Propositional Logic and Its Applications in Artificial Intelligence

Jussi Rintanen
Department of Computer Science
Aalto University
Helsinki, Finland

April 5, 2017

Contents

Table of Contents i

1 Introduction **1**

2 Definition **2**
2.1 Formulas 2
2.2 Equivalences 3
2.3 Truth-tables 3
2.4 Logical Consequence, Satisfiability, Validity 5
2.5 Normal Forms 6

3 Automated Reasoning in the Propositional Logic **9**
3.1 Truth-Tables 9
3.2 Tree-Search Algorithms 10
3.2.1 Unit Resolution 11
3.2.2 Subsumption 12
3.2.3 Unit Propagation 12
3.2.4 The Davis-Putnam Procedure 12

4 Applications **14**
4.1 State-Space Search 14
4.1.1 Representation of Sets 14
4.1.2 Set Operations as Logical Operations 15
4.1.3 Relations as Formulas 15
4.1.4 Mapping from Actions to Formulas 17
4.1.5 Finding Action Sequences through Satisfiability 20
4.1.6 Relation Operations in Logic 20
4.1.7 Symbolic State-Space Search 22

Index **24**

Chapter 1

Introduction

The classical propositional logic is the most basic and most widely used logic. It is a notation for Boolean functions, together with several powerful proof and reasoning methods.

The use of the propositional logic has dramatically increased since the development of powerful search algorithms and implementation methods since the later 1990ies. Today the logic enjoys extensive use in several areas of computer science, especially in Computer-Aided Verification and Artificial Intelligence. Its uses in AI include planning, problem-solving, intelligent control, and diagnosis.

The reason why logics are used is their ability to precisely express data and information, in particular when the information is *partial* or *incomplete*, and some of the *implicit* consequences of the information must be *inferred* to make them *explicit*.

The propositional logic, as the first known NP-complete problem [Coo71], is used for representing many types of co-NP-complete and NP-complete *combinatorial* search problems. Such problems are prevalent in artificial intelligence as a part of *decision-making*, *problem-solving*, *planning*, and other hard problems.

For many applications equally or even more natural choices would be various more expressive logics, including the *predicate logic* or various *modal logics*. These logics, however, lack the kind of efficient and scalable algorithms that are available for the classical propositional logic. The existence of high performance algorithms for reasoning with propositional logic is the main reason for its wide use in computer science.

Chapter 2

Definition

2.1 Formulas

Propositional logic is about Boolean functions, which are mappings from *truth-values* 0 and 1 (*false* and *true*) to truth-values. Arbitrary Boolean functions can be represented as *formulas* formed with *connectives* such as $\wedge$, $\vee$ and $\neg$, and it can be shown that any Boolean function can be represented by such a formula.

Boolean functions and formulas have important applications in several areas of Computer Science.

- Digital electronics and the construction of digital computation devices such as microprocessors, is based on Boolean functions.
- The theory of computation and complexity uses Boolean functions for representing abstract models of computation (for example Boolean circuits) and investigating their properties (for example the theory of Computational Complexity, where some of the fundamental results like NP-completeness [Coo71, GJ79] were first established with Boolean functions).
- Parts of Artificial Intelligence use Boolean functions and formulas for representing models of the world and the reasoning processes of intelligent systems.

The most primitive Boolean functions are represented by the connectives $\vee$, $\wedge$, $\neg$, $\rightarrow$ and $\leftrightarrow$ as follows.

α	β	$\alpha \vee \beta$
0	0	0
0	1	1
1	0	1
1	1	1

α	β	$\alpha \wedge \beta$
0	0	0
0	1	0
1	0	0
1	1	1

α	$\neg\alpha$
0	1
1	0

α	β	$\alpha \rightarrow \beta$
0	0	1
0	1	1
1	0	0
1	1	1

α	β	$\alpha \leftrightarrow \beta$
0	0	1
0	1	0
1	0	0
1	1	1

The connectives are used for forming complex Boolean functions from primitive ones.

Often only some of the connectives, typically $\neg$ and one or both of $\wedge$ and $\vee$ are viewed as the *primitive connectives*, and other connectives are defined in terms of the primitive connectives. The *implication* $\rightarrow$ and *equivalence* $\leftrightarrow$ connectives are often viewed as non-primitive: $\alpha \rightarrow \beta$ can be defined as $\neg\alpha \vee \beta$, and $\alpha \leftrightarrow \beta$ can be defined as $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$.

There is a close connection between the Boolean connectives $\vee$, $\wedge$ and $\neg$ and the operations $\cup$, $\cap$ and complementation in Set Theory. Observe the similarity between the truth-tables for the three connectives and the analogous tables for the set-theoretic operations.

α	β	$\alpha \cup \beta$
$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\{1\}$	$\{1\}$
$\{1\}$	$\emptyset$	$\{1\}$
$\{1\}$	$\{1\}$	$\{1\}$

α	β	$\alpha \cap \beta$
$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\{1\}$	$\emptyset$
$\{1\}$	$\emptyset$	$\emptyset$
$\{1\}$	$\{1\}$	$\{1\}$

α	$\overline{\alpha}$
$\emptyset$	$\{1\}$
$\{1\}$	$\emptyset$

Formed analogously to expressions in mathematics (parentheses, $+$, $\times$ etc.)

Definition 2.1 (Propositional Formulas) *Formulas a recursively defined as follows.*

- *Atomic propositions $x \in X$ are formulas.*
- *Symbols $\top$ and $\bot$ are formulas.*
- *If α and β are formulas, then so are*

  1. $\neg\alpha$
  2. $(\alpha \wedge \beta)$
  3. $(\alpha \vee \beta)$
  4. $(\alpha \rightarrow \beta)$
  5. $(\alpha \leftrightarrow \beta)$

Nothing else is a formula.

Parentheses (and) do not always need to be written if precedences between the connectives are observed. The highest precedence is with $\neg$, followed by $\wedge$ and $\vee$, then $\rightarrow$, and finally $\leftrightarrow$.

Definition 2.2 (Valuation of atomic propositions) *A* valuation $v : X \rightarrow \{0, 1\}$ *is a mapping from atomic propositions* $X = \{x_1, \ldots, x_n\}$ *to* truth-values *0 and 1.*

Valuations can be extended to *formulas* $\phi \in \mathcal{L}$, i.e. $v : \mathcal{L} \rightarrow \{0, 1\}$.

Definition 2.3 (Valuation of propositional formulas) *A given valuation $v : X \rightarrow \{0, 1\}$ of atomic propositions can be extended to a valuation of arbitrary propositional formulas over X.*

$v(\neg\alpha) = 1$ *iff* $v(\alpha) = 0$
$v(\top) = 1$
$v(\bot) = 0$
$v(\alpha \wedge \beta) = 1$ *iff* $v(\alpha) = 1$ *and* $v(\beta) = 1$
$v(\alpha \vee \beta) = 1$ *iff* $v(\alpha) = 1$ *or* $v(\beta) = 1$
$v(\alpha \rightarrow \beta) = 1$ *iff* $v(\alpha) = 0$ *or* $v(\beta) = 1$
$v(\alpha \leftrightarrow \beta) = 1$ *iff* $v(\alpha) = v(\beta)$

Example 2.4 Let $v(a) = 1, v(b) = 1, v(c) = 1, v(d) = 1$

$$\begin{aligned} v(a \vee b \vee c) &= 1 \\ v(\neg a \rightarrow b) &= 1 \\ v(a \rightarrow \neg b) &= 0 \end{aligned}$$

■

2.2 Equivalences

Table 2.1 lists equivalences that hold in the propositional logic. Replacing one side of any of these equivalences by the other - in any formula - does not change the Boolean function represented by the formula.

These equivalences have several applications, including translating formulas into *normal forms* (Section 2.5), and simplifying formulas. For example, all occurrences $\top$ and $\bot$ of the constant symbols (except one, if the whole formula reduces to $\top$ or $\bot$) can be eliminated with the equivalences containing these symbols.

2.3 Truth-tables

All valuations relevant to a formula are often tabulated as *truth-tables* so that each row corresponds to a valuation. Truth-tables are used for analyzing the most basic properties of formulas.

Valuations for formulas containing exactly one connective are as follows.

double negation	$\neg\neg\alpha$	$\equiv$	α
associativity $\vee$	$\alpha \vee (\beta \vee \gamma)$	$\equiv$	$(\alpha \vee \beta) \vee \gamma$
associativity $\wedge$	$\alpha \wedge (\beta \wedge \gamma)$	$\equiv$	$(\alpha \wedge \beta) \wedge \gamma$
commutativity $\vee$	$\alpha \vee \beta$	$\equiv$	$\beta \vee \alpha$
commutativity $\wedge$	$\alpha \wedge \beta$	$\equiv$	$\beta \wedge \alpha$
distributivity $\wedge \vee$	$\alpha \wedge (\beta \vee \gamma)$	$\equiv$	$(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$
distributivity $\vee \wedge$	$\alpha \vee (\beta \wedge \gamma)$	$\equiv$	$(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
idempotence $\vee$	$\alpha \vee \alpha$	$\equiv$	α
idempotence $\wedge$	$\alpha \wedge \alpha$	$\equiv$	α
absorption 1	$\alpha \wedge (\alpha \vee \beta)$	$\equiv$	α
absorption 2	$\alpha \vee (\alpha \wedge \beta)$	$\equiv$	α
De Morgan's law 1	$\neg(\alpha \vee \beta)$	$\equiv$	$(\neg\alpha) \wedge (\neg\beta)$
De Morgan's law 2	$\neg(\alpha \wedge \beta)$	$\equiv$	$(\neg\alpha) \vee (\neg\beta)$
contraposition	$\alpha \to \beta$	$\equiv$	$\neg\beta \to \neg\alpha$
negation $\top$	$\neg\top$	$\equiv$	$\bot$
negation $\bot$	$\neg\bot$	$\equiv$	$\top$
constant $\bot$	$\alpha \wedge \neg\alpha$	$\equiv$	$\bot$
constant $\top$	$\alpha \vee \neg\alpha$	$\equiv$	$\top$
elimination $\top \vee$	$\top \vee \alpha$	$\equiv$	$\top$
elimination $\top \wedge$	$\top \wedge \alpha$	$\equiv$	α
elimination $\bot \vee$	$\bot \vee \alpha$	$\equiv$	α
elimination $\bot \wedge$	$\bot \wedge \alpha$	$\equiv$	$\bot$
elimination $\bot \to$	$\bot \to \alpha$	$\equiv$	$\top$
elimination $\bot \to$	$\alpha \to \bot$	$\equiv$	$\neg\alpha$
elimination $\top \to$	$\top \to \alpha$	$\equiv$	α
elimination $\top \to$	$\alpha \to \top$	$\equiv$	$\top$
commutativity $\leftrightarrow$	$\alpha \leftrightarrow \beta$	$\equiv$	$\beta \leftrightarrow \alpha$
elimination $\top \leftrightarrow$	$\top \leftrightarrow \alpha$	$\equiv$	α
elimination $\bot \leftrightarrow$	$\bot \leftrightarrow \alpha$	$\equiv$	$\neg\alpha$

Table 2.1: Propositional Equivalences

$$
\begin{array}{rcl}
\alpha \wedge \beta & \models & \alpha \\
\alpha & \models & \alpha \vee \beta \\
\alpha & \models & \beta \rightarrow \alpha \\
\neg\alpha & \models & \alpha \rightarrow \beta \\
\alpha \wedge (\alpha \rightarrow \beta) & \models & \beta \\
\neg\beta \wedge (\alpha \rightarrow \beta) & \models & \neg\alpha
\end{array}
$$

Table 2.2: Useful Logical Consequences

α	$\neg\alpha$
0	1
1	0

α	β	$\alpha \wedge \beta$
0	0	0
0	1	0
1	0	0
1	1	1

α	β	$\alpha \vee \beta$
0	0	0
0	1	1
1	0	1
1	1	1

α	β	$\alpha \rightarrow \beta$
0	0	1
0	1	1
1	0	0
1	1	1

α	β	$\alpha \leftrightarrow \beta$
0	0	1
0	1	0
1	0	0
1	1	1

1. Columns for all n atomic propositions in the formula
2. Columns for all *subformulas* of the formula
3. Write 2^n rows corresponding to the valuations
4. Fill in truth-values in the remaining cells

Example 2.5 The truth-table for $\neg B \wedge (A \rightarrow B)$ is

A	B	$\neg B$	$(A \rightarrow B)$	$(\neg B \wedge (A \rightarrow B))$
0	0	1	1	1
0	1	0	1	0
1	0	1	0	0
1	1	0	1	0

2.4 Logical Consequence, Satisfiability, Validity

Definition 2.6 (Logical Consequence) *A formula ϕ is a* logical consequence *of $\Sigma = \{\phi_1, \ldots, \phi_n\}$, denoted by $\Sigma \models \phi$, if and only if for all valuations v, if $v \models \Sigma$ then $v \models \phi$.*

Logical consequences are what can be inferred with certainty.

Example 2.7 From "if it is a weekday today then most people are working today", and "it is a weekday today" it follows that "most people are working today".
$\{w \rightarrow p, w\} \models p$

Example 2.8 $\{a, a \rightarrow b, b \rightarrow c\} \models c$

Example 2.9 $\{m \vee t, m \rightarrow w, t \rightarrow w\} \models w$

Definition 2.10 (Validity) *A formula ϕ is* valid *if and only if $v(\phi) = 1$ for all valuations v.*

Example 2.11 $x \vee \neg x$ is valid.
$x \rightarrow (x \vee y)$ is valid.
$x \vee y$ is not valid.

Definition 2.12 (Satisfiability) *A formula ϕ is* satisfiable *if and only if there is at least one valuation v such that $v(\phi) = 1$.*

A set $\{\phi_1, \dots, \phi_n\}$ is satisfiable *if there is at least one valuation v such that $v(\phi_i) = 1$ for all $i \in \{1, \dots, n\}$.*

Example 2.13 The following formulas are satisfiable: x, $x \wedge y$, and $x \vee \neg x$. ■

Example 2.14 The following formulas are not satisfiable: $x \wedge \neg x$, $\perp$, $a \wedge (a \to b) \wedge \neg b$. ■

A satisfiable formula is also said to be *consistent*. A formula that is not satisfiable is *unsatisfiable*, *inconsistent*, or *contradictory*.

Satisfiability means *logically possible*: it is possible that the formula is true (if the truth-values of the atomic propositions in the formula are chosen right.)

Interestingly, there are close connections between satisfiability, validity, and logical consequence. These connections allow *reducing* questions concerning these concepts to each other, which allows choosing one of these concepts as the most basic one (for example implemented in algorithms for reasoning in the propositional logic), and reducing the other concepts to the basic one.

Theorem 2.15 (Validity vs. Logical Consequence 1) *ϕ is valid if and only if $\emptyset \models \phi$*

Theorem 2.16 (Validity vs. Logical Consequence 2) *$\{\phi_1, \dots, \phi_n\} \models \phi$ if and only if $(\phi_1 \wedge \cdots \wedge \phi_n) \to \phi$ is valid.*

Theorem 2.17 (Satisfiability vs. Validity) *ϕ is satisfiable if and only if $\neg\phi$ is* not valid.

Theorem 2.18 (Logical Consequence vs. Satisfiability) *$\Sigma \models \phi$ if and only if $\Sigma \cup \{\neg\phi\}$ is not satisfiable.*

In practice, algorithms for the propositional logic implement *satisfiability*. Other reasoning tasks are reduced to satisfiability testing.

Example 2.19 $\{m \vee t, m \to w, t \to w\} \models w$ if and only if $(m \vee t) \wedge (m \to w) \wedge (t \to w) \wedge \neg w$ is not satisfiable. ■

2.5 Normal Forms

Arbitrary propositional formulas can be translated to syntactically restricted forms. This can serve two main purposes. First, it may be more straightforward to define algorithms and inference methods when the formulas are in certain simple forms. The *resolution rule* (Section 3.2.1) is an example of this. Second, the process of translating a formula into a normal form does much of the work in solving important computational problems related to propositional formulas. For example, an answer to the question of whether a formula is valid is obtained as a by-product of translating the formula into certain normal forms.

In this course we focus on the best known normal form which is *the conjunctive normal form.*

Definition 2.20 (Literals) *If x is an atomic proposition, then x and $\neg x$ are* literals.

Definition 2.21 (Clauses) *If $l_1, \dots, l_n$ are literals, then $l_1 \vee \cdots \vee l_n$ is a* clause.

Definition 2.22 (Terms) *If $l_1, \dots, l_n$ are literals, then $l_1 \wedge \cdots \wedge l_n$ is a* term.

Definition 2.23 (Conjunctive Normal Form) *If $\phi_1, \dots, \phi_m$ are clauses, then the formula $\phi_1 \wedge \cdots \wedge \phi_m$ is in* conjunctive normal form.

Example 2.24 The following formulas are in conjunctive normal form.

$$\neg x$$
$$\neg x_1 \vee \neg x_2$$
$$x_3 \wedge x_4$$
$$(\neg x_1 \vee \neg x_2) \wedge \neg x_3 \wedge (x_4 \vee \neg x_5)$$

■

$$\alpha \leftrightarrow \beta \rightsquigarrow (\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha) \tag{2.1}$$
$$\alpha \rightarrow \beta \rightsquigarrow \neg\alpha \vee \beta \tag{2.2}$$
$$\neg(\alpha \vee \beta) \rightsquigarrow \neg\alpha \wedge \neg\beta \tag{2.3}$$
$$\neg(\alpha \wedge \beta) \rightsquigarrow \neg\alpha \vee \neg\beta \tag{2.4}$$
$$\neg\neg\alpha \rightsquigarrow \alpha \tag{2.5}$$
$$\alpha \vee (\beta \wedge \gamma) \rightsquigarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \tag{2.6}$$
$$(\alpha \wedge \beta) \vee \gamma \rightsquigarrow (\alpha \vee \gamma) \wedge (\beta \vee \gamma) \tag{2.7}$$

Table 2.3: Rewriting Rules for Translating Formulas to Conjunctive Normal Form

Table 2.3 lists rules that transform any formula into CNF. These rules are instances of the equivalence listed in Table 2.1

Algorithm 2.25 *Any formula can be translated into conjunctive normal form as follows.*

1. *Eliminate connective* $\leftrightarrow$ *(Rule 2.1).*
2. *Eliminate connective* $\rightarrow$ *(Rule 2.2).*
3. *Move* $\neg$ *inside* $\vee$ *and* $\wedge$ *(Rules 2.3 and 2.4).*
4. *Move* $\wedge$ *outside* $\vee$ *(Rules 2.6 and 2.7).*

Notice that the last step of the transformation multiplies the number of copies of subformulas α and γ. For some classes of formulas this transformation therefore leads to exponentially big normal forms.

Formulas in CNF are often represented as sets of clauses, with clauses represented as sets of literals. The placement of connectives $\vee$ and $\wedge$ can be left implicit because of the simple structure of CNF. The CNF $(A \vee B \vee C) \wedge (\neg A \vee D) \wedge E$ is represented as $\{A \vee B \vee C,\ \neg A \vee D,\ E\}$.

Example 2.26 Translate $A \vee B \rightarrow (B \leftrightarrow C)$ into CNF.

$\rightsquigarrow$ $A \vee B \rightarrow (B \rightarrow C) \wedge (C \rightarrow B)$
$\rightsquigarrow$ $\neg(A \vee B) \vee ((\neg B \vee C) \wedge (\neg C \vee B))$
$\rightsquigarrow$ $(\neg A \wedge \neg B) \vee ((\neg B \vee C) \wedge (\neg C \vee B))$
$\rightsquigarrow$ $(\neg A \vee ((\neg B \vee C) \wedge (\neg C \vee B))) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B)))$
$\rightsquigarrow$ $(\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B)))$
$\rightsquigarrow$ $(\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B) \wedge (\neg B \vee \neg B \vee C) \wedge (\neg B \vee \neg C \vee B)$

■

Another often used normal form is the Disjunctive Normal Form (DNF).

Definition 2.27 (Disjunctive Normal Form) *If* $\phi_1, \ldots, \phi_m$ *are terms, then the formula* $\phi_1 \vee \cdots \vee \phi_m$ *is in* disjunctive normal form.

Disjunctive normal forms are found with a procedure similar to that for CNF. The only difference is that a different set of distributivity rules is applied, for moving $\wedge$ inside $\vee$ instead of the other way round.

A less restrictive normal form that includes both CNF and DNF is the Negation Normal Form (NNF). It is useful in many types of automated processing of formulas because of its simplicity and the fact that translation into NNF only minimally affects the size of a formula, unlike the translations into DNF and CNF which may increase the size exponentially.

Definition 2.28 (Negation Normal Form) *Let* X *be a set of atomic propositions.*

1. $\perp$ *and* $\top$ *are in negation normal form.*
2. x *and* $\neg x$ *for any* $x \in X$ *are in negation normal form.*

3. *$\phi_1 \wedge \phi_2$ is in negation normal form if ϕ_1 and ϕ_2 both are.*

4. *$\phi_1 \vee \phi_2$ is in negation normal form if ϕ_1 and ϕ_2 both are.*

No other formula is in negation normal form.

After performing the first three steps in the translation into CNF the formula is in NNF. Notice that only the application of the distributivity equivalences increases the number of atomic propositions in the CNF translation. The translation into NNF does not affect the number of occurrences of atomic propositions or the connectives $\vee$ and $\wedge$, and the size of the resulting formula can be bigger only because of the higher number of occurrences of negation symbols $\neg$. Hence the NNF is never more than twice as big as the original formula.

Chapter 3

Automated Reasoning in the Propositional Logic

In this chapter we discuss the main methods for automating inference in the propositional logic.

The simplest method is based on truth-tables (Section 3.1), which enumerate all valuations of the relevant atomic propositions, and questions about satisfiability and logical consequence can be answered by evaluating the truth-values of the relevant formulas for every valuation.

Truth-tables are easy to implement as a program, but are impractical when the number of atomic propositions is higher than 20 or 30. The algorithms used in practice (Section 3.2) are based on searching a tree in which each path starting from the root node represents a (partial) valuation of the atomic propositions. The size of this tree is in practice orders of magnitudes smaller than corresponding truth-tables, and for finding one satisfying valuation not the whole tree needs to be traversed. Finally, only one path of the tree, from the root node to the current leaf node, needs to be kept in the memory at a time, which reduces the memory requirements substantially. This enables the use of these algorithms for formulas of the size encountered in solving important problems in AI and computer science in general, with up to hundreds of thousands of atomic propositions and millions of clauses.

3.1 Truth-Tables

The most basic method for testing satisfiability of a formula ϕ is to construct the truth-table for ϕ, representing all valuations of atomic propositions occurring in ϕ, and then check whether the column for ϕ contains at least one 1: ϕ is satisfiable if and only if the column for ϕ contains at least one 1. Obviously, this is because all possible valuations are represented in the truth-table, and a formula is satisfiable if and only if it is true in at least one valuation.

Example 3.1 The truth-table for $\neg B \wedge (A \rightarrow B)$:

A	B	$\neg B$	$(A \rightarrow B)$	$(\neg B \wedge (A \rightarrow B))$
0	0	1	1	**1**
0	1	0	1	0
1	0	1	0	0
1	1	0	1	0

$\neg B \wedge (A \rightarrow B)$ is satisfiable because it is true when both A and B are false, corresponding to the first row. ■

The second important problem is testing for logical consequence.

Algorithm 3.2 *To test whether ϕ a logical consequence of Σ do the following.*

1. *Construct the truth-table for ϕ and Σ.*

2. *Mark every row where members of Σ are 1.*

3. *Check that there is 1 in the column for ϕ for* every *marked row.*

This algorithm tests whether ϕ is true in every valuation in which Σ is true.

If there is marked row with ϕ false, then we have *a counterexample* to $\Sigma \models \phi$: Σ is true but ϕ is false.

Example 3.3 Test $\{a \rightarrow b, b \rightarrow c\} \models a \rightarrow c$:

a	b	c	$a \rightarrow b$	$b \rightarrow c$	$a \rightarrow c$
0	0	0	**1**	**1**	1
0	0	1	**1**	**1**	1
0	1	0	1	0	1
0	1	1	**1**	**1**	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	**1**	**1**	1

$\{a \rightarrow b, b \rightarrow c\}$ true on 4 rows, and we need to confirm that $a \rightarrow c$ is true on those rows. ■

Truth-tables are a practical method to be used by hand only up to a couple of atomic propositions. With 6 atomic propositions there are 64 rows, which barely fits on a sheet of paper. With computer it is possible to use the truth-table method for up to 20 or 25 variables. After that memory consumption and computation times will be impractically high. In many applications the number of atomic propositions is in the hundreds, thousands, or even hundreds of thousands, and completely different types of algorithms are necessary.

The exponentiality in the truth-table is inherent in all algorithms for testing satisfiability or logical consequence, similarly to many of challenging problems in AI For testing satisfiability, and many other problems, this exponentiality is an indication of NP-completeness [Coo71, GJ79]. The exponentiality is known as the *combinatorial explosion*, and is caused by the exponential number 2^n of combinations of values of n atomic propositions.NP-completeness of the satisfiability problem suggests that all algorithms for the problem necessarily need to do an exponential amount of computation, in the worst case, and the problem is *inherently hard* in this sense. Work on practically useful algorithms for the satisfiability problem try to avoid the exponential worst-case by using methods for pruning the search tree, and by using heuristics for choosing the traversal order and a tree structure that minimizes the time it takes to find a satisfying valuation or to determine that none exist.

3.2 Tree-Search Algorithms

More practical algorithms for solving the satisfiability problem do not explicitly go through all possible valuations of the atomic propositions, and do not store all of them in a big table that has to be kept in the memory.

The simplest algorithm for testing satisfiability that does not construct a truth-table does a depth-first traversal of a binary tree that represents all valuations. This is illustrated in Figure 3.1 for atomic propositions $X = \{a, b, c\}$.

During the traversal, this algorithm only needs to store in the memory the (partial) valuation on the path from the root node to the current node, and hence its memory consumption is linear in the number of atomic propositions.

The runtime of the algorithm, however, is still exponential in the number of atomic propositions for all unsatisfiable formulas, because the size of the tree. For satisfiable formulas the whole tree does not need to be traversed, but since the first satisfying valuation may encountered late in the traversal, in practice this algorithm is also in this usually exponential.

The improvements to this basic tree-search algorithm are based on the observation that many of the (partial) valuations in the tree make the formula false, and this is easy to detect during the traversal. Hence large parts of the search tree can be pruned.

In the following, we assume that the formula has been translated into the conjunctive normal form (Section 2.5). Practically all leading algorithms assume the input to be in CNF.

The fundamental observation in pruning the search tree is the following. Let v be a valuation and $l \vee l_1 \vee l_2 \vee \cdots \vee l_n$ a clause such that $v(l_1) = 0, \ldots, v(l_n) = 0$,

Since we are trying to find a valuation that makes our clause set *true*, this clause has to be *true* (similarly to all other clauses), and it can only be *true* if the literal l *must be true*.

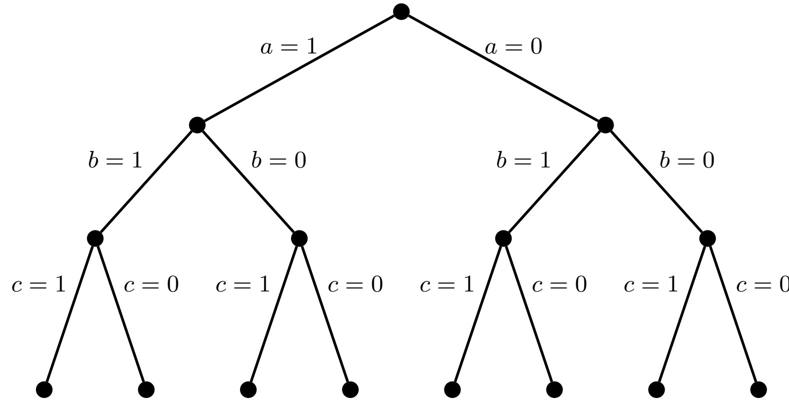

Figure 3.1: All valuations as a binary tree

Hence if the partial valuation corresponding to the current node in the search tree makes, in some clause, all literals *false* except one (call it l) that still does not have a truth-value, then l has to be made true. If l is an atomic propositions x, then the current valuation has to assign $v(x) = 1$, and if $l = \neg x$ for some atomic proposition x, then we have to assign $v(x) = 0$. We call these *forced assignments*.

Consider the search tree in Figure 3.1 and the clauses $\neg a \vee b$ and $\neg b \vee c$. The leftmost child node of the root node, which is reached by following the arc $a = 1$, corresponds to the partial valuation $v = \{(a, 1)\}$ that assigns $v(a) = 1$ and leaves the truth-values of the remaining atomic propositions open. Now with clause $\neg a \vee b$ we have the forced assignment $v(b) = 1$. Now, with clause $\neg b \vee c$ also $v(c) = 1$ is a forced assignment.

Hence assigning $a = 1$ forces the assignments of both of the remaining variables. This means that the leftmost subtree of the root node essentially only consists of one path that leads to the leftmost leaf node. The other leaf nodes in the tree, corresponding to valuations $\{a = 1, b = 1, c = 0\}$, $\{a = 1, b = 0, c = 1\}$, and $\{a = 1, b = 0, c = 1\}$, would not be visited, because it is obvious that they falsify at least one of the clauses. Since we are doing the tree search in order to determine the satisfiability of our clause set, the computation can be stopped here. The rightmost subtree of the root node, corresponding to all valuations with $a = 0$, is therefore not visited at all.

3.2.1 Unit Resolution

What we described as forced assignments is often formalized as the inference rule Unit Resolution.

Theorem 3.4 (Unit Resolution) *In*

$$\frac{l \quad \bar{l} \vee l_1 \vee l_2 \vee \cdots \vee l_n}{l_1 \vee l_2 \vee \cdots \vee l_n}$$

the formula below the line is a logical consequence of the formulas above the line.

Here the *complementation* $\bar{l}$ of a literal l is defined by $\overline{x} = \neg x$ and $\overline{\neg x} = x$.

Example 3.5 From $\{A \vee B \vee C, \neg B, \neg C\}$ one can derive A by two applications of the Unit Resolution rule. ■

A special case of the Unit Resolution rule is when both clauses are unit clauses (consisting of one literal only.) Since the $n = 0$ in this case, the formula below the line has 0 disjuncts. We identify a chain-disjunction with 0 literals with the constant false $\bot$, corresponding to the *empty clause*.

$$\frac{l \quad \bar{l}}{\bot}$$

Any clause set with the empty clause is *unsatisfiable*.

The Unit Resolution rule is a special case of the Resolution rule (which we will not be discussing in more detail.)

Theorem 3.6 (Resolution) *In*

$$\frac{l \vee l'_1 \vee \cdots \vee l'_m \quad \bar{l} \vee l_1 \vee \cdots \vee l_n}{l'_1 \vee \cdots \vee l'_m \vee l_1 \vee l_2 \vee \cdots \vee l_n}$$

the formula below the line is a logical consequence of the formulas above the line.

1: **procedure** DPLL(S)
2: S := $UP(S)$;
3: **if** $\bot \in S$ **then return** false;
4: x := any atomic proposition such that $\{x, \neg x\} \cap S = \emptyset$;
5: **if** no such x exists **then return** true;
6: **if** DPLL($S \cup \{x\}$) **then return** true;
7: **return** DPLL($S \cup \{\neg x\}$)

Figure 3.2: The DPLL procedure

3.2.2 Subsumption

If a clause set contains two clauses with literals $\{l_1, \ldots, l_n\}$ and $\{l_1, \ldots, l_n, l_{n+1}, \ldots, l_m\}$, respectively, then the latter clause can be removed, as it is a logical consequence of the former, and hence both clause sets are *true* in exactly the same valuations.

Example 3.7 Applying the Subsumption rule to $\{A \vee B \vee C, A \vee C\}$ yields the equivalent set $\{A \vee C\}$. ■

If the shorter clause is a unit clause, then this is called Unit Subsumption.

3.2.3 Unit Propagation

Performing all unit resolutions exhaustively leads to the Unit Propagation procedure, also known as Boolean Constraint Propagation (BCP).

In practice, when a clause $l_1 \vee \cdots \vee l_n$ with $n > 2$ can be resolved with a unit clause, the shorter clause of length $n - 1 > 1$ is not produced. This would often lead to producing $n - 1$ clauses, of lengths $1, 2, \ldots, n - 1$. Practical implementation of the unit propagation procedure only produce a new clause when the complements of all but 1 of the n literals of the clause have been inferred. This can be formalized as the following rule, with $n \geq 2$.

$$\frac{\overline{l_2}\ \overline{l_3}\ \ldots\ \overline{l_{n-1}}\quad l_1 \vee \cdots \vee l_n}{l_n}$$

We denote by $UP(S)$ the clause set obtained by performing all possible unit propagations to a clause set S, followed by applying the subsumption rule.

3.2.4 The Davis-Putnam Procedure

The Davis-Putnam-Logemann-Loveland procedure (DPLL), often known as simply the Davis-Putnam procedure, is given in Figure 3.2.

We have sketched the algorithm so that the first branch assigns the chosen atomic proposition x true in the first subtree (line 6) and false in the second (line 7). However, these two branches can be traversed in either order (exchanging x and $\neg x$ on lines 6 and 7), and efficient implementations use this freedom by using powerful heuristics for choosing first the atomic proposition (line 4) and then choosing its truth-value.

Theorem 3.8 *Let S be a set of clauses. Then DPLL(S) returns* true *if and only if S is satisfiable.*

References

Automation of reasoning in the classical propositional logic started in the works of Davis and Putnam and their co-workers, resulting in the Davis-Putnam-Logemann-Loveland procedure [DLL62] (earlier known as the Davis-Putnam procedure) and other procedures [DP60].

Efforts to implement the Davis-Putnam procedure efficiently lead to fast progress from mid-1990ies on.

The current generation of solvers for solving the SAT problem are mostly based on the Conflict-Driven Clause Learning procedure which emerged from the work of Marques-Silva and Sakallah [MSS96]. Many of the current leading implementation techniques and heuristics for selecting the decision variables come from the work on the Chaff solver [MMZ$^+$01]. Several other recent developments have strongly contributed to current solvers [PD07, AS09].

Unit propagation can be implemented to run in linear time in the size of the clause set, which in the simplest variants involve updating a counter every time a literal in a clause gets a truth-value [DG84]. Modern SAT solvers attempt to minimize memory accesses (cache misses), and even counter-based linear time procedures are considered too expensive, and unit propagation schemes that do not use counters are currently used [MMZ$^+$01].

Also stochastic local search algorithms have been used for solving the satisfiability problem [SLM92], but they are not used in many applications because of their inability to detect unsatisfiability.

The algorithms in this chapter assumed the formulas to be in CNF. As discussed in Section 2.5, the translation to CNF may in some cases exponentially increase the size of the formula. However, for the purposes of satisfiability testing, there are transformations to CNF that are of linear size and preserve satisfiability (but not logical equivalence.) These transformations are used for formulas that are not already in CNF [Tse68, CMV09].

Chapter 4

Applications

4.1 State-Space Search

State-space search is the problem of testing whether a state in a *transition system* is reachable from one or more *initial states*. Transition systems in the most basic cases can be identified with *graphs*, and the state-space search problem in this case is the s-t-reachability problem in graphs.

For small graphs the problem can be solved with standard graph search algorithms such as Dijkstra's algorithm. For graphs with an astronomically high number of states, 10^{10} or more, standard graph algorithms are impractical. Many transition systems can be compactly represented, yet their size as graphs is very high. This is because N Boolean (0-1) state variables together with a state-variable based representation of the possible actions or events can induce a state-space with the order of 2^n reachable states. This is often known as the *combinatorial explosion*. It turns out that the s-t-reachability problem for natural compactly represented graphs is PSPACE-complete [GW83, Loz88, LB90, Byl94].

Classical propositional logic has been proposed as one solution to state-space search problems for very large graphs, due to the possibility of representing and reasoning about large numbers of states with (relatively small) formulas.

4.1.1 Representation of Sets

A *state* in a transition system is an assignment of values to the state variables. In this work we restrict to finite-valued state variables. Without loss of generality we will limit to Boolean (two-valued, 0-1) state variables, because any n-valued state variables can be represented in terms of $\lceil \log_2 n \rceil$ Boolean state variables.

We will be representing states with n Boolean state-variables as n-bit bit-vectors.

Example 4.1 Let $X = \{A, B, C, D\}$. Now a valuation that assigns 1 to A and C corresponds to the bit-vector $\overset{ABCD}{1010}$, and the valuation assigning 1 only to B corresponds to the bit-vector $\overset{ABCD}{0100}$. ∎

Any propositional formula ϕ can be understood as a representation of those valuations v such that $v(\phi) = 1$. Since we have identified valuations and bit-vectors and states, a formula naturally represents a set of states or bit-vectors.

Example 4.2 Formula B represents all bit-vectors of the form ?1??, and the formula A represents all bit-vectors 1???. Formula B therefore represents the *set*

$$\{0\mathbf{1}00, 0\mathbf{1}01, 0\mathbf{1}10, 0\mathbf{1}11, 1\mathbf{1}00, 1\mathbf{1}01, 1\mathbf{1}10, 1\mathbf{1}11\}$$

and formula A represents the set

$$\{\mathbf{1}000, \mathbf{1}001, \mathbf{1}010, \mathbf{1}011, \mathbf{1}100, \mathbf{1}101, \mathbf{1}110, \mathbf{1}111\}.$$

∎

Similarly, $\neg B$ represents all bit-vectors of the form ?0??, which is the set

$$\{0\mathbf{0}00, 0\mathbf{0}01, 0\mathbf{0}10, 0\mathbf{0}11, 1\mathbf{0}00, 1\mathbf{0}01, 1\mathbf{0}10, 1\mathbf{0}11\}.$$

This is the *complement* of the set represented by B.

state sets	**formulas over X**
those $\frac{2^{\lvert X \rvert}}{2}$ bit-vectors where x is true	$x \in X$
$\overline{E}$ (complement)	$\neg E$
$E \cup F$	$E \vee F$
$E \cap F$	$E \wedge F$
$E \backslash F$ (set difference)	$E \wedge \neg F$
the empty set $\emptyset$	$\bot$ (constant *false*)
the universal set	$\top$ (constant *true*)
question about sets	**question about formulas**
$E \subseteq F$?	$E \models F$?
$E \subset F$?	$E \models F$ and $F \not\models E$?
$E = F$?	$E \models F$ and $F \models E$?

Table 4.1: Connections between Set-Theory and Propositional Logic

4.1.2 Set Operations as Logical Operations

There is a close connection between the Boolean connectives $\vee$, $\wedge$, $\neg$ and the set-theoretical operations of *union*, *intersection* and *complementation*, which is also historically the origin of Boole's work on Boolean functions.

If ϕ_1 and ϕ_2 represent bit-vector sets S_1 and S_2, then

1. $\phi_1 \wedge \phi_2$ represents set $S_1 \cap S_2$,

2. $\phi_1 \vee \phi_2$ represents set $S_1 \cup S_2$, and

3. $\neg\phi_1$ represents set $\overline{S_1}$.

Example 4.3 $A \wedge B$ represents the set $\{1100, 1101, 1110, 1111\}$ and $A \vee B$ represents the set $\{0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111\}$. ■

Questions about the relations between sets represented as formulas can be reduced to the basic logical concepts we already know, namely logical consequence, satisfiability, and validity.

1. "Is ϕ satisfiable?" corresponds to "Is the set represented by ϕ *non-empty*?"

2. $\phi \models \alpha$ corresponds to "Is the set represented by ϕ *a subset* of the set represented by α?".

3. "Is ϕ valid?" corresponds to "Is the set represented by ϕ *the universal set*?"

These connections allow using propositional formulas as a *data structure* in some applications in which conventional enumerative data structures for sets are not suitable because of the astronomic number of states. For example, if there are 100 state variables, then any formula consisting of just one atomic proposition represents a set of $2^{99} = 633825300114114700748351602688$ bit-vectors, which would require 7493989779944505344 TB in an explicit enumerative representation if each of the 100-bit vectors was represented with 13 bytes (wasting only 4 bits in the 13th byte.)

4.1.3 Relations as Formulas

Similarly to finite sets of atomic objects, *relations* on any finite set of objects can be represented as propositional formulas.

A binary relation $R \subseteq X \times X$ is a set of pairs $(a, b) \in X \times X$, and the representation of this relation is similar to representing sets before, except that the elements are pairs.

As before, we assume that the atomic objects are bit-vectors. A pair of bit-vectors of lengths n and m can of course be represented as a bit-vector of length $n + m$, simply by attaching the two bit-vectors together.

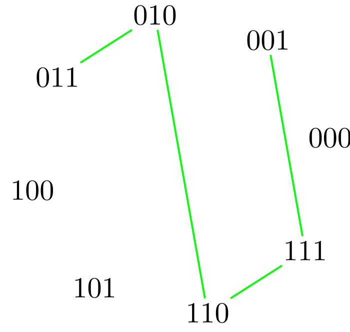

Figure 4.1: Transition relation as a graph

Example 4.4 To represent the *pair* $(0001, 1100)$ of bit-vectors, both expressed as valuations of atomic propositions A, B, C, D, instead use the atomic propositions $X_{01} = \{A_0, B_0, C_0, D_0, A_1, B_1, C_1, D_1\}$ as the index variables, instead of $X = \{A, B, C, D\}$.

The pair $(0001, 1100)$ is hence represented as 00011100, a valuation of X_{01}. ■

Pair $(\overset{A_0}{0}\ \overset{B_0}{0}\ \overset{C_0}{0}\ \overset{D_0}{1}\ ,\ \overset{A_1}{1}\ \overset{B_1}{1}\ \overset{C_1}{0}\ \overset{D_1}{0})$ therefore corresponds to the valuation that assigns 1 to D_0, A_1 and B_1 and 0 to all other variables.

Example 4.5 $(A_0 \leftrightarrow A_1) \wedge (B_0 \leftrightarrow B_1) \wedge (C_0 \leftrightarrow C_1) \wedge (D_0 \leftrightarrow D_1)$ represents the identity relation of 4-bit bit-vectors. ■

Example 4.6 The formula

$$\begin{aligned} inc_{01} = {} & (\neg C_0 \wedge C_1 \wedge (B_0 \leftrightarrow B_1) \wedge (A_0 \leftrightarrow A_1)) \\ & \vee(\neg B_0 \wedge C_0 \wedge B_1 \wedge \neg C_1 \wedge (A_0 \leftrightarrow A_1)) \\ & \vee(\neg A_0 \wedge B_0 \wedge C_0 \wedge A_1 \wedge \neg B_1 \wedge \neg C_1) \\ & \vee(A_0 \wedge B_0 \wedge C_0 \wedge \neg A_1 \wedge \neg B_1 \wedge \neg C_1) \end{aligned}$$

represents the successor relation of 3-bit integers

$$\{(000, 001), (001, 010), (010, 011), (011, 100), (100, 101), (101, 110), (110, 111), (111, 000)\}.$$

■

Example 4.7 Consider the transition relation $\{(001, 111), (010, 110), (011, 010), (111, 110)\}$ corresponding to the graph in Figure 4.1 The relation can be represented as the following truth-table (listing only those of the $2^6 = 64$ lines that have 1 in the column for ϕ.)

a_0	b_0	c_0	a_1	b_1	c_1	ϕ
⋮						
0	0	1	1	1	1	1
⋮						
0	1	0	1	1	0	1
⋮						
0	1	1	0	1	0	1
⋮						
1	1	1	1	1	0	1
⋮						

which is equivalent to the following formula.

$$(\neg a_0 \wedge \neg b_0 \wedge c_0 \wedge a_1 \wedge b_1 \wedge c_1) \vee$$
$$(\neg a_0 \wedge b_0 \wedge \neg c_0 \wedge a_1 \wedge b_1 \wedge \neg c_1) \vee$$
$$(\neg a_0 \wedge b_0 \wedge c_0 \wedge \neg a_1 \wedge b_1 \wedge \neg c_1) \vee$$
$$(a_0 \wedge b_0 \wedge c_0 \wedge a_1 \wedge b_1 \wedge \neg c_1)$$

■

Any binary relation (transition relation) over a finite set can be represented as a propositional formula in this way. These formulas can be used as components of formulas that represent *paths* in the graphs corresponding to the binary relation.

Given a formula for the binary relation (corresponding to edges of the graphs), a formula for any path of a given length T can be obtained by making T copies of the formula and renaming the atomic propositions for the state variables.

Example 4.8 Consider the increment relation from Example 4.6 and the corresponding formula inc_{01}. We make three copies of inc_{01} and change the subscripts of the variables in the last two copies by adding 1 and 2 to them, respectively.

$$inc_{01} \wedge inc_{12} \wedge inc_{23} \wedge inc_{34}.$$

The valuations that satisfy this formula exactly correspond to the sequences s_0, s_1, s_2, s_3, s_4 of consecutive nodes/states in the graph with edges represented by inc_{01}. ■

Example 4.9 Can bit-vector 100 be reached from bit-vector 000 by four steps?

This is equivalent to the satisfiability of the following formula.

$$\neg A_0 \wedge \neg B_0 \wedge \neg C_0 \wedge inc_{01} \wedge inc_{12} \wedge inc_{23} \wedge inc_{34} \wedge A_4 \wedge \neg B_4 \wedge \neg C_4$$

This formula represent the subset of the set of sequences s_0, s_1, s_2, s_3, s_4 that start from the state/node s_0 such that $s_0 \models \neg A \wedge \neg B \wedge \neg C$ is true and ends in the state s_4 such that $s_4 \models A \wedge \neg B \wedge \neg C$. ■

For most applications of state-space search there are nodes/states that have multiple successor nodes. This is because the event that may take place in the state is *non-deterministic*, or because the agent whose behavior we have modeled can take more than one alternative action in that state.

Consider the action that multiplies a 3-bit binary number by 2, which shifts all bits one step left, and loses the most significant bit.

$$ml2_{01} = (A_1 \leftrightarrow B_0) \wedge (B_1 \leftrightarrow C_0) \wedge \neg C_1$$

Example 4.10 Now we can ask the question whether the bit-vector 111 can be reached from bit-vector 000 by four steps, by using operations inc and $ml2$, with the following formula.

$$\neg A_0 \wedge \neg B_0 \wedge \neg C_0 \wedge$$
$$(inc_{01} \vee ml2_{01}) \wedge (inc_{12} \vee ml2_{12}) \wedge (inc_{23} \vee ml2_{23}) \wedge (inc_{34} \vee ml2_{34}) \wedge$$
$$A_4 \wedge B_4 \wedge C_4$$

■

4.1.4 Mapping from Actions to Formulas

In the following X denotes a set of state variables, e.g. $X = \{a, b, c\}$, and sets X_i for different $i \geq 0$ denote the state variables in X with subscript i denoting the values of the state variables at time i, e.g. $X_0 = \{a_0, b_0, c_0\}$ and $X_7 = \{a_7, b_7, c_7\}$. Finally, Θ_{ij} with $j = i + 1$ is the transition relation formula for change between times i and j, expressed in terms of atomic propositions in $X_i \cup X_{i+1}$.

Next we present a systematic mapping from a formal definition of actions to formulas in the propositional logic.

Definition 4.11 *An action is a pair* (p, e) *where*

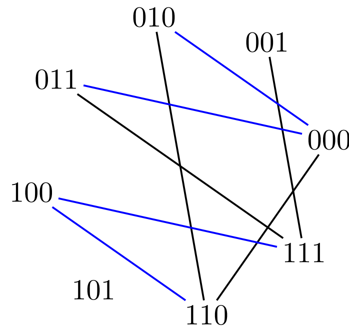

Figure 4.2: Actions as a graph

1. *p is a propositional formula over X (the precondition), and*
2. *e is a set of* effects *of the following forms.*
   (a) *$x := b$ for $x \in X$ and $b \in \{0,1\}$*
   (b) *IF ϕ THEN $x := b$ for $x \in X$ and $b \in \{0,1\}$ and formula ϕ*

Example 4.12 Let the state variables be $X = \{a, b, c\}$. Consider the following actions.

$$(\neg a, \{a := 1, b := 1\})$$
$$(b, \{b := 0, c := 0\})$$

These actions correspond to the following binary relation over the states.

$$\{(000,110),(001,111),(010,000),(010,110),(011,000),(011,111),(110,100),(111,100)\}$$

This relation corresponds to the formula

$$(\neg a_0 \wedge a_1 \wedge b_1 \wedge (c_0 \leftrightarrow c_1)) \vee (b_0 \wedge (a_0 \leftrightarrow a_1) \wedge \neg b_1 \wedge \neg c_1).$$

and can be depicted as the graph in Figure 4.2.

■

Next we present a systematic derivation of a translation from our action definition to the propositional logic. We could give the definition directly (and you can skip to the table given on the next page), but we can alternatively use the weakest precondition predicate familiar from programming logics [Dij75] to do the derivation systematically not only for our limited language for actions, but also for a substantially more general one.

The weakest precondition predicate $wp_\pi(\phi)$ gives for a program π (effect of an action) and a formula ϕ the weakest (most general) formula $\chi = wp_\pi(\phi)$ so that if a state s satisfies χ, that is $s \models \chi$, then the state obtained from s by executing π will satisfy ϕ.

Example 4.13 If a condition is not affected by the program, then the weakest precondition is the condition itself.

$$wp_{x:=1}(y = 1) = y = 1$$

If the program achieves the condition unconditionally, then the weakest precondition is the constant $\top$.

$$wp_{x:=1}(x = 1) = \top$$

■

Definition 4.14 (Expressions) *Only the constant symbols 0 and 1 are expressions.*

Definition 4.15 (Programs) *1. The empty program ϵ is a program.*

2. *If x is a state variable and f is an expression, then $x := f$ is a program*

3. *If ϕ is a formula over equalities $x = f$ where x is a state variable and f is an expression, and π_1 and π_2 are programs, then IF ϕ THEN π_1 ELSE π_2 is a program.*

4. *If π_1 and π_2 are programs, then their composition $\pi_1;\pi_2$ is a program. This program first executes π_2, and then π_2.*

Nothing else is a program.

Definition 4.16 (The Weakest Precondition Predicate)

$$wp_\epsilon(\phi) = \phi \quad (4.1)$$
$$wp_{x:=b}(\phi) = \phi \text{ with } x \text{ replaced by } b \quad (4.2)$$
$$wp_{\textit{IF } \theta \textit{ THEN } \pi_1 \textit{ ELSE } \pi_2}(\phi) = (\theta \wedge wp_{\pi_1}(\phi)) \vee (\neg\theta \wedge wp_{\pi_2}(\phi)) \quad (4.3)$$
$$wp_{\pi_1;\pi_2}(\phi) = wp_{\pi_1}(wp_{\pi_2}(\phi)) \quad (4.4)$$

Now we want to define a propositional formula that expresses the value of a state variable x in terms of the values of all state variables in the preceding state, when the current state was reached by executing the effect π of an action. This is obtained with the weakest precondition predicate simply as

$$wp_\pi(x)@0 \leftrightarrow x_1.$$

This works with effects defined as arbitrary programs, but here we limit our focus to some very simple forms of effects only. In particular, the effect of an action on a state variable x can be an assignment x := 0 or x := 1, or one or both of the conditional assignment IF ϕ THEN x := 1 and IF ϕ' THEN x := 0. With this formulation of effects, each state variable can be reasoned about independently of the other state variables.

We obtain the following in these cases.

$$\begin{aligned}
wp_\epsilon(x = 1) &= (x = 1) \\
wp_{x:=0}(x = 1) &= (0 = 1) = \bot \\
wp_{x:=1}(x = 1) &= (1 = 1) = \top \\
wp_{\textit{IF } \phi \textit{ THEN } x:=0 \textit{ ELSE } \epsilon}(x = 1) &= (\phi \wedge \bot) \vee (\neg\phi \wedge (x = 1)) = (\neg\phi \wedge (x = 1)) \\
wp_{\textit{IF } \phi \textit{ THEN } x:=1 \textit{ ELSE } \epsilon}(x = 1) &= (\phi \wedge \top) \vee (\neg\phi \wedge (x = 1)) = \phi \vee (\neg\phi \wedge (x = 1)) = \phi \vee (x = 1)
\end{aligned}$$

If we have as effects both IF ϕ THEN x := 0 and IF ϕ' THEN x := 1, we can combine these to

$$\textit{IF } \phi' \textit{ THEN } x := 1 \textit{ ELSE IF } \phi \textit{ THEN } x := 0 \textit{ ELSE } \epsilon$$

and obtain

$$\begin{aligned}
wp_{\textit{IF } \phi' \textit{ THEN } x:=1 \textit{ ELSE IF } \phi \textit{ THEN } x:=0 \textit{ ELSE } \epsilon}(x = 1) &= (\phi' \wedge \top) \vee (\neg\phi' \wedge ((\phi \wedge \bot) \vee (\neg\phi \wedge (x = 1)))) \\
&= \phi' \vee (\neg\phi \wedge (x = 1))
\end{aligned}$$

The propositional logic does not have expressions $x = 0$ or $x = 1$, but when we interpret 0 and 1 and falsity and truth, then these correspond to formulas $\neg x$ and x.

Below we summarize the translations of effects into formulas one state variable at a time. For any given state variable x the following table shows what propositional formula τ_x corresponds to the effects affecting x shown in the column *effect*.

case	effect	τ_x
1	x := 1	$\top \leftrightarrow x_1$
2	x := 0	$\bot \leftrightarrow x_1$
3	IF ϕ THEN x := 0	$(\neg\phi@0 \wedge x_0) \leftrightarrow x_1$
4	IF ϕ' THEN x := 1	$(\phi'@0 \vee x_0) \leftrightarrow x_1$
5	IF ϕ THEN x := 0 IF ϕ' THEN x := 1	$(\phi'@0 \vee (\neg\phi@0 \wedge x_0)) \leftrightarrow x_1$
6	-	$x_o \leftrightarrow x_1$

In the table, $\phi@t$ denotes ϕ with atomic propositions for state variables subscripted with t.

Case 6 is used when x does not appear in the effects of the action.

Case 5 is a generalization of cases 1 to 4. An unconditional assignment x := b is equivalent to a conditional assignment IF $\top$ THEN $x := b$. A missing assignment $x := b$ can be simply viewed as a conditional assignment IF $\bot$ THEN $x := b$. It is easy to verify that cases 1 to 4 (and 6) can be obtained from case 5 by defining one or both of ϕ and ϕ' as $\top$ or $\bot$.

Case 5 can be understood as saying that x will be *true* iff x was *true* before and doesn't become *false*, or x becomes *true*.

Now a formula given action can be obtained as the conjunction of its precondition p (with subscripts 0) and the formulas τ_x as obtained from the table.

$$p@0 \wedge \tau_{x^1} \wedge \tau_{x^2} \wedge \cdots \wedge \tau_{x^m}$$

where $X = \{x^1, \ldots, x^m\}$.

Theorem 4.17 *Let (p, e) be an action. Let s_0 and s_1 be any two states such that*

1. $s_0(p) = 1$
2. $s_1(x) = s_0(x)$ *for all $x \in X$ such that x does not appear in assignments in e, and*
3. $s_1(x) = b$ *if there is effect IF ϕ THEN $x := b$ in e such that $s_0(\phi) = 1$.*

Let $v(x_0) = s_0(x)$ and $v(x_1) = s_1(x)$ for all $x \in X$ (a valuation representing the two consecutive states s_0 and s_1.) Let ϕ be the formula representing (p, e). Then $v(\phi) = 1$.

4.1.5 Finding Action Sequences through Satisfiability

Let $\phi^1, \phi^2, \ldots, \phi^n$ be formulas for actions $a^1, a^2, \ldots, a^n$.

Choice between actions is represented by

$$\Theta_{01} = \phi^1 \vee \phi^2 \vee \cdots \vee \phi^n.$$

This formula says that at least one of the formulas for actions is true, and allows two or more true. This might seem problematic, because we have assumed that only action is taken at a time. However, the formulas ϕ^i for each of the actions *completely* determine how the values of state variables between the time points 0 and 1 are related. Two formulas ϕ^i and ϕ^j for $i \neq j$ can be true for a pair of consecutive states s_0 and s_1 only if both formulas describe exactly the same transition from state s_0 to s_1 (even if they described different transitions from states other than s_0). Hence there is no need for Θ_{01} to forbid two or more simultaneous actions.

Theorem 4.18 *A state s' such that $s'(G) = 1$ is reachable from a state s such that $s(I) = 0$ by a sequence of T actions if and only if $I@0 \wedge \Theta_{01} \wedge \cdots \wedge \Theta_{(T-1)T} \wedge G@T$ is satisfiable.*

4.1.6 Relation Operations in Logic

Above, we have used formulas representing relations for formulas that represent a sequence of states and actions. Another use for these formulas is the *computation* of sets of all possible successor states of a set of states. Here formulas are viewed as a data-structure which can be embedded in programs written in any conventional programming language.

To pursue this idea further, we first discuss the computation of sets of states in terms of operations of *relational algebra*, familiar for example from relational databases.

When viewing (sets of) actions as relations R, the *successor states* of a set S are obtained with the *image* operation.

$$img_R(S) = \{s' | s \in S, sRs'\}$$

In terms of operations from the relation algebra (familiar from relational databases), the image operation can be split to a *natural join* operation and a projection.

Example 4.19 (Image as natural join and projection) We will compute the possible successor states of $S = \{000, 010, 111\}$ with respect to the transition relation $R = \{(000, 011), (001, 010), (010, 001), (011, 000)\}$, expressed as $img_R(S)$.

First we select matching lines from S and R by *natural join*. Here the column names 0 and 1 could be viewed as representing the current time point and the next time point, respectively.

$$\begin{array}{c} 0 \\ \hline 000 \\ 010 \\ 111 \end{array} \bowtie \begin{array}{cc} 0 & 1 \\ \hline 000 & 011 \\ 001 & 010 \\ 010 & 001 \\ 011 & 000 \end{array} = \begin{array}{cc} 0 & 1 \\ \hline 000 & 011 \\ 010 & 001 \end{array}$$

The result is a binary relation that contains all members of S in the first column 0. The second step of the image computation is a projection operation for this binary relation to eliminate the first column 0.

$$\Pi_1 \left(\begin{array}{cc} 0 & 1 \\ \hline 000 & 011 \\ 010 & 001 \end{array} \right) = \begin{array}{c} 1 \\ \hline 011 \\ 001 \end{array}$$

The result is $img_R(S) = \{001, 011\}$. ■

The relation operations of natural join and projection can be defined for formulas that represent relations. This allows representing the computation of images $img_R(S)$ as manipulation of representation of S and R as formulas.

The first operation, natural join, corresponds to conjunction.

Example 4.20 Consider the natural join from the previous example.

$$\begin{array}{c} 0 \\ \hline 000 \\ 010 \\ 111 \end{array} \bowtie \begin{array}{cc} 0 & 1 \\ \hline 000 & 011 \\ 001 & 010 \\ 010 & 001 \\ 011 & 000 \end{array} = \begin{array}{cc} 0 & 1 \\ \hline 000 & 011 \\ 010 & 001 \end{array}$$

It is easy to see that when each of these relations is representing with a formula, the natural join operation is simply the forming of the conjunction of the two relations, when the column names are represented as the subscript of the atomic propositions.

$$\begin{array}{c} ((\neg A_0 \wedge \neg B_0 \wedge \neg C_0) \vee (\neg A_0 \wedge B_0 \wedge \neg C_0) \vee (A_0 \wedge B_0 \wedge C_0)) \\ \wedge \\ (\neg A_0 \wedge \neg A_1 \wedge (\neg B_0 \leftrightarrow B_1) \wedge (\neg C_0 \leftrightarrow C_1)) \\ = \\ \neg A_0 \wedge \neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1)) \end{array}$$

The logical equivalence of the formulas on both sides of = can be verified with truth-tables or some other method.

We have first eliminated the last disjuncts of $(\neg A_0 \wedge \neg B_0 \wedge \neg C_0) \vee (\neg A_0 \wedge B_0 \wedge \neg C_0) \vee (A_0 \wedge B_0 \wedge C_0))$ which is inconsistent with $\neg A_0$. From the remaining disjuncts we have removed conjunct $\neg A_0$, and then inferred additional conjuncts by using the equivalences $\neg B_0 \leftrightarrow B_1$ and $\neg C_0 \leftrightarrow C_1$. These equivalences are now redundant and are left out from the resulting formula. ■

$\exists$ and $\forall$ Abstraction

We will show that the projection operation in computing

$$\Pi_1 \left(\begin{array}{cc} 0 & 1 \\ \hline 000 & 011 \\ 010 & 001 \end{array} \right) = \begin{array}{c} 1 \\ \hline 011 \\ 001 \end{array}$$

can be performed with the Existential Abstraction (also known as *Shannon expansion*) operation.

This operation allows producing from From $\neg A_0 \wedge \neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1))$ the formula $(\neg A_1 \wedge B_1 \wedge C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge C_1)$.

Definition 4.21 Existential abstraction *of ϕ with respect to x:*

$$\exists x.\phi = \phi[\top/x] \vee \phi[\bot/x].$$

This operation allows eliminating atomic proposition x from any formula. Make two copies of the formula, with x replaced by the constant true $\top$ in one copy and with constant false $\bot$ in the other, and form their disjunction.

Analogously we can define *universal abstraction*, with conjunction instead of disjunction.

Definition 4.22 Universal abstraction *of ϕ with respect to x:*

$$\forall x.\phi = \phi[\top/x] \wedge \phi[\bot/x].$$

Example 4.23

$$\begin{aligned}
&\exists B.((A \to B) \wedge (B \to C)) \\
&= ((A \to \top) \wedge (\top \to C)) \vee ((A \to \bot) \wedge (\bot \to C)) \\
&\equiv C \vee \neg A \\
&\equiv A \to C
\end{aligned}$$

$$\begin{aligned}
&\exists AB.(A \vee B) = \exists B.(\top \vee B) \vee (\bot \vee B) \\
&= ((\top \vee \top) \vee (\bot \vee \top)) \vee ((\top \vee \bot) \vee (\bot \vee \bot)) \\
&\equiv (\top \vee \top) \vee (\top \vee \bot) \equiv \top
\end{aligned}$$

■

$\forall c$ and $\exists c$ *eliminate the column* for c by *combining lines* with the same valuation for variables other than c.

Example 4.24 $\exists c.(a \vee (b \wedge c)) \equiv a \vee b$ $\forall c.(a \vee (b \wedge c)) \equiv a$

$a\ b\ c$	$a \vee (b \wedge c)$
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

$a\ b$	$\exists c.(a \vee (b \wedge c))$
0 0	0
0 1	1
1 0	1
1 1	1

$a\ b$	$\forall c.(a \vee (b \wedge c))$
0 0	0
0 1	0
1 0	1
1 1	1

■

Example

From $\neg A_0 \wedge \neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1)$ produce $(\neg A_1 \wedge B_1 \wedge C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge C_1)$.

$$\begin{aligned}
&\Phi = \neg A_0 \wedge \neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1) \\
&\exists A_0 B_0 C_0.\Phi \\
&= \exists B_0 C_0.(\Phi[0/A_0] \vee \Phi[1/A_0]) \\
&= \exists B_0 C_0.(\neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1)) \\
&= \exists C_0.((\neg A_1 \wedge (\neg C_0 \wedge B_1 \wedge C_1)) \vee (\neg A_1 \wedge ((\neg C_0 \wedge \neg B_1 \wedge C_1)))) \\
&= (\neg A_1 \wedge B_1 \wedge C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge C_1)
\end{aligned}$$

4.1.7 Symbolic State-Space Search

Given a formula ϕ that represents a set S of states and a formula Θ_{01} that represents a binary relation R, a formula that represents $img_R(S)$ can be computed as follows.

1. Construct the formula $\exists X_0.(\phi@0 \wedge \Theta_{01})$ where X_0 is all state variables with subscript 0. After existentially abstracting X_0 the resulting formula only contains atomic propositions with subscript 1.

2. Replace all subscripts 1 by 0.

We denote the resulting formula by $img_{\Theta_{01}}(\phi)$.

This image operation can be applied repeatedly to compute formulas that represent the states reached with arbitrarily long sequences of actions. This leads to a *symbolic* breadth-first search algorithm for computing reachable states.

Algorithm 4.25 (Breadth-first search with formulas) *The algorithm takes as its first input a formula I that represents one or more initial states. This formula is over the set X of state variables. We denote the formula I with all atomic propositions with subscript 0 as $I@0$.*

The second input to the algorithm is a formula Θ_{01} which represents a binary relation over the set of states. It is over atomic propositions which are the state variables with subscripts 0 and 1.

1. $i := 0$
2. $\Phi_0 := I@0$
3. $i := i + 1$
4. $\Phi_i := \Phi_{i-1} \vee img_{\Theta_{01}}(\Phi_{i-1})$
5. *if* $\Phi_i \not\models \Phi_{i-1}$ *then go to step 3*

After the computation has terminated, the formula Φ_i represents the set of all reachable states.

References

For the solution of reachability problems in discrete transition systems the use of propositional logic as a representation of sets and relations was first proposed in the works by Coudert et al. [CBM90, CM90] and Burch et al. [BCL$^+$94].

The use of these logic-based techniques was possible with (ordered, reduced) Binary Decision Diagrams [Bry92], generally known as BDDs or OBDDs. BDDs are a restrictive normal form that guarantees a *canonicity* property: any two logically equivalent Boolean formulas are represented by a unique BDD. The canonicity property guarantees the polynomial-time solvability of many central problems about Boolean formulas, including *equivalence*, *satisfiability*, and *model-counting*. Though in some cases necessarily (even exponentially) larger than unlimited Boolean formulas, BDDs also in practice often lead to compact enough representations when unlimited formulas would in practice be too large.

Other similar normal forms, with useful polynomial-time operations, exist, including Decomposable Negation Normal form (DNNF) [Dar01, Dar02]. There are interesting connections between this type of normal forms and the Davis-Putnam procedure [HD05].

Problems with scalability of BDD-based methods was well-understood already in mid-1990ies. Another way of using the propositional logic for solving state-space search problems was proposed by Kautz and Selman [KS92] in 1992, and in 1996 they demonstrated it to be often far better scalable than alternative search methods for solving state-space search problems in AI [KS96]. This method was based on mapping one reachability query, whether a state satisfying a given goal formula is reachable in a given number of steps from a specified initial state. The method suffered from the large size of the formulas far less than BDDs, and was soon shown to be in many cases far better scalable than BDD-based methods for example in Computer Aided Verification, for solving the model-checking problem [BCCZ99].

In Section 4.1.5 we only presented the most basic translation of action sequences to propositional formulas, with the exactly one action at each step. This is what is known as a *sequential* encoding. This representation suffers from the possibility of sequences of actions $a_1, \ldots, a_n$ that are independent of each other, and which therefore can be permuted to every one of $n!$ different orders. So called *parallel* encodings allow several actions or events at the same time [KS96, RHN06]. Having several actions can simultaneous decreases the horizon length. This substantially improves the scalability of SAT-based methods in the main applications including the planning, diagnosis, and model-checking problems.

Index

BDD, 23
binary decision diagrams, 23
Boolean Constraint Propagation, 12
clause, 6
CNF, 6
combinatorial explosion, 14
complement (of a literal), 11
conjunctive normal form, 6
consistency, 6
Davis-Putnam procedure, 12
Davis-Putnam-Logemann-Loveland procedure, 12
disjunctive normal form, 7
DNF, 7
effect, 18
empty clause, 11
existential abstraction, 21
image operation, 20, 23
inconsistency, 6
literal, 6
negation normal form, 7
NNF, 7
OBDD, 23
precondition, 18
resolution, 11
satisfiability, 6
Shannon expansion, 21
term, 6
unit clause, 11
unit propagation, 12
unit resolution, 11
unit subsumption, 12
universal abstraction, 22
valuation, 3
weakest preconditions, 19

Bibliography

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 399–404. Morgan Kaufmann Publishers, 2009.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[BCL+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. MacMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Byl94] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[CBM90] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, 1990.

[CM90] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 126–129. IEEE Computer Society Press, 1990.

[CMV09] Benjamin Chambers, Panagiotis Manolios, and Daron Vroon. Faster SAT solving with better CNF generation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1590–1595, 2009.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Dar01] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

[Dar02] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002) and the 14th Conference on Innovative Applications of Artificial Intelligence (IAAI-2002)*, pages 627–634, 2002.

[DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, San Francisco, 1979.

[GW83] Hana Galperin and Avi Wigderson. Succinct representations of graphs. *Information and Control*, 56:183–198, 1983. See [Loz88] for a correction.

[HD05] Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In Leslie Pack Kaelbling, editor, *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 156–162. Professional Book Center, 2005.

[KS92] Henry Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley & Sons, 1992.

[KS96] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press, 1996.

[LB90] Antonio Lozano and José L. Balcázar. The complexity of graph problems for succinctly represented graphs. In Manfred Nagl, editor, *Graph-Theoretic Concepts in Computer Science, 15th International Workshop, WG'89*, number 411 in Lecture Notes in Computer Science, pages 277–286. Springer-Verlag, 1990.

[Loz88] Antonio Lozano. NP-hardness of succinct representations of graphs. *Bulletin of the European Association for Theoretical Computer Science*, 35:158–163, June 1988.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01)*, pages 530–535. ACM Press, 2001.

[MSS96] João P. Marques-Silva and Karem A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 220–227, 1996.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In Joao Marques-Silva and Karem A. Sakallah, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2007)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.

[RHN06] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 46–51, 1992.

[Tse68] G. S. Tseitin. On the complexity of derivations in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, 1968.