SAT solving using conflict-driven clause learning and its application to classical planning

Bachelor's thesis in Computer Science
submitted by

Jan Vanhove

born on 8 September 1986 in Dendermonde, Belgium

matriculation number: 10-219-186

supervised by

Prof. Dr. Thomas Studer
and
Borja Sierra Miranda

June 2024

University of Berne
Institute of Computer Science
Logic and Theory Group

Acknowledgements

I thank Prof. Dr. Thomas Studer for suggesting I write a Bachelor's thesis on a topic I knew next to nothing about. I found it surprisingly engrossing.

Many thanks also to Borja Sierra for the tea and for being a great sounding board.

I had fun writing this thesis. I hope it shows.

Jan Vanhove
jan.vanhove@students.unibe.ch, janvanhove@gmail.com
Fribourg, Switzerland, June 2024

Abstract

The Boolean satisfiability (SAT) problem asks if, given a propositional formula, there is some way to assign truth values to its variables that render the formula true. SAT is NP-complete, so a general efficient algorithm for solving it may not exist. However, SAT problems encountered in industrial applications tend to exhibit some structure that make them efficiently solvable in practice. This Bachelor's thesis presents a popular approach to solving such SAT problems that is based on so-called conflict-driven clause learning (CDCL). It also discusses a handful of tweaks commonly implemented, in some form or other, in current CDCL-based SAT solvers. Finally, it explains how planning problems can be solved using SAT solvers and illustrates this process using two types of examples.

Contents

1 Introduction **1**
1.1 Context and goal . . . 1
1.2 Preliminaries . . . 2

2 Bare-bones conflict-driven clause learning **5**
2.1 Unit propagation . . . 5
2.2 Conflict analysis and clause learning . . . 9
2.2.1 Conflict sets . . . 9
2.2.2 Basic clause learning algorithm . . . 12
2.3 Non-chronological backtracking . . . 18
2.4 Putting it all together . . . 21
2.5 CDCL vs brute-force search vs DPLL . . . 22
2.6 On the limits of CDCL-based SAT solving . . . 24

3 Common tweaks **27**
3.1 Learning more useful clauses . . . 27
3.1.1 Unique implication points . . . 28
3.1.2 Learnt clause minimisation . . . 33
3.2 Better value assignments by decision . . . 38
3.3 Forgetting learnt clauses . . . 39
3.4 Restarting . . . 39

4 Application to classical planning **41**
4.1 Basics of classical planning . . . 41
4.2 Conversion to SAT . . . 43
4.3 Examples . . . 46
4.3.1 The blocks world . . . 47
4.3.2 The knight's tour problem . . . 49
4.4 On the effects of encoding details . . . 52

Chapter 1

Introduction

1.1 Context and goal

The Boolean satisfiability problem, or SAT for short, asks if, given some formula in propositional logic, there exists a value assignment to the variables occurring in this formula that render the entire formula true. While SAT is known to be NP-complete (Cook, 1971), many large instances of SAT emerging from applications in industry are routinely and efficiently solved. To give the reader some sense of the size of such SAT instances, the 2023 SAT Competition featured forty-nine sequential (i.e., non-parallel, non-cloud-based) SAT solvers that attempted to solve 400 benchmark problems (see `https://satcompetition.github.io/2023/`). These benchmarks are biased against relatively easy problems and can feature thousands or even millions of variables and clauses; see Figure 1.1 on the following page. The winning submission for 2023 was able to solve 284 of these 400 benchmarks within the allotted time of 5,000 CPU seconds per problem, and 346 of the problems could be solved by at least one of the sequential competitors.

The watershed in SAT solver engineering was the development of solvers that learn new implicates of the propositional formula that they are given (Marques-Silva & Sakallah, 1996, 1999). For reasons that will become clear in the next chapter, the technique used to derive such new implicates is known as **conflict-driven clause learning** (CDCL). My first goal in this thesis is to describe a generic SAT solver that is based on conflict-driven clause learning (Chapter 2). To that end, I will spell out and prove key properties of such a generic solver. In descriptions of CDCL-based SAT solvers, these properties are typically explained through examples or by means of rough proof sketches. Where detailed proofs exist, I found it difficult to reconcile the formalisms used in different publications with each other. The proofs that I present are my own and represent my attempt to elucidate the *how* and *why* of CDCL-based SAT solving in a coherent way.

My second goal is to present a handful of tweaks that are commonly implemented in current CDCL-based SAT solvers (Chapter 3). As a glance at the solver descriptions from any SAT Competition will reveal, SAT solvers differ considerably in their precise implementations. Some of these tweaks are situated mostly at the computational level (e.g., involving data structures and caching), others are situated more at the logical and conceptual level. At the latter level,

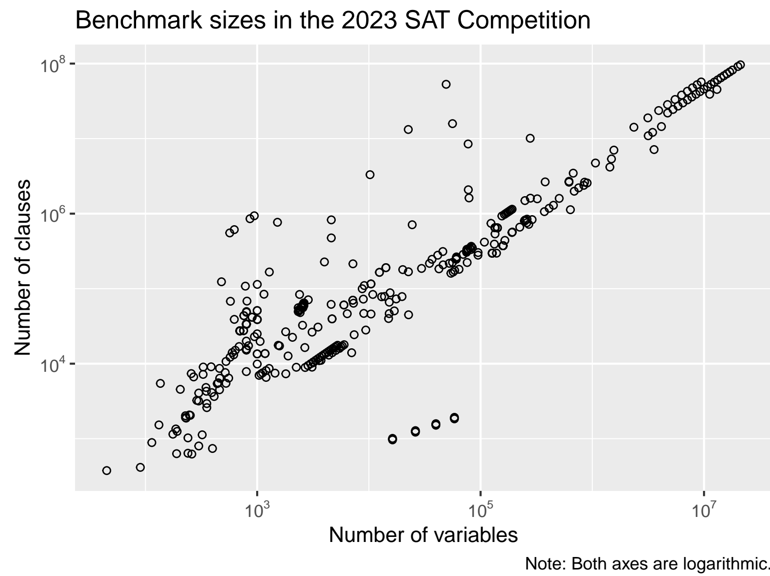

Figure 1.1: Sizes of the 400 benchmarks used in the 2023 SAT Competition. The benchmarks are available from `https://satcompetition.github.io/2023/`.

I have identified five tweaks that are implemented in some form or other in most current SAT solvers. I discuss two of these in some detail and the three others in broad strokes.

My third and final goal is to showcase an application of SAT solving: its use in solving planning problems (Chapter 4). I will explain what planning problems are and how they can be converted into SAT problems. Moreover, I will present two examples of what this conversion may look like in practice. I also provide computer code in the form of small package for the Julia programming language that can be used to generate similar examples.

1.2 Preliminaries

For ease of reference, I largely adopt the notation and basic definitions used by Marques-Silva et al. (2021) with a few minor modifications and additions. My wording unavoidably overlaps with theirs.

Definition 1.1. Propositional variables are denoted by lower-case Latin letters, with or without subscript (e.g., $x_1, x_2, \ldots, a, b, \ldots, z$). Given a set $\mathbb{V}$ of such propositional variables, **propositional formulas** – denoted by calligraphic uppercase letters such as $\mathcal{F}$ – are defined in the following way.

1. If $x \in \mathbb{V}$, then $\mathcal{F} = x$ is a propositional formula.

2. If $\mathcal{F}$ is a propositional formula, then so is $(\neg\mathcal{F})$ ('not $\mathcal{F}$').

3. If $\mathcal{F}, \mathcal{G}$ are both propositional formulas, then so is $(\mathcal{F} \vee \mathcal{G})$ ('$\mathcal{F}$ or $\mathcal{G}$').

4. If $\mathcal{F}, \mathcal{G}$ are both propositional formulas, then so is $(\mathcal{F} \wedge \mathcal{G})$ ('$\mathcal{F}$ and $\mathcal{G}$').

While one can include in this definition further connectives such as implication and equivalence, these are not needed in what follows. The set of all propositional variables that occur in a propositional formula $\mathcal{F}$ will be denoted by $\mathrm{var}(\mathcal{F})$.

If $x \in \mathbb{V}$ is a propositional variable, then we call both x and its complement $\neg x$ **literals**. We will write $\overline{x}$ in lieu of $\neg x$. Literals will be denoted by $\ell_1, \ell_2, \ldots$. Finite sets of literals are referred to as **clauses** and are denoted by $\mathfrak{c}_1, \mathfrak{c}_2, \ldots$. Throughout this thesis, such clauses are to be interpreted as disjunctions, that is, a clause

$$\mathfrak{c}_1 = \{\ell_1, \ell_2, \ldots, \ell_n\}$$

is to be interpreted as

$$\mathfrak{c}_1 = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_n.$$

The cardinality of a clause is referred to as its **width**.

If a formula $\mathcal{F}$ is represented as a conjunction of clauses, that is, if

$$\mathcal{F} = \mathfrak{c}_1 \wedge \cdots \wedge \mathfrak{c}_m,$$

we say that $\mathcal{F}$ is in **conjunctive normal form** (CNF). Finite sets of clauses are to be interpreted as conjunctions. ♢

Thus, for instance, the formula

$$\mathcal{F} = (x_1 \vee \overline{x}_2) \wedge (x_2 \vee x_3 \vee \overline{x}_4)$$

may be represented as

$$\mathcal{F} = \{\{x_1, \overline{x}_2\}, \{x_2, x_3, \overline{x}_4\}\}$$

and vice versa.

Definition 1.2. Propositional variables can be assigned values. For the purposes of SAT solving, a **value assignment** is defined as a function $\nu : \mathbb{V} \to \{0, 1, \mathfrak{u}\}$, where $\nu(x) = \mathfrak{u}$ signifies that the value of x is as yet undefined. If $\nu(x) = \mathfrak{u}$, we call the variable x **unassigned**. If $\nu(x) \in \{0, 1\}$, we call the variable x **assigned**. Literals inherit their assignment status (i.e., assigned or unassigned) from their variable.

If $\nu(\mathbb{V}) \subset \{0, 1\}$, ν is called a **complete assignment**. Otherwise, i.e., if $\mathfrak{u} \in \nu(\mathbb{V})$, ν is called a **partial assignment**.

If ν, ν' are value assignments over $\mathbb{V}$ such that both $\nu(x) = 1$ implies $\nu'(x) = 1$ and $\nu(x) = 0$ implies $\nu'(x) = 0$ for all $x \in \mathbb{V}$, we call ν' an **extension** of ν. We write $\nu \subset \nu'$, and we also say that ν' extends ν.

For a given assignment ν over $\mathbb{V}$, the value $\mathcal{F}^\nu$ of a propositional formula $\mathcal{F}$ is defined as follows.

1. If $\mathcal{F} = x, x \in \mathbb{V}$, then $\mathcal{F}^\nu = \nu(x)$.

2. If $\mathcal{F} = (\neg\mathcal{G})$, then

$$\mathcal{F}^\nu = \begin{cases} 0, & \text{if } \mathcal{G}^\nu = 1, \\ 1, & \text{if } \mathcal{G}^\nu = 0, \\ \mathfrak{u}, & \text{otherwise.} \end{cases}$$

3. If $\mathcal{F} = (\mathcal{E} \vee \mathcal{G})$, then

$$\mathcal{F}^\nu = \begin{cases} 1, & \text{if } \mathcal{E}^\nu = 1 \text{ or } \mathcal{G}^\nu = 1, \\ 0, & \text{if } \mathcal{E}^\nu = 0 \text{ and } \mathcal{G}^\nu = 0, \\ \mathfrak{u}, & \text{otherwise.} \end{cases}$$

4. If $\mathcal{F} = (\mathcal{E} \wedge \mathcal{G})$, then

$$\mathcal{F}^\nu = \begin{cases} 1, & \text{if } \mathcal{E}^\nu = 1 \text{ and } \mathcal{G}^\nu = 1, \\ 0, & \text{if } \mathcal{E}^\nu = 0 \text{ or } \mathcal{G}^\nu = 0, \\ \mathfrak{u}, & \text{otherwise.} \end{cases}$$

If $\mathcal{F}^\nu = 1$, the assignment ν is called a **model** (of $\mathcal{F}$) or a **satisfying assignment**. ♢

If a model ν is a partial assignment, we can easily obtain a complete satisfying extension ν' of ν, for instance by setting $\nu'(x) = 0$ for all $x \in \mathbb{V}$ with $\nu(x) = \mathfrak{u}$.

Definition 1.3. Let $\mathcal{F}$ and $\mathcal{G}$ be propositional formulas. If $\mathcal{F}^\nu = 1$ implies $\mathcal{G}^\nu = 1$ for any complete value assignment ν, then we call $\mathcal{G}$ an **implicate** of $\mathcal{F}$. ♢

Clauses of a CNF formula fall into one of four categories.

Definition 1.4 (Categories of CNF clauses)**.** Let $\mathfrak{c} = \{\ell_1, \ldots, \ell_n\}$ be a clause in some propositional CNF formula. If $\ell_i^\nu = 0, i = 1, \ldots, n$, then we call $\mathfrak{c}$ **falsified**. If $\ell_i^\nu = 1$ for some $1 \le i \le n$, then we call $\mathfrak{c}$ **satisfied**. If $\ell_i^\nu = \mathfrak{u}$ for exactly one $1 \le i \le n$ and $\ell_j^\nu = 0$ for all $1 \le j \le n, j \ne i$, then we call $\mathfrak{c}$ **unit**. In all other cases, we call $\mathfrak{c}$ **unresolved**. ♢

Throughout this thesis, it is assumed that the propositional formulas we are working with are in conjunctive normal form, and I will use whichever of these two representations is more convenient. For the purposes of SAT solving, the assumption that the formulas are in CNF is not a genuine restriction: Given an arbitrary formula $\mathcal{F}$ in propositional logic, efficient algorithms exist for generating a formula $\mathcal{F}'$ in CNF with the properties that $\mathcal{F}'$ is satisfiable if and only if $\mathcal{F}$ is satisfiable and that any model ν of $\mathcal{F}'$ is also a model of $\mathcal{F}$ (e.g., the Tseitin transformation, see Tseitin, 1983; also see Kuiter et al., 2022).

Since we define clauses to be sets of literals, we may assume in the following that any given literal occurs at most once in any given clause. Similarly, since we define CNF formulas as sets of clauses, we may assume that a CNF formula does not contain duplicated clauses. Moreover, since any disjunction featuring both a literal ℓ and its negation $\neg\ell$ are automatically satisfied by any complete assignment and since we interpret clauses as disjunctions, we may further assume that any given clause does not contain both ℓ and its negation.

Further definitions and conventions will be introduced when they are easier to motivate.

Chapter 2

Bare-bones conflict-driven clause learning

Conflict-driven clause learning SAT solvers are based on three key principles. The first is that if a clause is unit, this clause's unassigned literal needs to evaluate to 1 if the entire formula is to be satisfied. This principle is the basis of a subroutine known as unit propagation. The second key principle is that if a given provisional value assignment causes a clause to be falsified, this provides us with information about the problem at hand that we can leverage when looking for a solution. This principle leads to the twin practices of conflict analysis and clause learning. Third, conflict analysis yields information as to which provisional value assignments ought to be revised in a process known as backtracking. In this chapter, I first elucidate these three key principles and the routines they give rise to. I then explain how these can be combined into a no-frills CDCL-based algorithm for SAT solving. In the next chapter, I will discuss a few tweaks that are commonly implemented in SAT solvers.

2.1 Unit propagation

The backbone of SAT solvers is the **unit clause rule** (Davis & Putnam, 1960).

Lemma 2.1 (Unit clause rule)**.** *Let $\mathcal{F}$ be a propositional formula in* CNF*. Let ν be a value assignment over $\mathbb{V} \supset var(\mathcal{F})$. Let $\mathfrak{c} \in \mathcal{F}$ be a unit clause under ν with ℓ as its sole unassigned literal. Then any value assignment ν' that extends ν and satisfies $\mathcal{F}$ must yield $\ell^{\nu'} = 1$.*

Proof. The CNF formula $\mathcal{F}$ is satisfied if and only if all its clauses are satisfied. An extension ν' of ν satisfies the unit clause $\mathfrak{c}$ if and only if $\ell^{\nu'} = 1$. □

By the unit clause rule, the unassigned literal in any unit clause needs to be set to 1 in any satisfying extension of the current value assignment. That is, if we are in the process of constructing a value assignment ν and the unassigned literal in a unit clause has the form $x \in \mathbb{V}$, we need to set $\nu(x) = 1$; if the unassigned literal has the form $\overline{x}$, we need to set $\nu(x) = 0$. This may result in further clauses becoming unit, and the unit clause rule can again be applied to these. The iterated application of the unit clause rule is referred to as **unit propagation** (also known as **Boolean constraint propagation**). If a variable was assigned a value in $\{0, 1\}$ through unit propagation, it is said to be **implied**. Before introducing the unit propagation routine more formally, let's take a look at an example.

Example 2.2 (Unit propagation)**.** Consider the propositional formula

$$\mathcal{F} = \underbrace{\left(\bar{a} \vee \bar{b}\right)}_{=\mathfrak{c}_1} \wedge \underbrace{(b \vee \bar{c})}_{=\mathfrak{c}_2} \wedge \underbrace{(c \vee d)}_{=\mathfrak{c}_3} \wedge \underbrace{\left(e \vee \bar{f}\right)}_{=\mathfrak{c}_4} \wedge \underbrace{(b \vee g)}_{=\mathfrak{c}_5}.$$

and assume that $\nu(a) = 1$, whereas all other variables occurring in $\mathcal{F}$ are as yet unassigned. The clause $\mathfrak{c}_1$ contains exactly one unassigned variable and all of its other literals evaluate to 0. Hence, $\mathfrak{c}_1$ is unit. Applying the unit clause rule, we find that we need to set $\nu(b) = 0$ so that $\left(\bar{b}\right)^{\nu} = 1$. This causes the clauses $\mathfrak{c}_2$ and $\mathfrak{c}_5$ to become unit, which induces the assignments $\nu(c) = 0, \nu(g) = 1$. This, in turn, causes clause $\mathfrak{c}_3$ to become unit, leading to the assignment $\nu(d) = 1$. At this juncture, no unit clauses remain in $\mathcal{F}$, and the present bout of unit propagation ends. ♢

As the example demonstrates, unit propagation not only identifies some of the value assignments necessary in a model of a formula given the current partial assignment, it can also identify the unit clauses that led to these additional value assignments. In Example 2.2, the value assignment $\nu(b) = 0$ was derived from clause $\mathfrak{c}_1$, $\nu(c) = 0$ was derived from $\mathfrak{c}_2$, $\nu(g) = 1$ from $\mathfrak{c}_5$, and $\nu(d) = 1$ from $\mathfrak{c}_3$.

A variable can be also assigned a value by a decision if unit propagation terminates and the formula does not contain any falsified clauses. Such variables are known as **decision variables**. In this case, SAT solvers pick one of the as yet unassigned variables x and assign to it a value $\nu(x) \in \{0, 1\}$. While sophisticated procedures exist for deciding on this new value assignment, for our present purposes, we only need to assume that some helper function `PickBranchVariable()` exists that selects an unassigned variable and assigns either 0 or 1 to it. In the examples to follow, I assume that the function `PickBranchVariable()` selects the lexicographically first unassigned variable in $\mathbb{V}$ and assigns to it the value 1. More sophisticated approaches are discussed in Chapter 3.

Definition 2.3. Let $\mathcal{F}$ be a CNF formula and let $x \in \text{var}(\mathcal{F})$ be a propositional variable it contains. If x is an implied variable whose value was set after applying the unit clause rule to the clause $\mathfrak{c} \in \mathcal{F}$, we refer to $\mathfrak{c}$ as the **antecedent** of the value assignment of x. We write $\alpha(x) = \mathfrak{c}$. If a variable was assigned a value in $\{0, 1\}$ by a decision rather than by unit propagation, we set its antecedent to $\alpha(x) = \mathfrak{d}$, where we assume that $\mathfrak{d} \notin \mathcal{F}$. If a variable has not yet been assigned a value in $\{0, 1\}$, that is, if $\nu(x) = \mathfrak{u}$, we write $\alpha(x) = \mathfrak{n}$, where we assume that $\mathfrak{n} \notin \mathcal{F}$ and $\mathfrak{n} \neq \mathfrak{d}$. ♢

The antecedent of any given $x \in \text{var}(\mathcal{F})$ need not be unambiguously defined by the propositional formula $\mathcal{F}$ and the current partial value assignment ν. Consider, for instance, the propositional formula

$$\mathcal{F} = \underbrace{(a)}_{=\mathfrak{c}_1} \wedge \underbrace{(b)}_{=\mathfrak{c}_2} \wedge \underbrace{(\bar{a} \vee c)}_{=\mathfrak{c}_3} \wedge \underbrace{\left(\bar{b} \vee c\right)}_{=\mathfrak{c}_4}.$$

A single bout of unit propogation will yield the value assignments $\nu(a) = \nu(b) = \nu(c) = 1$. However, after having set the value assignments of a and b, the assignment $\nu(c) = 1$ can be obtained by applying the unit clause rule to either $\mathfrak{c}_3$ or $\mathfrak{c}_4$. That said, our unit propagation

routine will consider one unit clause at a time – not all unit clauses simultaneously. Whichever unit clause the routine used to determine the value assignment of x counts as its antecedent.

By alternating unit propagation and setting value assignments by decision, a decision stack is constructed. At the outset, before any value assignment by decision has been carried out, the decision stack has depth 0. Any value assignments carried out through unit propagation are added to the stack at this depth, without incrementing the stack's depth. If a value assignment by decision is carried out, the stack's depth is incremented, and both the newly set variable as well as all new value assignments that follow by unit propagation are added to the stack at its new depth.

Definition 2.4. The stack depth with which a value assignment is associated is referred to as its **decision level**; we write $\delta(x)$ for the decision level of value assignment of a variable x. If a variable x is unassigned, its decision level is defined to be $\delta(x) = \mathfrak{u}$. $\diamondsuit$

In the context of CDCL-based SAT-solving, it is convenient to keep track of both the current value assignments as well as the antecedents and decision levels associated with them by packing this information in a set of tuples, which I call an augmented value assignment.

Definition 2.5. Let $\mathbb{V}$ be a set of propositional formulas and let ν be a value assignment over $\mathbb{V}$. Then the **augmentation** of ν is defined as

$$N := \{(x, \nu(x), \delta(x), \alpha(x)) : x \in \mathbb{V}\}.$$

(N is the uppercase version of ν, the thirteenth letter in the Greek alphabet) We also call N an **augmented value assignment**. $\diamondsuit$

It is common in the literature to abuse notation and use the same symbol ν to refer to both the value assignment (a map from $\mathbb{V}$ to $\{0, 1, \mathfrak{u}\}$, i.e., technically a set of 2-tuples) and its augmentation (a set of 4-tuples). Since the value assignment proper may exist independently of a CDCL-based SAT solver but its augmentation only makes sense when discussing the workings of such solvers, it may make sense to keep both concepts distinct. Given an augmented value assignment N, it is clearly child's play to deduce the corresponding value assignment ν.

It is also convenient to be able to talk about a literal's decision level and antecedent, which are defined straightforwardly by inheritance from the literal's variable. In other words, if $\ell = x$ or $\ell = \overline{x}, x \in \mathbb{V}$, we define $\alpha(\ell) := \alpha(x)$ and $\delta(\ell) := \delta(x)$.

With the key terms defined, it is time to present the pseudocode for the unit propagation routine, see Algorithm 2.1. The `UnitClauseRule()` helper function on line 6 identifies the variable var of the unassigned literal in a unit clause and deduces which value val to set it to in order to satisfy the clause. The `Assign()` helper function on the next line removes the 4-tuple associated with var from the current augmented value assignment N and adds the 4-tuple that it is fed to N. The routine terminates with the return value `false` if the value assignment causes a clause to be falsified and with the return value `true` if each clause in $\mathcal{F}$ is either satisfied or unresolved. The meaning of the third line will become clear in the next definition.

```
Global: A set of clauses representing a propositional formula F in CNF
        A set of 4-tuples N representing an augmented value assignment, to be updated
        The current decision level DLevel
Output: F^ν ≠ 0?
1  UnitPropagation()
2      if exists falsified c ∈ F then
3          α(⊥) := c
4          return false
5      while exists unit c_i ∈ F do
6          (var, val) ← UnitClauseRule(c_i)
7          Assign(var, val, DLevel, c_i)
8          if exists falsified c_j ∈ F then
9              return false
10     return true
```

Algorithm 2.1: The unit propagation routine.

From now on, it is understood that any augmented value assignment N has been constructed by alternatingly applying unit propagation and value assignment by decision to some CNF formula $\mathcal{F}$ with $\text{var}(\mathcal{F}) \subset \mathbb{V}$.

Definition 2.6. Let N be an augmented value assignment. The **implication graph** corresponding to N is defined to be the tuple $I = (V_I, E_I)$, where V_I is the vertex set and E_I is the edge set, which are in turn defined as follows. First,

$$V_I := \begin{cases} \{x \in \mathbb{V} : \delta(x) \neq \mathfrak{u}\}, & \text{if no clause } \mathfrak{c} \in \mathcal{F} \text{ is falsified,} \\ \{x \in \mathbb{V} : \delta(x) \neq \mathfrak{u}\} \cup \{\perp\}, & \text{else,} \end{cases}$$

where we assume that $\perp \notin \mathbb{V}$. If unit propagation causes clause $\mathfrak{c} \in \mathcal{F}$ to become satisfied, we also define the antecedent of $\perp$ to be $\alpha(\perp) = \mathfrak{c}$, whereas the decision level of $\perp$ is the current decision level. More precisely, if $\perp \in V_I$, then

$$\delta(\perp) := \max\{\delta(x) : x \in \mathbb{V}, \delta(x) \neq \mathfrak{u}\}.$$

Second, we define

$$E_I := \{(a, b) \in V_I \times V_I : a \in \alpha(b) \text{ or } \overline{a} \in \alpha(b), \text{and } a \neq b\}. \qquad \diamondsuit$$

Since an implication graph's edges reflect the flow of unit propagation, it is a directed acyclic graph. Further, since each variable with an ingoing edge has exactly one clause as its antecedent, we may label any directed edge $(a, b) \in E_I$ with $\alpha(b)$. From now on, it is understood that any implication graph I is based on some augmented value assignment N. Further, it is common to slightly abuse notation when depicting implication graphs and to represent an assigned variable x as x if $\nu(x) = 1$ and as $\overline{x}$ if $\nu(x) = 0$. I, too, will adopt this convention.

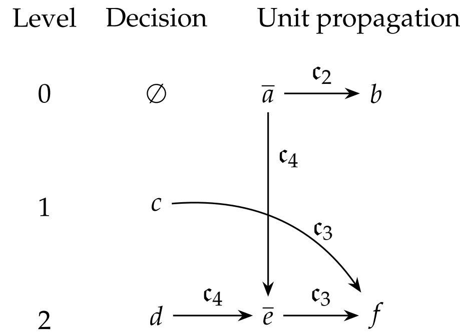

Figure 2.1: The implication graph to Example 2.7. It is assumed that value assignment by decision proceeds lexicographically and always assigns the value 1 to the variable.

Example 2.7 (Implication graph without falsified clauses)**.** Consider the formula

$$\mathcal{F} = \underbrace{\overline{a}}_{=c_1} \wedge \underbrace{(a \vee b)}_{=c_2} \wedge \underbrace{(\overline{c} \vee e \vee f)}_{=c_3} \wedge \underbrace{\left(a \vee \overline{d} \vee \overline{e}\right)}_{=c_4}.$$

Alternating unit propagation and our rudimentary `PickBranchVariable()` routine, we find a model for $\mathcal{F}$ without running into any conflicts; see Figure 2.1. ♢

Example 2.8 (Implication graph with a falsified clause)**.** Consider the formula

$$\mathcal{F} = \underbrace{\left(\overline{a} \vee \overline{d}\right)}_{=c_1} \wedge \underbrace{\left(\overline{a} \vee \overline{b} \vee \overline{f}\right)}_{=c_2} \wedge \underbrace{(\overline{c} \vee e \vee f)}_{=c_3} \wedge \underbrace{(\overline{c} \vee \overline{g})}_{=c_4} \wedge \underbrace{(\overline{e} \vee g)}_{=c_5}.$$

Alternating unit propagation and our rudimentary `PickBranchVariable()` routine, we obtain a falsified clause; see Figure 2.2. Note also that no value assignment can be made before the first value assignment by decision. ♢

2.2 Conflict analysis and clause learning

If unit propagation leads to a conflict, SAT solvers need to roll back one or several value assignments and explore a hitherto unexplored region of the search space. As their name suggests, conflict-driven clause-learning algorithms let guide the search for a satisfying assignment by adding to the propositional formula additional implicates that are deduced from conflicts. In what follows, I first discuss a general approach for deducing such implicates. I then show that the approach described by Marques-Silva & Sakallah (1999) is a special case of this more general approach.

2.2.1 Conflict sets

Darwiche & Pipatsrisawat (2021) discuss the importance of particular vertex subsets of implication graphs in SAT solving. The following definition is distilled from their discussion.

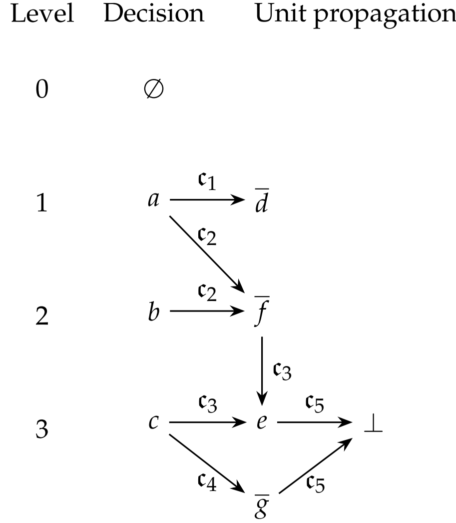

Figure 2.2: The implication graph to Example 2.8. It is assumed that value assignment by decision proceeds lexicographically and always assigns the value 1 to the variable.

Definition 2.9. Let $I = (V_I, E_I)$ be the implication graph associated with a value assignment that led to a conflict, i.e., $\perp \in V_I$. Then a **conflict partition** of I is any tuple (A, B) of vertex subsets satisfying the following conditions:

1. A, B is a partition of V_I. That is, $A \cup B = V_I, A \cap B = \emptyset$.
2. Any decision variable of I belongs to A. That is, for all $x \in V_I$, $\alpha(x) = \mathfrak{d}$ implies $x \in A$.
3. The conflict node belongs to B, i.e., $\perp \in B$.
4. For all $a \in A, b \in B$, we have $(b, a) \notin E_I$. In other words, no edges flow from B to A.

Given a conflict partition (A, B) of I, we define its associated **conflict set** C as

$$C := \{a \in A : (a, b) \in E_I \text{ for some } b \in B\}. \qquad \diamondsuit$$

As the next lemma shows, the current value assignments to the variables in the conflict set cannot be extended to a satisfying assignment.

Lemma 2.10. *Let (A, B) be a conflict partition of an implication graph associated with a conflict, with $C \subset A$ being this partition's conflict set. Let ν be the value assignment that led to the conflict. Let $\widetilde{\nu} \subset \nu$ be the value assignment defined through*

$$\widetilde{\nu}(x) = \begin{cases} \nu(x), & \text{if } x \in C, \\ \mathfrak{u}, & \text{else.} \end{cases}$$

Then unit propagation applied to $\widetilde{\nu}$ will yield a conflict.

Proof. We prove this lemma by induction on $n = \#B$. In the base case ($n = 1$), $B = \{\perp\}$. The conflict set comprises exactly all variables in $\alpha(\perp)$. So the value assignments to these variables imply the conflict.

Now consider the case $\#B = n + 1$. Since I is a directed acyclic graph, there exists at least one $b \in B \setminus \{\perp\}$ that does not have any edges entering it from another vertex in B. Either this b does not have any edges entering it at all (if its value was set by unit propagation at decision level 0) or all edges entering it originate in C. In either case, we can use unit propagation to determine the value assignment of this b in any satisfying extension of $\widetilde{\nu}$. Since there are no vertices in B with edges going into b, we may consider the conflict partition $(A \cup \{b\}, B \setminus \{b\})$. By the induction hypothesis, unit propagation applied to the value assignment to the conflict set associated with $(A \cup \{b\}, B \setminus \{b\})$ will yield a conflict. □

The relevance of this lemma is this. Given a conflict set $C = \{x_1, \dots, x_n\}$, we know that the value assignments $\nu(x_1), \dots, \nu(x_n)$ will lead to a conflict. Hence, any satisfying and complete value assignment $\widetilde{\nu}$ will need to set $\widetilde{\nu}(x_i)$ to $1 - \nu(x_i)$ for at least one $i \in \{1, \dots, n\}$. Hence, the single-clause CNF formula

$$\{\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n\},$$

where

$$\ell_i = \begin{cases} x_i, & \text{if } \nu(x_i) = 0, \\ \overline{x}_i, & \text{if } \nu(x_i) = 1, \end{cases}$$

for $i = 1, \dots, n$, is an implicate of $\mathcal{F}$.

Definition 2.11. Let $\mathcal{F}$ be a CNF formula and let ν be a partial value assignment over $\mathrm{var}(\mathcal{F})$ such that unit propagation yields a conflict in $\mathcal{F}$. Let $C = \{x_1, \dots, x_n\} \subset \mathrm{var}(\mathcal{F})$ be a conflict set. Then we call

$$\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n,$$

where

$$\ell_i = \begin{cases} x_i, & \text{if } \nu(x_i) = 0, \\ \overline{x}_i, & \text{if } \nu(x_i) = 1, \end{cases}$$

for $i = 1, \dots, n$, a **conflict-induced clause**. ♢

Lemma 2.12. *Let $\mathcal{F}$ be a propositional formula in* CNF. *Let $\{\mathfrak{c}\}$ be an implicate of $\mathcal{F}$. Then $\mathcal{F}$ is satisfiable if and only if $\mathcal{F} \cup \{\mathfrak{c}\}$ is satisfiable.*

Proof. If a value assignment ν satisfies $\mathcal{F}$, then it also satisfies all of the implicates of $\mathcal{F}$. So ν also satisfies the conjunction of $\mathcal{F}$ and any of its implicates. Since $\mathcal{F} \subset \mathcal{F} \cup \{\mathfrak{c}\}$, any satisfying assignment of $\mathcal{F} \cup \{\mathfrak{c}\}$ satisfies $\mathcal{F}$. □

This lemma implies that newly learnt implicates may be added to the CNF formula without affecting its satisfiability. Moreover, any satisfying assignment for the enlarged formula is also a satisfying assignment for the original formula. The advantage of adding such conflict-induced implicates to the CNF formula is that doing so prevents the same conflicting value assignments from being made. To see this, assume that the value assignment ν gave rise to a conflict and

that a conflict-induced clause with m literals was added to the CNF formula as a result. Unit propagation and value assignment by decision assign values to variables one at a time. So assume that in the process of constructing a value assignment, we obtain a partial value assignment $\widetilde{\nu}$ that evaluates exactly $m-1$ variables in this clause to the same values as ν did and leaves one variable unassigned. Then unit propagation guarantees that we will set the value assignment to this last variable x_m to $\widetilde{\nu}(x_m) = 1 - \nu(x_m)$. The value assignment $\widetilde{\nu}$ may of course still yield conflicts, but these would be the consequence of a different partial value assignment. For a related reason, it is particularly conflict-induced clauses that contain exactly one literal at the current decision level that are interesting in SAT solving.

Lemma 2.13. *Let $\mathcal{F}$ be a CNF formula and let ν be a value assignment over $\mathbb{V} \supset var(\mathcal{F})$ resulting in a conflict. Assume that $\delta(\perp) \geq 1$. Let $\mathfrak{c}$ be a clause induced by this conflict such that $\delta(\ell) = \delta(\perp)$ for exactly one $\ell \in \mathfrak{c}$. Then $\mathfrak{c} \notin \mathcal{F}$.*

Proof. All variables occurring in $\mathfrak{c}$ were set at a lower decision level than $\delta(\perp) \geq 1$ except for one. So if $\mathfrak{c} \in \mathcal{F}$, $\mathfrak{c}$ would have been a unit clause at a lower decision level, so that $\delta(\ell) < \delta(\perp)$. □

The assumption $\delta(\perp) \geq 1$ does not represent a genuine restriction: if $\delta(\perp) = 0$, the CNF formula is necessarily unsatisfiable.

2.2.2 Basic clause learning algorithm

A basic clause learning algorithm was proposed by Marques-Silva & Sakallah (1996, 1999). The following explanation of this algorithm is based on theirs with some modifications in notation. As we will see, this algorithm enables us to learn useful implicates in the sense of Lemma 2.13.

Definition 2.14. For every $x \in \mathbb{V} \cup \{\perp\}$, we let $A(x)$ denote the subset of variables whose value assignments directly implied the value assignment of x; if x is not an implied variable or the conflict node, $A(x) = \emptyset$. More precisely, we define

$$A(x) := \{y \in \mathbb{V} : y \in \alpha(x) \setminus \{x\} \text{ or } \overline{y} \in \alpha(x) \setminus \{\overline{x}\}\}.$$

We call this set the set of **antecedent variables** of x. ◇

Definition 2.15. We can partition $A(x)$ into those variables whose value assignments were effected at a lower decision level than $\delta(x)$ and whose value assignments were effected at $\delta(x)$, that is into

$$\Lambda(x) := \{y \in A(x) : \delta(y) < \delta(x)\}$$

and

$$\Sigma(x) := \{y \in A(x) : \delta(y) = \delta(x)\}.$$

We may now define for all $x \in \mathbb{V} \cup \{\perp\}$ a set of variables whose value assignments jointly imply the value assignment for x (if $x \in \mathbb{V}$) or that jointly lead to a conflict (if $x = \perp$). While several

such sets may exist, the one we are interested in is recursively defined as follows:

$$\texttt{causes_of} : V_{I'} \to 2^{V_{I'}},$$
$$x \mapsto \begin{cases} \{x\}, & \text{if } A(x) = \emptyset, \\ \Lambda(x) \cup \left[\bigcup_{y \in \Sigma(x)} \texttt{causes_of}(y)\right], & \text{else.} \end{cases}$$

◊

These definitions differ somewhat from those proposed by Marques-Silva & Sakallah (1999), who define $A(x)$ as a set of tuples $(y, \nu(y), \delta(y), \alpha(y)) \subset N$, which they called the antecedent assignment of x. Similarly, they define the sets $\Lambda(x), \Sigma(x)$, and $\texttt{causes_of}(x)$ as sets of tuples. However, I found that doing so clutters formulas and results in easily avoidable circumlocutions. On the basis of our $A(x)$, we may still retrieve the current value assignments, decision levels, and antecedent clauses using the ν, δ, and α functions.

If a conflict is identified, a new clause is constructed as follows:

$$\mathfrak{c}_{\text{new}} := \bigvee_{x \in \texttt{causes_of}(\bot)} x^{\nu(x)}, \tag{2.1}$$

where $x^{\nu(x)} := x$ if $\nu(x) = 0$ and $x^{\nu(x)} := \overline{x}$ if $\nu(x) = 1$. The case where $\nu(x) = \mathfrak{u}$ cannot occur for $x \in \texttt{causes_of}(\bot)$.

Below, we will see that $\texttt{causes_of}(\bot)$ can be interpreted as a kind of conflict set and that $\mathfrak{c}_{\text{new}}$ can similarly be interpreted as a kind of conflict-induced clause (cf. Definition 2.11). But first, a few examples are in order. In these, I slightly abuse notation and write $\overline{x}$ in lieu of x if $\nu(x) = 0$.

Example 2.16. Example 2.8 ended in a conflict. We obtain

$$\Lambda(\bot) = \emptyset$$

as well as

$$\Sigma(\bot) = \{e, \overline{g}\}.$$

The antecedent assignments of e and g are partitioned as

$$\Lambda(e) = \{\overline{f}\}, \Sigma(e) = \{c\}$$

and

$$\Lambda(g) = \emptyset, \Sigma(g) = \{c\}.$$

Since $A(c) = \emptyset$, we have

$$\texttt{causes_of}(\bot) = \{c, \overline{f}\}.$$

Hence,

$$\mathfrak{c}_{\text{new}} = \overline{c} \vee f.$$

This example also shows that $\texttt{causes_of}(\bot) = \{c, \overline{f}\}$ is but one subset of variables whose associated value assignments jointly lead to the conflict. The set $\{a, b, c\}$ would be another one, as would other sets encompassing $\{c, \overline{f}\}$. ♢

Example 2.17. Marques-Silva et al. (2021) discuss clause learning in a more informal, high-level way than do Marques-Silva & Sakallah (1999). The reader may find it instructive to compare how the former derive a clause to how $\texttt{clauses_of}(x)$ works.

In their example 4.2.5, Marques-Silva et al. (2021) consider the conflict that arises in the formula

$$\mathcal{F} = \underbrace{\left(\overline{a} \vee \overline{b} \vee c\right)}_{=\mathfrak{c}_1} \wedge \underbrace{(\overline{a} \vee d)}_{=\mathfrak{c}_2} \wedge \underbrace{\left(\overline{c} \vee \overline{d} \vee e\right)}_{=\mathfrak{c}_3} \wedge \underbrace{\left(\overline{h} \vee \overline{e} \vee f\right)}_{=\mathfrak{c}_4} \wedge \underbrace{(\overline{e} \vee g)}_{=\mathfrak{c}_5} \wedge \underbrace{\left(\overline{f} \vee \overline{g}\right)}_{=\mathfrak{c}_6}$$

after carrying out the following value assignments by decision: $\nu(h) = \nu(b) = \nu(y) = 1$. Figure 2.3 shows the implication graph for this problem. In their Table 4.1, they derive the learnt clause $(\overline{h} \vee \overline{b} \vee \overline{a})$ in a way that I found easy to understand but hard to put into pithy pseudocode without relying on $\texttt{causes_of}$. If we run $\texttt{causes_of}(\bot)$ on the same problem, we obtain the following derivation:

$$\begin{aligned}
\texttt{causes_of}(\bot) &= \underbrace{\Lambda(\bot)}_{=\emptyset} \cup \texttt{causes_of}(f) \cup \texttt{causes_of}(g) \\
&= \Lambda(f) \cup \texttt{causes_of}(e) \cup \underbrace{\Lambda(g)}_{=\emptyset} \cup \texttt{causes_of}(e) \\
&= \{h\} \cup \underbrace{\Lambda(e)}_{=\emptyset} \cup \texttt{causes_of}(c) \cup \texttt{causes_of}(d) \\
&= \{h\} \cup \Lambda(c) \cup \texttt{causes_of}(a) \cup \underbrace{\Lambda(d)}_{=\emptyset} \cup \texttt{causes_of}(a) \\
&= \{h\} \cup \{b\} \cup \{a\} \\
&= \{h, b, a\},
\end{aligned}$$

since $A(a) = \emptyset$. The learnt clause is hence again $\mathfrak{c}_{\text{new}} = \overline{h} \vee \overline{b} \vee \overline{a}$. ♢

In the light of our preceding discussion of conflict sets, it is tempting to try to make sense of the set $\texttt{causes_of}(\bot)$ as a conflict set associated with the implication graph that gave rise to the present conflict. As the next example shows, however, this is not always the case.

Example 2.18. Consider the CNF formula

$$\mathcal{F} = \underbrace{(\overline{a} \vee b)}_{=\mathfrak{c}_1} \wedge \underbrace{\left(\overline{a} \vee \overline{d} \vee e\right)}_{=\mathfrak{c}_2} \wedge \underbrace{\left(\overline{b} \vee \overline{c} \vee d\right)}_{=\mathfrak{c}_3} \wedge \underbrace{\left(\overline{b} \vee \overline{d} \vee \overline{f}\right)}_{=\mathfrak{c}_4} \wedge \underbrace{\left(\overline{d} \vee f\right)}_{=\mathfrak{c}_5}.$$

A possible implication graph associated with a conflict is shown in Figure 2.4. We have

$$\texttt{causes_of}(\bot) = \{b, c\},$$

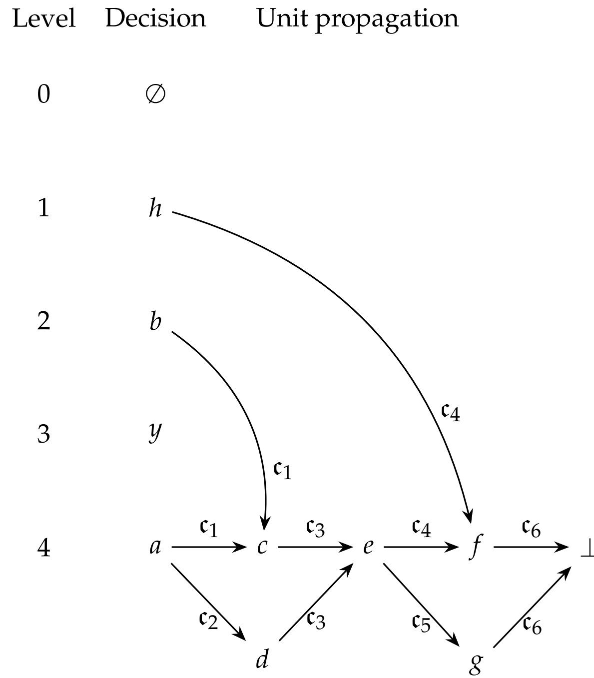

Figure 2.3: Implication graph for Example 4.2.5 in Marques-Silva et al. (2021). Redrawn from their Figure 4.2(a).

but this set does not constitute a conflict set. To see this, consider that any conflict partition (A, B) of which $\{b, c\}$ would be the conflict set would satisfy $d \in B$ on account of c having an outgoing edge to d only. Since no edges may leave B, it follows that $e \in B$. But then the conflict set would have to contain a, too. ♢

The problem that this example highlights is that there may be vertices representing variables whose values were assigned at the current decision level but that are not involved in the conflict. To get around this problem, we restrict our attention to the subgraph that comprises just those paths in the implication graph that terminate in $\perp$. This requires some further definitions.

Definition 2.19. Let $G = (V, E)$ be a directed acyclic graph. Then $x \in V$ is a **parent** of $y \in V$ if and only if there exists an edge (x, y) in the edge set E. In this case, we also call y a **child** of x. We call $x \in V$ an **ancestor** of $y \in V$ if and only if there exists a sequence $(z_0, z_1), (z_1, z_2), \dots, (z_{n-1}, z_n)$ in E with $z_0 = x$ and $z_n = y$. In this case, we also call y a **descendant** of x. ♢

If I is an implication graph, then x is a parent of y if and only if the variable $x \in A(y)$. Moreover, x is an ancestor of y if and only if there exist vertices $z_0, \dots, z_n \in V_I$ such that $x = z_0 \in A(z_1), z_1 \in A(z_2), \dots, z_{n-1} \in A(z_n)$ with $z_n = y$.

Definition 2.20. Let $I = (V_I, E_I)$ be the implication graph associated with a conflict. We define the **restricted implication graph** $I' = (V_{I'}, E_{I'})$ by setting

$$V_{I'} := \{x \in V_I : x \text{ is an ancestor of } \perp \text{ in } I\} \cup \{\perp\}$$

and

$$E_{I'} := \{(a, b) \in E_I : a, b \in V_{I'}\}.$$

♢

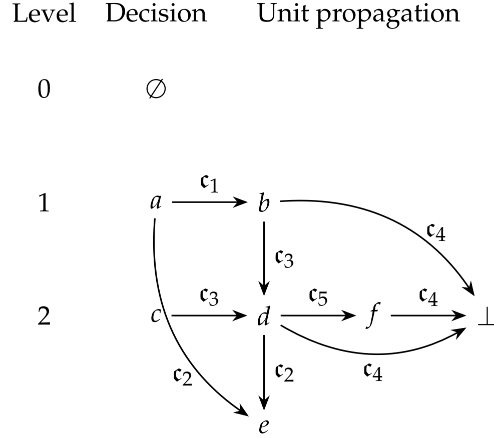

Figure 2.4: The set `causes_of`$(\bot)$ is not a conflict set.

If (A', B') is a conflict partition of the restricted implication graph I', then the value assignments to the elements in its associated conflict set C' already imply a conflict. The reason is that I' consists of precisely those vertices that represent value assignments that directly or indirectly gave rise to the conflict. We may hence apply the same reasoning as in the proof of Lemma 2.10 on page 10 to the restricted implication graph. It follows that any conflict-induced clause learnt on the basis of the restricted implication graph I' is an implicate of the CNF formula.

Lemma 2.21. *Given an implication graph I associated with a conflict, the set* `causes_of`$(\bot)$ *represents a conflict set of the restricted implication graph* $I' = (V_{I'}, E_{I'})$.

Proof. Let $x_0 \in V_{I'}$ be the unique vertex with $\alpha(x_0) = \mathfrak{d}, \delta(x_0) = \delta(\bot)$. That is, x_0 represents the latest decision variable. Define the set

$$B' := \{x \in V_{I'} : x \text{ is a descendant of } x_0 \text{ in } I'\}.$$

It is clear that

$$B' = \{x \in V_{I'} : A(x) \neq \emptyset, \delta(x) = \delta(\bot)\},$$

i.e., B' consists of all vertices at the latest decision level safe for the decision variable. So this set does not contain any decision variables, but it does contain $\bot$. Moreover, if $a \in B'$ and $(a, b) \in E_{I'}$, then b is a descendant of a in I'. Since a is a descendant of x_0, by transivity of the descendant relation, $b \in B'$. Hence, $(V_{I'} \setminus B', B')$ is a conflict partition of I'. The set

$$C' := \{y \in V_{I'} \setminus B' : \text{there exists an } x \in B' \text{ such that } y \text{ is a parent of } x\}$$

contains exactly those vertices in $V_{I'} \setminus B'$ with an edge into B', i.e., C' is the conflict set associated with the conflict partition $(V_{I'} \setminus B', B')$. We now show that

$$\texttt{causes_of}(\bot) = C'.$$

To this end, first assume that $y \in$ `causes_of`$(\bot)$. Further assume towards a contradiction that $y \in B'$. Then y is a descendant of x_0 in I'. This means that there exists some sequence of vertices

$z_0, \ldots, z_n$ in $V_{I'}$ with

$$x_0 = z_0 \in \Sigma(z_1), z_1 \in \Sigma(z_2), \ldots, z_{i-1} \in \Sigma(z_i), y = z_i \in \Sigma(z_{i+1}), \ldots, z_{n-1} \in \Sigma(z_n),$$

where $z_n = \perp$. By calling `causes_of`$(\perp)$, the definition of the `causes_of` function guarantees that we will eventually call `causes_of`(y). Since y is situated at the highest decision level, y is not part of the Λ set of any vertex. Moreover, since $\Sigma(y) \neq \emptyset$, `causes_of`(y) will result in a further recursive call to `causes_of()`. So there is no $x \in B'$ such that $y \in$ `causes_of`(x). Contradiction. Hence $y \in V_{I'} \setminus B'$.

We continue to consider $y \in$ `causes_of`$(\perp)$. Inspecting the recursion in the `causes_of` function, we see that there are only two ways in which y could have ended up in `causes_of`$(\perp)$. The first is that the recursive calls to `causes_of()` sketch out a sequence of vertices $z_0, \ldots, z_n$ in $V_{I'}$ with

$$y = z_0 \in \Lambda(z_1), z_1 \in \Sigma(z_2), \ldots, z_{n-1} \in \Sigma(z_n),$$

where $z_n = \perp$. The second possibility is that the recursive calls to `causes_of` sketch out a sequence of vertices $z_0, \ldots, z_n$ in $V_{I'}$ with

$$y = z_0 \in \Sigma(z_1), z_1 \in \Sigma(z_2), \ldots, z_{n-1} \in \Sigma(z_n),$$

where $z_n = \perp$ and with $A(y) = \emptyset$. In both cases, $\delta(z_1) = \delta(\perp), A(z_1) \neq \emptyset$. So $z_1 \in B'$ and $y \in C'$. Hence `causes_of`$(\perp) \subset C'$.

Now consider any $y \in C' \subset V_{I'} \setminus B'$. Then y has at least one child $x \in B'$. This x is either $\perp$ itself or an ancestor of $\perp$ with $\delta(x) = \delta(\perp)$. So there exists some sequence of vertices $z_0, \ldots, z_n$ in $V_{I'}$ with

$$y = z_0 \in A(z_1), z_1 \in \Sigma(z_2), \ldots, z_{n-1} \in \Sigma(z_n),$$

where $z_n = \perp$. The call `causes_of`$(\perp)$ eventually results in the call `causes_of`(z_1). Since $y \in A(z_1)$, the recursion continues. If $y \in \Lambda(z_1)$, we obtain

$$\texttt{causes_of}(\perp) \supset \texttt{causes_of}(z_1) \supset \Lambda(z_1) \ni y.$$

If $y \in \Sigma(z_1)$, we obtain

$$\texttt{causes_of}(\perp) \supset \texttt{causes_of}(z_1) \supset \texttt{causes_of}(y).$$

In the latter case, $\delta(y) = \delta(\perp)$. So $y \notin B'$ implies $A(y) = \emptyset$. So `causes_of`$(y) = \{y\}$. Hence $C' \subset$ `causes_of`$(\perp)$. □

Lemma 2.10 implies that the clause learnt after applying `causes_of`$(\perp)$ is a conflict-induced clause (but one based on the restricted rather than on the full implication graph) and that it is hence an implicate of the CNF formula and may be added to it. Moreover, the conflict set identified by `causes_of`$(\perp)$ has a useful property that arbitrary conflict sets do not necessarily have: it will contain exactly one variable whose value was set at the current decision level, namely the decision variable, alongside possibly some variables whose values were set at lower

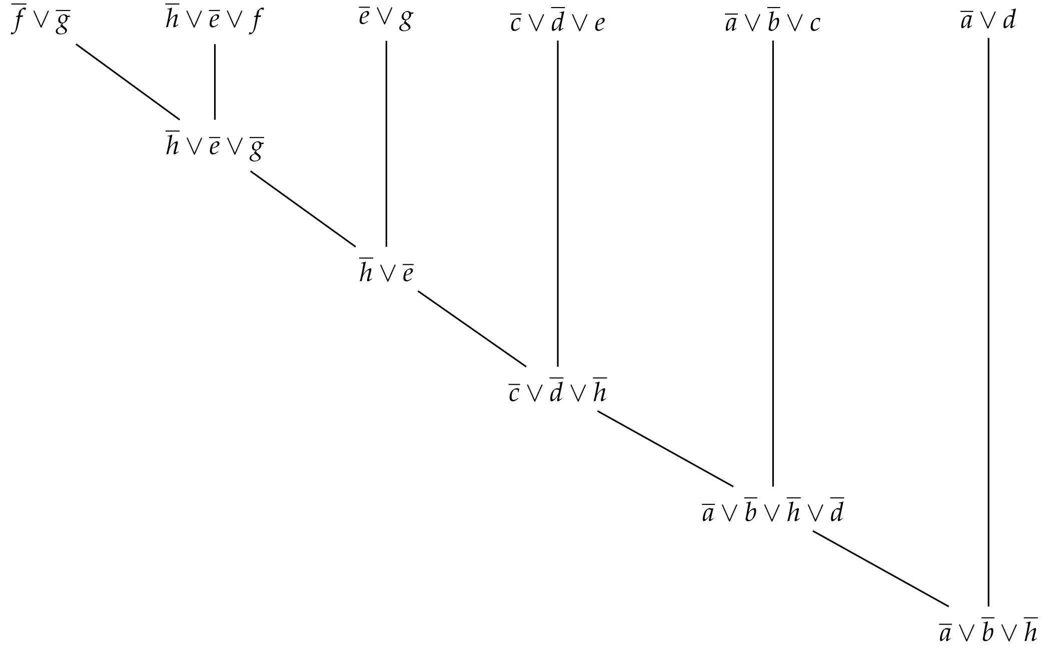

Figure 2.5: Learnt clauses can be obtained via resolution steps.

decision levels. Lemma 2.13 implies that the conflict-induced clause constructed on the basis of `causes_of`$(\bot)$ will be an implicate of $\mathcal{F}$ that is not yet contained in our CNF formula.

Remark 2.22. The clause learning algorithm discussed in this section implicitly works by resolution. For instance, the clause $\mathfrak{c}_{\text{new}} = \overline{h} \vee \overline{b} \vee \overline{a}$ learnt in Example 2.17 on page 14 may be obtained via the resolution steps shown in Figure 2.5. These steps trace out the recursive calls to `clauses_of()`. Since nothing in this thesis hinges on this remark, I do not prove that it holds generally. ♢

2.3 Non-chronological backtracking

Once a conflict has been analysed and a new implicate has been added to the CNF formula, some value assignments need to be reset so that the solver can explore an as yet uncharted part of the search space. This step is referred to as **backtracking**. Intuitively, it would seem to make sense to flip the value assignment of the last decision variable x which has not already been set to both 0 and 1 earlier and to set $\alpha(y) = \mathfrak{u}$ for all $y \neq x$ with $\delta(y) \geq \delta(x)$. This form of backtracking is known as **chronological backtracking** and would work in the sense that the resulting overall SAT solver would be sound and complete. However, following the lead of GRASP (e.g., Marques-Silva & Sakallah, 1999), current SAT solvers implement a form of **non-chronological backtracking**. This works as follows. Let d be the present decision level and let $\mathfrak{c}_{\text{new}}$ be the implicate just obtained by conflict analysis. Consider the set

$$D := \{\delta(\ell) : \ell \in \mathfrak{c}_{\text{new}}, \delta(\ell) < d\}$$

Global: A set of clauses representing a propositional formula $\mathcal{F}$ in CNF, to be updated
A set of 4-tuples ν representing a value assignment
The current decision level DLevel
Output: Highest decision level in the recorded clause that is still lower than the current decision level

```
1 ConflictAnalysis()
2   new ← CreateClause(causes_of(⊥))
3   MaxDLevel ← 0
4   if {ℓ ∈ new : δ(ℓ) < DLevel} ≠ ∅ then
5       MaxDLevel ← max{δ(ℓ) : ℓ ∈ new, δ(ℓ) < DLevel}
6   F ← F ∪ {new}
7   return MaxDLevel
```

Algorithm 2.2: The conflict analysis routine.

and define

$$\widetilde{d} := \begin{cases} 0, & \text{if } D = \emptyset, \\ \max(D), & \text{else.} \end{cases}$$

Now reset the value assignments, decision levels, and antecedents of all variables x with $\mathbb{N} \ni \delta(x) > \widetilde{d}$ and start another round of unit propagation with $\widetilde{d}$ as the new decision level. The newly added clause contains exactly one variable whose value has been reset to $\mathfrak{u}$, namely the decision variable that led to the conflict. All other literals in the clause evaluate to 0. Hence, the newly added clause is unit, and unit propagation can be applied to it, allowing the algorithm to explore new branches of the search tree.

Example 2.23 (Non-chronological backtracking)**.** Consider the propositional formula

$$\mathcal{F} = \underbrace{(\overline{a} \vee b)}_{=\mathfrak{c}_1} \wedge \underbrace{\left(\overline{b} \vee \overline{c} \vee d\right)}_{=\mathfrak{c}_2} \wedge \underbrace{(e \vee f)}_{=\mathfrak{c}_3} \wedge \underbrace{\left(\overline{d} \vee \overline{f} \vee g\right)}_{=\mathfrak{c}_4} \wedge \underbrace{\left(\overline{f} \vee h\right)}_{=\mathfrak{c}_5} \wedge \underbrace{\left(\overline{g} \vee \overline{h}\right)}_{=\mathfrak{c}_6}.$$

Picking the decision variables in lexicographical order and setting their values to 1 leads to a conflict, see Figure 2.6. Conflict analysis yields the new clause $(\overline{d} \vee \overline{f})$, resulting in the extended propositional formula

$$\widetilde{\mathcal{F}} = \underbrace{(\overline{a} \vee b)}_{=\mathfrak{c}_1} \wedge \underbrace{\left(\overline{b} \vee \overline{c} \vee d\right)}_{=\mathfrak{c}_2} \wedge \underbrace{(e \vee f)}_{=\mathfrak{c}_3} \wedge \underbrace{\left(\overline{d} \vee \overline{f} \vee g\right)}_{=\mathfrak{c}_4} \wedge \underbrace{\left(\overline{f} \vee h\right)}_{=\mathfrak{c}_5} \wedge \underbrace{\left(\overline{g} \vee \overline{h}\right)}_{=\mathfrak{c}_6} \wedge \underbrace{\left(\overline{d} \vee \overline{f}\right)}_{=\mathfrak{c}_7}.$$

The algorithm now backtracks to the new decision level $\delta(d) = 2$, where $\mathfrak{c}_7$ has become unit, allowing it to find a satisfying assignment; see Figure 2.7. ♢

Algorithm 2.2 presents pseudocode for the conflict analysis and clause learning routine, including the computation of the decision level the solver needs to backtrack to. The `CreateClause()` helper function implements Equation 2.1 on page 13.

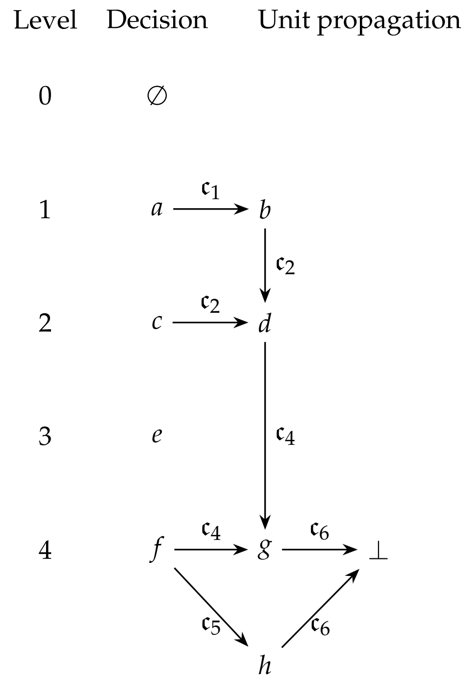

Figure 2.6: Implication graph of a value assignment leading to a conflict, see Example 2.23.

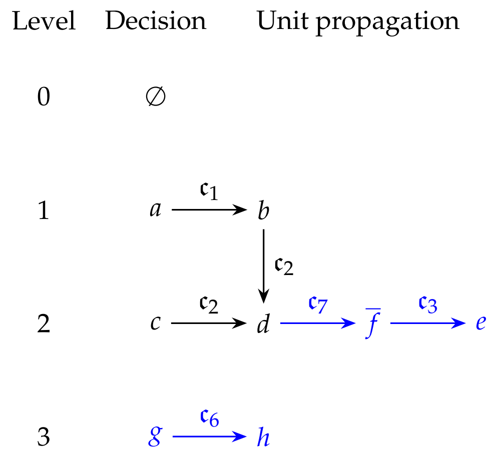

Figure 2.7: Continuation of Example 2.23. After backtracking to level 2, a satisfying assignment is found. Assignments effected after backtracking are in blue.

Global: A finite set of propositional variables $\mathbb{V}$
Input: A propositional formula $\mathcal{F}$ in CNF
Output: Satisfiability of $\mathcal{F}$

```
1  CDCL(𝓕)
2      N ← InitialiseAssignment()
3      DLevel ← 0
4      if not UnitPropagation() then
5          return false
6      while not AllVariablesAssigned() do
7          DLevel ← DLevel + 1
8          (var, val) ← PickBranchVariable()
9          Assign(var, val, DLevel, 𝔡)
10         while not UnitPropagation() do
11             if DLevel == 0 then
12                 return false
13             BLevel ← ConflictAnalysis()
14             Backtrack(BLevel)
15             DLevel ← BLevel
16     return true
```

Algorithm 2.3: The CDCL algorithm. From Marques-Silva et al. (2021, Algorithm 2, p. 141), with adaptations.

2.4 Putting it all together

All ingredients for a basic CDCL-based SAT solver are now in place; Algorithm 2.3 provides the pseudocode for the overarching algorithm. A few of the helper functions are not introduced in pseudocode as their logic is quite straightforward. The helper function `InitialiseAssignment()` identifies all variables in $\mathbb{V}$ and creates a set of tuples

$$\{(x, \mathfrak{u}, \mathfrak{u}, \mathfrak{n}) : x \in \mathbb{V}\}.$$

As its name suggests, the helper function `AllVariablesAssigned()` merely checks if there are any unassigned variables left in $\mathbb{V}$. The `Backtrack()` function resets the value assignments, decision levels, and antecedents of all variables whose decision level is above the input to `Backtrack()` back to $\mathfrak{u}, \mathfrak{u}$, and $\mathfrak{n}$, respectively.

Before discussing why bare-bones CDCL-based SAT solvers tend to work more efficiently than exhaustive search does, let's make sure that it works at all. While a soundness and completeness proof of an early CDCL-based SAT solver, GRASP, can be found in Marques-Silva & Sakallah (1999), my treatment of CDCL-based SAT solving differs somewhat from theirs, so that I present my own proofs.

Lemma 2.24. *Algorithm 2.3 terminates.*

Proof. Let $\mathcal{F}$ be a CNF formula that we feed into Algorithm 2.3. Define $n := \#\text{var}(\mathcal{F})$. The formula $\mathcal{F}$ has at most 3^n disjunctions with only variables in $\text{var}(\mathcal{F})$ as implicates: if we construct a new clause using the variables $\text{var}(\mathcal{F})$, we may for each variable x decide to leave it out of the

clause, to include x, or to include $\bar{x}$. When faced with a conflict with $\delta(\bot) = 0$, the algorithm immediately terminates (either on line 5 or on line 12). When faced with a conflict with $\delta(\bot) \geq 1$, the `ConflictAnalysis()` routine creates a conflict-induced clause that contains exactly one literal that was set at the current decision level and backtracks to the decision level at which this clause becomes unit. By Lemma 2.13, this clause has not yet been learnt before. Hence, the loop starting at line 10, during which a new implicate gets added to the CNF formula, will be executed at most 3^n times in total. Hence, the algorithm must terminate. □

If the algorithm terminates, it either outputs `true` or `false`, and we may both inspect the current extended CNF formula as well as the augmented value assignment N. Evidently, this termination guarantee is purely theoretical as it assumes infinite resources. In real-life applications, the CNF formula may become so large, and unit propagation may become so slow, that we may run out of computer memory or patience. The same caveat applies to the following theorem.

Theorem 2.25. *Algorithm 2.3 is sound and complete. That is, it outputs `true` if and only the CNF formula is satisfiable, in which case the value assignment constructed is a model of this CNF formula; moreover, it outputs `false` if and only if the CNF formula is unsatisfiable.*

Proof. Assume `CDCL`$(\mathcal{F})$ outputs `true`. Then all variables in $\mathcal{F}$ have been assigned (line 6 in Algorithm 2.3). If the algorithm never entered the while-loop starting on line 6, then all variables must have been assigned during unit propagation at decision level 0 (line 4), and the `UnitPropagation()` routine must have returned `true`. Inspecting Algorithm 2.1, it is clear that $\mathcal{F}^\nu \neq 0$. Since all variables have been assigned, it follows that $\mathcal{F}^\nu = 1$.

If, on the other hand, the algorithm did enter the while-loop starting on line 6, it can only have left it after `UnitPropagation()` on line 10 returned `true`. As before, this implies that $\mathcal{F}^\nu \neq 0$, and hence $\mathcal{F}^\nu = 1$. While it is possible that $\mathcal{F}$ has been extended with some of its implicates along the way, any value assignment that satisfies the extended formula also satisfies the original formula (Lemma 2.12). Hence, Algorithm 2.3 is sound, and the value assignment ν satisfies the original CNF formula.

By Lemma 2.24, if `CDCL`$(\mathcal{F})$ does not output `true`, it must output `false`. If the algorithm returns `false` at line 5, the formula $\mathcal{F}$ is clearly unsatisfiable: the `UnitPropagation()` routine will only have identified necessary value assignments, but these already cause a conflict. If the algorithm returns `false` on line 12, the algorithm must have backtracked to decision level 0 on line 14, during which process all value assignments by decision must have been erased. The current propositional formula on which `UnitPropagation()` operates on line 12 will only contain implicates of the original formula $\mathcal{F}$. So by the same token, `UnitPropagation()` will only have identified necessary value assignments, which already cause a conflict. Hence there does not exist any value assignment that satisfies $\mathcal{F}$ and its implicates. Hence there does not exist any value assignment that satisfies $\mathcal{F}$. □

2.5 CDCL vs brute-force search vs DPLL

Our bare-bones CDCL-based SAT-solving algorithm terminates and is both sound and complete. But these are not particularly high bars to clear: a brute-force search among all 2^n possible value

assignments to the n variables occurring in the CNF formula also clears them. CDCL-based SAT solvers tend to be more efficient than brute-force searches as unit propagation allows them to identify necessary value assignments implied by previous value assignments. This way, they do not need to potentially explore all branches of the search tree as a brute-force search would. Consider, for instance, Example 2.23 on page 19. The CNF formula contains eight variables, so a brute-force search would potentially have to consider $2^8 = 256$ different value assignments. Any algorithm relying on unit propagation, however, would immediately be able to exclude from its search space the $2^6 = 64$ value assignments that evaluate a to 1 but b to 0. Further branches of the search tree are similarly summarily dismissed at higher decision levels.

The utility of unit propagation was already recognised by the predecessors of CDCL-based algorithms, namely the DP algorithm (Davis & Putnam, 1960) and the computationally more feasible DPLL algorithm it gave rise to (Davis et al., 1962, named after Davis, Putnam, Logemann, and Loveland). The latter algorithm, like our CDCL-based solver, alternatingly applies unit propagation and value assignment by decision. It does not, however, deduce implicates of the input formula based on the conflicts it encounters. Instead, upon encountering a conflict, it backtracks to the last decision variable and sets its value to the other value; if both values $\{0, 1\}$ have already been tried, the algorithm backtracks to the penultimate decision variable, and so on. It outputs `true` if a satisfying assignment was found and `false` if both values have been tried to no avail for all decision variables. I do not provide pseudocode for the DPLL algorithm as this would require me to introduce new formalisms that will play no further role in this thesis (for two different pseudocode representations, see Darwiche & Pipatsrisawat, 2021, and Marques-Silva et al., 2021).

Example 2.26 (CDCL VS DPLL)**.** Consider the following propositional formula:

$$\mathcal{F} = \underbrace{(\overline{a} \vee b \vee c)}_{=\mathfrak{c}_1} \wedge \underbrace{\left(\overline{a} \vee \overline{b} \vee c \vee d\right)}_{=\mathfrak{c}_2} \wedge \underbrace{(\overline{a} \vee \overline{c} \vee d)}_{=\mathfrak{c}_3} \wedge \underbrace{(\overline{a} \vee d \vee e)}_{=\mathfrak{c}_4} \wedge \underbrace{\left(\overline{a} \vee \overline{d} \vee e\right)}_{=\mathfrak{c}_5} \wedge \underbrace{\left(\overline{a} \vee \overline{d} \vee \overline{e}\right)}_{=\mathfrak{c}_6}.$$

This formula is clearly satisfiable – we just need to set $\nu(\alpha) = 0$. But it is instructive to compare how the DPLL and CDCL algorithms go about discovering this fact if they pick the decision variables in lexicographical order and first assign to them the value 1.

The DPLL algorithm will first try out the value assignments $\nu(a) = \nu(b) = \nu(c) = 1$, at which point unit propagation can be applied. This yields $\nu(d) = 1$ due to clause $\mathfrak{c}_3$, which in turn results in the conflict $\nu(e) = 1$ (due to clause $\mathfrak{c}_5$) versus $\nu(e) = 0$ (due to clause $\mathfrak{c}_6$). At this point, the algorithm will reevaluate its latest guess and instead tries out the assignment $\nu(c) = 0$, keeping the assignments $\nu(a) = \nu(b) = 1$. Unit propagation now results in $\nu(d) = 1$ (clause $\mathfrak{c}_2$), which again yields the same conflict in the variable e. Having tried out both assignments for the c variable, the DPLL algorithm backtracks to the b variable. Setting $\nu(b) = 0$ while only keeping $\nu(a) = 1$, unit propagation results in $\nu(c) = 1$ (clause $\mathfrak{c}_1$), and then once more $\nu(d) = 1$ (clause $\mathfrak{c}_3$) and again the same conflict in the e variable. The algorithm now backtracks to the a variable, sets its value assignment to 0, and obtains a model. Figure 2.8 displays the branches of the search tree that the DPLL algorithm explores. Note that the DPLL algorithm backtracks chronologically, i.e., to the latest decision variable that it has not yet fully considered.

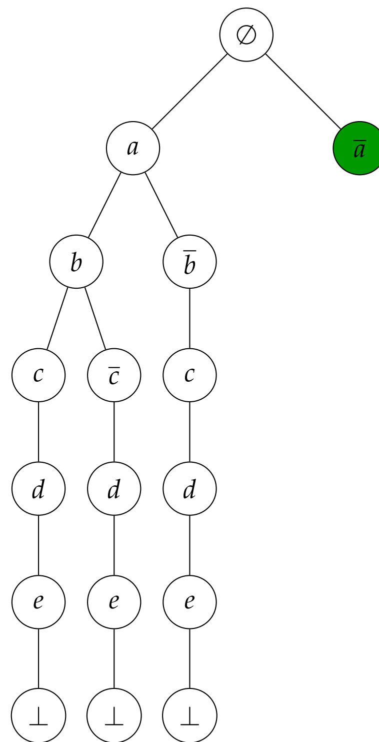

Figure 2.8: In Example 2.26, the DPLL algorithm first explores three branches of the search tree, always resulting in the same conflict. Only then does it reconsider to value assignment to the variable a, resulting in a satisfying partial assignment.

The CDCL algorithm, by contrast, will deduce from conflict that the value assignments to a and c cannot both be 1, see Figure 2.9. It adds the clause $\mathfrak{c}_7 = (\overline{a} \vee \overline{c})$ to the original formula and then backtracks more aggressively, namely to $\delta(a) = 1$. There, it immediately concludes that there are no models that evaluate a to 1, see Figure 2.10. From this conflict, it would learn the clause $\mathfrak{c}_8 = \overline{a}$. $\diamondsuit$

The combination of clause learning and non-chronological backtracking means that the CDCL algorithm tends to prune the search tree more aggressively than does the DPDL algorithm. Indeed, Knuth (2016) compared the performance of a couple of SAT-solving algorithms, including basic DPLL and CDCL implementations, on one hundred test cases and concluded that

> "it [the basic CDCL implementation, JV] is the clear method of choice in the vast majority of our test cases, and we can expect it to be the major workhorse for most of the satisfiability problems we encounter in daily work." [p. 122]

2.6 On the limits of CDCL-based SAT solving

Knuth's expectation testifies to the great successes that CDCL-based algorithms enjoy in real-life, industrial applications. Their Achilles' heel, however, are randomly-generated CNF formulas.[1]

[1]This is claimed in several publications (e.g., Ansótegui et al., 2019), and is reflected in the fact that CDCL-based algorithms rarely rarely entered the 'random' tracks of previous SAT competitions (see `https://satcompetition.github.io/`). I was unable to find useful data that backs up this claim, however.

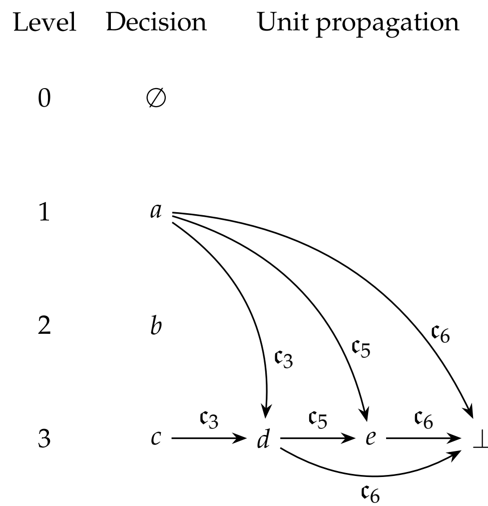

Figure 2.9: The CDCL algorithm encounters a conflict in Example 2.26 and deduces from it the implicate $(\bar{a} \vee \bar{c})$.

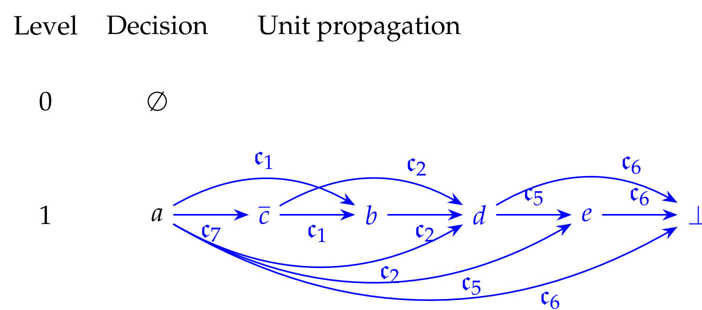

Figure 2.10: Continuation of Example 2.26: The CDCL algorithm has learnt that $\nu(a) = 1$ must imply $\nu(c) = 0$ for any model ν and backtracks to the decision level of a. Value assignments made after backtracking to decision level $\delta(a) = 1$ are coloured blue.

This suggests that CDCL-based SAT solvers implicitly capitalise on some properties typical of industrial SAT problems, which random CNF formulas do not exhibit. Perhaps surprisingly, it is still an open question what these properties are and how SAT solvers exploit them (Ganesh & Vardi, 2021).

One popular suggestion is that the problem's so-called community structure is one such property. What is meant by this is this. For each CNF formula, a variable-incidence graph may be constructed. This is an undirected weighted graph whose vertices are the formula's variables. The edges of this graph are drawn in a two-stage process. First, between each pair of variables that occur in the same clause, an edge whose weight is inversely proportional to the width of that clause is drawn. Then, all edges for each variable pair are coalesced and their weights summed. The problem is now said to have a good community structure to the extent that it is possible to partition the vertex set into clusters ('communities') in such a way that there are edges predominantly within clusters and few edges between clusters, more so than for a randomly constructed graph; see Definition 3 in Ansótegui et al. (2019) for a formal definition. Empirically, industrial – but by definition not random – SAT instances tend to be characterised by an excellent community structure (e.g., Ansótegui et al., 2019). Moreover, the time required to solve industrial SAT problems has been shown to be inversely correlated to the quality of their community structure (e.g., Newsham et al., 2014). That said, a solid theoretical explanation for how CDCL-based SAT solvers exploit community structure has so far proved elusive (Ganesh & Vardi, 2021), and, indeed, having good community structure does not necessarily make a SAT instance easy to solve (Mull et al., 2016). Ganesh & Vardi (2021) discuss a few additional proposals (also see Li et al., 2021; Zulkoski et al., 2018), but conclude that the questions as to what makes industrial SAT instances much more amenable to CDCL-based solving than random problems and why are as yet unanswered.

Chapter 3

Common tweaks

The previous chapter presented a no-frills conflict-driven clause learning SAT solver. Competitive CDCL-based SAT solvers implement a host of further refinements to this basic algorithm, allowing them to solve a range of real-life problem more efficiently (also see Fichte et al., 2020). As a quick perusal of the descriptions of the solvers submitted to the yearly SAT competition (see `https://satcompetition.github.io/`) will reveal, modern SAT solvers differ considerably in the tweaks they implement, and it is not possible to treat all of these – or even the most popular ones, for that matter – in detail. Some of these tweaks concern the computational nitty-gritty, with a prime example being the introduction of the 'watched literals' data structure in Chaff (Moskewicz et al., 2001) that reduces the number of memory accesses required in value (re)assignments. For the present chapter, however, I have opted to focus on a handful of more high-level conceptual tweaks to the basic CDCL algorithm. Specifically, I discuss in some detail two commonly implemented techniques for deducing more useful implicates from conflicts. I will also present in broad strokes three further ideas that are implemented in some form or other in modern SAT solvers.

3.1 Learning more useful clauses

Other things equal, shorter implicates are more useful to SAT solvers than larger ones as they tend to become unit earlier, allowing unit propagation to kick in more quickly. At the computational level, moreover, shorter clauses simply take up less memory. Empirical investigations (e.g., Audemard & Simon, 2009), however, resulted in the more powerful indicator of clause quality that is encapsulated in the following definition.

Definition 3.1. Let N be an augmented value assignment over $\mathbb{V} \supset \text{var}(\mathcal{F})$ and let $\mathfrak{c}$ be a clause with variables in $\mathbb{V}$ such that $\delta(x) \neq \mathfrak{u}$ for each variable x in $\mathfrak{c}$. Then the **literal block distance** (LBD) of $\mathfrak{c}$ is defined as

$$\text{LBD}(\mathfrak{c}) := \#\{\delta(x) : x \text{ is a variable in } \mathfrak{c}\}. \qquad \diamondsuit$$

I did not find any formal justification for the LBD metric, but going by Audemard & Simon (2009), the intuition behind this definition is as follows. If a SAT solver learns a new clause with a low LBD value, then the variables in this clause are more intimately linked in the region of the search

space that is currently being explored compared to clauses with high LBD values. As a result, the clause is more likely to become unit earlier compared to a clause with a high LBD value.

Furthermore, learnt clauses tend to be more useful when they allow the solver to backtrack to even lower decision levels. The reason for this is that the implied value assignments at lower decision levels are conditional on fewer value assignments by decision. Hence, implied value assignments at lower decision levels prune the search tree more aggressively than do implied value assignments at higher decision levels.

So CDCL-based SAT solvers typically implement techniques resulting in clauses that tend to be smaller, have lower LBD values, and allow for more aggressive backtracking than those obtained by the algorithm presented in the previous chapter.

3.1.1 Unique implication points

Consider again the implication graph in Figure 2.3 on page 15. In the previous chapter, we saw that from this implication graph, we would deduce the implicate $\overline{a} \vee \overline{b} \vee \overline{h}$. However, the set $\{e, h\}$ also represents a conflict set, which results in the shorter implicate $\overline{e} \vee \overline{h}$. Moreover, the latter implicate allows us to backtrack to decision level 1 rather than only to decision level 2. The lesson to be learnt from this example is that we do not have to backtraverse the implication graph all the way back to the latest decision variable: if the value assignment to a different variable at the current decision level, along with the value assignments at lower decision levels, already implies a conflict, we can stop the traversal at this variable. The next few definitions and lemmas spell out this idea, which was already present in the GRASP solver (Marques-Silva & Sakallah, 1996, 1999), more formally.

Definition 3.2. Let $G = (V_G, E_G)$ be a directed graph with $x, y \in V_G$. If there exists at least one path starting at the vertex x and terminating at the vertex y and there exists a vertex $z \neq y$ such that z lies on every such path, we say that z **dominates** x with respect to y. We also call z a **dominator** of x with respect to y. ♢

Note that if a path exists between x and y, then x itself dominates x with respect to y.

Definition 3.3. Let $I = (V_I, E_I)$ be the implication graph associated with a conflict. Let $\delta(\bot) \geq 1$ and let $x \in V_I$ be the sole vertex that satisfies $\delta(x) = \delta(\bot), \alpha(\bot) = \mathfrak{d}$ (that is, x represents the decision variable that led to the conflict). Then we call any dominator of x with respect to $\bot$ a **unique implication point** (UIP). Given an implication graph I associated with a conflict, we denote the set of its unique implication points as U_I. ♢

In this definition, we may disregard the case $\delta(\bot) = 0$ as this would immediately cause the solver to conclude that the CNF formula is unsatisfiable, obviating the need to learn further implicates. Further note that if u is a unique implication point, then we cannot backtraverse the implication graph starting at $\bot$ to a node x with $\delta(x) = \delta(\bot)$ that is not a descendant of u without encountering u.

Our immediate goal is to generalise the `causes_of` function defined on page 13 so that it stops backtraversing the implication graph at any implication point of our choosing rather than just at the current decision variable. To this end, let's define an overloaded version of this function as

follows:

$$\texttt{causes_of} : V_{I'} \times 2^{V_{I'}} \to 2^{V_{I'}},$$
$$(x, S) \mapsto \begin{cases} \{x\}, & \text{if } x \in S, \\ \Lambda(x) \cup \left[\bigcup_{y \in \Sigma(x)} \texttt{causes_of}(y, S)\right], & \text{else.} \end{cases}$$

Given a directed graph, finding the set of unique implication points is computationally easy (see, for instance, Lengauer & Tarjan, 1979). We can now use this set as the second input to the overloaded version of the causes_of function. As shown in the next three lemmas, this enables us to determine conflict sets that are no larger than the ones identified using the algorithm in the previous chapter; in practice, these conflict sets will in fact often be smaller.

The first lemma we need generalises Lemma 2.21 on page 16.

Lemma 3.4. *Let I be an implication graph associated with a conflict. Let U_I be the set of its unique implication points. Then the set $\texttt{causes_of}(\bot, \{u\}), u \in U_I$, represents a conflict set of the restricted implication graph I'. Furthermore, this set contains exactly one vertex associated with a variable with the same decision level as $\bot$.*

Proof. We adapt the proof of Lemma 2.21. Let $u \in U_I$ be an arbitrary unique implication point. Define the set

$$B' := \{x \in V_{I'} : x \text{ is a descendant of } u \text{ in } I'\}.$$

The partition $(V_{I'} \setminus B', B')$ is a conflict partition of I' for the same reason as in the proof of Lemma 2.21, substituting u for x_0. Now we define

$$C' := \{y \in V_{I'} \setminus B' : \text{there exists an } x \in B' \text{ such that } y \text{ is a parent of } x\}$$

and note that this is the conflict set associated with the conflict partition $(V_{I'} \setminus B', B')$. We want to show that

$$\texttt{causes_of}(\bot, \{u\}) = C'.$$

To this end, first assume that $y \in \texttt{causes_of}(\bot, \{u\})$ and further assume $y \in B'$. Again substituting u for x_0 in the proof of Lemma 2.21, we obtain a contradiction. Hence, $y \in V_{I'} \setminus B'$.

The recursive definition of the overloaded causes_of function and the fact that $u \in U_I$ guarantee that there are only two ways in which y could have ended up in $\texttt{causes_of}(\bot)$. The first is that the recursive calls to causes_of sketch out a sequence of vertices $z_0, \ldots, z_n$ in $V_{I'}$ with

$$y = z_0 \in \Lambda(z_1), z_1 \in \Sigma(z_2), \ldots, z_{n-1} \in \Sigma(z_n),$$

where $z_n = \bot$. The second possibility is that the recursive calls to causes_of sketch out a sequence of vertices $z_0, \ldots, z_n$ in $V_{I'}$ with

$$y = u = z_0 \in \Sigma(z_1), z_1 \in \Sigma(z_2), \ldots, z_{n-1} \in \Sigma(z_n),$$

where $z_n = \perp$. In both cases, the vertices $z_1, \ldots, z_n$ all are descendants of u. To see this, consider that $z_n = \perp$ is a descendant of u. In order to reach any non-descendant of u at the same decision level starting at $\perp$, we would have to backtraverse the implication graph at the same decision level (i.e., following the Σ sets). Since u is a UIP, we would eventually end up at u. But calling $\texttt{causes_of}(u, \{u\})$ terminates the recursion. So $\{z_1, \ldots, z_n\} \subset B'$. In particular, $z_1 \in B'$ and $y \in C'$. So $\texttt{causes_of}(\perp, \{u\}) \subset C'$.

Now consider any $y \in C' \subset V_{I'} \setminus B'$. Then y has at least one child $x \in B'$. This x is either $\perp$ itself or an ancestor of $\perp$ with $\delta(x) = \delta(\perp)$ that is also a descendant of u. So there exists some sequence of vertices $z_0, \ldots, z_n$ in $V_{I'}$ with

$$y = z_0 \in A(z_1), z_1 \in \Sigma(z_2), \ldots, z_{n-1} \in \Sigma(z_n),$$

where $z_n = \perp$ and $z_i \neq u$ for $i = 1, \ldots, n$. The call $\texttt{causes_of}(\perp, \{u\})$ eventually results in the call $\texttt{causes_of}(z_1, \{u\})$. Since $z_1 \neq u$ and $\Sigma(z_1) \neq \emptyset$, the recursion continues. If $y \in \Lambda(z_1)$, we obtain

$$\texttt{causes_of}(\perp, \{u\}) \supset \texttt{causes_of}(z_1, \{u\}) \supset \Lambda(z_1) \ni y.$$

If $y \in \Sigma(z_1)$, we obtain

$$\texttt{causes_of}(\perp, \{u\}) \supset \texttt{causes_of}(z_1, \{u\}) \supset \texttt{causes_of}(y, \{u\}).$$

In the latter case, $\delta(y) = \delta(\perp)$. If $y \neq u$, then y is itself a descendant of u: we can backtraverse the implication graph at the same decision level starting at $\perp$ without encountering u to reach y, hence y cannot be a non-descendant. This contradict $y \neq B'$. Hence, $\texttt{causes_of}(y, \{u\}) = \{y\}$. So $C' \subset \texttt{causes_of}(\perp, \{u\})$.

Lastly, note that $u \in C' = \texttt{causes_of}(\perp, \{u\})$. Since $\delta(u) = \delta(\perp)$, there is at least one vertex in $\texttt{causes_of}(\perp, \{u\})$ associated with a variable with the same decision level as $\perp$. Now let $y \in C'$ be a vertex with $\delta(y) = \delta(\perp)$. Since $y \in C'$, $y \notin B'$. Since u is a UIP, y is then either u or an ancestor of u. If y were an ancestor of u, the recursive calls to causes_of must have sketched out a path that included the call $\texttt{causes_of}(u, \{u\})$, which would have terminated the recursion. So y is not an ancestor of u. Hence $y = u$. Hence, $\texttt{causes_of}(\perp, \{u\})$ contains exactly one vertex associated with a variable with the same decision level as $\perp$. □

As the next lemma shows, given the set of unique implication points, we can easily construct the conflict set related to the first implication point as seen from the $\perp$ vertex. This strategy is referred to in the literature as the 1-UIP or First UIP scheme (e.g., Zhang et al., 2001), whereas the scheme discussed in the previous chapter is referred to as the Last UIP or the RELSAT scheme. (RELSAT was another clause-learning SAT solver; Bayardo & Schrag, 1997.)

Lemma 3.5 (1-UIP)**.** *Let I be an implication graph associated with a conflict. Let U_I be the set of its unique implication points. Then the set $\texttt{causes_of}(\perp, U_I)$ represents a conflict set of the restricted implication graph I'. Furthermore, this set contains exactly one vertex associated with a variable with the same decision level as $\perp$.*

Proof. Backtraversing the implication graph from the $\bot$ vertex to the vertex representing the decision variable x with $\delta(x) = \delta(\bot)$, the first $u \in U_I$ encountered is well-defined by virtue of its being a dominator of x with respect to $\bot$. The recursive calls of the overloaded $\texttt{causes_of}(y, U_I)$ function stop when $y \in U_I$, hence they already stop at $y = u$. Hence,

$$\texttt{causes_of}(\bot, U_I) = \texttt{causes_of}(\bot, \{u\}).$$

By the previous lemma, $\texttt{causes_of}(\bot, U_I)$ is a conflict set and contains exactly one variable at the same decision level as $\bot$, viz., u. □

The 1-UIP strategy is optimal in the sense of the following two lemmas (also see Hamadi et al., 2016).

Lemma 3.6 (Optimality w.r.t. clause width)**.** *Let I be an implication graph associated with a conflict. Let U_I be the set of its unique implication points. Then for all $u \in U_I$, we have*

$$\#\mathit{causes_of}(\bot, U_I) \leq \#\mathit{causes_of}(\bot, \{u\}). \qquad (*)$$

In particular,

$$\#\mathit{causes_of}(\bot, U_I) \leq \#\mathit{causes_of}(\bot).$$

Proof. As shown in the proof of the previous lemma, the first UIP as seen from the $\bot$ vertex is well-defined; call it u_0. We have already seen that

$$\texttt{causes_of}(\bot, U_I) = \texttt{causes_of}(\bot, \{u_0\}).$$

Pick any $u \in U_I$. If $u = u_0$, $(*)$ is trivial. If $u \neq u_0$, then u_0 is a descendant of u in I'. Hence, calling $\texttt{causes_of}(\bot, \{u\})$ will eventually result in calling $\texttt{causes_of}(u_0, \{u\})$. Hence, any $x \in \texttt{causes_of}(\bot, U_I)$ with $\delta(x) < \delta(\bot)$ also belongs to $\texttt{causes_of}(\bot, \{u\})$. Since both $\texttt{causes_of}(u_0, \{u\})$ and $\texttt{causes_of}(\bot, U_I)$ contain exactly one variable at decision level $\delta(\bot)$, $(*)$ follows.

Now let $x \in V_{I'}$ be the decision variable with $\delta(x) = \delta(\bot)$. Then $x \in U_I$ and so

$$\#\texttt{causes_of}(\bot, U_I) \leq \#\texttt{causes_of}(\bot, x) = \#\texttt{causes_of}(\bot).$$ □

Lemma 3.7 (Optimality w.r.t. LBD and backtracking level)**.** *Let I be an implication graph associated with a conflict. Let U_I be the set of its unique implication points and $u \in U_I$. Define*

$$D := \{\delta(x) : x \in \mathit{causes_of}(\bot, \{u\}), \delta(\ell) < \delta(\bot)\},$$
$$D_0 := \{\delta(x) : x \in \mathit{causes_of}(\bot, U_I), \delta(\ell) < \delta(\bot)\},$$

as well as

$$\widetilde{d} := \begin{cases} 0, & \text{if } D = \emptyset, \\ \max(D), & \text{else,} \end{cases}$$

$$\widetilde{d_0} := \begin{cases} 0, & \text{if } D_0 = \emptyset, \\ \max(D_0), & \text{else.} \end{cases}$$

Then $\#D_0 \leq \#D$ *and* $\widetilde{d_0} \leq \widetilde{d}$.

Proof. As before, denote the first UIP as seen from the $\perp$ vertex by u_0. From the proof of the previous lemma, we obtain

$$\texttt{causes_of}(\perp, U_I) \setminus \{u_0\} = \texttt{causes_of}(\perp, \{u_0\}) \setminus \{u_0\} \subset \texttt{causes_of}(\perp, \{u\}) \setminus \{u\}.$$

Hence, $D_0 \subset D$. Hence, $\#D_0 \leq \#D$ and $\widetilde{d_0} \leq \widetilde{d}$. □

In other words, the 1-UIP strategy enables us to learn shorter clauses with lower LBD values and to backtrack more aggressively, i.e., to lower decision levels, compared to the Last UIP scheme. The 1-UIP strategy can be accommodated by making a couple of minor changes to Algorithm 2.2 on page 19. Specifically, create the set U_I by identifying the dominators of the decision variable at the latest decision level with respect to $\perp$. Then run `CreateClause`($\texttt{causes_of}(\perp, U_I)$) instead of `CreateClause`($\texttt{causes_of}(\perp)$).

To my knowledge, current CDCL-based SAT solvers implement the 1-UIP learning scheme. However, GRASP learnt new implicates at *each* UIP. A possible pseudocode representation is provided in Algorithm 3.1. The idea is this. We first learn a new implicate using the 1-UIP scheme and jot down the first UIP u encountered. If u was not the only UIP, we use the `causes_of` function to figure out the root causes for our value assignment to u and encode this knowledge in a new clause. Since the value assignment to u must have been obtained through unit propagation, this new clause is also an implicate of the original CNF formula. We then learn the root causes for our value assignment to the second UIP, and so on, until we arrive at the decision variable at the current decision level.

Example 3.8. Based on the implication graph in Figure 2.3 on page 15, Algorithm 3.1 would learn the implicates $\mathfrak{c}_7 = (\overline{e} \vee \overline{h})$ (at the first implication point) and $\mathfrak{c}_8 = (\overline{a} \vee \overline{b} \vee e)$. The latter clause encodes the insight that can be gleaned from the implication graph that $\nu(a) = \nu(b) = 1$ implies $\nu(e) = 1$ for any model ν. ♢

It seems that the motivation for learning implicates at multiple UIPs in GRASP was to simply learn as much as possible from any conflict and thereby close off more unproductive branches of the search tree than the 1-UIP scheme would. Empirical results show that it succeeds in doing so, but that the 1-UIP scheme nonetheless has shorter runtimes. While learning the additional clauses incurs some minor overhead, the main problem with multiple clause learning seems to be that the increase in clauses slows down the unit propagation routine (Zhang et al., 2001).[1]

[1]Marques-Silva & Malik (2018) and Marques-Silva et al. (2021) cite a poster presentation by Sabharwal et al. (2012) as showing "promising results" in favour of multiple clause learning. But Sabharwal et al. (2012) did not compare

```
   Global: A set of clauses representing a propositional formula F in CNF, to be updated
           A set of 4-tuples N representing a value assignment
           Current decision level DLevel
   Output: Highest decision level in the recorded clause that is still lower than the current
           decision level
 1 ConflictAnalysis()
 2     U_I ← FindUIPs(N)
 3     c ← causes_of(⊥, U_I)
 4     new ← CreateClause(c)
 5     MaxDLevel ← 0
 6     if {ℓ ∈ new : δ(ℓ) < DLevel} ≠ ∅ then
 7         MaxDLevel ← max{δ(ℓ) : ℓ ∈ new, δ(ℓ) < DLevel}
 8     F ← F ∪ {new}
 9     u ← only variable in new with decision level DLevel
10     U_I ← U_I \ {u}
11     while U_I ≠ ∅ do
12         c ← causes_of(u, U_I)
13         new ← CreateClause(c) ∪ {u^(1−ν(u))}
14         F ← F ∪ {new}
15         u ← only variable in c with decision level DLevel
16         U_I ← U_I \ {u}
17     return MaxDLevel
```

Algorithm 3.1: Learning implicates at each UIP. For the definition of $u^{1-\nu(u)}$, see Equation 2.1 on page 13.

3.1.2 Learnt clause minimisation

Even when conflict-induced clauses are learnt using the 1-UIP scheme, it is often possible to reduce (or 'strengthen') the learnt clause to a proper subset of it that already prevents the same conflict from being reached. A first strategy used to identify superfluous literals in a learnt clause is known as **local minimisation** and is based on the principle of **self-subsuming resolution** (see, for instance, Marques-Silva & Malik, 2018): Let $\mathfrak{c}_1 \subset \mathfrak{c}_2$ be sets of literals over $\text{var}(\mathcal{F})$. If the newly learnt clause has the form $\{\ell\} \cup \mathfrak{c}_2$ and there exists a clause in $\mathcal{F}$ of the form $\{\neg\ell\} \cup \mathfrak{c}_1$, then resolving both clauses on ℓ yields the clause $\mathfrak{c}_1 \cup \mathfrak{c}_2 = \mathfrak{c}_2$. Hence, we may drop the literal ℓ from the newly learnt clause. Algorithm 3.2 outlines the resulting procedure.

Example 3.9 (Local minimisation)**.** Marques-Silva & Malik (2018) present the following example of local clause minimisation. Consider the CNF formula

$$\mathcal{F} = \underbrace{(\overline{x} \vee b)}_{=\mathfrak{c}_1} \wedge \underbrace{(\overline{z} \vee \overline{b} \vee c)}_{=\mathfrak{c}_2} \wedge \underbrace{(\overline{x} \vee \overline{y} \vee \overline{z} \vee a)}_{=\mathfrak{c}_3} \wedge \underbrace{(\overline{a} \vee \overline{c})}_{=\mathfrak{c}_4}.$$

and the implication graph in Figure 3.1 it could give rise to. The clause learnt after conflict analysis without minimisation is $\overline{x} \vee \overline{y} \vee \overline{z} \vee \overline{b}$. Resolving this clause on $\overline{b}$ with $\alpha(\overline{b}) = \alpha(b) = \mathfrak{c}_2 = \overline{x} \vee b$ yields $\overline{x} \vee \overline{y} \vee \overline{z}$, allowing us to strengthen the learnt clause by one literal. ♢

the GRASP scheme to the 1-UIP scheme. Rather, from what I was able to make out from their abstract, it seems that they compared the GRASP scheme to a multiple clause learning scheme in which new implicates were generated on

```
  Global: A set of clauses representing a propositional formula 𝓕 in CNF
          A set of 4-tuples N representing a value assignment
  Input: A learnt clause new
1 LocalMinimisation(new)
2     for ℓ ∈ new with α(ℓ) ≠ ∂ do
3         if α(ℓ) \ {¬ℓ} ⊂ new \ {ℓ} then
4             new ← new \ {ℓ}
5     return new
```

Algorithm 3.2: Strengthening learnt clauses using local minimisation. Note that the way we defined antecedents of literals means that $\alpha(\ell) = \alpha(\neg\ell)$.

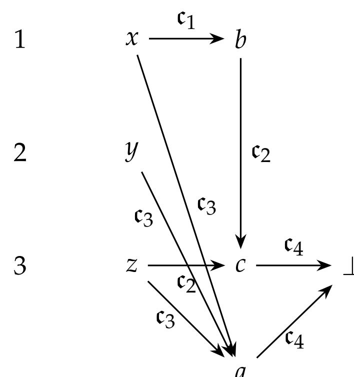

Figure 3.1: Implication graph of a value assignment leading to a conflict, see Example 3.9.

```
  Global: A set of clauses representing a propositional formula F in CNF
          A set of 4-tuples N representing a value assignment
  Input: A learnt clause new
1 RecursiveMinimisation(new)
2   for ℓ ∈ new with α(ℓ) ≠ 𝔡 do
3     if trace(ℓ, new \ {ℓ}) ⊂ new then
4       new ← new \ {ℓ}
5   return new
```

Algorithm 3.3: Strengthening learnt clauses using recursive minimisation.

Modern SAT solvers implement a more powerful strategy for eliminating superfluous literals from learnt clauses called **recursive minimisation**. Sörensson & Biere (2009) describe recursive minimisation as follows:

> "Generate the 1-UIP clause. Mark its literals. Implied variables in 1-UIP clause are candidates for removal. Search implication graph. Start from antecedent literals of candidate. Stop at marked literals or decisions. If search always ends at marked literals then the candidate can be removed."

Since it was not immediately clear to me what they meant by this nor why it works, I have attempted to spell out the logic behind this strategy more explicitly based on their examples as well as those in Marques-Silva & Malik (2018) and Marques-Silva et al. (2021). Let $\mathbb{L}$ be the set of literals over $\mathbb{V}$ supplemented with $\bot$. We define the recursive function

$$\mathbb{L} \times 2^{\mathbb{L}} \to 2^{\mathbb{L}},$$

$$\texttt{trace}(\ell, S) \mapsto \begin{cases} \{\ell\}, & \text{if } \ell \in S, \\ \{\bot\}, & \text{if } \ell \notin S, \alpha(\ell) = \mathfrak{d}, \\ \bigcup_{\ell' \in \alpha(\ell) \setminus \{\ell, \neg\ell\}} \texttt{trace}(\ell', S), & \text{otherwise.} \end{cases}$$

If $\alpha(\ell) = \mathfrak{u}$, we consider $\alpha(\ell) \setminus \{\ell, \neg\ell\}$ to be the empty set; in practice, we will apply `trace()` to literals occurring in the implication graph, and so this this latter case will not occur. Using this function, we define Algorithm 3.3. Note that if there exists a literal ℓ with $\alpha(\ell) \neq \delta$ in a learnt clause $\mathfrak{c}$ such that

$$\alpha(\ell) \setminus \{\neg\ell\} \subset \mathfrak{c} \setminus \{\ell\},$$

then Algorithm 3.3 will eliminate this literal from $\mathfrak{c}$. (The way conflict-induced clauses are constructed, $\ell \in \mathfrak{c}$ with $\alpha(\ell) \neq \mathfrak{d}$ implies that $\neg\ell \in \alpha(\ell)$.) Hence, recursive minimisation is stronger than local minimisation. Before proving that recursive minimisation produces useful learnt clauses in the sense of Lemma 2.13, let's look at an example.

---

the basis of $\texttt{causes_of}(\bot, \{u\})$ for each $u \in U_I$. In the example shown in Figure 2.3, this would result in the implicates $\mathfrak{c}_7 = (\bar{e} \vee \bar{h})$ and $\mathfrak{c}_8 = (\bar{a} \vee \bar{b} \vee \bar{h})$.

Example 3.10 (Recursive minimisation)**.** Sörensson & Biere (2009) present the following example. Consider the CNF formula

$$\begin{aligned}\mathcal{F} = \underbrace{a}_{=\mathfrak{c}_1} \wedge \underbrace{b}_{=\mathfrak{c}_2} \wedge \underbrace{(\overline{a} \vee \overline{c} \vee d)}_{=\mathfrak{c}_3} \wedge \underbrace{(\overline{b} \vee \overline{d} \vee e)}_{=\mathfrak{c}_4} \wedge \\ \underbrace{(\overline{d} \vee \overline{f} \vee g)}_{=\mathfrak{c}_5} \wedge \underbrace{(\overline{e} \vee \overline{g} \vee h)}_{=\mathfrak{c}_6} \wedge \underbrace{(\overline{h} \vee i)}_{=\mathfrak{c}_7} \wedge \underbrace{(\overline{k} \vee \ell)}_{=\mathfrak{c}_8} \wedge \\ \underbrace{(\overline{\ell} \vee \overline{r} \vee s)}_{=\mathfrak{c}_9} \wedge \underbrace{(\overline{d} \vee \overline{g} \vee \overline{s} \vee t)}_{=\mathfrak{c}_{10}} \wedge \underbrace{(\overline{s} \vee x)}_{=\mathfrak{c}_{11}} \wedge \underbrace{(\overline{x} \vee z)}_{=\mathfrak{c}_{12}} \wedge \\ \underbrace{(\overline{h} \vee \overline{i} \vee \overline{t} \vee y)}_{=\mathfrak{c}_{13}} \wedge \underbrace{(\overline{y} \vee \overline{z})}_{=\mathfrak{c}_{14}}.\end{aligned}$$

Figure 3.2 shows a possible implication graph with a conflict node. If we used the 1-UIP strategy without minimisation, we would obtain the learnt clause $\mathfrak{c}_{\text{new}} = \overline{d} \vee \overline{g} \vee \overline{h} \vee \overline{i} \vee \overline{s}$. If we applied local minimisation, we would be able to remove $\overline{i}$ from this clause, since $\alpha(\overline{i}) \setminus \{i\} = \{\overline{h}\} \subset \mathfrak{c}_{\text{new}}$. Using recursive minimisation, we would observe that

$$\texttt{trace}(\overline{i}, \{\overline{d}, \overline{g}, \overline{h}, \overline{s}\}) = \texttt{trace}(\overline{h}, \{\overline{d}, \overline{g}, \overline{h}, \overline{s}\}) = \{\overline{h}\} \subset \mathfrak{c}_{\text{new}},$$

so we can remove the literal $\overline{i}$ from $\mathfrak{c}_{\text{new}}$. We would further observe that

$$\begin{aligned}\texttt{trace}(\overline{h}, \{\overline{d}, \overline{g}, \overline{s}\}) &= \texttt{trace}(\overline{g}, \{\overline{d}, \overline{g}, \overline{s}\}) \cup \texttt{trace}(\overline{e}, \{\overline{d}, \overline{g}, \overline{s}\}) \\ &= \{\overline{g}\} \cup \texttt{trace}(\overline{d}, \{\overline{d}, \overline{g}, \overline{s}\}) \cup \texttt{trace}(\overline{b}, \{\overline{d}, \overline{g}, \overline{s}\}) \\ &= \{\overline{d}, \overline{g}\} \cup \emptyset \\ &\subset \mathfrak{c}_{\text{new}} \setminus \{\overline{i}\}.\end{aligned}$$

since $\alpha(\overline{b}) \setminus \{b\} = \emptyset$. So we would also remove $\overline{h}$ from the learnt clause. No further reductions can be achieved using recursive minimisation since

$$\texttt{trace}(\overline{d}, \{\overline{g}, \overline{s}\}) = \texttt{trace}(\overline{a}, \{\overline{g}, \overline{s}\}) \cup \texttt{trace}(\overline{c}, \{\overline{g}, \overline{s}\}) = \{c\} \not\subset \mathfrak{c}_{\text{new}}$$

and

$$\texttt{trace}(\overline{g}, \{\overline{d}, \overline{s}\}) = \texttt{trace}(\overline{f}, \{\overline{d}, \overline{s}\}) \cup \texttt{trace}(\overline{d}, \{\overline{d}, \overline{s}\}) = \{\overline{d}, f\} \not\subset \mathfrak{c}_{\text{new}}. \qquad \diamond$$

Lemma 3.11 (Recursive minimisation)**.** *Let $\mathcal{F}$ be a* CNF *formula and let $\mathfrak{c}$ be a conflict-induced implicate of $\mathcal{F}$ under the augmented value assignment N. Let $\ell \in \mathfrak{c}$ be a literal with $\alpha(\ell) \neq \mathfrak{d}$ such that $\texttt{trace}(\ell, \mathfrak{c} \setminus \{\ell\}) \subset \mathfrak{c}$. Then $\mathfrak{c} \setminus \{\ell\}$ is also an implicate of $\mathcal{F}$.*

Proof. Consider the recursion tree defined by the initial call $\texttt{trace}(\ell, \mathfrak{c} \setminus \{\ell\})$. The nodes of this tree are labelled with subsets of $\mathbb{L}$ in the following way. First, the root node is labelled $\{\ell\}$. Next, given a node labelled $\{\ell'\} \subset \mathbb{L} \setminus \{\bot\}$ in the recursion tree, its children are labelled as follows.

- If $\texttt{trace}(\ell', \mathfrak{c} \setminus \{\ell\}) = \{\ell'\}$, then $\{\ell'\}$ does not have any children, i.e., the node is a leaf.
- If $\texttt{trace}(\ell', \mathfrak{c} \setminus \{\ell\}) = \emptyset$, then it has exactly one child, which is labelled $\emptyset$. This child is a leaf.

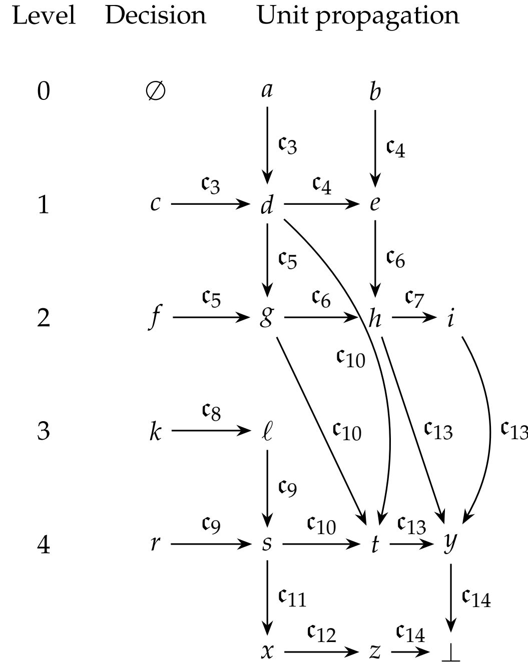

Figure 3.2: Implication graph of a value assignment leading to a conflict, see Example 3.10.

- If $\texttt{trace}(\ell', \mathfrak{c} \setminus \{\ell\}) = \{\bot\}$, then it has exactly one child, which is labelled $\{\bot\}$. This child is a leaf. Note that under the hypothesis that $\texttt{trace}(\ell, \mathfrak{c} \setminus \{\ell\}) \subset \mathfrak{c}$, no leaves are labelled $\{\bot\}$.

- Otherwise, the node has as many children as there are literals in $\alpha(\ell') \setminus \{\neg\ell'\}$, with the singleton sets containing these literals being their respective labels. To see why we only need to remove $\neg\ell'$ from $\alpha(\ell')$ and not ℓ', consider the following. As mentioned above, for $\ell \in \mathfrak{c}$ with $\alpha(\ell) \neq \mathfrak{d}$, we have $\neg\ell \in \alpha(\ell)$. Moreover, because of the unit clause rule, for any $\ell' \in \alpha(\ell) \setminus \{\neg\ell\}$ with $\alpha(\ell') \neq \mathfrak{d}$, we have $(\ell')^\nu = 0$ under the current value assignment ν. Hence, $\neg\ell' \in \alpha(\ell')$. Inductively, the claim follows.

Further note that different nodes may be labelled with the same subset of $\mathbb{L}$, but that subtrees with the same root label all look the same.

For a subtree with a node labelled $\{\ell'\}$ as its root, define descendants(ℓ', k) to be the union over the leaf labels in that subtree at a tree depth of at most k as well as over the node labels in that subtree at a tree depth of exactly k. Note that

$$\text{descendants}(\ell, k_{\text{max}}) = \texttt{trace}(\ell, \mathfrak{c} \setminus \{\ell\}),$$

where k_{max} is the depth of the full tree. We now show the following claim by induction on $k \geq 0$: Any model ν of $\mathcal{F}$ that satisfies $(\widetilde{\ell})^\nu = 0$ for each $\widetilde{\ell} \in \text{descendants}(\ell, k)$ also satisfies $\ell^\nu = 0$.

For $k = 0$, descendants$(\ell, k) = \{\ell\}$, and the claim holds.

For $k \geq 1$, consider a non-leaf node labelled ℓ' at depth $k-1$. We distinguish two cases. First, if $\text{descendants}(\ell', 1) = \emptyset$, then $\alpha(\ell') = \{\neg\ell'\}$. Hence any model ν of $\mathcal{F}$ satisfies $(\neg\ell')^\nu = 1$, that is, $(\ell')^\nu = 0$. Second, if $\text{descendants}(\ell', 1) = \alpha(\ell') \setminus \{\neg\ell'\}$, then any model ν that satisfies $(\widetilde{\ell})^\nu = 0$ for all $\widetilde{\ell} \in \text{descendants}(\ell', 1)$ also satisfies $(\neg\ell')^\nu = 1$ by the unit clause rule. Hence, in both cases, we have $(\ell')^\nu = 0$. We can now apply the induction hypothesis to $\text{descendants}(\ell, k-1)$, which yields the claim.

Returning to the claim of the lemma itself, we know that $\mathfrak{c}$ is an implicate of $\mathcal{F}$. By contraposing the claim just shown, any model ν of $\mathcal{F}$ with $\ell^\nu = 1$ also satisfies $(\ell')^\nu = 1$ for at least one $\ell' \in \texttt{trace}(\ell, \mathfrak{c} \setminus \{\ell\}) \subset \mathfrak{c} \setminus \{\ell\}$. Hence, the model ν already satisfies $(\mathfrak{c} \setminus \{\ell\})^\nu = 1$. □

Finally, note that if a newly learnt clause $\mathfrak{c}$ contains exactly one literal ℓ at the current decision level, then $\{\bot\} \in \texttt{trace}(\ell, \mathfrak{c} \setminus \{\ell\})$. Hence, this literal will not be deleted from the learnt clause. As a result, recursive minimisation produces useful clauses in the sense of Lemma 2.13 on page 12.

3.2 Better value assignments by decision

Up till now, we only assumed that the `PickBranchVariable()` function selected some as yet unassigned variable and assigned it some value in $\{0, 1\}$. For the examples, these variables were chosen in lexicographical order and they were always assigned the value 1. An alternative branching heuristic would be to randomly pick each decision variable from the set of unassigned variables and to randomly assign to it a value in $\{0, 1\}$. Particularly in the earlier SAT solvers, sundry heuristics were used that were more greedy in the sense that they gave precedence to value assignments that immediately satisfied the largest number of clauses or that brought the largest number of clauses a step closer to becoming unit. For a description and empirical comparison of some of these heuristics, I refer to Marques-Silva (1999).

The watershed in the development of branching heuristics was the introduction of the Variable State Independent Decaying Sum (VSIDS) strategy by Moskewicz et al. (2001). The original VSIDS heuristic works as follows. For each $x \in \mathbb{V}$, we maintain the counters $s(x)$ and $s(\overline{x})$, which are both initialised to 0. We further pick an $\alpha \in (0, 1)$, referred to as the decaying factor, and an interval size $i \in \mathbb{N}_{\geq 1}$. When a conflict-induced clause $\mathfrak{c}$ is learnt, we increment $s(\ell)$ by 1 for each $\ell \in \mathfrak{c}$; this logic can be implemented in the `CreateClause()` helper function. When choosing the next decision variable, `PickBranchVariable()` returns a variable var satisfying

$$\mathsf{var} \in \operatorname*{arg\,max}_{y \in \mathbb{V}: \nu(y) = \mathfrak{u}} \max\{s(y), s(\overline{y})\}.$$

If several variables satisfy this condition, one may be picked at random from them. This variable is then assigned the value 1 if $s(\mathsf{var}) > s(\overline{\mathsf{var}})$ and the value 0 if $s(\mathsf{var}) < s(\overline{\mathsf{var}})$. Otherwise, a value may be picked at random. After every i conflicts, we multiply all counters by α. As Moskewicz et al. (2001) explain, the motivating idea behind this heuristic is to prioritise satisfying recently learnt clauses.

The VSIDS strategy compared so favourably to its predecessors that some variation of it was implemented in most competitive SAT solvers. These variations dispensed with the idea of

maintaining counters for literals, instead keeping counters for variables only. This turned the algorithm into a decision variable strategy, with the task of assigning a value to the selected decision variable being delegated to a different algorithm (e.g., Jeroslow & Wang, 1990; Pipatsrisawat & Darwiche, 2007). For a discussion and empirical comparison of some variations on the basic idea of the original VSIDS strategy, see Biere & Fröhlich (2015).

However, it is still not entirely clear why VSIDS-based variable selection heuristics perform well in practice. An empirical study by Liang et al. (2015) suggests that VSIDS' success is related to the community structure often exhibited by SAT problems in practice (see Section 2.6 on page 24 on the notion of community structure). More specifically, VSIDS seems to favour variables that represent bridges between different communities of variables. Once values have been assigned to the bridges between two communities, these two communities become less directly dependent of one another, which may help the solver to take some kind of divide-and-conquer approach.

3.3 Forgetting learnt clauses

CDCL-based SAT solvers often learn an enormous amount of clauses, which both poses challenges in terms of memory management and slows down unit propagation (also see Krüger et al., 2022). Modern SAT solvers therefore regularly prune the set of learnt clauses by deleting learnt clauses that, according to some heuristic, are predicted to be of little further use to the solver. In order to remain complete, the solver needs to become ever less picky in terms of how many clauses it retains (see the termination lemma on page 21). To give an example, Audemard & Simon (2009) introduced the following heuristic. First, each time a clause is learnt, record its literal block distance (see Definition 3.1). After 20,000 conflicts (i.e., after 20,000 learnt clauses), delete the 50% of learnt clauses with the highest recorded literal block distances, but keep the most recently learnt clause. Repeat the action after every 500 conflicts.

3.4 Restarting

The final tweak I wanted to touch on are the restart policies that are commonly implemented in modern CDCL-based SAT solvers. These policies involve frequent backtracking to decision level 0, though solvers differ substantially in terms of when they restart. For an overview and empirical evaluation of different restart policies, I refer to Biere & Fröhlich (2019). According to Liang et al. (2018), it is not clear how restarts improve the solvers' performance, but their empirical evidence suggests that frequent restarts result in higher-quality learnt clauses, i.e., clauses with lower literal block distances. A key challenge facing solver engineers is to balance this advantage of frequent restarts against the computational overhead that restarting incurs.

Chapter 4

Application to classical planning

A commonly cited application of CDCL-based SAT solvers is solving classical planning problems (e.g., Fichte et al., 2023; Marques-Silva et al., 2021). Put simply, classical planning problems encode move-based games with well-defined, deterministic rules, starting positions, and desired final positions, and ask whether there exists a finite sequence of allowed moves that get us from the starting position to the desired position. In this chapter, I discuss the nuts and bolts of classical planning – a full-blown introduction would lead us down a few rabbit holes – and how classical planning problems can be solved using SAT solvers. I also present two concrete examples that illustrate the basic principles of encoding planning problems as SAT problems. For more advanced topics related to using SAT solvers for tackling planning problems, I refer to Rintanen (2012, 2021).

4.1 Basics of classical planning

Two somewhat different compact formalisations of classical planning can be found in Rintanen (2012) and Rintanen (2021). I found the formalisation in the earlier publication easier to reconcile with my treatment of CDCL-based SAT solvers, so the following definitions are largely based on Rintanen (2012), though still with some modifications.

Definition 4.1 (Classical planning problems)**.** Propositional variables are referred to as **state variables**. Finite sets of state variables are denoted by upper-case Latin letters; in the following, X will denote a generic finite set of state variables.

A **state** $s : X \to \{0, 1, \mathfrak{u}\}$ is a valuation of X. If $s(X) \subset \{0, 1\}$, we call s a **complete state**; if $\mathfrak{u} \in s(X)$, we call s a **partial state**. It is convenient to represent states as sets of literals over X, that is, as

$$S = \{x : s(x) = 1\} \cup \{\overline{x} : s(x) = 0\}.$$

As another matter of convenience, we allow ourselves to write $s(\ell)$, where $\ell \in \{x, \overline{x}\}$ is a literal over X. By this notation, we mean

$$s(\ell) = \begin{cases} \mathfrak{u}, & \text{if } s(x) = \mathfrak{u}, \\ s(x), & \text{if } s(x) \in \{0,1\}, \ell = x, \\ 1 - s(x), & \text{if } s(x) \in \{0,1\}, \ell = \overline{x}. \end{cases}$$

An **action** is a pair $\langle p, e \rangle$ such that p and e are consistent sets of literals over X. More precisely, $\ell \in p$ implies both that $\ell \in \{x, \overline{x}\}$ for some $x \in X$ and that $\neg\ell \notin p$, and mutatis mutandis for e. The set p is called the **precondition** of a, whereas e is called the **effects** of a.

An action $a = \langle p, e \rangle$ is called **executable** in a state s if $s \models \ell$ for each $\ell \in p$. If an action $a = \langle p, e \rangle$ is executable in s, we define the **successor state** $\text{exec}_a(s) : X \to \{0,1\}$ of s after executing a through

$$\text{exec}_a(s)(x) := \begin{cases} 1, & \text{if } x \in e, \\ 0, & \text{if } \overline{x} \in e, \\ s(x), & \text{else.} \end{cases}$$

Since $\text{exec}_a(s)$ is a state, we may represent it as a set of literals, like before.

A **problem instance** $\pi = \langle X, I, A, G \rangle$ consists of a finite set X of state variables, a complete state $I : X \to \{0,1\}$ called the **initial state**, a set A of actions, and another, possibly partial, state $G : X \to \{0,1\}$ called the **goal**. $\diamondsuit$

Given a problem instance $\pi = \langle X, I, A, G \rangle$, the classical planning problem asks if there exists a finite sequence $(a_1, \ldots, a_n) \subset A$ such that

$$\text{exec}_{a_n} \circ \cdots \circ \text{exec}_{a_1}(I) \supset G.$$

What this boils down to is that we want to know if there is a finite sequence of allowed moves ('executable actions') via which we can reach the goal starting from the initial state. If such a sequence exists, we call it a **plan**.

By adopting the definitions above, I have restricted the scope of this chapter to sequential plans, i.e., plans in which actions are executed one at a time. These contrast with parallel plans, in which several actions may be executed simultaneously – for instance, as when simultaneously sending one lorry from Basel to Härkingen and another one from Härkingen to Lucerne.

Intuition suggests that we may break down long plans into shorter ones by setting intermediate goals and that we may stitch these shorter plans back together. By the following two lemmas, which we will need shortly, this is indeed the case.

Lemma 4.2. *Let $\pi = \langle X, I, A, G \rangle$ be a problem instance with a plan $(a_1, \ldots, a_n)$, $n \geq 2$. For any $k \in \{1, \ldots, n-1\}$, define*

$$S_{k+1} := \mathit{exec}_{a_k} \circ \cdots \circ \mathit{exec}_{a_1}(I).$$

Then $\pi_1 := \langle X, I, A, S_{k+1}\rangle$ *and* $\pi_2 := \langle X, S_{k+1}, A, G\rangle$ *are problem instances with plans* $(a_1, \ldots, a_k)$ *and* $(a_{k+1}, \ldots, a_n)$*, respectively.*

Proof. Successor states of complete states are themselves complete states, so S_{k+1} is a valid initial state. Hence, π_1, π_2 are valid problem instances. It is clear that $(a_1, \ldots, a_k)$ is a plan for π_1. Since $(a_1, \ldots, a_n)$ is a plan for π, we have

$$\text{exec}_{a_n} \circ \ldots \text{exec}_{a_{k+1}} \left(\underbrace{\text{exec}_{a_k} \circ \cdots \circ \text{exec}_{a_1}(I)}_{=S_{k+1}} \right) \supset G,$$

so $(a_{k+1}, \ldots, a_n)$ is a plan for π_2. □

Lemma 4.3. *Let* $\pi_1 = \langle X, I, A, G_1\rangle$*, with* G_1 *a complete state, be a problem instance with a plan* $(a_1, \ldots, a_n)$ *and let* $\pi_2 = \langle X, G_1, A, G_2\rangle$ *be a problem instance with a plan* $(\widetilde{a}_1, \ldots, \widetilde{a}_{\widetilde{n}})$*. Then the problem instance* $\pi = \langle X, I, A, G_2\rangle$ *has a plan* $(a_1, \ldots, a_n, \widetilde{a}_1, \ldots, \widetilde{a}_{\widetilde{n}})$*.*

Proof. Since $(a_1, \ldots, a_n)$ is a plan for π_1, we have $\text{exec}_{a_n} \circ \cdots \circ \text{exec}_{a_1}(I) \supset G_1$. Since G_1 is a complete state, we even have $\text{exec}_{a_n} \circ \cdots \circ \text{exec}_{a_1}(I) = G_1$. Since $(\widetilde{a}_1, \ldots, \widetilde{a}_{\widetilde{n}})$ is plan for getting from G_1 to G_2, it follows that

$$\text{exec}_{\widetilde{a}_{\widetilde{n}}} \circ \cdots \circ \text{exec}_{\widetilde{a}_{\widetilde{1}}} (\text{exec}_{a_n} \circ \cdots \circ \text{exec}_{a_1}(I)) = \text{exec}_{\widetilde{a}_{\widetilde{n}}} \circ \cdots \circ \text{exec}_{\widetilde{a}_{\widetilde{1}}}(G_1) \supset G_2.$$ □

4.2 Conversion to SAT

Kautz & Selman (1992) proposed a general scheme for converting classical planning problems to SAT problems. The idea is this. First, we specify the number $t \geq 1$ of actions the plan must contain. Then, we express the initial state, goal, and actions using propositional variables and encode their interrelations in CNF form, as detailed below. We submit this CNF formula to a SAT solver. If the formula is satisfiable, we read out a model and derive a plan of length t from it; as we will see in the examples, this is trivial. If the formula is not satisfiable, we pick another value for t and start again.

Since classical planning is PSPACE-complete and SAT is NP-complete, it is still an open question whether classical planning can efficiently be reduced to SAT. Kautz and Selman's proposal sidesteps this issue by reformulating planning problems as *bounded* planning problems and then trying out different bounds. If there does not exist any $t \geq 1$ such that the CNF formula fed to the SAT solver is satisfiable, the search for a plan would in principle go on indefinitely.

Let $\pi = \langle X, I, A, G\rangle$ be a problem instance and let $t \geq 1$ be the desired plan length. We convert π to a propositional formula $\mathcal{F}$ as follows. First, we define the appropriate set of propositional variables:

$$\mathbb{V} := \{x@i : x \in X, i \in \{1, \ldots, t+1\}\} \cup \{a@i : a \in A, i \in \{1, \ldots, t\}\}.$$

That is, for each state variable, we create a propositional variable that encodes the state of this state variable before the first action ($x@1$) and after each action ($x@2, \ldots, x@(t+1)$). We similarly

associate each action with a propositional variable at each time point, indicating whether the action was executed at this time point. I call the suffix following the @ the time index of the propositional variable. In the following, $\ell@i$ is to be understood as $x@i$ if $\ell = x$ and as $\overline{x@i}$ if $\ell = \overline{x}$.

Second, we create sets of clauses that encode the initial state and the goal, namely

$$\mathcal{I} := \bigcup_{\ell \in I} \{\{\ell@1\}\}$$

and

$$\mathcal{G} := \bigcup_{\ell \in G} \{\{\ell@(t+1)\}\}.$$

Third, we require that actions can only be executed if their preconditions hold. To this end, we include in the propositional formula the clause

$$a@i \to \bigwedge_{\ell \in p} \ell@i$$

for each $a = \langle p, e \rangle \in A$ and for each $i = 1, \ldots, t$. We encode this as

$$\mathcal{P} := \bigcup_{i \in \{1,\ldots,t\}} \bigcup_{a = \langle p,e \rangle \in A} \bigcup_{\ell \in p} \{\{\overline{a@i}, \ell@i\}\}.$$

Fourth, executing an action causes its effects. To this end, we include in the propositional formula the clause

$$a@i \to \bigwedge_{\ell \in e} \ell@(i+1)$$

for each $a = \langle p, e \rangle \in A$ and for each $i = 1, \ldots, t$. This is similarly encoded as

$$\mathcal{E} := \bigcup_{i \in \{1,\ldots,t\}} \bigcup_{a = \langle p,e \rangle \in A} \bigcup_{\ell \in e} \{\{\overline{a@i}, \ell@(i+1)\}\}.$$

In other words, actions imply both their preconditions and their effects.

Fifth, we require that changes in the state only come about through relevant actions. In other words, if the state of x flips from 0 to 1 at some juncture, then some action $a = \langle p, e \rangle$ with $x \in e$ must just have been executed. To this end, we include in the propositional formula the clause

$$\overline{x@i} \wedge x@(i+1) \to \bigvee \{a@i : a = \langle p, e \rangle \in A, x \in e\}$$

for each $x \in X$ and for each $i = 1, \ldots$,t. We write this as

$$\mathcal{C}_{01} := \bigcup_{i \in \{1,\ldots,t\}} \bigcup_{x \in X} \Big\{ \{x@i, \overline{x@(i+1)}\} \cup \{a@i : a = \langle p, e \rangle \in A, x \in e\} \Big\}$$

By the same token, if the state of x flips from 1 to 0, then an action $a = \langle p, e\rangle$ with $\overline{x} \in e$ must just have been executed:

$$\mathcal{C}_{10} := \bigcup_{i\in\{1,\ldots,t\}} \bigcup_{x\in X} \{\{\overline{x@i}, x@(i+1)\} \cup \{a@i : a = \langle p, e\rangle \in A, \overline{x} \in e\}\}$$

Sixth, we encode that some action occurs at each time:

$$\mathcal{S} := \bigcup_{i\in\{1,\ldots,t\}} \{\{a@i : a \in A\}\}.$$

Seventh, and finally, we assert that at most one action occurs at each time. To this end, we include in the propositional formula the clause

$$a@i \to \overline{b@i}$$

for each $a, b \in A$ with $a \neq b$ and for each $i = 1, \ldots, t$. We write this as

$$\mathcal{O} := \bigcup_{i\in\{1,\ldots,t\}} \bigcup_{a\in A} \bigcup_{b\in A\setminus\{a\}} \left\{\{\overline{a@i}, \overline{b@i}\}\right\}.$$

The CNF formula submitted to the SAT solver is the conjunction of all these CNF formulas:

$$\mathcal{F} := \mathcal{I} \wedge \mathcal{G} \wedge \mathcal{P} \wedge \mathcal{E} \wedge \mathcal{C}_{01} \wedge \mathcal{C}_{10} \wedge \mathcal{S} \wedge \mathcal{O}.$$

The construction of this CNF formula is justified by the following theorem.

Theorem 4.4. *Let $\pi = \langle X, I, A, G\rangle$ be a problem instance, let $t \geq 1$, and let $\mathcal{F}$ be the* CNF *formula derived from π by the process outlined above. Then there exists a plan $(a_1, \ldots, a_t)$ for π if and only if there exists a model ν of $\mathcal{F}$ such that $\nu(a_i@i) = 1$ for $i = 1, \ldots, t$.*

Proof. First, assume that $t = 1$ and that $(a_1) = (\langle p_1, e_1\rangle)$ is a plan for getting from I to G. Then $\text{exec}_{a_1}(I) \supset G$, which means that two sets of conditions are fulfilled:

1. $p_1 \subset I$,
2. $G \subset e_1 \cup \{\ell \in I : \ell \notin e_1, \neg\ell \notin e_1\}$.

We construct a model ν of $\mathcal{F}$ by specifying the value assignments to disjoint sets of literals as follows.

1. For all $\ell \in I$, set $\ell@1^\nu = 1$. This immediately satisfies $\mathcal{I}$.
2. Set $\nu(a_1@1) = 1$ and $\nu(a_j@1) = 0$ for all $a_j \in A$ with $a_j \neq a_1$. This immediately satisfies both $\mathcal{S}$ and $\mathcal{O}$. Further, since $p_1 \subset I$, all clauses in $\mathcal{P}$ are now satisfied as well.
3. For all $\ell \in e_1$, set $\ell@2^\nu = 1$. Now, all clauses in $\mathcal{E}$ are satisfied.

4. For all $x \in X$ with $\{x, \overline{x}\} \cap e_1 = \emptyset$, set $\nu(x@2) = \nu(x@1)$. Now, both $\mathcal{C}_{01}$ and $\mathcal{C}_{10}$ are satisfied. Further, since G is a subset of the union of the disjoint sets e_1 and $\{\ell \in I : \ell \notin e_1, \neg\ell \notin e_1\}$, all clauses in $\mathcal{G}$ are satisfied, too.

Hence, all constituent formulas of $\mathcal{F}$ are satisfied, so ν is a model of $\mathcal{F}$.

Now assume that $t \geq 2$ and that $(a_1, \dots, a_t) = (\langle p_1, e_1 \rangle, \dots, \langle p_t, e_t \rangle)$. Use Lemma 4.3 to define problem instances $\pi_1 := \langle X, I, A, S_t \rangle$ and $\pi_2 := \langle X, S_t, A, G \rangle$ that have plans $(a_1, \dots, a_{t-1})$ and (a_t), respectively. By the induction hypothesis, the CNF formulas $\mathcal{F}_1$ and $\mathcal{F}_2$ corresponding to these problem instances for desired plan lengths $t-1$ and 1, respectively, are satisfiable. We can rename the propositional variables occurring in $\mathcal{F}_2$ by starting the time-indexing at t instead of 1. Let ν_1, ν_2 be models of $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively. It is clear that all propositional variables occurring in $\mathcal{F}$ occur in $\mathcal{F}_1$ or $\mathcal{F}_2$. The only propositional variables occurring in both $\mathcal{F}_1$ and $\mathcal{F}_2$ have t as their time index. These propositional variables encode the same complete state S_t, so we have $\nu_1(\ell@(t)) = \nu_2(\ell@(t))$ for all $\ell \in S_t$. Hence, we may define a model ν of $\mathcal{F}$ as follows:

$$\nu : \mathrm{var}(\mathcal{F}) \to \{0, 1\},$$
$$\nu(x) \mapsto \begin{cases} \nu_1(x), & \text{if } x \in \mathrm{var}(\mathcal{F}_1), \\ \nu_2(x), & \text{else.} \end{cases}$$

Now assume that ν is a model of the CNF formula $\mathcal{F}$ corresponding to π with $\nu(a_i@i) = 1$ for $i = 1, \dots, t$. Since ν satisfies $\mathcal{P}$ and $\mathcal{I}$, action $a_1 = \langle p_1, e_1 \rangle$ is executable in state I. Define $S_2 := \mathrm{exec}_{a_1}(I)$ and $\pi_1 := \langle X, I, A, S_2 \rangle$. Then (a_1) is a plan for π_1. Now consider $a_2 = \langle p_2, e_2 \rangle$. Let $\ell \in p_2$. Since subformula $\mathcal{P}$ is satisfied, we know that $\nu(\ell@2) = 1$. Assume towards a contradiction that $S_2(\ell) = 0$. Then $\ell \notin e_1$, so $I(\ell) = 0$. Subformula $\mathcal{I}$ guarantees that $\nu(\ell@1) = 0$. But then subformulas $\mathcal{C}_{01}$ and $\mathcal{C}_{10}$ ensure that $\ell \in e_1$. Contradiction. So $S_2(\ell) = 1$. Hence, action a_2 is executable in state S_2. Inductively, action a_i is executable in state $S_i := \mathrm{exec}_{a_{i-1}}(S_{i-1})$ for $i = 3, \dots, t$.

It remains to show that $S_{t+1} := \mathrm{exec}_{a_t} S_t \supset G$. To that end, consider $\ell \in G$. From subformula $\mathcal{G}$, we know that $\nu(\ell@(t+1)) = 1$. Assume towards a contradiction that $\ell \notin \mathrm{exec}_{a_t} S_t$, that is, $\mathrm{exec}_{a_t} S_t(\ell) = 0$. This implies $\neg\ell \in e_t$ (where e_t is the effects of a_t) or $S_t(\ell) = 0$. If $\neg\ell \in e_t$, then subformula $\mathcal{E}$ guarantees that $\nu(\ell@(t+1)) = 0$. Contradiction. If $S_t(\ell) = 0$, then $\nu(\ell@t) = 0$. Subformulas $\mathcal{C}_{01}$ and $\mathcal{C}_{10}$ now ensure that $\ell \in e_t$. Contradiction. So $\mathrm{exec}_{a_t} S_t \supset G$. Applying Lemma 4.2 $t-1$ times, we obtain the plan $(a_1, \dots, a_t)$ for problem instance π. □

Once a model ν of $\mathcal{F}$ is found, we can read off, for each $i = 1, \dots, t$, which action $a \in A$ satisfies $\nu(a@i) = 1$. These actions form a plan for getting from I to G.

4.3 Examples

In this section, I present two examples of fairly simple problems that can be formulated as planning problems and demonstrate how these can be solved using a SAT solver. In order to generate the CNF formulas corresponding to these problems and to display the output of the SAT solver in a more human-friendly format, I wrote a Julia package, `SatplanExamples.jl`, that is

available from `https://github.com/janhove/SatplanExamples.jl`. For more about the Julia language, I refer to `https://julialang.org/`. For these examples, I used the SAT solver Glucose, version 4.2.1, which is available from `https://github.com/audemard/glucose`.

4.3.1 The blocks world

The blocks world is a toy problem in the field of artificial intelligence. A number of labelled blocks are laid out in stacks on a table. We can pick up blocks that are at the top of a stack and put them on top of another stack or one the table one at a time. We want to obtain a plan for converting the initial arrangement into some desired arrangement.

Figure 4.1 shows the initial state and the goal of a blocks world problem. Since this particular problem cannot be solved by greedily satisfying subfeatures of the goal (e.g., putting B on C and then figuring out how to put A on B), it is known in the literature as Sussman's anomaly. We spell out the state variables as follows.

- The first set of state variables specifies for each tuple of blocks (x, y) if block x is directly on top of block y: $on(A, B), on(B, A)$, etc.
- The second set of state variables specifies for each block if it is directly on top of the table: $on(A, table), on(B, table)$, etc.
- The third set specifies for each block if it is clear, that is, if there is room on top of it: $clear(A), clear(B)$, etc.
- Further, the table is always clear: $clear(table)$.

Actions have the form $move(x, y, z)$, where x is a block, y is a block or the table, and z is a block or the table, such that $x \neq y, x \neq z, y \neq z$. The precondition of the action $move(x, y, z)$ is

$$\{clear(x), on(x, y), clear(z)\};$$

the effects of this action are

$$\{\neg on(x, y), on(x, z), clear(y), \neg clear(z)\},$$

if z is a block, and

$$\{\neg on(x, y), on(x, z), clear(y)\},$$

if z is the table. The initial state in Figure 4.1 may be specified as

$$\begin{aligned}
\{&on(A, table), \neg on(A, B), \neg on(A, C), \neg clear(A),\\
&on(B, table), \neg on(B, A), \neg on(B, C), clear(B),\\
&\neg on(C, table), on(C, A), \neg on(C, B), clear(C),\\
&clear(table)\}.
\end{aligned}$$

(a) Initial state. **(b)** Goal.

Figure 4.1: A blocks world problem known as Sussman's anomaly.

Similarly, the goal state may be specified as

$$\{\neg on(A, table), on(A, B), \neg on(A, C), clear(A),$$
$$\neg on(B, table), \neg on(B, A), on(B, C), \neg clear(B),$$
$$on(C, table), \neg on(C, A), \neg on(C, B), \neg clear(C),$$
$$clear(table)\}.$$

Note that this is not the only valid way to specify the problem instance. For example, we could have included $clear(table)$ in the effects of all actions. We could also have included actions such as $move(A, B, B)$ in the action set, that is, moves whose preconditions could never be met.

The `sussman()` function in the Julia package generates the CNF formula for the desired number of moves. Since it is clear that we will need three moves, we pick $t = 3$:

```
julia> cnf = sussman(3);
```

Using the `to_DIMACS()` function, the `cnf` object can be saved in the DIMACS format required by SAT solvers (`sussman.cnf`); the `sussman.code` serves as a codebook that will later be used for post-processing the output produced by the solver:

```
julia> to_DIMACS(cnf, "sussman.cnf", "sussman.code");
```

Both the `sussman.cnf` and `sussman.code` file are plain text files and are available from the Julia package's Github page.

For $t = 3$, the DIMACS file contains 106 propositional variables and 927 distinct clauses. These numbers differ markedly from those reported by Kautz & Selman (1992), whose SAT encoding of Sussman's anomaly comprised 127 variables and 2364 clauses. Their description suggests that they included moves whose preconditions can never be met (e.g., $move(A, B, B)$) in the action set, which would go some way towards explaining the difference. Since testing my encoding only resulted in valid plans (for $t \geq 3$) or in correct claims of unsatisfiability (for $t = 1, 2$), I did not attempt to recreate a CNF encoding that matches Kautz and Selman's exactly.

Having generated the DIMACS file, we can feed it to a SAT solver, for instance, Glucose. A model of the CNF formula is output as `sussman.soln`:

```
(base) jan@jan-xps:~$ glucose sussman.cnf sussman.soln
```

For this small a formula, the solver should find a solution in perhaps a few milliseconds or less. During preprocessing, some clauses get dropped, for instance, because they are supersets of

other clauses. The Julia package contains some functions for post-processing the output file. If we copy the `sussman.soln` file to our Julia working directory, we can output the plan in a more readable format:

```
julia> print_moves("sussman.soln", "sussman.code")
3-element Vector{String}:
 "move C from A to table at time 1"
 "move B from table to C at time 2"
 "move A from table to B at time 3"
```

We may also inspect the state variables that evaluate to true at a certain point in time, for instance like so:

```
julia> print_state("sussman.soln", "sussman.code", 2)
7-element Vector{String}:
 "C clear at time 2"
 "table clear at time 2"
 "B on table at time 2"
 "B clear at time 2"
 "C on table at time 2"
 "A clear at time 2"
 "A on table at time 2"
```

The other post-processing function provided in the package is `print_fullsolution()`, which does what it says on the tin.

4.3.2 The knight's tour problem

Figure 4.2 shows how knights move in the game of chess. Given a chess board of m ranks (rows) and n files (columns) that is otherwise empty, is it possible for a knight placed on some square to visit all squares without visiting any twice? If we require that the knight also finish on the square where it started, this question is known as the closed knight's tour problem; if the knight may finish at any square, we count its starting square as visited and refer to the problem as the open knight's tour problem.

Both the close and open knight's tour problems are solved problems, and efficient graph-based algorithms exist for finding tours, if they exist. But I find it nonetheless instructive to formulate them as planning problems to be solved using SAT solvers, partly *because* they are solved problems. For instance, it is known that no closed knight's tours exist on a $n \times n$ board when n is odd. Conversely, closed knight's tours exist on any $n \times n$ board if $n \geq 6$ is even (Schwenk, 1991). Hence, we can check any claims about the (un)satisfiability of a closed knight tour's problem against theory. The knight's tour problems are also useful for our present purposes because we can easily formulate problems that differ in size simply by varying m and n.

In contrast to the blocks world example, I did not construct the CNF formula for the knight's tour problem by converting a classical planning problem. Instead, I expressed directly expressed the problem in CNF form. There are different ways to do so, and I will discuss some differences between them at the end of this chapter. I opted for the following encoding. Given board

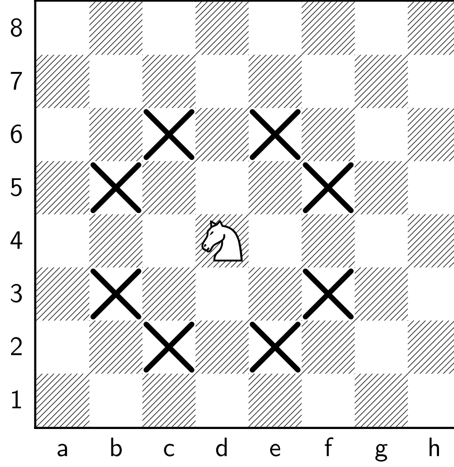

Figure 4.2: In chess, knights move in an L-shaped pattern.

dimensions m and n as well as a desired number of moves t, we define the following propositional variables. First, to indicate that the knight is on a given field at a given time, we use the propositional variables $on(x,y,i)$, where $1 \leq x \leq n, 1 \leq y \leq m, 1 \leq i \leq t+1$; deviating from standard chess notation, we use numbers to refer to files rather than letters. Second, to indicate that the knight moves to a certain square at a given time, we use $to(x,y,i)$, where $1 \leq x \leq n, 1 \leq y \leq m, 1 \leq i \leq t$.

We specify that exactly one move has to be made at each time $i = 1, \dots, t$ by including the clause

$$\{to(x,y,i) : x \in \{1,\dots,n\}, y \in \{1,\dots,m\}\}$$

for each $i = 1, \dots, t$ as well as the clause

$$\neg to(x,y,i) \vee \neg to(\widetilde{x},\widetilde{y},i)$$

for each $i = 1, \dots, t$ and each $(x,y),(\widetilde{x},\widetilde{y}) \in \{1,\dots,n\} \times \{1,\dots,m\}$ with $(x,y) \neq (\widetilde{x},\widetilde{y})$. We similarly specify that the knight cannot be on more than one square at a time by including the clause

$$\neg on(x,y,i) \vee \neg on(\widetilde{x},\widetilde{y},i)$$

for each $i = 1, \dots, t+1$ and each $(x,y),(\widetilde{x},\widetilde{y}) \in \{1,\dots,n\} \times \{1,\dots,m\}$ with $(x,y) \neq (\widetilde{x},\widetilde{y})$.

The knight moves in an L-shaped pattern, which we can encode as follows:

$$\{\neg to(x,y,i)\} \cup \{on(\widetilde{x},\widetilde{y},i) : 1 \leq \widetilde{x} \leq n, 1 \leq \widetilde{y} \leq m, |\widetilde{x}-x| \geq 1, |\widetilde{y}-y| \geq 1, |\widetilde{x}-x| + |\widetilde{y}-y| = 3\},$$

for each $i = 1, \dots, t$ and each $(x,y) \in \{1,\dots,n\} \times \{1,\dots,m\}$. This corresponds to the move's precondition. Of course, after the move, the knight is on the destination square:

$$\neg to(x,y,i) \vee on(x,y,i+1),$$

for each $i = 1, \ldots, t$ and each $(x, y) \in \{1, \ldots, n\} \times \{1, \ldots, m\}$. Note that we do not have to specify that the knight is not on its former square any more: we already specified that the knight cannot be on more than one square at a time.

Given a starting position (x_0, y_0), we spell out the initial state using the clause $on(x_0, y_0, 1)$; this immediately implies $\neg on(x, y, 1)$ for all $(x, y) \neq (x_0, y_0)$. To encode that each square must have been visited at the start of the $t + 1$-th turn, we use the clauses

$$on(x, y, 1) \vee \cdots \vee on(x, y, t + 1)$$

for all squares (x, y). If we want to obtain a closed knight's tour, we add the clause

$$on(x_0, y_0, t + 1)$$

and we pick $t = m \cdot n$. Since the knight needs to visit $m \cdot n$ squares and land on its starting square at the end of the t-th move, this guarantees that the knight will not visit any square other than the starting square twice. For an open knight's tour, we pick $t = m \cdot n - 1$, since the starting square counts as visited.

Incidentally, we could add further restrictions to the problem. If, for instance, we were interested in finding a knight's tour where the knight is on *E*7 (i.e., on $(5, 7)$) at the start of the 23rd turn, we just add the one-literal clause $on(5, 7, 23)$ to the formula. By the same token, if we already had a knight's tour, but we wanted to know if a different one existed, we could just add the clause

$$\neg to(x_1, y_1, 1) \vee \cdots \vee \neg to(x_t, y_t, t),$$

where $to(x_1, y_1, 1), \ldots, to(x_t, y_t, t)$ represents the known knight's tour.

The Julia package features the function `knights_problem()` that constructs the CNF formula in the manner outlined above. In order to generate the formula corresponding to a closed knight's tour on a 5-by-6 board where the knight starts on C2 (i.e., on $(3, 2)$), we would use

```
julia> cnf = knights_problem(5, 6, 5*6, [3, 2]);
```

We can save this formula in the DIMACS format and feed it to a SAT solver just like in the previous example. Since a closed knight's tour exists in this setting (Schwenk, 1991), a model for the CNF exists and Glucose duly finds one in about five CPU seconds on my machine:[1]

```
julia> to_DIMACS(cnf, "knight5by6.cnf", "knight5by6.code");
(base) jan@jan-xps:~$ glucose -no-adapt knight5by6.cnf knight5by6.soln
[output not shown]
```

[1] A note on the timings is in order. First, Glucose's performance, like that of several other SAT solvers, depends, among other things, on the order of the clauses (also see Biere & Heule, 2019; Björk, 2009). The effect can be quite dramatic at times. To see this, we can export `Set(cnf)` to the DIMACS format instead of `cnf`. This puts the clauses in the `cnf` object through a hash function, effectively scrambling their order. For the example on the 5-by-6 board, this increases the time it takes Glucose to find a solution to 19 seconds. Second, I noticed that, at least for the knight's tour problems, Glucose was able to find a solution *considerably* more quickly when I switched off one of its default features, namely adapting its solver strategy (see Audemard & Simon, 2016). For comparison, instead of finding a solution in about five seconds without allowing for adaptive solver strategies, it took Glucose over three minutes to find one with its default settings. All in all, the timings reported should be taken with a grain of salt as they depend on factors that were not discussed in any detail in this thesis.

```
julia> print_moves("knight5by6.soln", "knight5by6.code")
30-element Vector{String}:
 "move to 1 1 at time 1"
 "move to 2 3 at time 2"
 "move to 1 5 at time 3"
 "move to 3 4 at time 4"
 "move to 5 5 at time 5"
 "move to 6 3 at time 6"
 "move to 5 1 at time 7"
 "move to 4 3 at time 8"
 "move to 6 2 at time 9"
 "move to 4 1 at time 10"
 "move to 2 2 at time 11"
 "move to 1 4 at time 12"
 "move to 3 5 at time 13"
 "move to 5 4 at time 14"
 "move to 4 2 at time 15"
 "move to 6 1 at time 16"
 "move to 5 3 at time 17"
 "move to 6 5 at time 18"
 "move to 4 4 at time 19"
 "move to 2 5 at time 20"
 "move to 1 3 at time 21"
 "move to 2 1 at time 22"
 "move to 3 3 at time 23"
 "move to 1 2 at time 24"
 "move to 3 1 at time 25"
 "move to 5 2 at time 26"
 "move to 6 4 at time 27"
 "move to 4 5 at time 28"
 "move to 2 4 at time 29"
 "move to 3 2 at time 30"
```

For an 8-by-8 board, the CNF formula contains 8,256 propositional variables and 268,386 clauses; on my machine, the solver finds a closed knight's tour in under six minutes. For a 9-by-9 board, the formula contains 13,203 variables and 541,406 clauses, and the solver correctly concludes after about 10 seconds that no closed knight's tours exist.

To generate the CNF formula for an open knight's tour, we can use

```
julia> cnf = knights_problem(8, 9, 8*9 - 1, [5, 6]; open_tour = true);
```

4.4 On the effects of encoding details

In the previous section, I mentioned in passing that the same planning problem can be represented by different CNF encodings. Indeed, Kautz & Selman (1992, 1996) pointed out that propositional

variables that take three or more arguments can be replaced by combinations of propositional variables that take no more than two arguments. For instance, in the blocks world problem, the variable $move(A, B, table, 4)$ can be replaced by the variables $object(A, 4), from(B, 4), to(table, 4)$. This alternative encoding can reduce the number of propositional variables and clauses the CNF formula contains, sometimes specularly so (see Kautz & Selman, 1992, Table 1). As a result, the CNF formula can become easier to solve.

The knight's tour problem offers a case in point of how recodings can result in simpler formulas. Originally, I had specified the moves not as $to(x, y, i)$ but as $move(x_0, y_0, x_1, y_1, i)$, that is, the moves had encoded both the source and the destination squares as opposed to just the destination squares. Moreover, I had explicitly included variables that encoded whether a square had been visited on the i-th turn, i.e., $visited(x, y, i)$. The corresponding CNF formula for an 8-by-8 board, which can be generated using the `knights_problem_old` function, contained 29,824 propositional variables and just shy of 3.8 million clauses, and it took Glucose about eleven minutes to find a knight's tour using this encoding – compared to under six minutes when using the current encoding. I refer the interested reader to the source code for `knights_problem_old` to see how the CNF formula is constructed.

That said, I did not have any success adopting Kautz and Selman's strategy of replacing variables with three or more arguments by combinations of variables with two arguments: While it is possible to recode variables of the form $on(x, y, i)$ as $on_file(x, i) \land on_rank(y, i)$ (and similarly for $to(x, y, i)$), the goal condition then consists of the clauses

$$(on_file(x, 1) \land on_rank(y, 1)) \lor \cdots \lor (on_file(x, t + 1) \land on_rank(y, t + 1))$$

for each square (x, y). Converting one such clause to an equivalent conjunction of disjunctions would result in a subformula with 2^{t+1} clauses, whereas converting it to an equisatisfiable CNF formula using the Tseitin transformation would entail introducing a new variable for each subclause $on_file(x, i) \land on_rank(y, i)$, which boils down to reintroducing the $on(x, y, i)$ variables into the formula.

Bibliography

Ansótegui, Carlos, Maria Luisa Bonet, Jesús Giráldez-Cru, Jordi Levy & Laurent Simon. 2019. Community structure in industrial SAT instances. *Journal of Artificial Intelligence Research* 66. 443–472. doi:10.1613/jair.1.11741.

Audemard, Gilles & Laurent Simon. 2009. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI'09: Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 399–404. doi:10.5555/1661445.1661509.

Audemard, Gilles & Laurent Simon. 2016. Glucose and Syrup in the SAT'16. In *Proceedings of the SAT Competition 2016: Solver and benchmark descriptions*, 40–41.

Bayardo, Roberto J., Jr. & Robert C. Schrag. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, 203–208.

Biere, Armin & Andreas Fröhlich. 2015. Evaluating CDCL variable scoring schemes. In *Theory and Applications of Satisfiability Testing – SAT 2015*, 405–422. doi:10.1007/978-3-319-24318-4_29.

Biere, Armin & Andreas Fröhlich. 2019. Evaluating CDCL restart schemes. In *Proceedings of Pragmatics of SAT 2015 and 2018*, 1–17. doi:10.29007/89dw.

Biere, Armin, Marijn Heule, Hans van Maaren & Toby Walsh (eds.). 2021. *Handbook of satisfiability*. IOS Press.

Biere, Armin & Marijn J. H. Heule. 2019. The effect of scrambling CNFs. In *Proceedings of Pragmatics of SAT 2015 and 2018*, vol. 59 EPiC Series in Computing, 111–126. doi:10.29007/9dj5.

Björk, Magnus. 2009. Successful SAT encoding techniques. *Journal of Satisfiability, Boolean Modeling, and Computation* 7(4). 189–201. doi:10.3233/SAT190085.

Cook, Stephen A. 1971. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, 151–158. doi:10.1145/800157.805047.

Darwiche, Adnan & Knot Pipatsrisawat. 2021. Complete algorithms. In Biere et al. (2021) 101–132. doi:10.3233/FAIA200986.

Davis, Martin, George Logemann & Donald Loveland. 1962. A machine program for theorem-proving. *Communications of the ACM* 5(7). 394–397. doi:10.1145/368273.368557.

Davis, Martin & Hilary Putnam. 1960. A computing procedure for quantification theory. *Journal of the ACM* 7(3). 201–215. doi:10.1145/321033.321034.

Fichte, Johannes K., Markus Hecher & Stefan Szeider. 2020. A time leap challenge for SAT-solving. In *Principles and Practice of Constraint Programming (CP 2020)*, 267–285. doi:10.1007/978-3-030-58475-7_16.

Fichte, Johannes K., Daniel Le Berre, Markus Hecher & Stefan Szeider. 2023. The silent (r)evolution of SAT. *Communications of the ACM* 66(6). 64–72. doi:10.1145/3560469.

Ganesh, Vijay & Moshe Y. Vardi. 2021. On the unreasonable effectiveness of SAT solvers. In Tim Roughgarden (ed.), *Beyond the worst-case analysis of algorithms*, 547–566. Cambridge University Press. doi:10.1017/9781108637435.032.

Hamadi, Youssef, Saïd Jabbour & Lakhdar Saïs. 2016. What we can learn from conflicts in propositional satisfiability. *Annals of Operations Research* 240. 13–37. doi:10.1007/s10479-015-2028-9.

Jeroslow, Robert G. & Jinchang Wang. 1990. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence* 1. 167–187. doi:10.1007/BF01531077.

Kautz, Henry & Bart Selman. 1992. Planning as satisfiability. In *ECAI '92: Proceedings of the 10th European Conference on Artificial Intelligence*, 359–363. doi:10.5555/145448.146725.

Kautz, Henry & Bart Selman. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1194–1201. doi:10.5555/1864519.1864564.

Knuth, Donald E. 2016. *Satisfiability*, vol. 4 (The Art of Computer Programming Facsimile 6). Addison-Wesley.

Krüger, Tom, Jan-Hendrik Lorenz & Florian Wörz. 2022. Too much information: Why CDCL solvers need to forget learned clauses. *PLOS ONE* 17(8). e0272967. doi:10.1371/journal.pone.0272967.

Kuiter, Elias, Sebastian Krieter, Chico Sundermann, Thomas Thüm & Gunter Saake. 2022. Tseitin or not Tseitin? The impact of CNF transformations on feature-model analyses. In *ASE '22: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, doi:10.1145/3551349.3556938.

Lengauer, Thomas & Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems* 1(1). 121–141. doi:10.1145/357062.357071.

Li, Cunxiao, Jonathan Chung, Soham Mukherjee, Marc Vinyals, Noah Fleming, Antonina Kolokolova, Alice Mu & Vijay Ganesh. 2021. On the hierarchical community structure of practical Boolean formulas. In *Theory and Applications of Satisfiability Testing – SAT 2021*, 359–376. doi:10.1007/978-3-030-80223-3_25.

Liang, Jia Hui, Vijay Ganesh, Ed Zulkoski, Atulan Zaman & Krzysztof Czarnecki. 2015. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In

Hardware and Software: Verification and Testing – HVC 2015, 225–241. doi:10.1007/978-3-319-26287-1_14.

Liang, Jia Hui, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li & Vijay Ganesh. 2018. Machine learning-based restart policy for CDCL SAT solvers. In *Theory and Applications of Satisfiability Testing – SAT 2018*, 94–110. doi:10.1007/978-3-319-94144-8_6.

Marques-Silva, João. 1999. The impact of branching heuristics in propositional satisfiability algorithms. In *Progress in artificial intelligence. 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Evora, Portugal, September 21-24, 1999, proceedings*, 62–74. doi:10.1007/3-540-48159-1_5.

Marques-Silva, João, Ines Lynce & Sharad Malik. 2021. Conflict-driven clause learning SAT solvers. In Biere et al. (2021) 133–182. doi:10.3233/FAIA200987.

Marques-Silva, João & Sharad Malik. 2018. Propositional SAT solving. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith & Roderick Bloem (eds.), *Handbook of model checking*, 247–275. Springer. doi:10.1007/978-3-319-10575-8_9.

Marques-Silva, João P. & Karem A. Sakallah. 1996. GRASP: A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design, San Jose, CA, USA, 1996*, 220–227. doi:10.1109/ICCAD.1996.569607.

Marques-Silva, João P. & Karem A. Sakallah. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48(5). 506–521. doi:10.1109/12.769433.

Moskewicz, Matthew W., Conor F. Madigan, Ying Zhao, Lintao Zhang & Sharad Malik. 2001. Chaff: Engineering an efficient SAT solver. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, 530–535. doi:10.1145/378239.379017.

Mull, Nathan, Daniel J. Fremont & Sanjit A. Seshia. 2016. On the hardness of SAT with community structure. In *Theory and Applications of Satisfiability testing – SAT 2016*, 141–159. doi:10.1007/978-3-319-40970-2_10.

Newsham, Zack, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard & Laurent Simon. 2014. Impact of community structure on SAT solver performance. In *Theory and Applications of Satisfiability Testing – SAT 2014*, 252–268. doi:10.1007/978-3-319-09284-3_20.

Pipatsrisawat, Knot & Adnan Darwiche. 2007. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing – SAT 2007*, 294–299. doi:10.1007/978-3-540-72788-0.

Rintanen, Jussi. 2012. Planning as satisfiability: Heuristics. *Artificial Intelligence* 193. 45–86. doi:10.1016/j.artint.2012.08.001.

Rintanen, Jussi. 2021. Planning and SAT. In Biere et al. (2021) 765–789. doi:10.3233/FAIA201003.

Sabharwal, Ashish, Horst Samulowitz & Meinolf Sellmann. 2012. Learning back-clauses in SAT. In *Theory and Applications of Satisfiability Testing – SAT 2012*, 498–499. doi:10.1007/978-3-642-31612-8_53.

Schwenk, Allen J. 1991. Which rectangular chessboards have a knight's tour? *Mathematics Magazine* 64(5). 325–332. doi:10.2307/2690649.

Sörensson, Niklas & Armin Biere. 2009. Minimizing learned clauses. In *Theory and Applications of Satisfiability Testing – SAT 2009*, 237–243. doi:10.1007/978-3-642-02777-2_23.

Tseitin, G. S. 1983. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann & Graham Wrightson (eds.), *Automation of reasoning 2*, 466–483. Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-81955-1_28.

Zhang, Lintao, Conor F. Madigan, Matthew H. Moskewicz & Sharad Malik. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, 279–285. doi:10.1109/ICCAD.2001.968634.

Zulkoski, Edward, Ruben Martins, Christoph M. Wintersteiger, Jia Hui Liang, Krzysztof Czarnecki & Vijay Ganesh. 2018. The effect of structural measures and merges on SAT solver performance. In *Principles and practice of constraint programming: 24th International Conference, CP 2018, Lille, France, August 27–31, 2018*, 436–452. doi:10.1007/978-3-319-98334-9_29.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Jan Vanhove
Freiburg i. Üe., 20. Juni 2024