnf([[1, -2], [-1, 3]]), 1, TRUE)  == cnf([[3]])
    assert avic(cnf([[1, -2], [-1, 3]]), 1, FALSE) == cnf([[-2]])

    # Test for idempotence.
    _cnf = cnf([[1, -2], [-1, 3]])
    assert avic(avic(_cnf, 1, FALSE), 1, FALSE) == avic(_cnf, 1, FALSE)

    pytest.raises(ValueError, avic, [[]], 1, TRUE)
```

5.6.4 Assign

We define a function that applies a (possibly partial) Boolean assignment to a given Cnf. This function carries out the following computation –

$$x[A] = x[\{l_1, \ldots, l_n\}] = (((x[l_1])[l_2]) \cdots)[l_n],$$

where x is a Cnf, A is an assignment, and each l_i is a literal.

For positive literals in the assignment, we set the underlying variable to true using `assign_variable_in_cnf`. For negative literals in the assignment, we use the same function to set the underlying variable to false.

We note that this function is idempotent in the first argument (i.e. keeping the assignment fixed, applying this function twice to a Cnf is the same as applying it once).

A complete assignment might return a trivial Cnf value (a true or false Cnf). However, in general, since the assignments can be partial, the function can return a nontrivial Cnf instead.

```
(python3.9) (cnf.py) <<assign>>
def assign(cnf_instance: Cnf, assignment: Assignment) -> Cnf:
    <<assign-docstring>>
    cnf_copy: frozenset[Clause] = cnf_instance.copy()
    for variable_, boolean in assignment.items():
        cnf_copy = assign_variable_in_cnf(cnf_copy, variable_, boolean)
    return tautologically_reduce_cnf(cnf_copy)
```

We add a docstring and some unit-tests for the function.

```
(python3.9) (cnf.py) <<assign-docstring>>
"""Assign Bool values to Variables if present in Cnf.

For each Variable (key) in ``assignment``, replace the Variable and its
negation in ``cnf_instance`` using the Bool (value) in ``assignment``. The
final output is always tautologically reduced. This function is idempotent.

Args:
   cnf_instance (:obj:`Cnf`)
   assignment (:obj:`Assignment`): a dict with keys being Variables to be
      replaced and values being Bools that the Variables are to be assigned
      to. The ``assignment`` dict need not be complete and can be partial.

Edge case:
   An empty assignment dict results in ``cnf_instance`` simply getting
   topologically reduced.

Return:
   Tautologically-reduced Cnf formed by replacing every key in the
   ``assignment`` dict (and those keys' negations) by corresponding Bool values.
"""
```

```
(python3.9) (test/test_cnf.py) <<test_assign>>
def test_assign():
    assert assign(cnf([[1, -2], [-1, 3]]), {1: TRUE}) == cnf([[3]])
    assert assign(cnf([[1, -2], [-1, 3]]), {1: TRUE, 2: FALSE}) == cnf([[3]])
```

```
    assert assign(cnf([[1, -2], [-1, 3]]),
                  {1: TRUE, 2: FALSE, 3: FALSE}) == cnf([[FALSE]])
    assert assign(cnf([[TRUE]]), {1: TRUE}) == cnf([[TRUE]])
    assert assign(cnf([[TRUE]]), {}) == cnf([[TRUE]])
    assert assign(cnf([[FALSE]]), {}) == cnf([[FALSE]])
    assert assign(cnf([[1]]), {}) == cnf([[1]])

    # Test for idempotence.
    _cnf = cnf([[1, -2], [-1, 3]])
    assert assign(assign(_cnf, {1: TRUE}), {1: TRUE}) \
        == assign(_cnf, {1: TRUE})

    pytest.raises(ValueError, assign, [[]], {1: TRUE})
```

5.7 Standalone script run commands

We now add in some commands that will be run when the module is called from the command line by a user, say by invoking `python3 -m cnf.py` at the terminal prompt. The purpose of these lines is to test that the user has installed the module correctly and it also serves to demonstrate the construction of a Cnf.

To invoke these lines only when the script is run as stand-alone, and not during regular imports, we put these lines in a `if __name__ =` 'main'= block.

We use a logger to send messages and output to *stdout*.

```
(python3.9) (cnf.py) <<standalone_run_commands>>
logger.info('Cnfs can be constructed using the cnf() function.')
logger.info('>>> cnf([[1, -2], [3, 500]])')
logger.info(cnf([[1, -2], [3, 500]]))
```

5.8 Tangling

We now tangle all the code blocks defined in this chapter into a single file (`cnf.py`) following the overview laid out in §5.2.2. Using the source file for this chapter, the tangling can be carried out by running the following script

```
(sh) (scratch) <<tangle>>
cp subchapter_cnf.org cnf.org  # create a temp file
# cnf.org -> cnf.py + test_cnf.py (tangle)
```

```
emacs --batch cnf.org -f org-babel-tangle --kill
black cnf.py test/test_cnf.py  # auto-format for PEP8 compliance
rm cnf.org                     # remove temp file
```

We similarly tangle all the unit-tests into a single file (`test/test_cnf.py`) using the following layout.

```
(python3.9) (test/test_cnf.py) <<test_cnf.py-full>>
#!/usr/bin/env python3.9
import pytest
from cnf import *

<<test_variable>>
<<test_Bool>>
<<test_lit>>
<<test_clause>>
<<test_cnf>>

<<test_neg>>
<<test_absolute_value>>
<<test_lits>>

<<test_tautologically_reduce_clause>>
<<test_tautologically_reduce_cnf>>

<<test_assign_variable_in_lit>>
<<test_assign_variable_in_clause>>
<<test_assign_variable_in_cnf>>
<<test_assign>>
```

Finally, we can run all the tests on module functions as follows –

```
(sh) (scratch) <<run-tests>>
python3 -m mypy cnf.py  # static typechecking of code
python3 -m py.test      # run all the unit tests
python3 -m cnf.py       # run the module as a stand-alone script
```

5.9 Concluding remarks

In this chapter we present the first of several modules from the `graphsat` Python package. The interested reader can check out a more complete implementation of all the other modules at the GitHub repository `vaibhavkarve/graphsat`.

Chapter 6

3GraphSAT and computational results

In this chapter, we summarize some of the key findings enabled by the local rewriting theorem from §4.4, and by the `graphsat` python package introduced in §5. This experimental style of math combines programming and proofs to classify graphs, hypergraphs, and infinite graphs as totally satisfiable or unsatisfiable. We use programming for number-crunching and dealing with the vast amount of cases that need to be checked in each calculation. For pattern-matching in graphs, we often also rely on the human brain's ability to spot invariants when shown a list of graph drawings.

In §6.1 we list tables of standard graph disjunctions that one may encountered when carrying out local graph rewriting calculations. These tables are all generated using the `graph_or` function in the `operations.py` module. In §6.2 we then use these disjunction results to derive graph reduction rules – global rewrites that do not affect the satisfiability of a graph. In §6.3, we describe an ongoing effort to describe the criterion for hypergraph minimality taking into account a growing list of graph reduction rules.

In §§6.4–6.5, we present computationally obtained satisfiability and unsatisfiability results on mixed hypergraphs and triangulations. The choice of structures we present in those sections is less systematic and more driven by ease of calculation.

In §6.6 we discuss the graph decision problem as it relates to infinite graphs. We also present some examples of totally satisfiable infinite graphs. In §6.7 we discuss the logistical setup used for carrying out all the calculations in this chapter, along with a mention of the challenges posed by continuing these computations on bigger graphs in the face of exponentially more cases that need checking.

6.1 Standard graph disjunctions

Essential to carrying out local graph rewriting is the calculation of graph disjunctions of the form $\boldsymbol{g_i} \vee \boldsymbol{h_i}$. A graph disjunction, even if we start with $\boldsymbol{g_i}$ and $\boldsymbol{h_i}$ being graphs, frequently yields a set of Cnfs that might not be a graph. For example,

$$\begin{aligned}\boldsymbol{a} \vee \boldsymbol{a} &= \{x \vee y \mid (x, y : \text{Cnf}) \in \boldsymbol{a}\} \\ &= \{a \vee a,\ a \vee \overline{a},\ \overline{a} \vee a,\ \overline{a} \vee \overline{a}\} \\ &= \{a,\ \overline{a},\ \top\}\end{aligned}$$

We note that these three Cnfs do not belong to the set corresponding to any single graph because the Cnfs do not have the same vertex set. The Cnfs a and $\overline{a}$ have a vertex set of $\boldsymbol{a}$; the true Cnf has an empty vertex set. Nevertheless, we can write it as a union of graphs. We can write $\boldsymbol{a} \vee \boldsymbol{a} = \top \cup \boldsymbol{a}$.

Nevertheless, we cannot always write a disjunction even as a union of graphs. For example,

$$\begin{aligned}\boldsymbol{a}\vee(\boldsymbol{a} \wedge \boldsymbol{b}) = \{&a \vee (a \wedge b), \quad a \vee (a \wedge \overline{b}), \quad a \vee (\overline{a} \wedge b), \quad a \vee (\overline{a} \wedge \overline{b}), \\ &\overline{a} \vee (a \wedge b), \quad \overline{a} \vee (a \wedge \overline{b}), \quad \overline{a} \vee (\overline{a} \wedge b), \quad \overline{a} \vee (\overline{a} \wedge \overline{b})\} \\ = \{&a \wedge (a \vee b), \quad a \wedge (a \vee \overline{b}), \quad a \vee b, \quad a \vee \overline{b}, \\ &\overline{a} \vee b, \quad \overline{a} \vee \overline{b}, \quad \overline{a} \wedge (\overline{a} \vee b), \quad \overline{a} \wedge (\overline{a} \vee \overline{b})\} \\ = \boldsymbol{ab} \cup \{&a \wedge (a \vee b), \quad a \wedge (a \vee \overline{b}), \quad \overline{a} \wedge (\overline{a} \vee b), \quad \overline{a} \wedge (\overline{a} \vee \overline{b})\}\end{aligned}$$

Although the remaining Cnfs all belong to the same graph, namely $\boldsymbol{a} \wedge \boldsymbol{ab}$, we note that we do not have the complete set. For example, we are missing the Cnf $a \wedge (\overline{a} \vee b)$. Hence we cannot write $\boldsymbol{a}\vee(\boldsymbol{a} \wedge \boldsymbol{b}) = \boldsymbol{ab} \cup \boldsymbol{a} \wedge \boldsymbol{ab}$. The best we can do is –

$$\boldsymbol{ab} \subset \boldsymbol{a}\vee(\boldsymbol{a} \wedge \boldsymbol{b}) \subset \boldsymbol{ab} \cup \boldsymbol{a} \wedge \boldsymbol{ab}.$$

These subset-superset pairs give us a lower and upper bound for the set in the middle. The utility of these subset-superset pairs becomes apparent when we view them in the context of satisfiability statuses. In the following

equation, let $\boldsymbol{s}$ an arbitrary graph. Then, we have –

$$\boldsymbol{\gamma}(\boldsymbol{s} \wedge \boldsymbol{ab}) \leftarrow \boldsymbol{\gamma}(\boldsymbol{s} \wedge (\boldsymbol{a} \vee (\boldsymbol{a} \wedge \boldsymbol{b}))) \leftarrow \boldsymbol{\gamma}(\boldsymbol{s} \wedge \boldsymbol{ab}) \wedge \boldsymbol{\gamma}(\boldsymbol{s} \wedge \boldsymbol{a} \wedge \boldsymbol{ab}).$$

This means that if $\boldsymbol{s} \wedge \boldsymbol{ab}$ is unsatisfiable, then is $\boldsymbol{s} \wedge (\boldsymbol{a} \vee (\boldsymbol{a} \wedge \boldsymbol{b}))$ too. On the other hand, if $\boldsymbol{s} \wedge \boldsymbol{ab}$ is totally satisfiable, the by checking if $\boldsymbol{s} \wedge \boldsymbol{a} \wedge \boldsymbol{ab}$ is also totally satisfiable, we can conclude that $\boldsymbol{s} \wedge (\boldsymbol{a} \vee (\boldsymbol{a} \wedge \boldsymbol{b}))$ is totally satisfiable as well.

Below, we include tables of such graph disjunctions. These tables come in handy performing local rewrites, as seen in §6.2. The first two tables list graph disjunctions that can be written exactly as a union of graphs; the third table lists graph disjunctions that can only be listed as a subset-superset pair.

Table 6.1: Graph disjunctions where size of $\boldsymbol{h_1}$ + size of $\boldsymbol{h_2}$ is 2.

$\boldsymbol{h_1}$		$\boldsymbol{h_2}$		$\boldsymbol{h_1} \vee \boldsymbol{h_2}$
$\boldsymbol{a}$	$\vee$	$\boldsymbol{a}$	=	$\top \cup \boldsymbol{a}$
$\boldsymbol{a}$	$\vee$	$\boldsymbol{b}$	=	$\boldsymbol{ab}$
$\boldsymbol{a}$	$\vee$	$\boldsymbol{bc}$	=	$\boldsymbol{abc}$
$\boldsymbol{a}$	$\vee$	$\boldsymbol{ab}$	=	$\top \cup \boldsymbol{ab}$
$\boldsymbol{ab}$	$\vee$	$\boldsymbol{cd}$	=	$\boldsymbol{abcd}$
$\boldsymbol{ab}$	$\vee$	$\boldsymbol{ac}$	=	$\top \cup \boldsymbol{abc}$
$\boldsymbol{ab}$	$\vee$	$\boldsymbol{ab}$	=	$\top \cup \boldsymbol{ab}$

Table 6.2: Graph disjunctions where size of $\boldsymbol{h_1}$ + size of $\boldsymbol{h_2}$ is 3.

$\boldsymbol{h_1}$		$\boldsymbol{h_2}$		$\boldsymbol{h_1} \vee \boldsymbol{h_2}$
$\boldsymbol{a}$	$\vee$	$\boldsymbol{a}^2$	=	$\boldsymbol{a}$
$\boldsymbol{a}$	$\vee$	$\boldsymbol{b}^2$	=	$\boldsymbol{a}$
$\boldsymbol{a}$	$\vee$	$(\boldsymbol{ab})^2$	=	$\top \cup \boldsymbol{a} \cup \boldsymbol{ab}$
$\boldsymbol{a}$	$\vee$	$(\boldsymbol{b} \wedge \boldsymbol{ab})$	=	$\boldsymbol{a} \cup \boldsymbol{ab}$
$\boldsymbol{ab}$	$\vee$	$\boldsymbol{c}^2$	=	$\boldsymbol{ab}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{a} \wedge \boldsymbol{c})$	=	$\boldsymbol{ab} \cup \boldsymbol{abc}$
$\boldsymbol{ab}$	$\vee$	$\boldsymbol{a}^2$	=	$\boldsymbol{ab}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{a} \wedge \boldsymbol{b})$	=	$\top \cup \boldsymbol{ab}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{c} \wedge \boldsymbol{ac})$	=	$\boldsymbol{ab} \cup \boldsymbol{abc}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{c} \wedge \boldsymbol{ab})$	=	$\boldsymbol{ab} \cup \boldsymbol{abc}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{a} \wedge \boldsymbol{cd})$	=	$\boldsymbol{ab} \cup \boldsymbol{abcd}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{a} \wedge \boldsymbol{ac})$	=	$\top \cup \boldsymbol{ab} \cup \boldsymbol{abc}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{a} \wedge \boldsymbol{bc})$	=	$\top \cup \boldsymbol{ab} \cup \boldsymbol{abc}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{a} \wedge \boldsymbol{ab})$	=	$\top \cup \boldsymbol{ab}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{ab} \wedge \boldsymbol{cd})$	=	$\boldsymbol{ab} \cup \boldsymbol{abcd}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{ab} \wedge \boldsymbol{ac})$	=	$\top \cup \boldsymbol{ab} \cup \boldsymbol{abc}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{ac})^2$	=	$\top \cup \boldsymbol{ab} \cup \boldsymbol{abc}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{ac} \wedge \boldsymbol{bc})$	=	$\top \cup \boldsymbol{ab} \cup \boldsymbol{abc}$
$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{ab})^2$	=	$\top \cup \boldsymbol{ab}$

The above tables show that the possible graph disjunctions grow quickly with the edges participating in the disjunction. This is why we stop at a maximum of three edges. For calculating the graph disjunction of more

Table 6.3: Subset-superset pairs for graph disjunctions where size of $\boldsymbol{h_1}$ + size of $\boldsymbol{h_2}$ is at most 3.

Subset		$\boldsymbol{h_1}$		$\boldsymbol{h_2}$		Superset
		$\boldsymbol{a}$	$\vee$	$(\boldsymbol{b} \wedge \boldsymbol{c})$	$\subset$	$(\boldsymbol{ab} \wedge \boldsymbol{ac})$
$\boldsymbol{ab}$	$\subset$	$\boldsymbol{a}$	$\vee$	$(\boldsymbol{a} \wedge \boldsymbol{b})$	$\subset$	$\boldsymbol{ab} \cup (\boldsymbol{a} \wedge \boldsymbol{ab})$
		$\boldsymbol{a}$	$\vee$	$(\boldsymbol{b} \wedge \boldsymbol{cd})$	$\subset$	$(\boldsymbol{ab} \wedge \boldsymbol{acd})$
$\boldsymbol{abc}$	$\subset$	$\boldsymbol{a}$	$\vee$	$(\boldsymbol{a} \wedge \boldsymbol{bc})$	$\subset$	$\boldsymbol{abc} \cup (\boldsymbol{a} \wedge \boldsymbol{abc})$
$\boldsymbol{ab}$	$\subset$	$\boldsymbol{a}$	$\vee$	$(\boldsymbol{b} \wedge \boldsymbol{ac})$	$\subset$	$\boldsymbol{ab} \cup (\boldsymbol{ab} \wedge \boldsymbol{ac})$
		$\boldsymbol{a}$	$\vee$	$(\boldsymbol{b} \wedge \boldsymbol{bc})$	$\subset$	$(\boldsymbol{ab} \wedge \boldsymbol{abc})$
$\top \cup \boldsymbol{a} \cup \boldsymbol{ab}$	$\subset$	$\boldsymbol{a}$	$\vee$	$(\boldsymbol{a} \wedge \boldsymbol{ab})$	$\subset$	$\top \cup \boldsymbol{a} \cup \boldsymbol{ab} \cup (\boldsymbol{a} \wedge \boldsymbol{ab})$
		$\boldsymbol{a}$	$\vee$	$(\boldsymbol{bc} \wedge \boldsymbol{de})$	$\subset$	$(\boldsymbol{abc} \wedge \boldsymbol{ade})$
		$\boldsymbol{a}$	$\vee$	$(\boldsymbol{bc} \wedge \boldsymbol{bd})$	$\subset$	$(\boldsymbol{abc} \wedge \boldsymbol{abd})$
		$\boldsymbol{a}$	$\vee$	$(\boldsymbol{bc})^2$	$\subset$	$(\boldsymbol{abc})^2$
$\boldsymbol{acd}$	$\subset$	$\boldsymbol{a}$	$\vee$	$(\boldsymbol{ab} \wedge \boldsymbol{cd})$	$\subset$	$\boldsymbol{acd} \cup (\boldsymbol{ab} \wedge \boldsymbol{acd})$
$\boldsymbol{ab} \cup \boldsymbol{abc}$	$\subset$	$\boldsymbol{a}$	$\vee$	$(\boldsymbol{ab} \wedge \boldsymbol{bc})$	$\subset$	$\boldsymbol{ab} \cup \boldsymbol{abc} \cup (\boldsymbol{ab} \wedge \boldsymbol{abc})$
$\top \cup \boldsymbol{ab} \cup \boldsymbol{ac}$	$\subset$	$\boldsymbol{a}$	$\vee$	$(\boldsymbol{ab} \wedge \boldsymbol{ac})$	$\subset$	$\top \cup \boldsymbol{ab} \cup \boldsymbol{ac} \cup (\boldsymbol{ab} \wedge \boldsymbol{ac})$
		$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{c} \wedge \boldsymbol{d})$	$\subset$	$(\boldsymbol{abc} \wedge \boldsymbol{abd})$
		$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{c} \wedge \boldsymbol{de})$	$\subset$	$(\boldsymbol{abc} \wedge \boldsymbol{abde})$
$\boldsymbol{abc}$	$\subset$	$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{c} \wedge \boldsymbol{cd})$	$\subset$	$\boldsymbol{abc} \cup (\boldsymbol{abc} \wedge \boldsymbol{abcd})$
$\boldsymbol{abc}$	$\subset$	$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{c} \wedge \boldsymbol{ad})$	$\subset$	$\boldsymbol{abc} \cup (\boldsymbol{abc} \wedge \boldsymbol{abd})$
		$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{cd} \wedge \boldsymbol{ef})$	$\subset$	$(\boldsymbol{abcd} \wedge \boldsymbol{abef})$
		$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{cd} \wedge \boldsymbol{ce})$	$\subset$	$(\boldsymbol{abcd} \wedge \boldsymbol{abce})$
		$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{cd})^2$	$\subset$	$(\boldsymbol{abcd})^2$
$\boldsymbol{abcd}$	$\subset$	$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{cd} \wedge \boldsymbol{ac})$	$\subset$	$\boldsymbol{abcd} \cup (\boldsymbol{abc} \wedge \boldsymbol{abcd})$
$\top \cup \boldsymbol{abc} \cup \boldsymbol{abd}$	$\subset$	$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{ac} \wedge \boldsymbol{ad})$	$\subset$	$\top \cup \boldsymbol{abc} \cup \boldsymbol{abd} \cup (\boldsymbol{abc} \wedge \boldsymbol{abd})$
$\top \cup \boldsymbol{abc} \cup \boldsymbol{abd}$	$\subset$	$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{ac} \wedge \boldsymbol{bd})$	$\subset$	$\top \cup \boldsymbol{abc} \cup \boldsymbol{abd} \cup (\boldsymbol{abc} \wedge \boldsymbol{abd})$
$\boldsymbol{abcd}$	$\subset$	$\boldsymbol{ab}$	$\vee$	$(\boldsymbol{ac} \wedge \boldsymbol{cd})$	$\subset$	$\boldsymbol{abcd} \cup (\boldsymbol{abc} \wedge \boldsymbol{abcd})$

edges, we can always use the `graph_or` function from the `operations` module on each individual disjunction.

6.2 Graph reduction rules

This section states some global graph rewriting rules that leave the satisfiability status of a graph unchanged. We call these *graph reduction rules*. Using the graph local rewriting theorem, we prove the invariance of GRAPH-SAT under these reduction rules.

These reduction rules yield a set of simple search-and-replace rules that can be deployed computationally to simplify a graph, make it smaller, and then subject it to a graph-satchecker.

In the following subsections, we will always label the sphere of the graph $\boldsymbol{g}$ at vertex $\mathbf{1}$ as $\boldsymbol{s}$. Since $\boldsymbol{s}$ has no edges incident on $\mathbf{1}$, when decomposing locally at that vertex, we can always write $\boldsymbol{s}[\mathbf{1}] = \boldsymbol{s}$.

6.2.1 Deleting leaf vertices

We start by looking at vertices of degree 1 (not counting edge multiplicities), i.e. at *leaf vertices*. We prove that leaf vertices can be deleted without affecting the satisfiability status of a graph. First, we prove that we can delete a leaf vertex that has only edges of size 1 of any multiplicity incident on it.

For a multiplicity 1 edge, we get $\boldsymbol{s} \wedge \mathbf{12} \sim \boldsymbol{s}$.

For multiplicity 2 edges, we get $\boldsymbol{s} \wedge \mathbf{12}^2 \sim \boldsymbol{s} \wedge (\mathbf{2} \vee \mathbf{2}) = \boldsymbol{s} \wedge (\top \cup \mathbf{2}) = \boldsymbol{s} \wedge \mathbf{2}$.

For multiplicity 3 edges, we get $\boldsymbol{s} \wedge \mathbf{12}^3 \sim \boldsymbol{s} \wedge (\mathbf{2} \vee \mathbf{2}^2) \sim \boldsymbol{s} \wedge (\mathbf{2} \vee \bot) = \boldsymbol{s} \wedge \mathbf{2}$.

For multiplicity 4 edges, we get $\boldsymbol{s} \wedge \mathbf{12}^4 \sim \boldsymbol{s} \wedge (\mathbf{2} \vee \mathbf{2}^3) \cup \boldsymbol{s} \wedge (\mathbf{2}^2 \vee \mathbf{2}^2) = \boldsymbol{s} \wedge (\mathbf{2} \vee \bot) \cup \boldsymbol{s} \wedge (\bot \vee \bot) \sim \boldsymbol{s} \wedge \mathbf{2} \cup \bot \sim \bot$.

All the graph disjunctions used in the above calculation can be found in Table 6.1. We can write these results more concisely as $\boldsymbol{s} \wedge \mathbf{12}^n \sim \boldsymbol{s} \wedge \mathbf{2}^{\lfloor n/2 \rfloor}$, $\forall n \in \mathbb{N}$, such that $\mathbf{2}^2 \sim \bot$.

Next, we show that a leaf vertex incident only on hyperedges can be rewritten to edges of size 2. For a leaf vertex $\mathbf{1}$, incident on a hyperedge of multi-

plicity 1, we get the reduction $\boldsymbol{s} \wedge \mathbf{123}^1 \sim \boldsymbol{s}$.

For a hyperedge of multiplicity 2, we get $\boldsymbol{s} \wedge \mathbf{123}^2 \sim \boldsymbol{s} \wedge (\mathbf{23} \vee \mathbf{23}) = \boldsymbol{s} \wedge (\top \cup \mathbf{23})$.

For a hyperedge of multiplicity 3, we get $\boldsymbol{s} \wedge \mathbf{123}^3 \sim \boldsymbol{s} \wedge (\mathbf{23} \vee \mathbf{23}^2) = \boldsymbol{s} \wedge (\mathbf{23} \cup \top) = \boldsymbol{s} \wedge \mathbf{23} \cup \boldsymbol{s} = \boldsymbol{s} \wedge \mathbf{23}$.

For a hyperedge of multiplicity 4, we get $\boldsymbol{s} \wedge (\mathbf{123})^4 \sim \boldsymbol{s} \wedge (\mathbf{23} \vee \mathbf{23}^3) \cup \boldsymbol{s} \wedge (\mathbf{23}^2 \vee \mathbf{23}^2) = \boldsymbol{s} \wedge (\mathbf{23} \cup \top) \cup \boldsymbol{s} \wedge (\mathbf{23}^2 \cup \mathbf{23} \cup \top) = \boldsymbol{s} \wedge \mathbf{23}^2 \cup \boldsymbol{s} \wedge \mathbf{23} \cup \boldsymbol{s} = \boldsymbol{s} \wedge \mathbf{23}^2$.

Some of the graph disjunctions in the above calculations can be found in Table 6.2. The remaining disjunctions are computed using the `operations.graph_or` function from the `graphsat` package. Code-snippets and their outputs are provided below for reference.

```
_____ (python3.9) (scratch) <<calculation1>> _____
import cnf, mhgraph
from operations import graph_or

g1 = mhgraph.mhgraph([[2, 3]])
g3 = mhgraph.mhgraph([[2, 3]]*3)

for x in graph_or(g1, g3):
    print(x)
```

Output $\mathbf{23} \vee \mathbf{23}^3$:

```
(<Bool: TRUE>)
(2,3)
(2,-3)
(-2,3)
(-2,-3)
```

```
_____ (python3.9) (scratch) <<calculation2>> _____
import cnf, mhgraph
from operations import graph_or

g2 = mhgraph.mhgraph([[2, 3]]*2)

for x in graph_or(g2, g2):
    print(x)
```

Output $\mathbf{23}^2 \vee \mathbf{23}^2$:

```
(<Bool: TRUE>)
(2,3)
(2,-3)
(-2,3)
(-2,-3)
(2,3)(2,-3)
(2,3)(-2,3)
(2,3)(-2,-3)
(2,-3)(-2,3)
(2,-3)(-2,-3)
(-2,3)(-2,-3)
```

Looking at the pattern, we can derive similar rules for higher multiplicities:

- $\boldsymbol{s} \wedge (\mathbf{123})^5 \sim \boldsymbol{s} \wedge \mathbf{23}^2$
- $\boldsymbol{s} \wedge (\mathbf{123})^6 \sim \boldsymbol{s} \wedge \mathbf{23}^3$
- $\boldsymbol{s} \wedge (\mathbf{123})^7 \sim \boldsymbol{s} \wedge \mathbf{23}^3$
- $\boldsymbol{s} \wedge (\mathbf{123})^8 \sim \boldsymbol{s} \wedge \mathbf{23}^4 \sim \bot$

In general, when rewriting locally at the vertex $\mathbf{1}$, we can write $\forall n \in \mathbb{N}$, $\boldsymbol{s} \wedge \mathbf{123}^n \sim \boldsymbol{s} \wedge \mathbf{23}^{\lfloor n/2 \rfloor}$, such that $\mathbf{23}^4 \sim \bot$.

6.2.2 Smoothing edges

Having dealt with leaf vertices (i.e. vertices of degree 1), we now consider vertices of degree 2. Edges and hyperedges incident at such vertices can be "smoothed" without affecting the satisfiability status of a graph. We call these operations smoothing because each operation results in graphs with fewer bends (by having fewer vertices).

We can smoothe out the intersection of two edges as $\boldsymbol{s} \wedge \mathbf{12} \wedge \mathbf{13} \sim \boldsymbol{s} \wedge (\mathbf{2} \vee \mathbf{3}) = \boldsymbol{s} \wedge \mathbf{23}$. Similarly, we can smoothe out the intersection of two hyperedges sharing a common edge as $\boldsymbol{s} \wedge \mathbf{123} \wedge \mathbf{124} \sim \boldsymbol{s} \wedge (\mathbf{23} \vee \mathbf{24}) \sim \boldsymbol{s} \wedge (\top \cup \mathbf{234}) = \boldsymbol{s} \cup \boldsymbol{s} \wedge \mathbf{234} \sim \boldsymbol{s} \wedge \mathbf{234}$.

Smoothing out the intersection of two hyperedges sharing a common vertex results in a size 4 hyperedge – $\boldsymbol{s} \wedge \mathbf{123} \wedge \mathbf{145} \sim \boldsymbol{s} \wedge (\mathbf{23} \vee \mathbf{45}) = \boldsymbol{s} \wedge \mathbf{2345}$. Smoothing at an edge-hyperedge pair with a common vertex yields $\boldsymbol{s} \wedge \mathbf{12} \wedge \mathbf{134} \sim \boldsymbol{s} \wedge (\mathbf{2} \vee \mathbf{34}) = \boldsymbol{s} \wedge \mathbf{234}$.

6.2.3 Tucking edges

We now prove a series of reduction rules that allow deletion of degree 2 or higher vertices, resulting in graphs with fewer edges. Visually, these operations look like tucking-in of an extended fin of the graph.

Tucking-in at an edge-hyperedge intersection incident at a common edge yields $\boldsymbol{s} \wedge \mathbf{12} \wedge \mathbf{123} \sim \boldsymbol{s} \wedge (\mathbf{2} \vee \mathbf{23}) = \boldsymbol{s} \wedge (\mathbf{23} \cup \top) = \boldsymbol{s} \wedge \mathbf{23} \cup \boldsymbol{s} \sim \boldsymbol{s} \wedge \mathbf{23}$.

Degree 3 intersections have reduction rules for the following cases –

1. A hyperedge with an edge incident on two of its three sides, i.e. $\boldsymbol{s} \wedge \mathbf{12} \wedge \mathbf{13} \wedge \mathbf{123}$.

2. A hyperedge of multiplicity 2, with an edge incident on one of its sides, i.e. $\boldsymbol{s} \wedge \mathbf{12} \wedge \mathbf{123}^2$.

3. A hyperedge with an edge of multiplicity two incident on one of its sides, i.e. $\boldsymbol{s} \wedge \mathbf{12}^2 \wedge \mathbf{123}$.

For instance 1, we get –

$$\begin{aligned}
\boldsymbol{s} \wedge \mathbf{12} \wedge \mathbf{13} \wedge \mathbf{123} &\sim \boldsymbol{s} \wedge (\mathbf{2} \wedge \mathbf{23} \vee \mathbf{3}) \cup \boldsymbol{s} \wedge (\mathbf{3} \wedge \mathbf{23} \vee \mathbf{2}) \cup \boldsymbol{s} \wedge (\mathbf{2} \wedge \mathbf{3} \vee \mathbf{23}) \\
&\sim \boldsymbol{s} \wedge (\mathbf{3} \cup \mathbf{23}) \cup \boldsymbol{s} \wedge (\mathbf{2} \cup \mathbf{23}) \cup \boldsymbol{s} \wedge (\mathbf{23} \cup \top) \\
&= \boldsymbol{s} \cup \boldsymbol{s} \wedge \mathbf{2} \cup \boldsymbol{s} \wedge \mathbf{3} \cup \boldsymbol{s} \wedge \mathbf{23} \\
&\sim \boldsymbol{s} \wedge \mathbf{2} \cup \boldsymbol{s} \wedge \mathbf{3} \cup \boldsymbol{s} \wedge \mathbf{23} \\
&\sim \boldsymbol{s} \wedge \mathbf{2} \cup \boldsymbol{s} \wedge \mathbf{3}
\end{aligned}$$

For instance 2, we get –

$$\begin{aligned}
\boldsymbol{s} \wedge \mathbf{12} \wedge \mathbf{123}^2 &\sim \boldsymbol{s} \wedge (\mathbf{2} \vee \mathbf{23}^2) \cup \boldsymbol{s} \wedge (\mathbf{23} \vee \mathbf{2} \wedge \mathbf{23}) \\
&\sim \boldsymbol{s} \wedge (\top \cup \mathbf{2} \cup \mathbf{23}) \cup \boldsymbol{s} \wedge (\top \cup \mathbf{23}) \\
&= \boldsymbol{s} \cup \boldsymbol{s} \wedge \mathbf{2} \cup \boldsymbol{s} \wedge \mathbf{23} \\
&\sim \boldsymbol{s} \wedge \mathbf{2} \cup \boldsymbol{s} \wedge \mathbf{23} \\
&\sim \boldsymbol{s} \wedge \mathbf{2}
\end{aligned}$$

For instance 3, we get $\boldsymbol{s} \wedge \mathbf{12}^2 \wedge \mathbf{123} \sim \boldsymbol{s} \wedge (\mathbf{2} \vee \mathbf{2} \wedge \mathbf{23}) \cup \boldsymbol{s} \wedge (\mathbf{23} \vee \mathbf{2}^2)$. Since we have $(\top \cup \mathbf{2} \cup \mathbf{23}) \subset \mathbf{2} \vee \mathbf{2} \wedge \mathbf{23} \subset (\top \cup \mathbf{2} \cup \mathbf{23} \cup \mathbf{2} \wedge \mathbf{23})$, we can write –

$$\boldsymbol{s} \wedge (\top \cup \mathbf{2} \cup \mathbf{23}) \cup \boldsymbol{s} \wedge \mathbf{23} \subset \boldsymbol{s} \wedge \mathbf{12}^2 \wedge \mathbf{123} \subset \boldsymbol{s} \wedge (\top \cup \mathbf{2} \cup \mathbf{23} \cup \mathbf{2} \wedge \mathbf{23}) \cup \boldsymbol{s} \wedge \mathbf{23}$$

Applying the $\boldsymbol{\gamma}$ graph-satisfiability map, yields –

$$\boldsymbol{\gamma}(\boldsymbol{s}\wedge\top\cup\boldsymbol{s}\wedge\mathbf{2}\cup\boldsymbol{s}\wedge\mathbf{23}) \leftarrow \boldsymbol{\gamma}(\boldsymbol{s}\wedge\mathbf{12}^2\wedge\mathbf{123}) \leftarrow \boldsymbol{\gamma}(\boldsymbol{s}\wedge\top\cup\boldsymbol{s}\wedge\mathbf{2}\cup\boldsymbol{s}\wedge\mathbf{23}\cup\boldsymbol{s}\wedge\mathbf{2}\wedge\mathbf{23})$$

This can in turn be simplified to –

$$\boldsymbol{\gamma}(\boldsymbol{s}\wedge\mathbf{2}) \leftarrow \boldsymbol{\gamma}(\boldsymbol{s}\wedge\mathbf{12}^2\wedge\mathbf{123}) \leftarrow \boldsymbol{\gamma}(\boldsymbol{s}\wedge\mathbf{2}\wedge\mathbf{23})$$

This last result can be seen as a partial rewrite rule owing to the presence of the subset-superset pair.

6.2.4 Opening a triple-intersection vertex

We can replace a three-hyperedge intersection incident on a common vertex with three simple edges on the boundary. The proof of this reduction rule is as follows –

$$\begin{aligned}
&\boldsymbol{s}\wedge\mathbf{123}\wedge\mathbf{124}\wedge\mathbf{134}\\
&\sim \boldsymbol{s}\wedge(\mathbf{23}\vee\mathbf{24}\wedge\mathbf{34})\cup\boldsymbol{s}\wedge(\mathbf{24}\vee\mathbf{23}\wedge\mathbf{34})\cup\boldsymbol{s}\wedge(\mathbf{34}\vee\mathbf{23}\wedge\mathbf{24})\\
&= \boldsymbol{s}\wedge(\top\cup\mathbf{23}\cup\mathbf{234})\cup\boldsymbol{s}\wedge(\top\cup\mathbf{24}\cup\mathbf{234})\cup\boldsymbol{s}\wedge(\top\cup\mathbf{34}\cup\mathbf{234})\\
&= \boldsymbol{s}\cup\boldsymbol{s}\wedge\mathbf{23}\cup\boldsymbol{s}\wedge\mathbf{24}\cup\boldsymbol{s}\wedge\mathbf{34}\cup\boldsymbol{s}\wedge\mathbf{234}\\
&\sim \boldsymbol{s}\wedge\mathbf{23}\cup\boldsymbol{s}\wedge\mathbf{24}\cup\boldsymbol{s}\wedge\mathbf{34}
\end{aligned}$$

6.3 Minimality of unsatisfiable hypergraphs

In the case of multi-graphs, we showed that satisfiability is invariant under homeomorphisms. We could then define a minimal unsatisfiable multi-graph to be one which is unsatisfiable, with every proper topological minor being totally satisfiable.

In the case of multi-hypergraphs, we have instead a list of reduction rules. The reduction rules make it harder to define minimality owing to several independent reasons –

1. Given a list of reduction rule, it is a computationally expensive to check if any of these rules apply to a given graph.

2. For most reduction rule, the right side (which is what we obtain after rewriting) is not a single graph – it is instead a union of graphs. This means that any notion of minimality for hypergraphs must incorporate the effect of rewriting as a union of graphs instead of a single graph.

3. Our list of reduction rules is not complete. We may find more reduction rules by rewriting at higher degree vertices and carrying out longer computations. Each additional reduction rule could make the minimality criterion stricter and would shrink the size of any minimal set of unsatisfiable hypergraphs.

4. Uniqueness of the minimal set of unsatisfiable hypergraphs is not guaranteed owing to the complexity of the reduction rules.

6.4 Computational results concerning mixed hypergraphs

In this section we provide a list of known totally satisfiable and unsatisfiable mixed-hypergraphs i.e. hypergraphs which have edges of size 1, 2 or 3. List of candidate hypergraphs are generated programmatically using SageMath's `nauty` package [15], which has various tools for generating canonically-labeled, non-isomorphic graphs with certain properties.

We list below several methods by which the satisfiability of a graph may be checked –

1. If a graph is finite and small (fewer than 6 hyperedges), we can satcheck it by passing it to a graph satchecker. Such a satchecker can be found in the `mhgraph_pysat_satcheck` function in the `sat.py` module.

2. If a graph is finite but not too small (7 to 20 hyperedges), we can decompose it using local rewriting at the min-degree vertex. For this, we use the `decompose` function found in the `graph_rewrite.py` module.

3. If a graph is finite and big (20+ hyperedges), then we can reduce it to a smaller graph by passing it to the `make_tree` function in the `operations.py` module.

4. Lastly, if a graph is infinite, we have to work out its satisfiability status manually using reduction rules. Reduction rules themselves can be generated by the `local_rewrite` function in the `graph_rewrite.py` module.

Presented in Figure 6.1 is a selection of unsatisfiable hypergraphs up to vertex size 5. A larger list can be found in the Appendix.

6.5 Computational results concerning triangulations

In this section we check a list of common/standard hypergraphs having edges of size exactly 3. These hypergraphs can be drawn as triangulations of various surfaces and are thus of interest when viewing GRAPHSAT from the topological point of view.

6.5.1 Thickening of graph edges

We outline below a method to create triangulations starting with simple graphs, such that the satisfiability status in going from the simple graph to the triangulation remains unchanged.

The process involves a local rewrite of every simple edge $\boldsymbol{ab}$ in a graph with the hyperedges $\boldsymbol{abc} \wedge \boldsymbol{acd} \wedge \boldsymbol{bcd}$, where $\boldsymbol{c}$ and $\boldsymbol{d}$ are new vertices not previously appearing in the graph.

For example, since $\boldsymbol{ab}^4$ is an unsatisfiable graph, we can thicken all its edges into hyperedges to form the triangulation

$$\boldsymbol{ab1} \wedge \boldsymbol{a12} \wedge \boldsymbol{b12} \wedge \boldsymbol{ab3} \wedge \boldsymbol{a34} \wedge \boldsymbol{b34} \wedge \boldsymbol{ab5} \wedge \boldsymbol{a56} \wedge \boldsymbol{b56} \wedge \boldsymbol{ab7} \wedge \boldsymbol{a78} \wedge \boldsymbol{b78}.$$

This thickening process is shown in Figure 6.2.

Similarly, thickening of the unsatisfiable graph $\boldsymbol{ab}^2 \wedge \boldsymbol{bc}^2$ results in the unsatisfiable triangulation shown in Figure 6.3. This triangulation is planar and can therefore be embedded in any surface. Thus, every surface has an unsatisfiable triangulation.

We also note that not all unsatisfiable triangulations are mere thickenings of unsatisfiable simple graphs. To prove that graph satisfiability is invariant

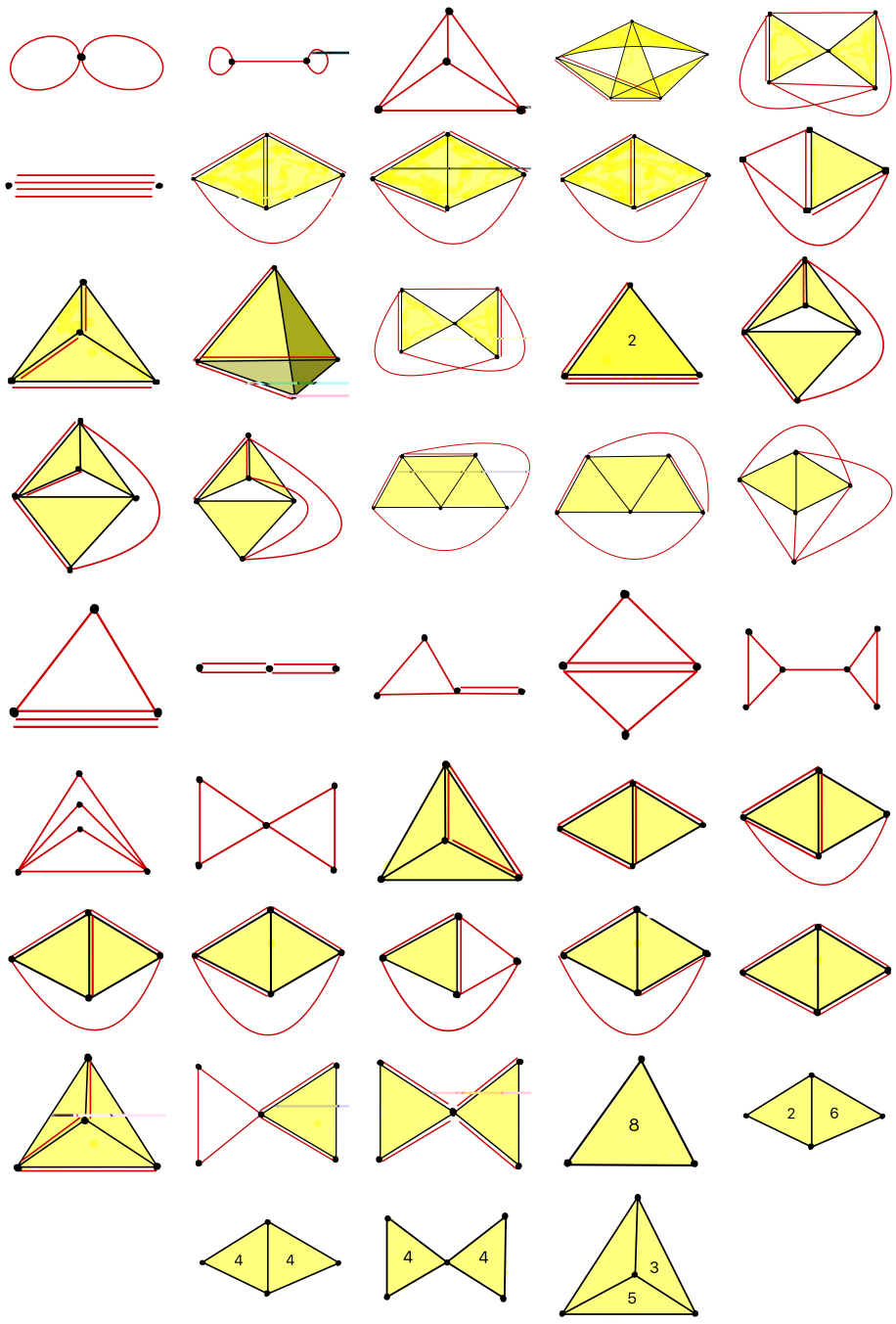


Figure 6.1: A selection of known unsatisfiable looped-multi-hypergraphs.

under thickening of edges, we observe the following –

$$\boldsymbol{s} \wedge \mathbf{123} \wedge \mathbf{134} \wedge \mathbf{234} \sim \boldsymbol{s} \wedge \mathbf{123}^2 \quad \text{(using edge-smoothing reduction rule)}$$
$$\sim \boldsymbol{s} \wedge \mathbf{12} \quad \text{(using leaf vertex reduction rule)}$$

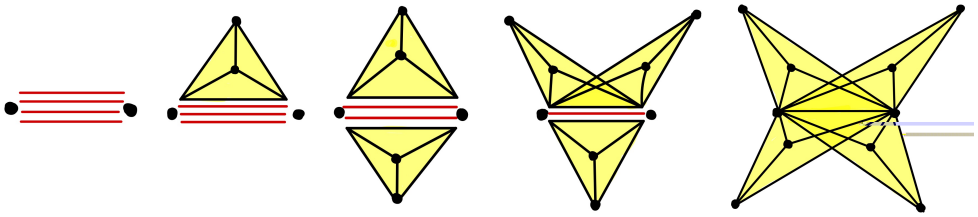


Figure 6.2: Thickening of the edges of $\boldsymbol{ab}^4$ shown step-by-step.

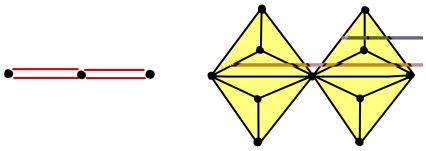


Figure 6.3: Thickening of the edges of $\boldsymbol{ab}^2 \wedge \boldsymbol{bc}^2$.

These reduction rules can be applied only because $\boldsymbol{s}$ does not have any edges incident on vertices **3** and **4** since there vertices are newly introduced by the thickening process.

6.5.2 Tetrahedron and prisms

The tetrahedron's wire-frame structure, i.e. its edges form the graph $\boldsymbol{K_4}$, a known unsatisfiable graph. The faces form the triangulation $\boldsymbol{abc} \wedge \boldsymbol{acd} \wedge \boldsymbol{abd} \wedge \boldsymbol{bcd}$. Using the reduction rule from §6.2.4 gives

$$\begin{aligned} \text{Tetrahedron} &\sim \boldsymbol{abc} \wedge \boldsymbol{ab} \cup \boldsymbol{abc} \wedge \boldsymbol{ac} \cup \boldsymbol{abc} \wedge \boldsymbol{bc} \\ &\sim \boldsymbol{ab} \cup \boldsymbol{ac} \cup \boldsymbol{bc} \\ &\sim \top \cup \top \cup \top \\ &= \top \end{aligned}$$

Thus the tetrahedron is totally satisfiable.

On the other hand the triangular prism has two possible minimal triangulations –

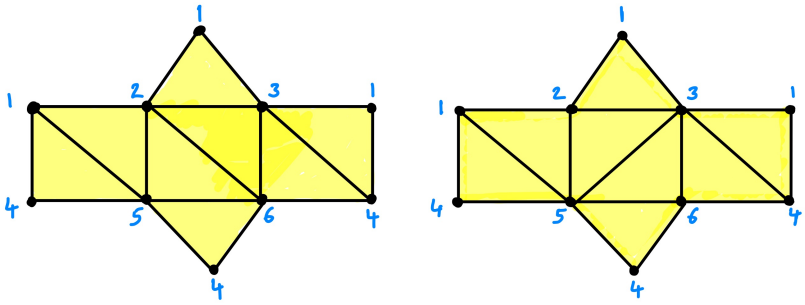
1. The symmetric triangulation, given by

$$\mathbf{123} \wedge \mathbf{125} \wedge \mathbf{134} \wedge \mathbf{145} \wedge \mathbf{236} \wedge \mathbf{256} \wedge \mathbf{346} \wedge \mathbf{456}.$$

2. The asymmetric triangulation, given by

$$\mathbf{123} \wedge \mathbf{125} \wedge \mathbf{136} \wedge \mathbf{145} \wedge \mathbf{146} \wedge \mathbf{236} \wedge \mathbf{256} \wedge \mathbf{456}.$$

These triangulations are shown in Figure 6.4. Passing them to the `decompose` function from the `graph_rewite` module tells us that both triangulations are totally satisfiable.



(a) The symmetric triangulation.

(b) The asymmetric triangulation.

Figure 6.4: Minimal triangulations of a triangular prism

```
_____ (python3.9) (scratch) <<prism-calculation>> _____
import mhgraph as mhg
import graph_rewrite as grw

prism1: mhg.MHGraph = mhg.mhgraph([[1,2,3], [1,2,5], [1,3,4], [1,4,5],
                                   [2,3,6], [2,5,6], [3,4,6], [4,5,6]])
grw.decompose(prism1)

prism2: mhg.MHGraph = mhg.mhgraph([[1,2,3], [1,2,5], [1,3,6], [1,4,5],
                                   [1,4,6], [2,3,6], [2,5,6], [4,5,6]])
grw.decompose(prism2)
```

Output:

```
(1, 2, 3)¹,(1, 2, 5)¹,(1, 3, 4)¹,(1, 4, 5)¹,(2, 3, 6)¹,(2, 5, 6)¹,(3, 4, 6)¹,(4, 5, 6)¹
    is SAT
(1, 2, 3)¹,(1, 2, 5)¹,(1, 3, 6)¹,(1, 4, 5)¹,(1, 4, 6)¹,(2, 3, 6)¹,(2, 5, 6)¹,(4, 5, 6)¹
    is SAT
```

6.5.3 Triangulation of a Möbius strip

A Möbius strip can be triangulated as $\mathbf{124} \wedge \mathbf{146} \wedge \mathbf{235} \wedge \mathbf{245} \wedge \mathbf{346} \wedge \mathbf{356}$ (also shown in Figure 6.5). Using `graph_rewrite.decompose`, we conclude that this triangulation is totally satisfiable.

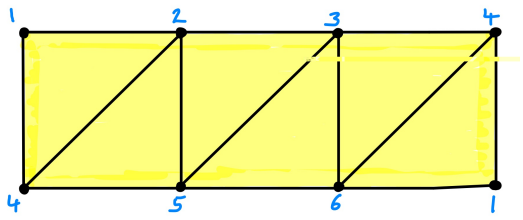


Figure 6.5: Triangulation of a Möbius strip

```
(python3.9) (scratch) <<mobius>>
import mhgraph as mhg
import graph_rewrite as grw

mobius_strip: mhg.MHGraph = mhg.mhgraph([[1, 2, 4], [1, 4, 6], [2, 3, 5],
                                         [2, 4, 5], [3, 4, 6], [3, 5, 6]])
grw.decompose(mobius_strip)
```

Output:

```
(1, 2, 4)¹,(1, 4, 6)¹,(2, 3, 5)¹,(2, 4, 5)¹,(3, 4, 6)¹,(3, 5, 6)¹ is SAT
```

6.5.4 Minimal triangulation of the real projective plane

The minimal triangulation of $\mathbb{RP}^2$ has six vertices and is unique up to relabeling of vertices. This triangulation is a classic result and is often referred to in literature as $\mathbb{RP}^2_6$. It can be written as
$\mathbf{123} \wedge \mathbf{326} \wedge \mathbf{461} \wedge \mathbf{412} \wedge \mathbf{526} \wedge \mathbf{561} \wedge \mathbf{153} \wedge \mathbf{364} \wedge \mathbf{425} \wedge \mathbf{534}$, and is shown in Figure 6.6. Using `graph_rewrite.decompose`, we conclude that this triangulation is totally satisfiable.

```
(python3.9) (scratch) <<RP2>>
import mhgraph as mhg
import graph_rewrite as grw
```

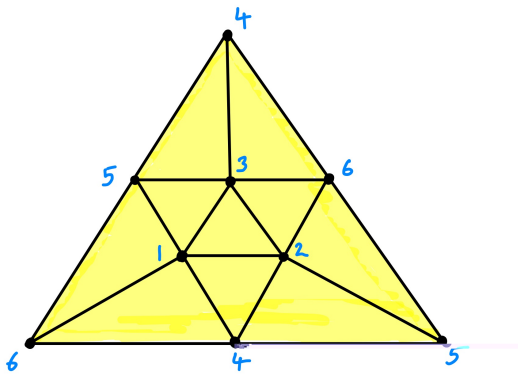


Figure 6.6: Minimal triangulation of the real projective plane, denoted $\mathbb{RP}^2_6$.

```
rp2: mhg.MHGraph
rp2 = mhg.mhgraph([[1, 2, 3], [3, 2, 6], [4, 6, 1], [4, 1, 2], [5, 2, 6],
                   [5, 6, 1], [1, 5, 3], [3, 6, 4], [4, 2, 5], [5, 3, 4]])
grw.decompose(rp2)
```

Output:

```
(1, 2, 3)¹,(3, 2, 6)¹,(4, 6, 1)¹,(4, 1, 2)¹,(5, 2, 6)¹,(5, 6, 1)¹,(1, 5, 3)¹,
    (3, 6, 4)¹,(4, 2, 5)¹,(5, 3, 4)¹ is SAT
```

6.5.5 Minimal triangulation of a Klein bottle

A Klein bottle has six distinct 8-vertex triangulations [16]. These triangulations all contain 16 distinct hyperedges and have a minimum vertex degree of 6. These large numbers make it difficult to determine the satisfiability status of these triangulations without committing to significant computational resources.

Of the six distinct triangulations, we checked but one – the "242 triangulation", given by the faces
123 $\wedge$ **372** $\wedge$ **153** $\wedge$ **175** $\wedge$ **147** $\wedge$ **162** $\wedge$ **642** $\wedge$ **168** $\wedge$ **148** $\wedge$ **248** $\wedge$ **643** $\wedge$**374** $\wedge$ **685** $\wedge$ **653** $\wedge$ **825** $\wedge$ **275**.

Passing it to `graph_rewrite.decompose` and waiting for several hours of computations results in the discovery that the 242 configuration is unsatisfiable. In fact, it is unsatisfiable even if we remove the **825** $\wedge$ **275** subgraph!

The 242 triangulation and its unsatisfiable subgraph are shown in Figure 6.7 for reference.

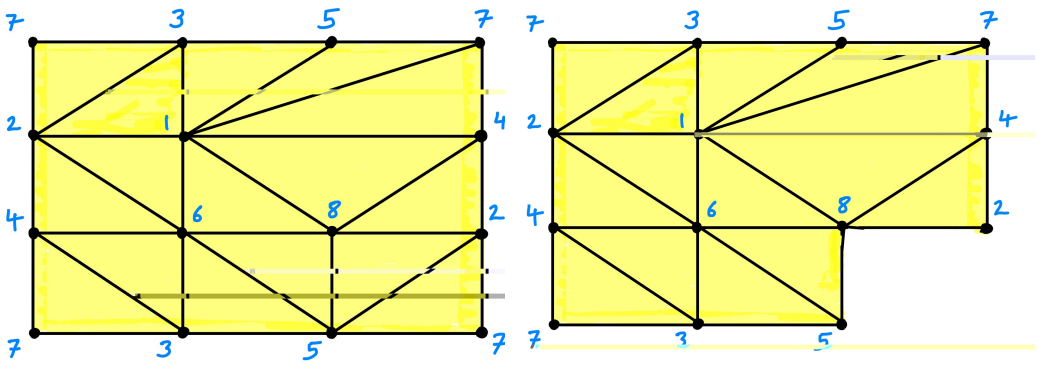


Figure 6.7: The "242 triangulation" of a Klein bottle and its unsatisfiable subgraph.

6.5.6 Minimal triangulation of a torus

The torus can be minimally triangulated [17] as –
126 $\wedge$ **267** $\wedge$ **237** $\wedge$ **371** $\wedge$ **674** $\wedge$ **745** $\wedge$ **715** $\wedge$ **156** $\wedge$ **412** $\wedge$ **452** $\wedge$ **523** $\wedge$**563** $\wedge$ **634** $\wedge$ **431**.

This triangulation is shown in Figure 6.8 and is found to be unsatisfiable by the `graph_rewrite.decompose` function. In fact, it is unsatisfiable even if we remove the **634** $\wedge$ **431** subgraph.

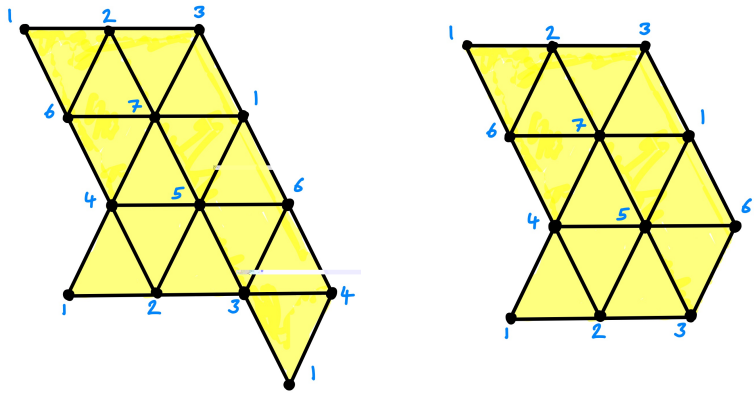


Figure 6.8: Minimal triangulation of a torus and its unsatisfiable subgraph.

6.6 Infinite GRAPHSAT

We recall from §2.3 that V is an arbitrary countable set, edges are nonempty sets of vertices, and graphs are nonempty multisets of edges. This definition does not exclude edges or graphs from being of countably infinite size. A graph with infinite edges, or edges of infinite size is an *infinite graph*.

We note that Cnfs, in a similar vein, can also be infinite. *Infinite Cnfs* either have infinitely many clauses, or have clauses of infinite size.

The notions of assignment, satisfiability, unsatisfiability – all carry over to infinite graphs and infinite Cnfs. The only notions that do not carry over are the questions of computational complexity since we cannot talk of program run-time for infinite instances of GRAPHSAT.

6.6.1 Infinitely many disconnected loops

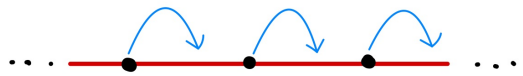
The graph $\mathbf{1} \wedge \mathbf{2} \wedge \mathbf{3} \wedge \ldots$ is a graph made of countably infinite disconnected self-loops. This graph is totally satisfiable because every connected component of it is.

6.6.2 Uniform infinite trees

For examples of totally satisfiable infinite graphs that are connected, we consider a family of tree graphs. For positive integer n, let $\boldsymbol{T_n}$ denote an infinite tree graph with each vertex being connected to exactly n different vertices via edges of size 2. These are also sometimes referred to in the literature as infinite trees of uniform degree n.

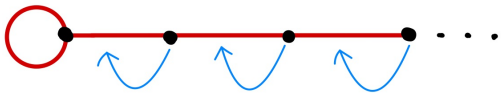
Each $\boldsymbol{T_n}$ is in fact totally satisfiable since we proved in §3.2 that every tree is totally satisfiable (and since the proof of Theorem 1 did not depend on the finiteness of the graph.

Another intuitive way to see that $\boldsymbol{T_2}$, for example, is totally satisfiable is that (at the level of Cnfs) each vertex can be used to satisfy its adjacent clause (see Figure 6.9). This results in a chain of assignments and each clause is satisfied in a style reminiscent of Hilbert's famous infinite hotel.



(a) T_2 (the infnite line graph) can be satisfed by vertex assignments shown using blue arrows.

(b) The infnite ray graph is similarly totally satisfable.



(c) The infnite ray graph with a looped on the tail is similarly totally satisfable.

Figure 6.9: Infinite line and ray graphs are totally satisfiable using vertex assignment. Each edge is satisfied by a unique vertex.

6.6.3 Infinite ray graph

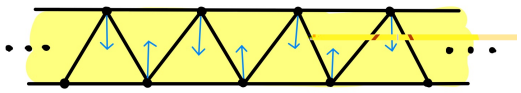
Proof by demonstrating a vertex assignment also for other infinite graphs. We should keep in mind that a valid vertex assignment can only help us remove a single adjacent edge (or hyperedge) per vertex. Also, the existence of a vertex assignment implies that the graph in question is totally satisfiable, but its nonexistence does not prove that the graph is unsatisfiable.

We use this technique to argue that the infinite ray graph with a looped tail (see Figure 6.9) is totally satisfiable. At the level of Cnfs, we can see that the tail vertex can be used to satisfy the loop. The vertex next to the loop satisfies the last edge, the vertex after that satisfies that last-but-one edge, and so on. This assignment is shown in the figure using arrows. This proves that the infinite ray as well as the infinite ray with looped tail are both totally satisfiable graphs.

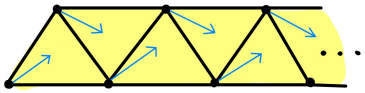
6.6.4 Bi-infinite strip

We next consider a thickened version of $\boldsymbol{T_2}$ made of hyperedges, as shown in Figure 6.10. We call this is bi-infinite strip and claim that it is totally satisfiable. The assignment that satisfies a given Cnf in this graph can be derived by using the arrows shown in the figure. Similarly, the mono-infinite

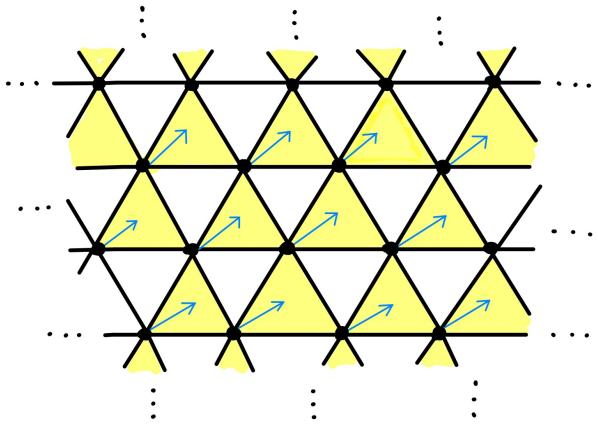
strip shown in Figure 6.10 is also totally satisfiable by the vertex assignment shown in the figure.



(a) The bi-infnite strip is totally satisfable using the vertex assignments shown using blue arrows.



(b) The mono-infnite strip is similarly totally satisfable.



(c) This triangulation is formed by taking a triangular uniform plane tiling and removing alternate tiles. It is totally satisfable using the vertex assignments shown.

Figure 6.10: Triangulations having vertex-assignments that use a single vertex to satisfy each hyperedge are totally satisfiable.

6.6.5 Plane tiling with missing alternate tiles

Lastly, we consider the tiling of the plane with alternate triangles and holes as shown in Figure 6.10. This triangulation can be satisfied by the vertex assignment shown in the same figure.

6.6.6 Compactness theorem and infinite GRAPHSAT

A graph is totally satisfiable if and only if every Cnf in it is satisfiable. The condition for every Cnf being satisfiable can itself be translated into a large Cnf if we allow the introduction of new variables.

For example, consider the single-edge graph $\boldsymbol{ab}$. There is a set of 4 Cnfs in the set $\boldsymbol{ab}$, and by relabeling the vertices, we can write $\boldsymbol{\gamma}(\boldsymbol{ab}) = \sigma\big((a_1 \vee b_1) \wedge (a_2 \vee \overline{b_2}) \wedge (\overline{a_3} \vee b_3) \wedge (\overline{a_4} \vee \overline{b_4})\big)$. We call this translation map from Graphs to Cnfs $\boldsymbol{\tau}$ (short for translation).

We use this map $\boldsymbol{\tau}$ to change the total-satisfiability question of a graph $\boldsymbol{g}$ from a universal quantification over all Cnfs in $\boldsymbol{g}$ to an existential quantification over all truth-assignments for the Cnf $\boldsymbol{\tau}(\boldsymbol{g})$. This change to existential quantification allows us to apply the Compactness Theorem.

In mathematical logic, the Compactness Theorem states that a set of first-order sentences has a model if and only if every finite subset of it has a model. In the context of GRAPHSAT, this means that an infinite graph is totally satisfiable if and only if every finite subgraph of it is totally satisfiable. This means we can always restrict out attention to studying only finite graphs. It also means that any unsatisfiable infinite graph must have an unsatisfiable finite subgraph.

Note of acknowledgment: The author wishes to thank Prof.s Anush Tserunyan and Yuliy Baryshnikov for pointing out this argument using the Compactness Theorem in the case of Cnfs.

6.7 Computational logistics

A computational procedure for finding all unsatisfiable looped-multi-hypergraphs can be carried out as follows —

Step 1. Start with all looped-multi-hypergraphs sorted from smallest to largest. This can be done by calling `nauty` from inside SageMath. We use `nauty` to generate all graphs with the following properties inside a specified vertex range –

- the graph must be connected.
- total number of vertices must lie within specified range.

- edge sizes can be 1 (loops), 2 (simple edges), or 3 (hyperedges).
- we disallow edges of size 4 in order to keep things simple.
- we specify that the minimum vertex degree of the graph should be 2 (since leaf vertices are known to be reducible).
- we allow an edge of size k to only have multiplicity less than 2^k.

Step 2. Pick a graph and apply all known reduction rules to it.

Step 3. Sat-check the irreducible part of the graph left over from Step 2 using brute-force strategy.

Step 4. If totally satisfiable, then pick the next graph and go back to Step 2.

Step 5. If unsatisfiable, then add the irreducible part to the list of known "minimal criminals".

While this procedure allows us to search for small unsatisfiable graphs, it is clear that we have to contend with an exponential blowup in the number of graphs as well as an exponential blowup in the number of Cnfs that need to be sat-checked as we keep increasing the vertex range. Table 6.4 tabulates the number of graphs for different vertex ranges.

Table 6.4: Exponential blowup in the number of graphs with increasing vertex count.

Number of connected graphs with less than 7 vertices	143
Number of minimal unsatisfiable irreducible simple graphs	4
Number of connected graphs with less than 6 vertices	10080
Number of minimal unsatisfiable irreducible L-M-H-graphs	202
Number of connected graphs with less than 7 vertices	48,364,386
Number of minimal unsatisfiable irreducible L-M-H-graphs	unknown

Chapter 7

Conclusion

An outcome of this work is the creation of a new graph decision problem – GRAPHSAT. We have demonstrated that 2GRAPHSAT is in complexity class P and has a finite obstruction set containing four simple graphs. The natural next step of exploring 3GRAPHSAT gave rise to the local graph rewriting theorem, which leveraged the fact that taking a union over all possible vertex-assignments preserves the satisfiability status of a graph. Using this theorem, we were able to generate a list of graph reduction rules and an incomplete list of obstructions to satisfiability of multi-hypergraphs.

7.1 Key results from this work

1. Notation and grammar for using graph-theoretic language in the context of Cnfs, SAT, and GRAPHSAT.

2. Set of all totally satisfiable simple graphs
   = Graphs forbidding $\{\boldsymbol{K_4}$, **butterfly graph**, **bow-tie graph**, $\boldsymbol{K_{1,1,3}}\}$ as topological minors.

3. Set of all totally satisfiable looped-multi-graphs = Language of 2GRAPHSAT decision problem
   = Graphs forbidding $\{\boldsymbol{K_4}$, **double-loop graph**, **dumbbell graph**, $\boldsymbol{ab}^4\}$ as topological minors.

4. There is a P-time algorithm for 2GRAPHSAT.

5. A proof of the graph local rewriting theorem –

$$\boldsymbol{g}[\boldsymbol{v}] \sim \text{sphere}(\boldsymbol{g},\boldsymbol{v}) \wedge \text{star}(\boldsymbol{g},\boldsymbol{v})[\boldsymbol{v}] = \bigcup_{\substack{\boldsymbol{g_i}\ \boldsymbol{h_i} : \text{Graph} \\ \boldsymbol{g_i} \wedge \boldsymbol{h_i} = \text{link}(\boldsymbol{g},\boldsymbol{v})}} \text{sphere}(\boldsymbol{g},\boldsymbol{v}) \wedge (\boldsymbol{g_i} \vee \boldsymbol{h_i})$$

6. An open-license Python package called `graphsat` [12] that implements the following –
    - For formulae in conjunctive normal forms (Cnfs), it implements variables, literals, clauses, Boolean formulae, and truth-assignments. It includes an API for reading, parsing and defining new instances.
    - For graph theory, the package includes graphs with self-loops, edge-multiplicities, hyperedges, and multi-hyperedges. It includes an API for reading, parsing and defining new instances.
    - For satisfiability of Cnfs and graphs, it contains a bruteforce algorithm, an implementation that uses the open-source sat-solver PySAT [13], and an implementation using the MiniSAT solver [14].
    - Additionally, for graph theory, the library also implements vertex maps, vertex degree, homeomorphisms, homomorphisms, subgraphs, and isomorphisms. This allows us to encode local rewriting rules as well as parallelized grid-based searching for forbidden structures.
    - Finally, `graphsat` has a tree-based recursive reduction algorithm that uses known local-rewrite rules as well as algorithms for checking satisfiability invariance of proposed reduction rules.
7. A list of graph reduction rules –
    - $\boldsymbol{s} \wedge \mathbf{12}^n \sim \boldsymbol{s} \wedge \mathbf{2}^{\lfloor n/2 \rfloor}$ i.e. leaf edge deletion.
    - $\boldsymbol{s} \wedge \mathbf{123}^n \sim \boldsymbol{s} \wedge \mathbf{23}^{\lfloor n/2 \rfloor}$ i.e. leaf hyperedge deletion.
    - Smoothing at degree 2 vertices, yielding graphs with fewer vertices.
    - Tucking in of extended fins, yielding graphs with fewer edges.
    - Opening a triple-intersection vertex.
8. Distinct ways to check the satisfiability status of a graph –
    - If a graph is finite and small (fewer than 6 hyperedges), then satcheck it by passing it to a graph satchecker. Such a satchecker can be found in the `mhgraph_pysat_satcheck` function in the `sat.py` module.

- If a graph is finite but not too small (7 to 20 hyperedges), then decompose it using local rewriting at the min-degree vertex. One can use the `decompose` function found in the `graph_rewrite.py` module.
- If a graph is finite and big (20+ hyperedges), then reduce it to a smaller graph by passing it to the `make_tree` function in the `operations.py` module.
- Lastly, if a graph is infinite, then ascertain its satisfiability status manually using reduction rules. Reduction rules themselves can be generated by the `local_rewrite` function in the `graph_rewrite.py` module.

9. An incomplete list of known unsatisfiable looped-multi-hypergraphs (pictured in Figure 6.1 and listed in the Appendix.

7.2 Future directions

Complexity class We showed that the complexity class of 2GRAPHSAT is P while the complexity class for 3GRAPHSAT is not known. Moreover, the effect of local graph rewriting on 3GRAPHSAT's complexity class is not known. Hence a key research question that arises is whether local rewriting preserves complexity, and whether it makes 3GRAPHSAT easier in practice.

3graphsat vs. 3sat We have an incomplete list of unsatisfiable looped-multi-hypergraphs. Questions that arise within this context are – whether the number of essential SAT-invariant graph reduction rules is finite? Even if the reduction rules are not finite, are they implementable in polynomial-time. Even if they are not implementable in polynomial time, it is possible that there is a polynomial-time check for demonstrating that none of the reduction rules apply to a given graph.

It is also not known if the number of minimal unsatisfiable graphs under these reduction rules is finite. So far we have found more than 200 distinct unsatisfiable and irreducible looped-multi-hypergraphs with less

Table 7.1: A table showing the satisfiability statuses of complete uniform hypergraphs. Non-obvious results are shown in boldface.

	b = 1	b = 2	b = 3	b = 4	b = 5	b = 6	b = 7	⋯
a = 1	sat	sat	sat	sat	sat	sat	sat	⋯
a = 2	-	sat	sat	**unsat**	unsat	unsat	unsat	⋯
a = 3	-	-	sat	sat	**unsat**	unsat	unsat	⋯
a = 4	-	-	-	sat	sat	**sat**	**unknown**	⋯
a = 5	-	-	-	-	sat	sat	**unknown**	⋯
a = 6	-	-	-	-	-	sat	sat	⋯
a = 7	-	-	-	-	-	-	sat	⋯
⋮	-	-	-	-	-	-	-	

than 7 vertices. If the complete list is infinite, it would imply that 3GRAPHSAT is not in complexity class P.

If 3GRAPHSAT is in P, this would give us an easy P-time heuristic check for 3SAT, simplifying some 3SAT cases, while not directly affecting the complexity class of 3SAT.

Complete-uniform hypergraphs Let $\boldsymbol{\mathcal{K}_{a,b}}$ denote the complete a-uniform hypergraph on b vertices. To construct $\boldsymbol{\mathcal{K}_{a,b}}$, we can start with b vertices and connect all $\binom{b}{a}$ combinations with a hyperedge of size a. We can also think of this as the $(a-1)$-skeleton of a $(b-1)$-simplex.

The hypergraph $\boldsymbol{\mathcal{K}_{a,b}}$'s satisfiability status is interesting because it combines the extreme of having all possible hyperedges connected (which can force unsatisfiability) with the extreme of each hyperedge being incident on a large number of vertices (which can force satisfiability).

For example, we know that $\boldsymbol{\mathcal{K}_{2,4}}$ is K_4, i.e. the complete simple graph on 4 vertices and is known to be totally satisfiable. On the other hand, the graph $\boldsymbol{\mathcal{K}_{2,3}}$ is C_3 is known to be totally satisfiable. Table 7.1 summarizes the known satisfiability statuses of various $\boldsymbol{\mathcal{K}_{a,b}}$ graphs. As seen in the table, the satisfiability-status of $\boldsymbol{\mathcal{K}_{4,7}}$ and $\boldsymbol{\mathcal{K}_{5,7}}$ are not known.

Random GraphSAT Both random instances of GraphSAT as well as graph analogues of random SAT were not covered as part of this research. We leave these questions for further inquiry in the future.

Generalized triangle intersection The generalized rule for n triangular hyperedges meeting at a common free vertex is not known. We do know the reduction rule only for $n = 3$ –

$$\boldsymbol{s} \wedge \mathbf{123} \wedge \mathbf{124} \wedge \mathbf{134} \sim \boldsymbol{s} \wedge \mathbf{23} \cup \boldsymbol{s} \wedge \mathbf{24} \cup \boldsymbol{s} \wedge \mathbf{34}$$

Reduction rules for $n \geq 4$ yield massive data-tables of resulting Cnfs, which we have so far been unable to group into a convenient set of graphs.

References

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71, Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158.

[2] L. A. Levin, "Universal enumeration problems," *Problemy Peredači Informacii*, vol. 9, no. 3, pp. 115–116, 1973, ISSN: 0555-2923.

[3] M. R. Krom, "The decision problem for a class of first-order formulas in which all disjunctions are binary," *Mathematical Logic Quarterly*, vol. 13, no. 1-2, pp. 15–20, 1967.

[4] S. Even, A. Itai, and A. Shamir, "On the complexity of timetable and multicommodity flow problems," *SIAM Journal on Computing*, vol. 5, no. 4, pp. 691–703, 1976.

[5] B. Aspvall, M. F. Plass, and R. E. Tarjan, "A linear-time algorithm for testing the truth of certain quantified boolean formulas," *Information Processing Letters*, vol. 8, no. 3, pp. 121–123, 1979, ISSN: 0020-0190.

[6] T. Feder, "Network flow and 2-satisfiability," *Algorithmica*, vol. 11, no. 3, pp. 291–319, Mar. 1994, ISSN: 1432-0541.

[7] T. J. Schaefer, "The complexity of satisfiability problems," in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, ser. STOC '78, San Diego, California, USA: ACM, 1978, pp. 216–226.

[8] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, Springer, 1972, pp. 85–103.

[9] N. Robertson and P. Seymour, "Graph minors I. Excluding a forest," *Journal of Combinatorial Theory, Series B*, vol. 35, no. 1, pp. 39–61, 1983, ISSN: 0095-8956.

[10] ——, "Graph minors XX. Wagner's conjecture," *Journal of Combinatorial Theory, Series B*, vol. 92, no. 2, pp. 325–357, 2004, Special Issue Dedicated to Professor W.T. Tutte, ISSN: 0095-8956.

[11] V. Karve and A. N. Hirani, "The complete set of minimal simple graphs that support unsatisfiable 2-cnfs," *Discrete Applied Mathematics*, vol. 283, pp. 123–132, 2020, ISSN: 0166-218X. DOI: 10.1016/j.dam.2019.12.017.

[12] ——, *Github: Vaibhavkarve/graphsat*, version v0.1.0, Apr. 2021. DOI: 10.5281/zenodo.4662169. [Online]. Available: github.com/vaibhavkarve/graphsat.

[13] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *SAT*, 2018, pp. 428–437. DOI: 10.1007/978-3-319-94144-8_26. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_26.

[14] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518, ISBN: 978-3-540-24605-3.

[15] B. D. McKay and A. Piperno, "Practical graph isomorphism, ii," *Journal of Symbolic Computation*, vol. 60, pp. 94–112, 2014, ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2013.09.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747717113001193.

[16] D. P. Cervone, "Vertex-minimal simplicial immersions of the Klein bottle in three space," en, *Geometriae Dedicata*, vol. 50, no. 2, pp. 117–141, Apr. 1994, ISSN: 0046-5755, 1572-9168. DOI: 10.1007/BF01265307. [Online]. Available: http://link.springer.com/10.1007/BF01265307.

[17] A. F. Möbius, "Zur theorie der polyëder und der elementarverwandtschaft," *Gesammelte werke*, vol. 2, pp. 513–560, 1886.

Appendix: List of known unsatisfiable hypergraphs

Presented below is a list of known unsatisfiable hypergraphs. This list was generated using SageMath's `nauty` module and then filtering for unsatisfiable graphs.

```
 1  (1)²
 2  (1), (2), (1,2)
 3  (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)
 4  (1,2), (1,3), (2,3), (1,2,3), (1,2,4), (1,2,5), (3,4,5)
 5  (1,2), (1,3), (1,4), (2,3), (2,4), (1,2,5), (3,4,5)
 6  (1,2), (1,3), (1,4), (2,3), (2,4), (1,3,4)
 7  (1,2), (1,3), (1,4), (2,3), (2,4), (1,3,5), (2,4,5)
 8  (1,2)², (1,4), (1,2,4)²
 9  (1,2), (1,3), (1,4), (2,3), (1,2,4), (1,3,4)
10  (1,2), (1,3), (1,4), (2,3), (1,2,4), (1,4,5), (2,3,5)
11  (1,2), (1,3), (1,4), (2,3), (1,2,4), (1,3,5), (2,4,5)
12  (1,2), (1,3), (1,4), (2,3), (1,2,4), (2,3,4)
13  (1,2), (1,3), (1,4), (2,3), (1,2,4), (1,2,5), (3,4,5)
14  (1,2), (1,3), (1,4), (2,3), (1,2,5), (1,4,5), (3,4,5)
15  (1,2), (1,3), (1,4), (2,3), (1,4,5), (2,3,5), (2,4,5)
16  (1,2), (1,3), (1,4), (2,3), (1,4,5), (2,4,5), (3,4,5)
17  (1)², (1,3)
18  (1,2)², (1,4)², (1,2,4)
19  (1,2), (1,3), (1,4)², (2,3), (2,3,4)
20  (1,2)², (1,3)², (2,3)
21  (1,2)², (1,4), (1,5), (1,4,5), (2,4,5)
22  (1,2), (1,3), (1,4), (1,5), (2,3), (2,4,5), (3,4,5)
23  (1,2), (1,3)², (2,4), (1,2,4), (2,3,4)
24  (1,2), (1,3), (1,5), (2,4), (3,5), (2,3,4), (2,4,5)
25  (1,2), (1,3), (2,4), (1,2,3), (1,2,4), (1,3,4)
```

26 (1), (1,3), (1,5), (1,3,5), (3,4,5)

27 (1,2), (1,3), (1,5), (2,4), (1,2,4), (2,3,5), (3,4,5)

28 (1), $(1,3)^2$, (1,3,4)

29 (1,2), (1,3), (2,4), (1,2,4), (2,3,4), (1,3,4)

30 $(1,2)^2$, (1,3), (2,4), (1,3,4), (2,3,4)

31 (1,2), (1,3), (2,4), (3,4), (1,2,3), (1,2,4)

32 (1,2), (1,3), (2,4), (3,4), (1,2,5), (1,3,4), (3,4,5)

33 (1,2), (1,3), (2,4), (3,4), (1,2,4), (1,4,5), (2,3,5)

34 (1), $(1,3)^2$, (3,4)

35 (1,2), (1,3), (1,5), (2,4), (3,4), (1,2,5), (3,4,5)

36 (1,2), (1,3), (2,4), (1,2,3), (1,2,4), (1,3,5), (2,4,5)

37 (1,2), (1,3), (2,4), (1,2,3), (1,2,4), (1,4,5), (2,3,5)

38 (1,2), (1,3), (2,4), (1,2,3), (1,2,4), (1,2,5), (3,4,5)

39 (1,2), (1,3), (2,4), (1,2,3), (1,2,5), (2,4,5), (3,4,5)

40 (1,2), (1,3), (2,4), (1,2,3), (1,4,5), (2,3,4), (2,3,5)

41 (1,2), (1,3), (2,3), (2,4), (3,5), (4,5), (1,4,5)

42 (1,2), (1,3), (2,4), (3,5), (4,5), (1,2,3), (1,4,5)

43 $(2)^2$, (3,5)

44 (1,2), (1,3), (2,4), (3,5), (4,5), (1,2,5), (1,3,4)

45 (1,2), (1,3), (2,4), (4,5), (1,2,3), (1,3,5), (1,4,5)

46 (1,2), (1,3), (2,4), (4,5), (1,2,3), (1,3,5), (3,4,5)

47 $(2)^2$, (2,3,5)

48 (1,2), (1,3), (2,4), (4,5), (1,2,3), (1,4,5), (2,3,5)

49 (1,2), (1,3), (2,4), (4,5), (1,2,3), (1,4,5), (3,4,5)

50 (2), (1,2), (1,3), (1,2,3), (1,3,5)

51 (1,2), (1,3), (2,4), (1,2,3), (1,3,5), (1,4,5), (2,4,5)

52 (1,2), (1,3), (2,4), (1,2,3), (1,4,5), (2,3,5), (2,4,5)

53 (1,2), (1,3), (2,4), (1,3,5), (1,4,5), (2,3,5), (2,4,5)

54 (1,2), (1,3), (2,4), (1,2,3), (1,3,5), (2,4,5), (3,4,5)

55 (1,2), (1,3), (4,5), (1,2,4), (1,2,5), (1,3,4), (1,3,5)

56 (1,2), (1,3), (4,5), (1,2,4), (1,3,5), (2,4,5), (3,4,5)

57 (1,2), (1,3), (4,5), (1,2,4), (1,3,4), (1,3,5), (2,4,5)

58 (1,2), (1,3), (4,5), (1,2,4), (1,3,4), (2,3,5), (2,4,5)

59 (1,2), (1,3), (4,5), (1,2,4), (1,3,4), (2,4,5), (3,4,5)

60 (1,2), (1,3), $(2,4)^2$, (1,3,4), (2,3,4)

61 (1,2), (1,3), (2,4), (4,5), (1,2,5), (1,3,4), (3,4,5)

```
62  (1,2), (1,3), (2,4), (4,5), (1,2,5), (1,3,4), (1,3,5)
63  (1,2), (1,3), (2,4), (4,5), (1,3,4), (2,3,5), (3,4,5)
64  (1,2), (1,3), (2,4), (4,5), (1,2,5), (1,3,5), (2,3,4)
65  (1,2), (1,3), (2,4), (4,5), (1,3,5), (2,3,4), (3,4,5)
66  (1,2), (1,3), (2,4), (4,5), (1,2,4), (1,3,5), (2,3,5)
67  (1,2), (1,3), (2,4), (1,2,3), (1,4,5), (2,4,5), (3,4,5)
68  (1,2), (1,3), (2,4), (1,2,3), (1,3,5), (2,3,4), (2,4,5)
69  (1,2), (1,3), (2,4), (1,2,4), (1,2,5), (1,3,4), (3,4,5)
```