Design of Logic-based Intelligent Systems

Design of Logic-based Intelligent Systems

Klaus Truemper
University of Texas at Dallas

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data is available.

ISBN 0-471-48403-2

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Contents

Preface

The computer revolution of the 20th and 21st centuries has bestowed upon mankind powerful tools for creative work. For the first time, one can dare to begin the construction of machines and systems first envisioned by the mathematician, physicist, engineer, and philosopher G. W. Leibniz (1646–1716). He proposed that human disagreements should be settled by computation, and foresaw collection and encoding of the facts of the real world. The implementation of his daring vision is possible now, some 300 years later.

There is some disagreement among researchers as to which approaches, concepts, and tools are appropriate for the implementation of Leibniz's vision. In addition, there is much debate about how results should be interpreted. Even use of the word "intelligence" can set off a discussion. We would like to steer clear of these disagreements by focusing instead on what can be accomplished, and by sidestepping philosophical questions about the meaning of the results. It may well be that the future will settle the philosophical questions rather pragmatically.

Despite our desire to avoid controversy, we would like to use the term "intelligent" in connection with computer-based systems. That is, we define a system to be *intelligent* if it accomplishes feats which, when carried out by humans, require a substantial amount of intelligence.

In this book, we use an extension of propositional logic for the design and implementation of intelligent systems. The results build upon work started by us in 1985. That effort produced some effective techniques for computation in propositional logic, described in Truemper (1998), and a related software package called *Leibniz System* in honor of G. W. Leibniz. Thus, this book constitutes a third step in a quest for effective design tools for intelligent systems.

Despite the close connection with our prior work, the results of this book can be implemented independently from the results of the earlier effort. This is possible since, in the meantime, some researchers have produced other effective methods for computation in propositional logic. These methods may be used here instead of the algorithms of Truemper (1998). In that sense, this book is completely independent of the results of that reference and of the Leibniz System software.

This book is intended to be both a handbook for practitioners and a text for senior level or graduate level courses covering intelligent systems. Each chapter contains a number of exercises that reinforce learning of the material. The only prerequisite is command of a procedural programming language like C or C++, and the ability to program simple combinatorial algorithms typically covered in first-level programming courses. There is one exception. In Chapter 8, Sections 8.4–8.6 require basic knowledge of probability theory and statistics, and Section 8.7 uses basic results of linear programming.

Acknowledgments

The research underlying this book was supported over many years by the Office of Naval Research (ONR). Additional support was provided by the Italian research institute CNR-IASI "Antonio Ruberti," the Technical Support Working Group (TSWG), and the author's home institution, the University of Texas at Dallas.

Many people directly or indirectly contributed to the research. H. Almubaid, A. Remshagen, J. Straach, F.-S. Sun, and Y. Zhao carried out directly related research as Ph.D. students. Joint projects with G. Felici and R. Rinaldi of CNR-IASI were essential for some of the algorithms and applications, as were recent research projects with students S. Bartnikowski, M. Granberry, and J. Mugan. Furthermore, B. Belknap and J. W. Davidson III supplied information about two applications in Chapter 11.

Typesetting was done in TEX, and initial checking was accomplished with the Laempel System. The latter system was constructed in joint work with Y. Zhao and H. Almubaid, using the Leibniz System for the processing of the numerous logic modules.

Much help with the final checking of content was provided by T. Achterberg, C. Buchheim, S. Hachul, F. Liers, J. Mugan, M. Percan, B. Randerath, I. Truemper, and U. Truemper.

To the many people who gave so much support in so many ways, I express my sincere thanks. This book could not have been created without their help.

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 1

Introduction

1.1 Overview

We begin with a definition of intelligent systems.

Intelligent System

One may define a computer-based system to be *intelligent* if it accomplishes feats which, when carried out by humans, require a substantial amount of intelligence. Examples of such accomplishments are the diagnosis of complex diseases and the determination of the sense of words in texts.

Expert systems and *intelligent agents* are two particular types of intelligent systems. We define them next.

Expert System

An *expert system* is an intelligent system which in an interactive setting asks a person for information and, based upon the response, draws conclusions or gives advice. For example, diagnostic expert systems identify defects or diseases.

Intelligent Agent

An *intelligent agent* is an intelligent system which perceives its environment by sensors and which uses that information to act upon the environment.

Since this definition is quite general, one might conjecture that all intelligent systems essentially are intelligent agents. For example, expert systems can be viewed as particular intelligent agents. But there are intelligent systems that are not intelligent agents. An example is an intelligent system that constructs expert systems from data. That intelligent system is not an intelligent agent, unless one accepts the odd notion that creating an expert system constitutes acting upon an environment.

Logic as Tool

This book covers the design and implementation of intelligent systems where logic is the main tool of construction. That choice allows rather easy encoding and evaluation of complex relationships. As a result, one can create intelligent systems for a large variety of settings.

Besides techniques rooted in logic, a number of other methods and tools have been proposed and used in intelligent systems. We cannot do justice to the breadth and depth of these methods here. The reader interested in an overview should consult any one of the introductory texts on artificial intelligence cited in Section 2.8 of Chapter 2.

The remainder of this chapter proceeds as follows.

Section 1.2 resolves a conflicting use of certain terms of logic.

Section 1.3 introduces levels of thinking and relates them to the levels of a certain hierarchy of the theory of computation.

Section 1.4 reviews logic-based concepts and tools available for the construction of intelligent systems. From the available options, we select an extension of propositional logic and related algorithms.

Section 1.5 covers various approaches in which intelligent systems can be formulated.

Section 1.6 deals with the computational complexity of the algorithms of this book.

Section 1.7 outlines options for implementing the algorithms in computer programs.

Section 1.8 suggests ways for reading the remainder of the book.

1.2 Terminology

In the first part of the 20th century, logicians such as B. Russell and A. N. Whitehead as well as philosophers such as L. Wittgenstein expended a major effort to devise a clear and consistent terminology for logic. Meticulous adherence to these definitions is essential for work in logic. Unfortunately, some of the definitions clash badly with definitions employed in other areas of science such as computer science, engineering, and operations research.

We cite two examples taken from propositional logic. There, the word "model" means an assignment of *True/False* values for the variables of a propositional formula such that these values produce a value of *True* for the formula. The word "valid" characterizes a propositional formula that evaluates to *True* under all *True/False* assignments for the variables. On the other hand, in other areas of science, "model" typically means a mathematical formulation of a real-world situation, and "valid" means that a formulation correctly represents that part of the real world.

We do not know of a reasonable way to reconcile these and other conflicting definitions, short of eliminating their use in this book. But these terms are part and parcel of almost any treatment of real-world problems outside logic, and we cannot agree to forgo their use. Thus, we employ these terms as done in areas such as computer science, engineering, and operations research, and apologize to logicians for this unaccustomed use.

1.3 Levels of Thinking

When we use logic for reasoning, we may distinguish between *levels of thinking*. We define the levels in an informal discussion.

At the first level, we reason about problems using some logic modules in a certain, direct way. The logic modules represent the relationships and facts of the problems. Questions such as "Do symptoms s and t imply that disease d is present?" or "Is this log-in behavior typical for a hacker?" can be answered that way. We refer to thinking at the first level also as *thinking about problems*.

At the second level, a number of logic modules are given, and we select which one is to be used for reasoning. We call this *thinking about thinking*. At the third level, we think about which selection process is to be used. This is *thinking about thinking about thinking*. Evidently, we can go on recursively and define ever higher levels of thinking.

One may argue, rightly so, that all cases of thinking at the second and higher levels are reducible to thinking at the first level. To exclude the possibility of trivial reductions for the general case, we demand, as part of the definition of the levels, that such a reduction generally cannot be achieved by a *polynomial algorithm*, where the number of computing steps is bounded by a polynomial of the size of the instance. The term "cannot be accomplished" is meant in a pragmatic sense. That is, the word "cannot" signals that currently nobody knows how to accomplish a reduction of the general case by a polynomial algorithm.

The propositional logic problems at the first level of the so-called polynomial hierarchy of the theory of computation, as well as related versions that involve, for example, optimization, are examples of thinking at the first level and thus of thinking about problems.

The propositional logic problems at the second level of the hierarchy require thinking about thinking. This is a case where currently it is not known whether a polynomial reduction to the first level is possible. Another instance of thinking about thinking is reasoning that decides whether a first-level logic module should be introduced, or revised, or even deleted. One may view this as a generalization of the above example involving the second level of the polynomial hierarchy.

Basic results for first-level thinking are described in Chapters 2 and 3, and for second-level thinking in Chapter 4. Applications for both levels are given in Chapter 11.

1.4 Logic Tools

Logic can be used in a number of ways to construct intelligent systems. We give a brief summary of current tools. For references, see Section 2.8 of Chapter 2.

Production Rules

Production rules encode if–then relationships. An example rule is, "If symptoms s and t are present, then the patient has disease d." Of course, this statement is encoded in suitable notation. The rules are evaluated by two recursive chaining procedures. *Forward chaining* requires some initial facts and derives conclusions by a recursive application of the rules. *Backward chaining* begins with given conclusions and deduces which facts can produce those results by a recursive and, in some sense, inverted use of the rules.

Prolog

Prolog is an elaboration of the production rule concept where *universal quantification* is allowed. An example Prolog statement is, "For all x, y, and z: If x is the ancestor of y, and if y is the parent of z, then x is the ancestor z." Of course, the encoding uses suitable notation. Prolog uses forward and backward evaluation processes which in spirit are similar to forward and backward chaining, but which, owing to the universal quantification, are much more complex and may involve tree searches. With that process, one may deduce results from given facts and explore facts that produce given conclusions.

Propositional Logic

Propositional logic uses *True/False* variables and several operators such as $\neg$ ("not"), $\wedge$ ("and"), and $\vee$ ("or"). Here, the "or" operator is *inclusive*. For example, $r \vee s$ evaluates to *True* if at least one of r and s has the value *True*. Formulas of propositional logic are produced from variables and operators by simple rules. Of particular interest are two forms of formulas called *Conjunctive Normal Form* (CNF) and *Disjunctive Normal Form* (DNF).

A key problem called the *satisfiability problem* (SAT) asks that one determines whether there are *True/False* values for the variables of a given CNF formula such that the values cause the formula to evaluate to *True*. This problem is difficult, but for reasonably sized formulas can be solved rapidly and reliably. Many decision problems of intelligent systems can be formulated as cases of SAT. There also is a variation of SAT called MINSAT, where costs are affiliated with the *True/False* values of variables, and where a satisfying solution with minimum total cost is desired.

First-order Logic

First-order logic is a generalization of propositional logic. Here, formulas are constructed using *predicates*, which are also called truth functions; operators such as $\neg$, $\wedge$, and $\vee$; and *existential* and *universal quantifiers*. The analog of the satisfying solution of propositional logic is of much interest. The existence of such a solution generally is very difficult to test, and existing algorithms often require interactive processing where a human directs the computing effort. For some applications, in particular in mathematics, one gladly suffers such a demanding solution process when the outcome potentially is a major mathematical result.

Fuzzy Logic and Bayesian Networks

The methods described so far can only deal with deterministic relationships. The following two techniques handle uncertainty. *Fuzzy logic* combines production rules and likelihood measures to express implications involving uncertainty. *Bayesian networks* use directed acyclic graphs and certain probability tables to achieve the same goal.

Choice of Propositional Logic with Extensions

This book uses propositional logic with some extensions so that one can:

- Accommodate uncertainty of facts and optimization of plans.
- Validate logic formulations by effective techniques.
- Automatically handle nonmonotonicity and incompleteness, which are two types of inconsistencies between a logic formulation and the real world.
- Produce explanations for decisions.
- Create cost-effective question-and-answer systems where information can only be obtained at some cost.
- Handle thinking at the first and second level, and sometimes even at higher levels. For example, one can decide when a system should give up on a quest.
- Learn logic formulas from data. One may merge such formulas seamlessly with other, possibly manually constructed, formulas.

Comparison

Production rules, Prolog, fuzzy logic, and Bayesian networks do not support all of the above features. However, the universal quantification of Prolog simplifies the modeling and evaluation of general facts, and fuzzy logic and Bayesian networks represent uncertainty in a more exact manner than done here. An appropriate extension of first-order logic would support more features than those listed above, but using them would lead to a large increase of computational effort. Thus, one may view the approach taken here as a way to achieve both a large number of features and computational effectiveness.

In the next section, we sketch ways in which models may be formulated.

1.5 Formulation of Models

We may build a logic model of a real-world setting in several ways.

First, we may analyze the problem and manually define logic formulas that model the observed relationships. Chapter 5 covers this approach.

Second, we may analyze data connected with the problem and extract logic formulas that correctly represent relationships implicit in the data. Chapters 7 and 8 contain results for such learning.

Third, we may build an intelligent system which accepts data of the problem as input and which, virtually without human assistance, outputs an intelligent system that correctly handles the problem. Section 11.8 of Chapter 11 covers a class of problems treatable this way.

Fourth, ..., well, we could go on and on and never come to an end. There are just too many possibilities for the construction of systems. For

example, a formulation may analyze its performance and correct itself. Such behavior could be called *implicit learning*, in contrast to the above mentioned extraction of logic formulas from data, which one could call *explicit learning*. This book covers aspects of implicit learning in Chapters 9 and 10.

1.6 Computational Complexity

The problems SAT and MINSAT defined earlier are core problems that must be solved in most algorithms of this book. These problems are theoretically difficult, but reasonably sized instances can be solved rapidly and reliably. If we consider the solution of any SAT or MINSAT instance to require one step, then all algorithms of this book require a polynomial number of steps, where the polynomial is a function of the length of the input for the algorithms. Because of this fact, we dispense with complexity arguments throughout the book.

1.7 Software

The subsequent chapters contain a large number of algorithms that cover all problems except the basic problems SAT and MINSAT. Section 2.8 of Chapter 2 provides references for methods handling the latter problems. Following this approach, we suggest two options for the implementation of the algorithms of this book.

First, the reader may obtain programs for SAT and MINSAT, typically via the Internet. The reader then implements the algorithms of this book using the programs for SAT and MINSAT as subroutines.

Second, the reader may obtain via the Internet a software package called *Leibniz System* which has been created by the author and which has most, though not all, algorithms of this book implemented in the programming language C. The reader then implements additional algorithms as needed. If the reader owns a copy of this book, then by this very fact the reader is licensed to download and use all code of the Leibniz System for any commercial or non-commercial application without additional charge or license fee. For details and downloading instructions, the reader should visit the website www.leibnizsystem.com.

1.8 Suggested Reading Sequences

As next step, the reader should cover Chapter 2, which contains introductory material about logic and related concepts. For a first pass, the reader may just scan the material.

After reading Chapter 2, there are two independent paths. If the reader wants to learn direct formulation of intelligent systems, Chapters 3, 5, and 6 provide basic results, and Chapters 4, 9, and 10 present advanced material. If, instead, the reader is just interested in the derivation of logic formulas from data, then Chapters 7 and 8 provide that information.

For some typical applications, the reader should turn to Chapter 11. With each application, the chapter includes a list of the preceding chapters containing relevant theoretical results and tools. Thus, the reader need only study those chapters to handle a specific application.

Dependence of Chapters

The dependence among chapters is as follows. Suppose that Chapter 2 has been read. Then Chapters 4–6 require Chapter 3, but otherwise can be read independently. Chapter 7 can be read independently. Chapter 8 requires Chapter 7. Chapter 9 requires Chapter 6. Chapter 10 requires most material of the preceding chapters.

Part I

Logic Problems

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 2

Introduction to Logic and Problems SAT and MINSAT

2.1 Overview

In Chapter 1, it is claimed that logic may be used to encode knowledge and carry out reasoning. In this chapter, we take the first steps to back up that claim. Specifically, we cover definitions and basic results of logic and show how one may encode simple facts in logic and how one may draw conclusions from such facts.

The simplest type of logic, called *propositional logic*, involves *logic variables* that may take on the value *True* or *False*. When we combine logic variables in a certain way using *logic operators*, we obtain *logic formulas*. When a logic formula has a certain format, it is said to be in *conjunctive normal form* (CNF). The logic relationships of many real-world situations can be readily expressed by CNF formulas.

A more general type of logic, called *first-order logic*, involves functions that for each input value produce *True* or *False*. The functions are called *predicates* or *truth functions*. Analogously to the propositional logic case, we combine predicates using certain *logic operators* to create *logic formulas*. Under certain restrictions, a first-order logic formula can be transformed to a propositional logic formula. In this book, we only consider first-order logic formulas for which this transformation is possible.

When we make decisions or reason about a situation, we rely on some knowledge to deduce conclusions. Formulas of propositional or first-order logic often are an attractive way of encoding such knowledge. Once such an encoding has been accomplished, we deduce conclusions using a mathematical technique called *theorem proving*. The technique requires that

for a given logic formula one either finds a so-called *satisfying solution* or concludes that no such solution exists. For propositional CNF formulas, the problem of finding a satisfying solution or concluding the nonexistence of such a solution is called the SAT *problem.* The name "SAT" is not an acronym, but is an abbreviation of the word "satisfiability."

The conclusions must sometimes take costs into account. Specifically, two costs are associated with each logic variable. One of the costs applies when the variable takes on the value *True*, and the second cost applies to the case of *False*. To deduce a conclusion, one either obtains a satisfying solution that minimizes total cost or concludes that there is no satisfying solution. This problem is called the *logic minimization problem.* For propositional CNF formulas, the logic minimization problem is the MINSAT *problem.* The name "MINSAT" stands for "minimization subject to satisfiability."

Finally, we introduce the concept of *likelihood* in connection with logic statements. That concept allows us to encode facts that do not hold with certainty.

The chapter proceeds as follows.

Section 2.2 provides detail about variables, operators, formulas of propositional logic, and the special case of CNF formulas.

Section 2.3 discusses predicates, operators, and formulas of first-order logic. The section includes a transformation of such formulas to propositional formulas under certain conditions.

Section 2.4 defines certain classes of formulas of propositional logic.

Section 2.5 shows how decision making can be reduced to theorem proving and introduces the SAT problem.

Section 2.6 defines the general logic minimization problem and the MINSAT problem.

Section 2.7 introduces the notion of likelihood in connection with logic formulas and discusses other kinds of logic.

Section 2.8 suggests material for further reading.

Section 2.9 contains exercises.

2.2 Propositional Logic

We define basic concepts of propositional logic.

Variable

A *logic variable* of *propositional logic* is a variable that may take on just two values, the *logic constants True* and *False*. When we use logic variables to represent facts or knowledge, we must first define the meaning

of *True/False* for each variable. For example, *True* for a variable called *sunshine* might mean that the sun is shining, and *False* that it is not. For a second variable *rain*, *True* might mean that it is raining, and *False* that it is not.

As a matter of convenience, we abbreviate such definitions. We state explicitly the meaning for *True* and assume implicitly the opposite meaning for *False*. So it suffices to say that *True* for *sunshine* means that the sun is shining and that *True* for *rain* means that it is raining.

Logic Operators

Several *operators* are used to build *logic formulas* from logic variables.

The simplest operator is the unary operator $\neg$, denoting "not." That operator turns *True* into *False* and *False* into *True*. Evidently, the operator negates the meaning. For example, since *True* for *rain* means that it is raining, *True* for $\neg$*rain* means that it is not raining.

Two binary operators are the *conjunction* $\wedge$, denoting "and," and the *disjunction* $\vee$, denoting "or." For example, from *sunshine* and *rain* we obtain *sunshine* $\wedge$ *rain* and *sunshine* $\vee$ *rain*. We explain the effect of the operators using these example expressions.

Expression *sunshine* $\wedge$ *rain*: If both *sunshine* and *rain* have the value *True*, then *sunshine* $\wedge$ *rain* has the value *True*. For all other ways of assigning *True/False* values to the two variables, *sunshine* $\wedge$ *rain* has the value *False*.

Expression *sunshine* $\vee$ *rain*: If *sunshine* or *rain* has the value *True*, then *sunshine* $\vee$ *rain* has the value *True*. Here and later, "or" is meant to be *inclusive*. Thus, both *sunshine* and *rain* may have the value *True*. If both *sunshine* and *rain* have the value *False*, *sunshine* $\vee$ *rain* has the value *False*.

The expressions *sunshine* $\wedge$ *rain* and *sunshine* $\vee$ *rain* are examples of *logic formulas*. The general definition of a logic formula relies on repeated use of the operators $\neg$, $\wedge$, and $\vee$. First, each variable by itself is considered to be a formula. Second, if S is a formula, then $\neg S$ is a formula. Third, if S and T are formulas, then $S \wedge T$ and $S \vee T$ are formulas as well.

For example, let S be the formula *sunshine* $\wedge$ *rain*, and declare T to be the variable, and thus formula, *snow*. Here, the value *True* for *snow* means that it is snowing. The formula $\neg S$ is $\neg$(*sunshine* $\wedge$ *rain*), $S \wedge T$ is (*sunshine* $\wedge$ *rain*) $\wedge$ *snow*, and $S \vee T$ is (*sunshine* $\wedge$ *rain*) $\vee$ *snow*. Note the added parentheses in the formulas. They indicate how the formulas were created. *True/False* values for formulas are calculated in the expected way. That is, $\neg S$ has the value opposite to that of S, $S \wedge T$ has the value *True* precisely when both S and T have the value *True*, and $S \vee T$ has the value *True* precisely when S or T has the value *True*.

Literal

A propositional variable may occur any number of times in a formula. Each such occurrence, possibly negated, is a *literal* of the formula.

Equality of Formulas

Two formulas are equal if, for all possible *True/False* values assigned to the variables of the formulas, the resulting *True/False* values of the two formulas are in agreement.

Basic Equations

We state elementary equations of propositional logic. Exercise (2.9.1) asks the reader to verify validity.

$$\begin{aligned} \neg(\neg S) &= S \\ S \wedge T &= T \wedge S \\ S \vee T &= T \vee S \\ (R \wedge S) \wedge T &= R \wedge (S \wedge T) \\ (R \vee S) \vee T &= R \vee (S \vee T) \end{aligned} \tag{2.2.1}$$

The second and third equation of (2.2.1) express the *commutative law* of propositional logic, while the last two equations express the *associative law*. The latter law allows us to simplify $(R \wedge S) \wedge T$ and $R \wedge (S \wedge T)$ to $R \wedge S \wedge T$ without risk of confusion. Similarly, $(R \vee S) \vee T$ and $R \vee (S \vee T)$ may be simplified to $R \vee S \vee T$.

Another useful result for formulas is the *distributive law*. It says that

$$\begin{aligned} R \wedge (S \vee T) &= (R \wedge S) \vee (R \wedge T) \\ R \vee (S \wedge T) &= (R \vee S) \wedge (R \vee T) \end{aligned} \tag{2.2.2}$$

The $\neg$ operator switches the roles of $\wedge$ and $\vee$ according to the equations $\neg(S \wedge T) = (\neg S) \vee (\neg T)$ and $\neg(S \vee T) = (\neg S) \wedge (\neg T)$. As a matter of clarity, we prefer to omit parentheses whenever the $\neg$ operator applies to just one term. For example, instead of $(\neg S)$ in the two equations, we prefer to write $\neg S$. Using that rule, the two equations can be simplified to

$$\begin{aligned} \neg(S \wedge T) &= \neg S \vee \neg T \\ \neg(S \vee T) &= \neg S \wedge \neg T \end{aligned} \tag{2.2.3}$$

Implications

Reasoning often involves statements of the form "If S takes on the value *True*, then T takes on *True* as well." A shorter, equivalent statement is "S

implies T." We use the operator $\rightarrow$ for "implies" and write $S \rightarrow T$. We call S the *condition* of the implication and call T the *conclusion*. The converse case is "S is implied by T," denoted by $S \leftarrow T$. When both implications hold simultaneously, we say "S holds if and only if T holds," denoted by $S \leftrightarrow T$.

The *True/False* values of $S \rightarrow T$, $S \leftarrow T$, and $S \leftrightarrow T$ are computed as follows.

$S \rightarrow T$: If S has the value *False* or T has the value *True*, then $S \rightarrow T$ has the value *True*. In all other cases, $S \rightarrow T$ has the value *False*. The rule may seem odd, since, in the case of *False* for S, we assign to $S \rightarrow T$ the value *True* regardless of the value of T. But that choice turns out to be a good one for the encoding of knowledge and computing in logic. The rule for assigning *True/False* values to $S \rightarrow T$ is expressed by the equation

$$S \rightarrow T = \neg S \vee T \tag{2.2.4}$$

$S \leftarrow T$: We declare $S \leftarrow T$ to be equal to $T \rightarrow S$ and apply the above rule for $\rightarrow$. In equation form, we have

$$S \leftarrow T = T \rightarrow S = \neg T \vee S = S \vee \neg T \tag{2.2.5}$$

$S \leftrightarrow T$: Both $S \rightarrow T$ and $S \leftarrow T$ hold, so we have

$$S \leftrightarrow T = (S \rightarrow T) \wedge (S \leftarrow T) = (\neg S \vee T) \wedge (S \vee \neg T) \tag{2.2.6}$$

An alternate definition is that $S \leftrightarrow T$ has the value *True* precisely when the value of S agrees with that of T. According to that definition,

$$S \leftrightarrow T = (S \wedge T) \vee (\neg S \wedge \neg T) \tag{2.2.7}$$

Exercise (2.9.2) asks the reader to prove that the two definitions are equivalent. That is,

$$(\neg S \vee T) \wedge (S \vee \neg T) = (S \wedge T) \vee (\neg S \wedge \neg T) \tag{2.2.8}$$

At times, we link a number of formulas, say $S_1, S_2, \ldots, S_n$, by the $\wedge$ operator, getting $S_1 \wedge S_2 \wedge \ldots \wedge S_n$. Note that the associative law permits us to dispense with parentheses. It is convenient to abbreviate the expression to $\bigwedge_{i=1}^{n} S_i$ analogously to the use of the Greek capital sigma, $\sum$, for the addition of several terms. In the same spirit, we abbreviate $S_1 \vee S_2 \vee \ldots \vee S_n$ to $\bigvee_{i=1}^{n} S_i$.

CNF and DNF System

Two types of formulas are of particular interest. One type is called *conjunctive normal form system*, for short CNF *system*. Such a formula is,

for some $n \geq 1$, a conjunction of formulas $S_1, S_2, \ldots, S_n$, that is, $\bigwedge_{i=1}^{n} S_i$, where each S_i is a disjunction of literals. Each S_i is called a CNF *clause*. For example, $(s_1 \vee \neg s_2) \wedge \neg s_3 \wedge (s_1 \vee s_4)$ is a CNF system, with CNF clauses $S_1 = s_1 \vee \neg s_2$, $S_2 = \neg s_3$, and $S_3 = s_1 \vee s_4$. Another example is $s_1 \vee \neg s_2$ or just s_1, each consisting of just one CNF clause.

We permit the explicit declaration of the variables as part of the definition of a CNF system. Indeed, due to such specifications, we have selected the term "CNF system" instead of "CNF formula." For the above formula $(s_1 \vee \neg s_2) \wedge \neg s_3 \wedge (s_1 \vee s_4)$, one could specify s_1, s_2, s_3, and s_4 as variables. But one could also list additional variables even though they do not occur in the given formula. For example, s_1, $s_2, \ldots, s_{10}$ would be an acceptable list of variables. We see later—for example, when we define trivial and empty CNF systems—why it is advantageous to admit such a specification of variables.

When we reverse the roles of disjunction and conjunction in a CNF system, we obtain a *disjunctive normal form system*, for short DNF *system*. Thus, a DNF system is a disjunction of the form $\bigvee_{i=1}^{n} S_i$, where each S_i is a conjunction of literals. Each S_i is a DNF *clause* of the system. An example DNF system is $(s_1 \wedge \neg s_2) \vee \neg s_3 \vee (s_1 \wedge s_4)$, with DNF clauses $S_1 = s_1 \wedge \neg s_2$, $S_2 = \neg s_3$, and $S_3 = s_1 \wedge s_4$.

In this book, we typically work with CNF systems and thus omit the DNF case from the definitions to follow. For the same reason, we often reduce the term CNF *clause* to just *clause* without risk of confusion. In the second half of the book, we do need some definitions for the DNF case. We introduce them at that time as needed.

Empty Clause

It is convenient that we consider clauses that do not contain any literals. Such a clause is *empty*. We assign to such a clause the value *False*. If in a CNF system $\bigwedge_{i=1}^{n} S_i$ one of the clauses S_i is empty, then the *False* value for that clause causes the entire CNF system to have the value *False*.

Trivial and Empty CNF Systems

Due to the separate specification of variables and clauses in CNF systems, the following degenerate cases are possible. First, we may have variables but no clauses. Second, we may have no variables, but do have one or more clauses, all of which must be empty. Third, we may have no variables and no clauses. In the first two cases, the CNF system is *trivial*, and in the third one, *empty*.

We declare a trivial CNF system with variables but no clauses to have the value *True* regardless of the *True/False* values assigned to the variables.

Since any empty clause has the value *False*, a trivial CNF system without variables but with clauses, each of which must be empty, has the value *False*. Finally, the empty CNF system, which has no variables and no clauses, has the value *True*.

CNF Subsystem

Let S be a given CNF system. We *reduce* S to a CNF *subsystem* by deleting clauses or variables. If a variable is being deleted, then all literals arising from that variable must be deleted from the clauses. A subsystem of S is *proper* if at least one clause or variable has been deleted.

Conversion to CNF System

There is a general method for converting an arbitrary logic formula to a CNF system. For our purposes, we do not need to cover that general method. Instead, we treat some special cases that will arise later in various contexts.

(2.2.9) Case. Conversion of a DNF system.

Using the distributive law repeatedly, we convert the DNF system to a CNF system. For example, let the DNF system be $(\neg s_1 \wedge s_2) \vee (t_1 \wedge \neg t_2)$. Application of the distributive law produces the CNF system $(\neg s_1 \vee t_1) \wedge (\neg s_1 \vee \neg t_2) \wedge (s_2 \vee t_1) \wedge (s_2 \vee \neg t_2)$.

The resulting CNF system may be large relative to the size of the DNF system. When that is the case, an alternate method, described in Exercise (2.9.4), is useful. The method produces a CNF system that is not much larger than the given DNF system and that, though not equal to the given DNF system, nevertheless is *equivalent* in a certain sense. For details of that definition, see Exercise (2.9.4).

(2.2.10) Case. Conversion of $S \wedge T$ where S and T are arbitrary formulas.

Convert each of S and T to a CNF system, then combine the two systems to one CNF system for $S \wedge T$.

(2.2.11) Case. Conversion of $S \to T$ where S is a conjunction and T is a disjunction of literals.

We treat an example case with $S = \neg s_1 \wedge s_2$ and $T = t_1 \vee \neg t_2$. From (2.2.4), we know that $S \to T = \neg S \vee T$. Using that result and the rule (2.2.3) for handling negations, $(\neg s_1 \wedge s_2) \to (t_1 \vee \neg t_2) = [\neg(\neg s_1 \wedge s_2)] \vee (t_1 \vee \neg t_2) = s_1 \vee \neg s_2 \vee t_1 \vee \neg t_2$. The resulting formula is a CNF clause. Evidently, we have replaced each literal of S by its negation, each $\wedge$ of S by $\vee$, and finally $\to$ by $\vee$. That rule also works in the general case.

(2.2.12) Case. Conversion of $S \to T$ where S is a disjunction and T is a conjunction or disjunction, or where S is a conjunction or disjunction and T is a conjunction. In each instance, the terms of the conjunction or disjunction are literals.

The conditions imposed on S and T imply that $S \to T$, which is equal to $\neg S \vee T$, is a DNF system that may be converted to a CNF system using the method of (2.2.9).

For example, if $S = \neg s_1 \wedge s_2$ and $T = t_1 \wedge \neg t_2$, then $\neg S \vee T = s_1 \vee \neg s_2 \vee (t_1 \wedge \neg t_2)$, which is a DNF system. We use the distributive law to convert that system to the CNF system $(s_1 \vee \neg s_2 \vee t_1) \wedge (s_1 \vee \neg s_2 \vee \neg t_2)$.

(2.2.13) Case. Conversion of $S \leftrightarrow T$ where each one of S and T is a conjunction or disjunction of literals.

Rewrite $S \leftrightarrow T$ as $(S \to T) \wedge (S \leftarrow T)$ and apply (2.2.10)–(2.2.12).

2.3 First-order Logic

First-order logic is an extension of propositional logic. It relies on predicates, also called truth functions, plus universal and existential quantifiers. In this book, we make a restricted use of first-order logic. Indeed, the restrictions are such that all formulas of first-order logic considered here are readily transformed to formulas of propositional logic.

Predicate

Let U be some nonempty set, called the *universe*. In general, U is taken to be infinite. For any $k \geq 1$, denote the set $\{(u_1, u_2, \ldots, u_k) \mid u_i \in U, \text{ for all } i\}$, which is the *product* of k copies of U, by $\prod_{i=1}^{k} U$. When k is small, we may denote $\prod_{i=1}^{k} U$ by $U \times U \times \cdots \times U$, with k U-terms. A *predicate* or *truth function* is a function p that, for some $k \geq 1$, takes the elements of $\prod_{i=1}^{k} U$ to the set $\{True, False\}$.

Universal and Existential Quantifiers

Predicates are employed in connection with the *universal quantifier* $\forall$, denoting "for all" or "for each," and the *existential quantifier* $\exists$, denoting "there exists." We give two example statements involving these quantifiers and a predicate p that takes the pairs of $U \times U$ to $\{True, False\}$. The first statement is $\forall x\big(\forall y\big[p(x, y)\big]\big)$. It means "for all $x \in U$ and for all $y \in U$, $p(x, y)$ holds." The second statement is $\forall x\big(\exists y\big[p(x, y)\big]\big)$. It means "for each $x \in U$, there exists $y \in U$ such that $p(x, y)$ holds." Note the implicit use of U in each quantification.

Construction of Formulas

Analogously to the case of propositional formulas, a simple construction rule creates all formulas of first-order logic, as follows.

First, any propositional variable or predicate constitutes a formula. In the predicate case, each variable occurring as part of an argument of the predicate is *free*.

Second, let S and T be formulas. Then $\neg S$, $S \wedge T$, $S \vee T$, $S \rightarrow T$, $S \leftarrow T$, and $S \leftrightarrow T$ are all formulas. Any free variable of S is also free in $\neg S$, and any free variable of S or T is also free in $S \wedge T$, $S \vee T$, $S \rightarrow T$, $S \leftarrow T$, and $S \leftrightarrow T$.

Third, let x be a free variable of a formula S. Then $\forall x(S)$ and $\exists x(S)$ are formulas, with free variables as in S except for x.

Fourth, the construction process is allowed to stop only if the formula at hand has no free variables. Such a formula is *completed*.

Finite Quantification

In this book, we always require the set U to be finite. Indeed, instead of assuming each quantification term to implicitly involve U, we permit specification of a particular finite set with each term. Thus, $\forall x$ becomes, for some finite set X, $\forall x \in X$, and $\exists x$ becomes $\exists x \in X$.

For example, let *family* be the set consisting of the elements *dad*, *mom*, and *son*; that is, $family = \{dad, mom, son\}$. Define a predicate *has_car* on the set *family*. If x is an element of *family*, then the value *True* for $has\_car(x)$ is to mean that x has a car. To express that each member of *family* has a car, we write $\forall x \in family\ (has\_car(x))$. To state that some member of *family* has a car, we write $\exists x \in family\ (has\_car(x))$.

Conversion to Propositional Formula

A completed formula with finite quantification can be translated to a propositional formula in a two-step process.

First, we replace universal quantification by conjunctions and replace existential quantification by disjunctions. Specifically, $\forall x \in X$ is replaced by $\bigwedge_{x \in X}$, and $\exists x \in X$ by $\bigvee_{x \in X}$. For example, $\forall x \in family\ (has\_car(x))$ becomes $\bigwedge_{x \in family} has\_car(x)$, which is a short form of $has\_car(dad) \wedge has\_car(mom) \wedge has\_car(son)$. The expression $\exists x \in family\ (has\_car(x))$ becomes $\bigvee_{x \in family} has\_car(x)$, which is a short form of $has\_car(dad) \vee has\_car(mom) \vee has\_car(son)$.

Second, we define propositional variables from the predicates and the sets over which they are defined in the obvious way. For example, from the predicate *has_car* defined on $family = \{dad, mom, son\}$, we derive the propositional variables $has\_car(dad)$, $has\_car(mom)$, and $has\_car(son)$.

Below, we classify propositional formulas and cover related results. Since formulas of first-order logic with finite quantification are readily converted to propositional formulas, a separate treatment of such first-order logic formulas is not needed.

2.4 Classification of Propositional Formulas

One may classify propositional formulas according to the *True/False* values they can produce.

Tautology, Contradiction, Satisfiability

Let S be a propositional formula.

S is a *tautology* if, for any assignment of *True/False* values to the variables, the value of S is *True*. An example is $S = s \vee \neg s$.

S is a *contradiction* if, for any assignment of *True/False* values to the variables, the value of S is *False*. An example is $S = s \wedge \neg s$.

Note that S is a tautology if and only if $\neg S$ is a contradiction.

S is *satisfiable* if, for some assignment of *True/False* values to the variables, S produces the value *True*. Any *True/False* values for the variables that produce *True* for S constitute a *satisfying solution*. For example, $S = s \wedge t$ has the value *True* when s and t have the value *True*. Hence, S is satisfiable, and $s = t = True$ is a satisfying solution. An S that is not satisfiable is *unsatisfiable*.

Evidently, if S is a contradiction, then it is unsatisfiable, and *vice versa*. Also, if S is a tautology, then it is satisfiable. However, if S is satisfiable, then it need not be a tautology. For example, $S = s \wedge t$ has the value *True* for $s = t = True$ and the value *False* for $s = t = False$, and thus is satisfiable but not a tautology.

Suppose we have a method for deciding whether an arbitrary propositional formula is satisfiable. That method can be used to decide whether a given formula S is a contradiction or a tautology, as follows.

For the contradiction case, we apply the method to S and declare S to be a contradiction if the algorithm determines S to be unsatisfiable.

For the tautology case, we apply the method to $\neg S$ and declare S to be a tautology if $\neg S$ is unsatisfiable.

We conclude that the problem of classifying a formula S as a contradiction or tautology essentially is the problem of deciding satisfiability of S or $\neg S$.

Satisfiability Problem SAT

Deciding whether an arbitrary propositional formula is satisfiable is difficult. That is, no algorithm is known that efficiently carries out such a classification for all formulas. On the other hand, a number of classification algorithms have been constructed that efficiently handle propositional formulas of certain classes. Trivial examples are the classification instances of this chapter, including those of the exercises. Each such instance involves at most four variables and is readily handled by trying out all possible *True/False* values for the variables.

Solving real-world satisfiability problems often amounts to deciding whether certain CNF subsystems of some CNF system are satisfiable. We have stated that deciding satisfiability for all propositional formulas is difficult. The same conclusion applies if we restrict ourselves to CNF systems. Thus, solving practical logic problems is potentially difficult.

In computational logic, the problem of deciding satisfiability for CNF systems is called SAT. Thus, SAT is the following problem.

(2.4.1) SAT

Instance: CNF system S.

Solution: Either: A satisfying solution of S. Or: The conclusion that S is not satisfiable.

Much research has been done to find effective solution algorithms for subclasses of SAT. Section 2.8 provides references.

The satisfiability problem for propositional formulas is closely tied to the notion of theorems and decision making. We establish that link next.

2.5 Theorem Proving and Decision Making

Let S and T be two propositional formulas defined on the same set of variables. The formula T is a *theorem* of S if any satisfying solution of S is a satisfying solution for T. Equivalently, we could demand that the formula $S \rightarrow T$ is a tautology, or that $\neg(S \rightarrow T)$ is a contradiction, or that $\neg(S \rightarrow T)$ is unsatisfiable. Since $\neg(S \rightarrow T) = \neg(\neg S \vee T) = S \wedge \neg T$, yet another definition of *theorem* is possible. It demands that $S \wedge \neg T$ is unsatisfiable.

Link Between Theorems and Decision Making

When we make choices, we typically rely on prior knowledge or insight. In many cases, we can encode such knowledge compactly in a logic formula,

say S. The satisfying solutions of S correspond to the situations or settings that are possible according to prior knowledge.

Similarly, one may encode choices or decisions in a formula T. That is, each satisfying solution of T corresponds to a particular choice or action. Below, we call T a *conclusion*. We justify that definition by the fact that in a moment we connect S and T by the formula $S \rightarrow T$. According to an earlier definition, S is the condition of the implication, and T is the conclusion.

We consider two types of decision problems.

In problems of the first type, we need to establish whether a given conclusion T is *mandatory* for S. This is the case if S implies T. Equivalently, we may require T to be a theorem of S or $S \wedge \neg T$ to be unsatisfiable.

In problems of the second kind, we are to settle whether a certain conclusion T is *allowed* for S or, equivalently, if $S \wedge T$ is satisfiable. If the answer is affirmative, each satisfying solution of $S \wedge T$ corresponds to a situation encoded by S that allows for a choice encoded by T.

The two types of decision problems are related. We look at the connection after discussion of an example problem.

Example: Using an Umbrella

Suppose the knowledge producing S concerns the weather and a related action, as follows. We have either sunshine or rain, and we use an umbrella if and only if it is raining. Our weather knowledge is rather defective, but let us ignore that shortcoming.

We define two variables *sunshine* and *rain* for the given types of weather and a variable *umbrella* for the use of that device, with the obvious meaning for the value *True*. Using these variables, we represent our knowledge by the formula $S = (sunshine \leftrightarrow \neg rain) \wedge (rain \leftrightarrow umbrella)$.

The reader should verify that S has the value *True* precisely if the *True/False* values for the variables reflect a situation allowed by our knowledge. For example, $sunshine = False$ and $rain = umbrella = True$ depict the situation where there is rain and no sunshine and where an umbrella is used.

Suppose we want to know whether we use an umbrella when the sun is shining. The problem may seem accurately stated, but actually it is not. Suppose the sun is shining. Do we want to know (1) whether we must use an umbrella, or (2) whether we may use an umbrella, or (3) whether we are not allowed to use an umbrella, or (4) whether we are allowed not to use an umbrella? Cases (1) and (3) are decision problems where we settle whether a conclusion is mandatory. Cases (2) and (4) represent decision problems where we settle whether a conclusion is allowed. We treat each one of the four cases.

For the first case, consider the statement $T_1 = \mathit{sunshine} \rightarrow \mathit{umbrella}$. We must use an umbrella when the sun is shining, if and only if T_1 is a mandatory conclusion, or theorem, of S. To check whether T_1 is a theorem of S, we test if $S \wedge \neg T_1$ is satisfiable. We urge the reader to carry out that test by trying all possible *True/False* values for the variables of $S \wedge \neg T_1 = (\mathit{sunshine} \leftrightarrow \neg \mathit{rain}) \wedge (\mathit{rain} \leftrightarrow \mathit{umbrella}) \wedge \neg(\mathit{sunshine} \rightarrow \mathit{umbrella})$. The test should establish that $S \wedge \neg T_1$ has the value *True* when $\mathit{sunshine} = \mathit{True}$ and $\mathit{rain} = \mathit{umbrella} = \mathit{False}$. Thus, $S \wedge \neg T_1$ is satisfiable, and T_1 is not a theorem of S. Accordingly, sunshine does not force us to use an umbrella. Note that the *True/False* values for the variables resulting in *True* for $S \wedge \neg T_1$ demonstrate that T_1 is not a theorem of S. For that reason, those *True/False* values constitute a *counterexample* to any claim that T_1 is a theorem of S.

We proceed to the second case. We must settle whether we may use an umbrella if the sun is shining. Consider the statement $T_2 = \mathit{sunshine} \wedge \mathit{umbrella}$. Sunshine and the use of an umbrella are compatible if and only if $S \wedge T_2$ has a satisfying solution. The reader should check that $S \wedge T_2 = (\mathit{sunshine} \leftrightarrow \neg \mathit{rain}) \wedge (\mathit{rain} \leftrightarrow \mathit{umbrella}) \wedge (\mathit{sunshine} \wedge \mathit{umbrella})$ has no satisfying solution. Indeed, for $S \wedge T_2$ to have the value *True*, both *sunshine* and *umbrella* must have the value *True*. But for those values, both $\mathit{rain} = \mathit{True}$ and $\mathit{rain} = \mathit{False}$ produce the value *False* for $S \wedge T_2$. We conclude that there is no situation where sunshine occurs and we would use an umbrella.

For the third case, we check whether $T_3 = \mathit{sunshine} \rightarrow \neg \mathit{umbrella}$ is a theorem of S. We leave it to the reader to verify that this is so.

Finally, the fourth case involves checking whether, for $T_4 = \mathit{sunshine} \wedge \neg \mathit{umbrella}$, the formula $S \wedge T_4$ is satisfiable. The reader should confirm that this is the case.

The first and fourth case are related. We have $S \wedge \neg T_1$ satisfiable. Since $T_1 = \mathit{sunshine} \rightarrow \mathit{umbrella}$ is not a theorem, $S \wedge \neg T_1$ is satisfiable. But $\neg T_1 = \neg(\mathit{sunshine} \rightarrow \mathit{umbrella}) = \mathit{sunshine} \wedge \neg \mathit{umbrella} = T_4$, so $S \wedge T_4 = S \wedge \neg T_1$ is satisfiable, and we have an affirmative answer for the fourth case.

Similarly, the second and third case are related. We know that T_3 of the third case is a theorem of S, that is, $S \wedge \neg T_3$ is unsatisfiable. This implies that for any satisfying solution of S, the corresponding value of $\neg T_3 = \neg(\mathit{sunshine} \rightarrow \neg \mathit{umbrella}) = \mathit{sunshine} \wedge \mathit{umbrella} = T_2$ is *False*. Hence, $S \wedge T_2 = S \wedge \neg T_3$ is unsatisfiable, and we have a negative answer for the second case.

Link Between the Two Decision Problems

In general, settling whether some T is a theorem of some S provides additional information about satisfying solutions of S for which T or $\neg T$ has a

certain value. The next theorem tells the relationships.

(2.5.1) Theorem. *Let S and T be propositional formulas. Then* (a) *and* (b) *below hold.*

(a) *If S is satisfiable and T is a theorem of S, then there exists a satisfying solution of S for which $\neg T$ has the value False.*
(b) *If T is not a theorem of S, then there exists a satisfying solution of S for which $\neg T$ has the value True.*

Proof. For (a), let T be a theorem of S. Since $S \wedge \neg T$ is unsatisfiable, each satisfying solution of S must produce the value *False* for $\neg T$. Since S is satisfiable, there is at least one such solution.

We turn to (b). If T is not a theorem of S, then $S \wedge \neg T$ is satisfiable. Hence, there exists a satisfying solution of S that produces the value *True* for $\neg T$. □

Seemingly Odd Decisions

There are cases of S where any T is a theorem of S. There are also cases of T where T is a theorem of any S. We examine the circumstances producing the two situations.

The first situation occurs if S is unsatisfiable. Under that assumption, $S \wedge \neg T$ is unsatisfiable regardless of the form of T. Hence, we have the seemingly odd result that any formula T is a theorem of an unsatisfiable S.

Suppose we construct an S as part of a decision making system. Further suppose that S, which should be satisfiable, is unsatisfiable due to some error committed during the construction process. If we are not aware of that shortcoming and use S to decide whether formulas T are theorems of S, then we will deduce that every such T is a theorem of S. This error can be avoided only if we check S for satisfiability. Of course, satisfiability of S does not guarantee that S is correctly formulated. But unsatisfiability of S means that S is in error and must be corrected.

The propositional formulas S of practical applications can be large and complicated. Suppose that we have an efficient algorithm for deciding satisfiability of S and that we find S to be unsatisfiable. We know that S contains a formulation error, but otherwise may not have a clue which part of S is in error. Hence, we need some diagnostic tool to localize the error. Chapter 5, which covers the validation of formulas, provides such a tool.

We turn to the second situation, where T is a theorem of any S. That case is at hand if T is a tautology. Indeed, $\neg T$ is then unsatisfiable, and, for any S, the formula $S \wedge \neg T$ is unsatisfiable as well. Accordingly, T is a theorem of any S as claimed. Such T typically does not represent a reasonable decision problem and is the result of a formulation error that needs to be corrected.

In practical applications, it usually is easy to detect whether a given T is a tautology. In such settings, T tends to be rather simple. Indeed, it often is one CNF clause. Exercise (2.9.9) asks the reader to prove that a CNF system, and hence a CNF clause, is a tautology if and only if each clause of the system contains, for some variable s, the literals s and $\neg s$. Thus, we can check by inspection whether T is a tautology and identify the formulation error if this is so.

So far, we have handled decisions by satisfiability testing. In the next section, we meet decisions that require optimization.

2.6 Logic Minimization

Let S be a propositional formula, say with variables $s_1, s_2, \ldots, s_n$. Suppose for each variable s_j of S we have two costs. One of the costs is c_j. It is incurred if we assign *True* to s_j. The second cost is d_j and is incurred if we assign *False* to s_j. For any satisfying solution for S, define $\sum_{s_j=True} c_j$ to be the sum of the c_j for which the corresponding variable s_j has the value *True*. Similarly, define $\sum_{s_j=False} d_j$ to be the sum of the d_j for which s_j has the value *False*. Define the *total cost* of the satisfying solution to be $\sum_{s_j=True} c_j + \sum_{s_j=False} d_j$.

Consider the following problem. We are given a formula T and must decide if $S \wedge T$ is satisfiable. If this is the case, we must obtain a satisfying solution for $S \wedge T$ such that the total cost associated with that solution is minimal. We call that problem the *logic minimization problem.* We discuss an example.

Example: Weather and Transportation

Suppose we have the following setting. The weather is either sunshine or rain. We take either the bus or a taxi to go to work. Suppose it is raining. If we use the bus, we must walk to the bus stop and hence use an umbrella. If we use a taxi, we do not have to use an umbrella.

We use the variables *sunshine*, *rain*, *bus*, *taxi*, and *umbrella*. The variables *sunshine* and *rain* are interpreted as before, and *bus* and *taxi* have the expected meaning. Define S to be the formula $S = (sunshine \leftrightarrow \neg rain) \wedge (bus \leftrightarrow \neg taxi) \wedge [(rain \wedge bus) \leftrightarrow umbrella]$. That formula represents the above facts.

Suppose that the fare for the bus is \$3 and for the taxi \$4, and that we view the inconvenience of handling an umbrella to be equivalent to a cost of \$2.

Accordingly, we declare the cost associated with the value *True* for *bus* to be 3 and define the corresponding cost for *taxi* and *umbrella* to be

4 and 2, respectively. For all other *True/False* values of the variables of S, we declare the cost to be 0.

We want to find the least-cost way of going to work when the sun is shining. We express that condition by $T = sunshine$. It is easy to see that $S \wedge T$ has a satisfying solution. Due to T, any such solution must have the value *True* for *sunshine*. The reader should try all possible *True/False* values for the remaining variables *rain*, *bus*, and *taxi* and should compute the total cost for each satisfying solution of $S \wedge T$ found that way. That process confirms that $sunshine = bus = True$ and $rain = taxi = umbrella = False$, with total cost = 3, is the unique best solution.

Now consider the case of rain. We specify it by $T = rain$. This time, the best solution is $sunshine = bus = umbrella = False$ and $rain = taxi = True$, with total cost = 4.

Logic Minimization Problem MINSAT

Define MINSAT to be the logic minimization problem where the given formula is a CNF system. Thus, MINSAT is the following problem.

(2.6.1) MINSAT

Instance: CNF system S. For each variable of S, two rational cost values associated with the values *True* and *False* for that variable.

Solution: Either: A satisfying solution of S for which the total cost is minimum. Or: The conclusion that S is unsatisfiable.

The logic minimization instances discussed above are not MINSAT instances since they are not in CNF form. However, one readily converts them to CNF form using the method of Section 2.2.

MINSAT is at least as difficult as SAT, and no algorithm is known that effectively solves all MINSAT instances. Much research has been done to find algorithms for special subclasses of MINSAT. Section 2.8 provides references.

2.7 Other Kinds of Logic

Decision making or reasoning based on theorem proving is called *deduction*. There are other types of reasoning.

Fuzzy Logic and Bayesian Networks

Let $S = rain \rightarrow umbrella$. The formula S says that, whenever it rains, an umbrella is used. But suppose that most, but not all, people use an

umbrella when it is raining. Then the deduction "umbrella is used" from "it rains" is frequently but not always correct. How can we express that fact? A simple way is to allow *likelihoods* in connection with implications. In the example case, the formula becomes *rain* $\rightarrow$ *umbrella frequently.*

There are several ways to derive conclusions from formulas involving likelihoods. To-date, the most successful methods have been fuzzy logic and Bayesian networks. *Fuzzy logic* is based on *production rules.* Each such rule is an implication where the condition is a conjunction of literals and where the conclusion is just one literal. The axioms of fuzzy logic specify how likelihoods are handled in connection with production rules. The deductive process relies on these axioms and the forward chaining procedure discussed in Chapter 1. *Bayesian networks* use directed acyclic graphs and certain probability tables to express uncertain implications. The evaluation is carried out by exact or approximate methods.

In Chapter 6, we adapt one of the notions of fuzzy logic and obtain a theorem-proving method that accommodates likelihoods. That method has advantages and disadvantages when compared with fuzzy logic or Bayesian networks. A disadvantage is that the method is not as sophisticated as the cited methods with regard to the handling of the likelihoods. An advantage is that the method processes formulations other than production rules and that it deduces results that cannot be obtained by the forward chaining of fuzzy logic or the evaluation methods of Bayesian networks.

Abduction

Suppose the formula *rain* $\rightarrow$ *umbrella* represents all we know. Assume we see a person with an umbrella. We are tempted to declare that it is raining, but cannot deduce that conclusion by theorem proving from *rain* $\rightarrow$ *umbrella*. But reasoning by *abduction* allows us to draw that conclusion. At first, abduction may seem inappropriate. But outside mathematics, it is often employed.

For example, suppose a medical doctor examines a patient with elevated temperature. The doctor may claim an infection to be present, applying abduction to the knowledge that infections cause elevated temperature. Indeed, most diagnoses of diseases, malfunctions, or accidents are based on abduction, as are many everyday decisions.

One may convert abduction to deduction with likelihoods. Returning to the medical diagnosis example, let us express the knowledge that infections cause elevated temperature by *infection* $\rightarrow$ *elevated_temperature*. We turn that formula into *elevated_temperature* $\rightarrow$ *infection frequently*, with the obvious interpretation, and deduce from any instance of elevated temperature that an infection likely is present.

Induction

In mathematics, *induction* is a principle that supports certain proofs. For example, suppose we have a function $f(n)$ defined for the integers $n \geq 1$. We want to show that the function has a certain form. We can establish that result via induction as follows.

We first show that $f(1)$ has the claimed form. Then we assume that, up to an arbitrary positive integer k, $f(1), f(2), \ldots, f(k)$ have the claimed form and prove that $f(k+1)$ has the desired form as well. When this is accomplished, then the principle of mathematical induction allows us to conclude that $f(n)$ has the claimed form for all positive n.

In the everyday use of the word "induction," we mean a more general concept. In terms of the function example $f(n)$, suppose we have seen that, say, $f(1)$, $f(4)$, $f(9)$, $f(17)$, $f(23)$, and $f(89)$ have the claimed form. Using the nonmathematical but useful induction employed in everyday life, we might claim, or rather postulate, that $f(n)$ has the desired form for all positive n. Of course, the claim might be in error, so we cannot develop mathematical or scientific results with this imprecise notion of induction. But that concept of induction serves us well in learning processes where we extract principles or relationships from data. In Chapters 7 and 8, we see how such learning can be accomplished.

Nonmonotonic Reasoning

Recall that in propositional logic a formula T is a theorem of a formula S if $S \wedge \neg T$ is unsatisfiable. Suppose we enlarge S using a formula R, say to $S' = R \wedge S$. Let a formula T be a theorem of S. Since $S \wedge \neg T$ is unsatisfiable, $R \wedge S \wedge \neg T = S' \wedge T$ is unsatisfiable as well. Hence, T is a theorem of S'.

Suppose R represents additional knowledge. The above observation about T may be paraphrased as, "If additional knowledge becomes available, then previously established theorems remain theorems." Reasoning having that property is called *monotonic*.

Practical reasoning often is not monotonic. For example, when we see a small object with wings soaring in the sky, we might reason that it is a bird. Once we have more information, we may change our opinion and declare the object to be a kite or an airplane. As another example, an initial medical diagnosis claiming a heart problem might be downgraded to the claim of an upset stomach as more information becomes available.

In this book, we allow for nonmonotonic reasoning whenever knowledge involves likelihoods. For example, we may deduce that some conclusion holds with a certain likelihood and later discover the conclusion to be incorrect. In Chapter 9, we see how such situations are handled within our theorem-proving framework.

Other Kinds of Reasoning

Other kinds of reasoning are *default reasoning*, which is used in the absence of specific knowledge, and *metareasoning*, which decides which type of reasoning to employ. The two types of reasoning are cases of thinking at the second level, as defined in Section 1.3 of Chapter 1. In this book, we do not cover default reasoning or metareasoning. We mention only that both kinds can be accommodated by certain multilevel intelligent systems.

Looking Ahead

There are several variations of the SAT and MINSAT problems that arise from real-world applications. Chapter 3 discusses these variations in detail. There also exist decision problems that according to current knowledge are substantially more difficult than the SAT or MINSAT problems. We see in Chapter 4 that these decision problems can be formulated as so-called quantified SAT or MINSAT problems.

The subsequent chapters frequently invoke two algorithms for solving the problems SAT and MINSAT. The algorithms are called SOLVE SAT and SOLVE MINSAT, respectively. For the purposes of this book, the specific form of these algorithms is not important, as long as they are reasonably effective. The next section contains references. A number of commercially or publicly available software implementations of the two algorithms may be obtained from the Internet. Exercise (2.9.13) below discusses a related, optional programming project.

2.8 Further Reading

An excellent overview of the historical developments of propositional logic and first-order logic may be found in Newman (1956) or Kneale and Kneale (1984).

A number of books cover theorem proving and problem SAT in detail. See, for example, Wos, Overbeek, Lusk, and Boyle (1992) or Kleine Büning and Lettmann (1999).

We have solved small example instances of the satisfiability problem or of the logic minimization problem by enumerating all possible *True/False* values for the variables. That method is impractical when the number of variables is anything but very small. Methods exist that solve instances with hundreds or even thousands of variables, see Gent, van Maaren, and Walsh (2000) and Franco, Kautz, Kleine Büning, van Maaren, Selman, and Speckenmeyer (2004).

MINSAT is treated in detail in Truemper (1998). The reference includes decomposition-based solution methods for SAT and MINSAT.

For a discussion of the various kinds of reasoning of Section 2.7, see introductory artificial intelligence texts such as Nilsson (1998), Luger (2002), Negnevitsky (2002), and Russell and Norvig (2003).

2.9 Exercises

The calculations required by some of the exercises may be accomplished with the software of Exercise (2.9.13).

(2.9.1) By assigning all possible *True/False* values to R, S, and T of (2.2.1)–(2.2.3), confirm validity of those equations.

(2.9.2)
(a) Prove that, for any formula S, $S \wedge \neg S = \mathit{False}$ and $S \vee \neg S = \mathit{True}$.
(b) Prove that, for any formula S, $S = S \vee \mathit{False} = S \wedge \mathit{True}$.
(c) Use (a) and (b) and the distributive law of (2.2.2) to prove the equation of (2.2.8).

(2.9.3) Convert each of the formulas below to a CNF system.
(a) $(r \wedge \neg s \wedge t) \rightarrow (\neg r \vee \neg v)$.
(b) $(\neg r \wedge s) \rightarrow (v \wedge \neg w \wedge z)$.
(c) $(r \vee s \vee \neg t) \rightarrow (p \wedge \neg q \wedge z)$.
(d) $(\neg r \vee s \vee \neg t) \rightarrow (\neg p \vee \neg q \vee z)$.
(e) $(p \wedge r) \vee (r \wedge s) \vee (u \wedge v)$.
(f) $[(p \vee r) \wedge (r \vee \neg s)] \vee \neg(u \wedge \neg w)$.
(g) $\neg[(p \vee \neg r) \wedge (\neg r \vee s)] \vee u$.

(2.9.4) (Conversion of a DNF system to an equivalent CNF system) Define a CNF system T to be *equivalent* to a DNF system S if the following conditions are satisfied:
(i) T contains the variables of S plus some additional auxiliary variables.
(ii) Any satisfying solution for S can be extended to one for T by assignment of unique *True/False* values to the auxiliary variables.
(iii) Any satisfying solution for T becomes one for S if one disregards the *True/False* values for the auxiliary variables.

A method for determining an equivalent CNF system T from a given DNF system S is as follows. Suppose $S = \bigvee_{i=1}^{k} S_i$, where each S_i is a conjunction of literals. Then T has the variables of S plus k additional auxiliary variables, say $t_1, t_2, \ldots, t_k$. For each i, the auxiliary variable t_i corresponds to the DNF clause S_i. The CNF clauses of T are derived as follows. First, the clause $t_1 \vee t_2 \vee \ldots \vee t_k$ is needed. Second, for $i = 1, 2, \ldots, k$, the formula $t_i \leftrightarrow S_i$ is converted to CNF form using the distributive law, as

described in Section 2.2. The CNF clauses constructed by the two steps are the CNF clauses of T.

(a) Prove that the CNF system T derived from the DNF system S is equivalent to S.

(b) Apply the method to convert the DNF system $(s_1 \wedge \neg s_2 \wedge s_3) \vee (v_1 \wedge \neg v_2 \wedge \neg v_3) \vee (w_1 \wedge w_2 \wedge \neg w_3)$ to an equivalent CNF system.

(c) Convert the DNF example system of (b) to a CNF system just by using the distributive law. Compare the CNF system obtained for (b) with the one found here and confirm that the CNF system of (b) is substantially smaller.

(d) Derive an improved version of the conversion method that takes advantage of cases where some DNF clauses S_i of S consist of just one literal. (*Hint*: For each such S_i, the literal of S_i can play the role of the auxiliary variable t_i. Thus, t_i need not be introduced.)

(e) Convert $[(p \vee q \vee r) \rightarrow (s \wedge \neg t)] \vee (\neg u \wedge v)$ to CNF.

(2.9.5) Let *persons* be the set with elements *Mary*, *John*, and *Steve*, and let *subject* be the set with elements *geology*, *physics*, and *mathematics*. Let *takes_class*(x, y) be the predicate representing that $x \in$ *persons* takes a class in the area $y \in$ *subject*. Write first-order logic formulas for the following statements, then convert each such statement into a propositional formula.

(a) At least one of *Mary*, *John*, and *Steve* takes a class in *geology*, *physics*, or *mathematics*.

(b) *Mary*, *John*, and *Steve* take a class in *mathematics*.

To express the statements below, define appropriate subsets of *persons* and *subjects*.

(c) At least one of *Mary* and *John* takes a class in *geology* or *physics*.

(d) *Mary* and *Steve* take classes in *mathematics* and *physics*.

(2.9.6) Classify the following formulas as tautology, contradiction, satisfiable, or unsatisfiable.

(a) $(s \wedge t) \vee (\neg s \wedge \neg t)$.

(b) $(s \vee t) \wedge (\neg s \vee \neg t)$.

(c) $s \leftrightarrow \neg s$.

(d) $(s \leftrightarrow t) \wedge (t \leftrightarrow v) \wedge (v \leftrightarrow \neg s)$.

(2.9.7) Show that a propositional formula S is a CNF system if and only if $\neg S$ is a DNF system.

(2.9.8) Consider the following cases of S and T.

(i) $S = (p \rightarrow q) \wedge (q \rightarrow r)$; $T = p \rightarrow r$.

(ii) $S = (\neg p \vee \neg q) \wedge (p \vee r)$; $T = \neg q \vee r$.

(iii) $S = (p \wedge q) \vee (q \wedge r)$; $T = q \vee r$.

(a) Convert S of (i) and (iii) to a CNF system.

(b) For each one of the cases (i)–(iii) of S and T, answer the following

questions by trying out all possible *True/False* values for the variables.
Is T a theorem of S?
Does S have a satisfying solution for which T has the value *True*?
Is $\neg T$ a theorem of S?
Does S have a satisfying solution for which T has the value *False*?

(2.9.9)
(a) Prove that a CNF system is a tautology if and only if for every clause of the system there is some variable such that both that variable and its negation occur in the clause.
(b) Prove that a DNF system is a contradiction if and only if for every clause of the system there is some variable such that both that variable and its negation occur in the clause.
(c) Prove or disprove: For any propositional formulas S and T, either T or $\neg T$ is a theorem of S.

Note: (a) and (b) show that testing whether a CNF clause is a tautology or whether a DNF system is a contradiction can be done by inspection.

(2.9.10) Solve the following MINSAT instances by trying out all possible *True/False* values for the variables. The CNF system S is one of (a), (b), and (c).
(a) $S = (\neg p \vee q) \wedge (\neg q \vee r) \wedge q$.
(b) $S = (\neg p \vee \neg q) \wedge (q \vee r) \wedge p$.
(c) $S = (p \vee q) \wedge (q \vee r) \wedge \neg q$.
The cost for the value *True* for each one of the variables p, q, and r is 1, and the cost for the value *False* for these variables is 0.

(2.9.11) Let S be a satisfiable formula with variables $s_1, s_2, \ldots, s_n$ and $t_1, t_2, \ldots, t_k$. You have been asked to decide for $j = 1, 2, \ldots, k$ whether $T = t_j$ is a theorem of S. Instead of answering that question directly by solving k satisfiability problems, you assign a cost of 1 to the value *True* for $t_1, t_2, \ldots, t_k$ and assign a cost of 0 to the value *False* for these variables. You assign a cost of 0 to both *True* and *False* for the variables $s_1, s_2, \ldots, s_n$. You solve the MINSAT instance so derived from S. Suppose that, in the least-cost solution so found, the variables $t_1, t_2, \ldots, t_l$ have the value *False* and that the variables $t_{l+1}, t_{l+2}, \ldots, t_k$ have the value *True*.
(a) What can you conclude at this point?
(b) How would you use the insight gained from the MINSAT solution to partially answer the original question about $T = t_j$, for $j = 1, 2, \ldots, k$?

(*Hint*: See Chapter 5 about accelerated theorem proving.)

(2.9.12) Let a MINSAT instance define the costs for *True* and *False* for one of the variables, say, p, to be α and β, respectively.
(a) Suppose each occurrence of p (resp. $\neg p$) is replaced by $\neg p$ (resp. p). (This is called a replacement of p by its *complement*.) Further assume

that the new MINSAT instance has the same costs as the original MINSAT instance, except that for p the roles of α and β are reversed. Show that an optimal solution for the new MINSAT instance becomes an optimal solution for the original MINSAT instance and *vice versa* when the optimal *True/False* value for p is replaced by the opposite value.

(b) Show that, if in the original MINSAT instance one replaces α by $\alpha' = \alpha - \beta$ and β by $\beta' = 0$, then an optimal solution for the new MINSAT instance is an optimal solution for the original instance and *vice versa*.

(c) Using (a) and (b), show that any MINSAT instance can be transformed to another MINSAT instance where the costs for each variable are nonnegative for the value *True* and are 0 for the value *False*. Furthermore, any optimal solution for the new MINSAT instance can be transformed to an optimal solution for the original MINSAT instance or *vice versa* by changing, for a specified set of variables, the optimal *True/False* values to the opposite values.

(2.9.13) (Optional programming project) There are two options if the reader wants to avoid the manual checking of cases demanded in some of the exercises. First, the reader may acquire a commercially or publicly available software package for SAT and MINSAT to solve those problems. Any search engine on the Internet will point to such software. Second, the user may carry out Steps (a) and (b) below to create simple computer programs for solving small SAT and MINSAT instances.

(a) (Program for small SAT instances) Code a simple program that accepts the CNF systems of small SAT instances, say, with up to 10 variables and 100 clauses. The program enumerates all possible *True/False* values for the variables and checks which of the assignments constitute a satisfying solution. The output of the program either is "the given SAT instance is unsatisfiable" or consists of a satisfying solutions.

(b) (Program for small MINSAT instances) Extend the program of (a) so that small MINSAT instances can be solved. Thus, cost values for *True* and *False* for each variable are part of the input, and the output either is the statement "the given MINSAT instance is unsatisfiable" or consists of a satisfying solution whose total cost is minimum.

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 3

Variations of SAT and MINSAT

3.1 Overview

We establish variations of problem SAT (2.4.1) and problem MINSAT (2.6.1). These variations arise from several tasks, for example, when one must eliminate erroneous clauses from a CNF system or must determine the reasons for a decision. For each problem considered here, a CNF system S is given, and one must derive another CNF system S' from S having specified features. In almost all cases, the feature demands a certain maximality or minimality. We define these two terms.

Let X be a set, and define P to be any property of sets. Then a subset $X' \subseteq X$ is *maximal* (resp. *minimal*) *with respect to* P if X' has property P and if every subset of X that properly contains X' (resp. that is properly contained in X') does not have property P. For example, let $X = \{a, b, c\}$. Declare a subset $X' \subseteq X$ to have property P if X' is a proper subset of X that contains the element a or has exactly two elements. Thus, the sets $\{a\}$, $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$ are precisely the subsets of X having property P. Then the sets $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$ are the maximal subsets of X with respect to P, and the sets $\{a\}$ and $\{b, c\}$ are the minimal subsets of X with respect to that property.

We extend these definitions to CNF systems S using satisfiability or unsatisfiability as property P. Let S' be a CNF subsystem of S obtained by deletion of some clauses. Then S' is *maximal with respect to satisfiability* (resp. *minimal with respect to unsatisfiability*) if S' is satisfiable (resp. unsatisfiable) and if every CNF subsystem of S that can be obtained from

S' by adding (resp. deleting) clauses, does not have that property. The problem of finding such a maximal (resp. minimal) S' under the additional constraint that S' must contain a specified CNF subsystem $\overline{S}$ of S, is called MAXCLS SAT (resp. MINCLS UNSAT).

We define two additional problems called MAXVAR SAT and MINVAR UNSAT. Let us call the assignment of *True* or *False* to a variable a *fixing* of that variable. Declare the undoing of a fixing a *freeing* of a variable. An instance of MAXVAR SAT or MINVAR UNSAT consists of a CNF system S and a set T whose members specify *True/False* values for some of the variables of S. For example, we may have $T = \{p = \mathit{True}, q = \mathit{False}, r = \mathit{True}\}$. The set T is such that, when the variables of S are fixed according to T, then an unsatisfiable CNF system results. For both MAXVAR SAT and MINVAR UNSAT, the solution consists of a subset of T with certain features, as follows.

In the case of MAXVAR SAT, fixing of the variables according to the selected subset must produce a satisfiable CNF system, and fixing of any additional variable in agreement with T must lead to an unsatisfiable CNF system.

In the case of MINVAR UNSAT, fixing according to the selected subset must produce an unsatisfiable CNF system, and freeing of any of those fixings must lead to a satisfiable CNF system.

We show that MAXVAR SAT and MINVAR UNSAT effectively are special cases of MAXCLS SAT and MINCLS UNSAT, respectively. Indeed, one may represent the fixing of a variable t to *True* (resp. *False*) by adding the clause t (resp. $\neg t$). The freeing of a variable corresponds to the removal of such a clause. When this way of fixing and freeing of variables is used in MAXVAR SAT and MINVAR UNSAT, then it is easy to see that these problems become special cases of MAXCLS SAT and MINCLS UNSAT.

We should emphasize that an instance of MAXCLS SAT, MINCLS UNSAT, MAXVAR SAT, or MINVAR UNSAT may have a number of solutions. For example, for a given instance S of MAXCLS SAT, there may be several maximal satisfiable subsystems S'. Indeed, the number of clauses in such S' may differ significantly. Fortunately, this fact is often of little importance when we use solutions to these problems in applications.

The chapter also covers a variation of MINSAT called MAXSAT. The latter problem involves a CNF system S and rational nonnegative *weights* associated with the clauses. One must find a subsystem of satisfiable clauses for which the sum of the corresponding weights is maximum. We include simple transformations that convert any instance of MAXSAT to one of MINSAT and *vice versa*.

The algorithms of this chapter, as well as of subsequent chapters, frequently invoke two schemes called Algorithms SOLVE SAT and SOLVE MINSAT as subroutines. Recall from Chapter 2 that, for the purposes of this book, the specific form of Algorithms SOLVE SAT and SOLVE MIN-

SAT is not important, as long as they are reasonably effective. Section 2.8 of Chapter 2 contains references for such schemes.

The chapter proceeds as follows.

Sections 3.2–3.6 cover MAXCLS SAT, MINCLS UNSAT, MAXVAR SAT, MINVAR UNSAT, and MAXSAT, in that order. The sections also include small example applications for the five problems. For a first reading, it suffices that the reader scans Section 3.3, which deals with the MINCLS UNSAT case.

Section 3.7 points out further reading.

Section 3.8 contains exercises.

3.2 Problem MAXCLS SAT

We motivate the problem by an example application.

Example

Let the CNF system S given by

$$\begin{aligned} &p \vee q \vee r \\ &\neg p \vee \neg q \vee \neg r \end{aligned} \tag{3.2.1}$$

be a partial logic formulation of some real-world situation. Besides the clauses of (3.2.1), somebody has supplied the clauses

$$\begin{aligned} &\neg p \vee r \\ &p \vee \neg q \\ &q \vee \neg r \end{aligned} \tag{3.2.2}$$

and has claimed that the combined clauses of (3.2.1) and (3.2.2), that is,

$$\begin{aligned} &p \vee q \vee r \\ &\neg p \vee \neg q \vee \neg r \\ &\neg p \vee r \\ &p \vee \neg q \\ &q \vee \neg r \end{aligned} \tag{3.2.3}$$

give an accurate logic formulation for the given real-world situation.

However, when we check the CNF system of (3.2.3) for satisfiability, we discover that it is unsatisfiable. Thus, the logic formulation of (3.2.3) is inconsistent, and we should not add all clauses of (3.2.2) to (3.2.1). Which clauses of (3.2.2), if any, should be added? Chapter 5 addresses this difficult problem. Here, we take a simplistic approach, as follows.

Solution of Example

We tentatively add the clauses of (3.2.2) one by one to (3.2.1). After the addition of one clause, we check the enlarged CNF system for satisfiability. If the enlarged CNF system is satisfiable, we permanently accept the added clause. Otherwise, we remove that clause again.

The method requires that we select an order in which the clauses of (3.2.2) are added to (3.2.1). For simplicity, let us add the clauses in the order in which they are listed in (3.2.2). We carry out the steps of the above scheme.

First, we tentatively add the clause $\neg p \vee r$ of (3.2.2) to (3.2.1), getting

$$\begin{aligned} & p \vee q \vee r \\ & \neg p \vee \neg q \vee \neg r \\ & \neg p \vee r \end{aligned} \tag{3.2.4}$$

That CNF system is satisfiable. For example, $p = q = \mathit{False}$ and $r = \mathit{True}$ is a satisfying solution. Hence, we permanently accept the added clause.

Next, we tentatively add the clause $p \vee \neg q$ of (3.2.2) to (3.2.4), getting

$$\begin{aligned} & p \vee q \vee r \\ & \neg p \vee \neg q \vee \neg r \\ & \neg p \vee r \\ & p \vee \neg q \end{aligned} \tag{3.2.5}$$

The new CNF system is satisfiable since the values $p = q = \mathit{False}$ and $r = \mathit{True}$ for (3.2.4) satisfy the added clause. Thus, we permanently accept the added clause.

Finally, we add the third clause of (3.2.2) to (3.2.5). The resulting CNF system is given by (3.2.3), which we already know to be unsatisfiable. Hence, we remove the clause again. Since all clauses of (3.2.2) have been processed, we stop. The final CNF system is given by (3.2.5).

Problem Definition

We introduce some notation. Let S be the CNF system given by (3.2.3), define $\overline{S}$ to be the CNF system of (3.2.1), and let S' be the CNF subsystem of S given by (3.2.5). Then S is unsatisfiable, $\overline{S}$ is satisfiable, and S' is obtained from $\overline{S}$ by adding some clauses of S. Moreover, S' is satisfiable, but addition of any clause of S that is not in S' results in an unsatisfiable CNF system. Hence, S' is a satisfiable subsystem of S that is maximal under clause addition. Section 3.1 says finding such S' is the task of problem MAXCLS SAT. For later reference, we include a compact specification of that problem using the present notation.

(3.2.6) MAXCLS SAT

Instance: Unsatisfiable CNF system S. A satisfiable subsystem $\overline{S}$ obtained from S by the deletion of some clauses.

Solution: A maximal satisfiable subsystem S' obtained from $\overline{S}$ by adding clauses of S.

Solution Algorithm

We formalize the above calculations to obtain a solution algorithm for MAXCLS SAT. In the description of the algorithm, indeed of all algorithms of the book, we use the convention that any statement in parentheses is a comment.

(3.2.7) Algorithm SOLVE MAXCLS SAT. *Solves instance of* MAXCLS SAT.

Input: Unsatisfiable CNF system S. A satisfiable subsystem $\overline{S}$ obtained from S by the deletion of some clauses.

Output: A maximal satisfiable subsystem S' obtained from $\overline{S}$ by adding clauses of S.

Requires: Algorithm SOLVE SAT.

Procedure:

1. Select an order of the clauses of S that are not in $\overline{S}$. (The choice of the order depends on the application. For example, a random order may be appropriate.) Initialize $S' = \overline{S}$.
2. Process the clauses of S that are not in $\overline{S}$ in the selected order one by one.
   For each such clause, do the following: Update S' by adding the clause; use Algorithm SOLVE SAT to decide satisfiability of S'; if S' is unsatisfiable, remove the clause from S' again.
3. Output the final S' as the desired CNF subsystem of S.

Proof of Validity. By the satisfiability of $\overline{S}$ and the rules of Step 2, the final S' of Step 3 is satisfiable. Consider any clause of S that is not in the final S'. When the clause was processed in Step 2, it was added to S' on hand, say S'', was determined to lead to unsatisfiability, and was removed again. Since the final S' has S'' as subsystem, addition of the clause to the final S' must cause unsatisfiability as well. Thus, the final S' is a satisfiable subsystem of S that is maximal under the addition of clauses. □

We emphasize that different orders selected in Step 1 of Algorithm SOLVE MAXCLS SAT may result in different solutions S'. Indeed, the number of clauses in such S' may vary considerably. The analogous observation applies to the problems MINCLS UNSAT, MAXVAR SAT, and

MINVAR UNSAT discussed in subsequent sections. Fortunately, this fact is often of little importance when we use solutions to these problems in applications.

We turn to the problem MINCLS UNSAT.

3.3 Problem MINCLS UNSAT

We begin with an example application. For a more elaborate example, see the treatment of inconsistent clauses in Section 5.4 of Chapter 5.

Example

Let R be the following CNF system.

$$\begin{aligned} &p \vee q \vee r \\ &\neg p \vee \neg q \vee \neg r \\ &\neg p \vee r \\ &p \vee \neg q \end{aligned} \tag{3.3.1}$$

We want to decide whether the statement

$$r \rightarrow \neg q \tag{3.3.2}$$

is a theorem of R. We negate the statement (3.3.2), add that negated statement to R, and test the resulting CNF system for satisfiability. If the enlarged CNF system is unsatisfiable, then the statement (3.3.2) is a theorem of R, and otherwise it is not. We carry out these steps.

Since $r \rightarrow \neg q = \neg r \vee \neg q$, the negation of (3.3.2) is

$$r \wedge q \tag{3.3.3}$$

When we add statement (3.3.3) to the CNF system R of (3.3.1), we obtain the following CNF system S.

$$\begin{aligned} &p \vee q \vee r \\ &\neg p \vee \neg q \vee \neg r \\ &\neg p \vee r \\ &p \vee \neg q \\ &r \\ &q \end{aligned} \tag{3.3.4}$$

The reader may want to verify that S is unsatisfiable. Hence, $r \rightarrow \neg q$ is a theorem of R.

Someone might ask us to explain why statement (3.3.2) is a theorem of R. More specifically, the person might ask us to point out a minimal subset of the clauses of R that establish statement (3.3.2) to be a theorem.

Solution of Example

We find such a minimal subset by iteratively examining each clause of R in (3.3.4).

In the first iteration, we remove from (3.3.4) the clause $p \vee q \vee r$, which is the first clause of R. We get

$$\begin{array}{l} \neg p \vee \neg q \vee \neg r \\ \neg p \vee r \\ p \vee \neg q \\ r \\ q \end{array} \tag{3.3.5}$$

That CNF system turns out to be unsatisfiable. This means that the statement (3.3.2) is a theorem of the following subsystem of R.

$$\begin{array}{l} \neg p \vee \neg q \vee \neg r \\ \neg p \vee r \\ p \vee \neg q \end{array} \tag{3.3.6}$$

Put differently, we only need the clauses of R given by (3.3.6) to explain why (3.3.2) is a theorem of R.

Maybe even fewer clauses of R constitute an explanation. To find out, we remove from (3.3.5) the clause $\neg p \vee \neg q \vee \neg r$, which is the second clause of R. We get

$$\begin{array}{l} \neg p \vee r \\ p \vee \neg q \\ r \\ q \end{array} \tag{3.3.7}$$

The CNF system of (3.3.7) has the satisfying solution $p = q = r = \mathit{True}$. Thus, (3.3.2) cannot be explained to be a theorem of R just using the clauses

$$\begin{array}{l} \neg p \vee r \\ p \vee \neg q \end{array} \tag{3.3.8}$$

of R. Hence, we declare the clause $\neg p \vee \neg q \vee \neg r$ to be part of the explanation that (3.3.2) is a theorem of R.

There are two other clauses of (3.3.5) that are potentially not needed: the clauses $\neg p \vee r$ and $p \vee \neg q$, which are the third and fourth clauses of R.

Hence, we remove $\neg p \vee r$ from (3.3.5), getting

$$\begin{array}{l} \neg p \vee \neg q \vee \neg r \\ p \vee \neg q \\ r \\ q \end{array} \tag{3.3.9}$$

That CNF system is unsatisfiable. Thus, the statement (3.3.2) is a theorem of the following subsystem of R.

$$\begin{array}{l} \neg p \vee \neg q \vee \neg r \\ p \vee \neg q \end{array} \tag{3.3.10}$$

We carry out one more iteration. This time, we remove the clause $p \vee \neg q$ from (3.3.9), getting

$$\begin{array}{l} \neg p \vee \neg q \vee \neg r \\ r \\ q \end{array} \tag{3.3.11}$$

That CNF system is clearly satisfiable. Hence, the statement (3.3.2) cannot be explained to be a theorem of R just using the single clause

$$\neg p \vee \neg q \vee \neg r \tag{3.3.12}$$

of R. We conclude that the clauses $\neg p \vee \neg q \vee \neg r$ and $p \vee \neg q$ of (3.3.10) by themselves establish the statement (3.3.2) to be a theorem of R. Furthermore, no proper subsystem of (3.3.10) obtained by the deletion of clauses can do so.

Problem Definition

We introduce some notation. Let S be the CNF system of (3.3.4), and define $\overline{S}$ to be the subsystem of S representing the negation of (3.3.2); that is, $\overline{S}$ is given by (3.3.3). Let S' be the CNF subsystem of S given by (3.3.9). Then S' is an unsatisfiable subsystem of S that is minimal under the deletion of clauses not in $\overline{S}$. According to Section 3.1, finding such a minimal subsystem is the task of problem MINCLS UNSAT. For later reference, we list a compact statement of that problem using the present notation.

(3.3.13) MINCLS UNSAT

Instance: Unsatisfiable CNF system S. A subsystem $\overline{S}$ obtained from S by the deletion of some clauses.

Solution: A minimal unsatisfiable subsystem S' obtained from S by the deletion of clauses not in $\overline{S}$.

Solution Algorithm

The above calculations can be formalized to the following solution algorithm for MINCLS UNSAT.

(3.3.14) Algorithm SOLVE MINCLS UNSAT. *Solves instance of* MINCLS UNSAT.

Input: Unsatisfiable CNF system S. A subsystem $\overline{S}$ obtained from S by the deletion of some clauses.

Output: A minimal unsatisfiable subsystem S' obtained from S by the deletion of clauses not in $\overline{S}$.

Requires: Algorithm SOLVE SAT.

Procedure:

1. Select an order of the clauses of S that are not in $\overline{S}$. (The choice of the order depends on the application. For example, a random order may be appropriate.) Initialize $S' = S$.
2. Process the clauses of S that are not in $\overline{S}$ in the selected order one by one.
   For each such clause, do the following: Update S' by deleting the clause; use Algorithm SOLVE SAT to decide satisfiability of S'; if S' is satisfiable, add the clause to S' again.
3. Output the final S' as the desired CNF subsystem of S.

Exercise (3.8.3) asks the reader to supply the proof of validity, which is similar to that for Algorithm SOLVE MAXCLS SAT (3.2.7).

We discuss problem MAXVAR SAT next.

3.4 Problem MAXVAR SAT

We begin with an example application.

Example

Suppose the CNF system S given by

$$\begin{aligned} &p \vee \neg q \vee r \\ &\neg p \vee q \\ &\neg p \vee \neg r \vee \neg s \end{aligned} \tag{3.4.1}$$

is the logic formulation of some real-world process. Somebody hands us the *True/False* values $p = True$, $q = False$, and $r = True$ and demands that

we use these values to obtain a satisfying solution for S. However, when we fix the variables p, q, and r of S to these values, we discover that there is no satisfying solution. Indeed, the second clause cannot be satisfied since $p = \mathit{True}$ and $q = \mathit{False}$.

The person now changes the demand and asks that we find a satisfying solution so that a maximal subset of the given *True/False* values for p, q, and r is used. We proceed to find such a satisfying solution.

Solution of Example

First, we tentatively fix the variable p to the given value *True* and check if S is satisfiable. This is indeed so. For example, $p = q = s = \mathit{True}$ and $r = \mathit{False}$ is a satisfying solution. Accordingly, we permanently enforce $p = \mathit{True}$.

Second, we tentatively fix q to the given value *False*. Since $p = \mathit{True}$ and $q = \mathit{False}$ leave the second clause unsatisfied, the fixing of q to *False* does not lead to a satisfying solution. Hence, we free q again.

Third, we tentatively fix r to the given value *True*. Since $p = r = \mathit{True}$ can be extended to the satisfying solution $p = q = r = \mathit{True}$ and $s = \mathit{False}$, we permanently enforce $r = \mathit{True}$.

At this point, all specified *True/False* values have been processed, and the values $p = r = \mathit{True}$ have been retained. These values represent a maximal subset of the originally specified values for which an extension to a satisfying solution is possible.

Problem Definition

If T is the set of originally specified *True/False* values, that is, $T = \{p = \mathit{True}, q = \mathit{False}, r = \mathit{True}\}$, then $T' = \{p = \mathit{True}, r = \mathit{True}\}$ is the desired maximal subset of T. According to Section 3.1, finding such T' is the task of problem MAXVAR SAT. For later reference, we define that problem using the present notation plus an added set $\overline{T}$ that for the above example would be empty.

(3.4.2) MAXVAR SAT

Instance: A satisfiable CNF system S. A set T of *True/False* values for some of the variables of S. A subset $\overline{T}$ of T. The CNF system S becomes unsatisfiable if the variables are fixed according to T, but is satisfiable if the variables are fixed according to $\overline{T}$.

Solution: A maximal subset T' of T such that T' includes $\overline{T}$ and such that fixing of the variables of T' results in a satisfiable CNF system.

Solution Algorithm

The above example calculations can be formalized to a solution algorithm for MAXVAR SAT.

(3.4.3) Algorithm SOLVE MAXVAR SAT. *Solves instance of* MAXVAR SAT.

Input: A satisfiable CNF system S. A set T of *True/False* values for some of the variables of S. A subset $\overline{T}$ of T. The CNF system S becomes unsatisfiable if the variables are fixed according to T, but is satisfiable if the variables are fixed according to $\overline{T}$.

Output: A maximal subset T' of T such that T' includes $\overline{T}$ and such that fixing of the variables of T' results in a satisfiable CNF system.

Requires: Algorithm SOLVE SAT.

Procedure:

1. Select an order of the *True/False* values listed in T but not in $\overline{T}$. (The choice of the order depends on the application. For example, a random order may be appropriate.) Fix variables in S according to $\overline{T}$. Initialize $T' = T$.
2. Process the *True/False* values listed in T but not in $\overline{T}$ in the selected order one by one.
   For each such *True/False* value, do the following: Update S by fixing the specified variable to the *True/False* value; use Algorithm SOLVE SAT to decide satisfiability of the updated S; if S is unsatisfiable, free the specified variable again, and remove the *True/False* listing for that variable from T'.
3. Output the final T' as the desired list of *True/False* values.

Exercise (3.8.5)(a) demands that problem MAXVAR SAT (3.4.2) and Algorithm SOLVE MAXVAR SAT (3.4.3) are proved to be special cases of problem MAXCLS SAT (3.2.6) and Algorithm SOLVE MAXCLS SAT (3.2.7), respectively. That proof also validates the above algorithm.

We move on to problem MINVAR UNSAT.

3.5 Problem MINVAR UNSAT

The problem MINVAR UNSAT arises in situations where we want to acquire as little information as possible to prove a given conclusion. We begin with an example application.

Example

Let the CNF system S given by

$$\begin{aligned} & p \vee \neg q \vee r \vee s \\ & \neg p \vee \neg q \vee s \vee t \\ & p \vee \neg r \vee s \end{aligned} \tag{3.5.1}$$

model a real-world setting. The variables p, q, and r represent questions that can be answered by "yes" or "no;" the corresponding values for the variables are *True* and *False*, respectively. The variables s and t represent conclusions. Given *True/False* values for p, q, and r, we attempt to prove if s or t is a theorem.

For example, suppose we know that $p = False$ and $q = r = True$. To check if s is a theorem for these values, we fix p, q, and r to the given values, enforce the added clause $\neg s$ by fixing $s = False$, and check if the updated S is satisfiable. The conclusion s must hold if and only if the updated S is unsatisfiable.

Since the given values do not satisfy the third clause of (3.5.1), we have the case where conclusion s must hold. The analogous process applies to the conclusion t. In that case, it turns out that t cannot be proved to be a theorem since $p = t = False$ and $q = r = s = True$ constitute a satisfying solution of (3.5.1).

We change the scenario and confine ourselves to the conclusion s. Assume that as yet we do not have *True/False* values for p, q, and r. However, we can obtain a *True/False* value for any one of these variables at some cost. To decide whether s is a theorem, we employ the following scheme. We ask for the *True/False* value of one of the variables p, q, and r and then attempt to prove s. If s can be proved, then we have the desired result and stop. Otherwise, we ask for the *True/False* value of a second variable of p, q, and r and again try to prove s. Proceeding in this fashion, either we prove s to be a theorem at some point, or we conclude that s cannot be proved.

The efficiency of the process depends on the order in which we ask for the *True/False* values of p, q, and r. In the absence of prior knowledge, selecting a sequence that minimizes total cost of the process is not possible. However, if the above process is carried out repeatedly—say, in various real-world settings—then one has an opportunity to learn a cost-effective selection. Details are presented in Chapter 10. Here, we cover one aspect of the learning described in that chapter, using the above example and the given *True/False* values $p = False$ and $q = r = True$. We know that, for these values, s can be proved to be a theorem. But there may be a proper subset of these *True/False* values that already allows us to prove s. If there is such a subset, and if we had known about it in advance, then we

could have proved s without acquiring the remaining *True/False* values. Of course, we would be interested in minimal subsets. Let us find one such subset.

Solution of Example

To start, we fix in S the variables p, q, and r to the given values $p = \mathit{False}$ and $q = r = \mathit{True}$ and also impose $s = \mathit{False}$. We know that, for these values, S is unsatisfiable. We relax these fixings in an iterative scheme, as follows.

First, we tentatively free p and check for satisfiability. We discover that $p = q = r = t = \mathit{True}$ and $s = \mathit{False}$ is a satisfiable solution. Hence, we permanently fix $p = \mathit{False}$.

Second, we tentatively free q and check for satisfiability. Since the given values $p = s = \mathit{False}$ and $r = \mathit{True}$ leave the third clause of (3.5.1) unsatisfied, we can prove s without knowing the value of q. Accordingly, we leave q free.

Third, we tentatively free r. For that case, $p = q = r = s = t = \mathit{False}$ is a satisfying solution, and s cannot be proved. Hence, we permanently enforce $r = \mathit{True}$.

Evidently, $p = \mathit{False}$ and $r = \mathit{True}$ comprise a minimal subset of the given *True/False* values for p, q, and r such that s can be proved when the *True/False* values of that subset are enforced.

Problem Definition

Let $T = \{p = \mathit{False}, q = \mathit{True}, r = \mathit{True}, s = \mathit{False}\}$. The set T provides a listing of *True/False* values whose fixing in S results in an unsatisfiable CNF system. Define $\overline{T} = \{s = \mathit{False}\}$ and $T' = \{p = \mathit{False}, r = \mathit{True}, s = \mathit{False}\}$. Then T' is a minimal subset of T such that T' includes $\overline{T}$ and such that fixing variables in S according to T' results in an unsatisfiable CNF system. According to Section 3.1, finding such a subset is the task of problem MINCLS UNSAT. For later reference, we state that problem using the present notation.

(3.5.2) MINVAR UNSAT

Instance: CNF system S. A set T of *True/False* values for some of the variables of S. A subset $\overline{T}$ of T. The CNF system S becomes unsatisfiable if the variables are fixed according to T.

Solution: A minimal subset T' of T such that T' includes $\overline{T}$ and such that S becomes unsatisfiable if the variables are fixed according to T'.

Solution Algorithm

The above calculations for the example case can be formalized to a solution algorithm for MINVAR UNSAT.

(3.5.3) Algorithm SOLVE MINVAR UNSAT. *Solves instance of* MINVAR UNSAT.

Input: CNF system S. A set T of *True/False* values for some of the variables of S. A subset $\overline{T}$ of T. The CNF system S becomes unsatisfiable if the variables are fixed according to T.

Output: A minimal subset T' of T such that T' includes $\overline{T}$ and such that S becomes unsatisfiable if the variables are fixed according to T'.

Requires: Algorithm SOLVE SAT.

Procedure:

1. Select an order of the *True/False* values listed in T but not in $\overline{T}$. (The choice of the order depends on the application. For example, a random order may be appropriate.) Fix variables in S according to T. Initialize $T' = \overline{T}$.
2. Process the *True/False* values listed in T but not in $\overline{T}$ in the selected order one by one.
   For each such *True/False* value, do the following: Update S by freeing the specified variable; use Algorithm SOLVE SAT to decide satisfiability of the updated S; if S is satisfiable, fix the specified variable again to the listed value, and add the listed *True/False* to T'.
3. Output the final T' as the desired list of *True/False* values.

Exercise (3.8.3) asks the reader to show that problem MINVAR UNSAT (3.5.2) and Algorithm SOLVE MINVAR UNSAT (3.5.3) are special cases of problem MINCLS UNSAT (3.3.13) and Algorithm SOLVE MINCLS UNSAT (3.3.14), respectively. That result also validates the above algorithm.

Up to this point, we have considered variations of SAT. In the next section, we encounter a variation of MINSAT called MAXSAT.

3.6 Problem MAXSAT

Recall that the problem MAXCLS SAT (3.2.6) asks one to find a maximal subset of satisfiable clauses of an unsatisfiable CNF system S. There may be several such maximal subsets, and their size may vary. Suppose we want a maximal subset that has largest size. We call such a subset a *max cardinality satisfiable subset*. There is a more general case.

Problem Definition

We are given an unsatisfiable CNF system S. With each clause i of S, a nonnegative rational number d_i is associated and called the *weight* of clause i. One is to find a satisfiable subset of the clauses of S such that the sum of the weights of the selected clauses is maximized. We call that subset a *max weight satisfiable subset*. The problem of finding such a subset is called MAXSAT. Thus, that problem can be summarized as follows.

(3.6.1) MAXSAT

Instance: Unsatisfiable CNF system S. For each clause i, a nonnegative rational weight d_i.

Solution: A satisfiable subset S' of the clauses of S for which the sum of the corresponding weights is maximum.

Each instance of MAXSAT is readily transformed to an instance of MINSAT. We demonstrate this first by an example.

Example Transformation to MINSAT

Let the CNF system S and the weights be given by

$$\begin{array}{ll} p \vee \neg q & d_1 = 7 \\ \neg q \vee r & d_2 = 9 \\ \neg p \vee \neg r & d_3 = 2 \\ \neg p & d_4 = 1 \\ q & d_5 = 4 \end{array} \tag{3.6.2}$$

Evidently, clauses 1, 4, and 5 of (3.6.2) cannot be simultaneously satisfied. Thus, S is unsatisfiable. Indeed, by direct enumeration of cases, it is not difficult to confirm that the subsystem S' consisting of clauses 1, 2, and 5 is satisfiable and has maximum weight among the satisfiable subsystems. The values $p = q = r = \mathit{True}$ constitute a satisfying solution for that subset. We confirm these results once we have discussed the transformation to an equivalent MINSAT problem.

To effect the transformation to MINSAT, we add to each clause i of S a new variable u_i. For each original variable of S, we declare the cost of *True* or *False* to be 0. For each variable u_i, we declare the cost of *True* to be d_i and the cost of *False* to be 0. Thus, we get the following MINSAT

instance, where "s.t." stands for "subject to."

$$
\begin{array}{ll}
\min & \sum_{u_i = True} d_i \\
\text{s. t.} & p \vee \neg q \vee u_1 \\
& \neg q \vee r \vee u_2 \\
& \neg p \vee \neg r \vee u_3 \\
& \neg p \vee u_4 \\
& q \vee u_5
\end{array}
\tag{3.6.3}
$$

A bit of checking verifies that (3.6.3) has the unique optimal solution $p^* = q^* = r^* = u_3^* = u_4^* = \mathit{True}$ and $u_1^* = u_2^* = u_5^* = \mathit{False}$. The sum of the d_i for which $u_i^* = \mathit{True}$ is equal to $d_3 + d_4 = 2 + 1 = 3$. Thus, 3 is the optimal objective function value of (3.6.3).

We translate the optimal solution of the MINSAT instance (3.6.3) to one for the MAXSAT instance (3.6.2).

Take any clause i of (3.6.3) for which $u_i^* = \mathit{False}$. That clause is satisfied when the solution values for p^*, q^*, and r^* are used. Hence, the corresponding clause of (3.6.2) is satisfied by these same p^*, q^*, and r^* values.

Now consider any clause i of (3.6.3) with $u_i^* = \mathit{True}$. We claim that such a clause is not satisfied if we use the p^*, q^*, and r^* values and if we omit the u_i term. Indeed, if that clause was satisfied, we could have improved the optimal solution of (3.6.3) by defining $u_i^* = \mathit{False}$.

We conclude that the subset of clauses i of (3.6.2) for which $u_i^* = \mathit{False}$ contains precisely all clauses of (3.6.2) that are satisfied by the p^*, q^*, and r^* values. We argue that this subset has max weight and thus solves the MAXSAT instance. Suppose, to the contrary, that a satisfying solution exists for a clause subset of (3.6.2) with larger weight. That solution can be extended to a solution of (3.6.3) by defining $u_i = \mathit{False}$ for the clauses i in the subset and by declaring $u_i = \mathit{True}$ for the remaining clauses. Under this translation, a larger weight of the MAXSAT instance (3.6.2) becomes a smaller objective function value for the MINSAT instance (3.6.3). Thus, we have a solution for (3.6.3) whose objective function value is less than that of the optimal solution p^*, q^*, r^*, $u_1^*, \ldots, u_5^*$, which is not possible.

Since $u_1^* = u_2^* = u_5^* = \mathit{False}$ and $u_3^* = u_4^* = \mathit{True}$, the max weight subset of clauses of (3.6.2) consists of clauses 1, 2, and 5, and $p^* = q^* = r^* = \mathit{True}$ constitute a satisfying solution for that subset. This conclusion confirms the result obtained earlier for (3.6.2).

Solution Algorithm

Algorithm SOLVE MAXSAT below translates MAXSAT instances to MINSAT instances using the above ideas. Exercise (3.8.10) asks the reader to

prove validity of the scheme. In view of the above discussion, the proof should not be difficult.

(3.6.4) Algorithm SOLVE MAXSAT. *Solves instance of* MAXSAT.

Input: Unsatisfiable CNF system S. For each clause i, a nonnegative rational weight d_i.

Output: A satisfiable subset S' of the clauses of S for which the sum of the corresponding weights is maximum.

Requires: Algorithm SOLVE MINSAT.

Procedure:

1. Add to each clause i of S a new variable u_i, getting a CNF system R. Assign costs to the variables of R as follows. For each variable of R occurring in S, declare the cost of *True* or *False* to be 0. For each variable u_i of R, declare the cost of *True* to be d_i, and define the cost of *False* to be 0.
2. Use Algorithm SOLVE MINSAT to solve the MINSAT instance given by R and the assigned costs. For each clause i of R, let u_i^* be the optimal *True/False* value.
3. Output the subset S' of clauses i of S for which $u_i^* = False$ as an optimal solution for the MAXSAT instance.

Transformation of MINSAT to MAXSAT

One can also transform any MINSAT instance to a MAXSAT instance. Here is the three-step process.

First, use the steps of Exercise (2.9.12) to achieve an equivalent MINSAT instance, where, for each variable, the costs associated with *True* is nonnegative, and where the cost associated with *False* is 0. Let S be the CNF system of that equivalent MINSAT instance.

Second, for each variable s_j of S, add one clause to S that consists just of $\neg s_j$. Let S' be the resulting CNF system.

Third, define weights for the clauses of S' as follows. For each clause $\neg s_j$ that was added in the second step, the weight is equal to the cost of s_j associated with *True* in the MINSAT instance. Let M be the sum of these assigned weights. For each remaining clause of S', the weight is equal to $M+1$. Exercise (3.8.12) asks for a proof of validity of this transformation.

Looking Ahead

The next chapter treats additional logic problems arising from several real-world applications. These problems are called quantified SAT and MINSAT problems.

3.7 Further Reading

Many variants of SAT are covered in Kleine Büning and Lettmann (1999). For further details about MAXSAT, see Truemper (1998).

3.8 Exercises

The calculations required by some of the exercises may be accomplished with the software of Exercise (3.8.14).

(3.8.1) Verify that the CNF system of (3.2.3) is unsatisfiable.

(3.8.2) Solve the MAXCLS SAT instances given by (a) and (b) below.
(a) S is the CNF system

$$p \vee q$$
$$\neg p \vee q$$
$$p \vee \neg q$$
$$\neg p \vee \neg q$$

and $\overline{S}$ consists of the last two clauses of S.
(b) S is the CNF system

$$r \vee \neg s \vee t$$
$$\neg r$$
$$\neg s \vee \neg t$$
$$r \vee s$$
$$\neg s$$

and $\overline{S}$ consists of the last three clauses of S.

(3.8.3) Prove validity of Algorithm SOLVE MINCLS UNSAT (3.3.14).

(3.8.4) Solve the MINCLS UNSAT instances given by (a) and (b) below.
(a) S is the CNF system

$$p \vee q$$
$$\neg p \vee q$$
$$p \vee \neg q$$
$$\neg p \vee \neg q$$

and $\overline{S}$ consists of the last clause of S.

(b) S is the CNF system

$$r \vee \neg s \vee t$$
$$r \vee s$$
$$\neg s \vee \neg t$$
$$\neg r$$
$$\neg s$$

and $\overline{S}$ consists of the last two clauses of S.

(3.8.5) Each variable fixing/freeing in a CNF system S can be represented by the addition/removal of a clause with one literal. Use this fact to answer (a) and (b) below.

(a) Show that problem MAXVAR SAT (3.4.2) and Algorithm SOLVE MAXVAR SAT (3.4.3) are special cases of problem MAXCLS SAT (3.2.6) and Algorithm SOLVE MAXCLS SAT (3.2.7), respectively.

(b) Show that problem MINVAR UNSAT (3.5.2) and Algorithm SOLVE MINVAR UNSAT (3.5.3) are special cases of problem MINCLS UNSAT (3.3.13) and Algorithm SOLVE MINCLS UNSAT (3.3.14), respectively.

(3.8.6) Solve the MAXVAR SAT instances given by (a) and (b) below.

(a) S is the CNF system

$$\neg p \vee \neg q \vee r \vee s$$
$$\neg p \vee \neg q \vee \neg s$$
$$p \vee \neg q \vee r \vee s$$

$T = \{p = \mathit{True}, q = \mathit{True}, r = \mathit{False}\}$, and $\overline{T} = \emptyset$.

(b) S is the CNF system

$$p \vee q$$
$$q \vee r$$
$$r \vee s$$
$$p \vee s$$

$T = \{p = \mathit{True}, q = \mathit{False}, r = \mathit{False}\}$, and $\overline{T} = \emptyset$.

(3.8.7) Transform the instances (3.8.6)(a) and (b) of MAXVAR SAT into instances of MAXCLS SAT.

(3.8.8) Solve the MINVAR UNSAT instances given by (a) and (b) below.

(a) S is the CNF system

$$\neg p \vee q$$
$$\neg q \vee r$$
$$\neg r \vee s$$

$T = \{p = \mathit{True}, q = \mathit{True}, r = \mathit{True}, s = \mathit{False}\}$, and $\overline{T} = \{s = \mathit{False}\}$.

(b) S is the CNF system

$$p \vee q \vee r \vee s$$
$$\neg p \vee \neg q \vee s$$
$$\neg q \vee \neg r \vee s$$

$T = \{p = True, q = True, r = False, s = False\}$, and $\overline{T} = \{s = False\}$.

(3.8.9) Transform the instances (3.8.8)(a) and (b) of MINVAR UNSAT to instances of MINCLS UNSAT.

(3.8.10) Give a proof of validity of Algorithm SOLVE MAXSAT (3.6.4).

(3.8.11) Apply Algorithm SOLVE MAXSAT (3.6.4) to the MAXSAT instances of (a) and (b) below.

(a)

$p \vee q$	$d_1 = 2$
$\neg p \vee q$	$d_2 = 1$
$p \vee \neg q$	$d_3 = 5$
$\neg p \vee \neg q$	$d_4 = 3$

(b)

$r \vee \neg s \vee t$	$d_1 = 2$
$\neg s \vee \neg t$	$d_2 = 3$
$r \vee s$	$d_3 = 1$
$\neg r \vee \neg s$	$d_4 = 7$
$\neg r \vee s$	$d_5 = 6$

(3.8.12) Prove validity of the transformation from MINSAT to MAXSAT given in Section 3.6 after Algorithm SOLVE MAXSAT (3.6.4). In particular, show how any optimal solution of a MINSAT instance leads to an optimal solution of the corresponding MAXSAT instance and *vice versa*.

(3.8.13) Transform the MINSAT instances of Exercise (2.9.10) to equivalent MAXSAT instances.

(3.8.14) (Optional programming project) There are two options if the reader wishes to avoid manual solution of some of the exercises. First, the reader may obtain commercially or publicly available software for at least some of the problems. Any search engine on the Internet will point to such software. Second, the user may opt to write programs that implement the algorithms of (a)–(e) below, and then use that software to solve the exercises. The task is easy if the reader has already created programs for SAT and MINSAT as described in Exercise (2.9.13).

(a) Algorithm SOLVE MAXCLS SAT (3.2.7).

(b) Algorithm SOLVE MINCLS UNSAT (3.3.14).

(c) Algorithm SOLVE MAXVAR SAT (3.4.3).
(d) Algorithm SOLVE MINVAR UNSAT (3.5.3).
(e) Algorithm SOLVE MAXSAT (3.6.4).

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 4

Quantified SAT and MINSAT

4.1 Overview

Some decision problems of propositional logic seemingly defy a compact encoding in the format of any one of the problems discussed in Chapters 2 and 3. We say "seemingly" since at present there is much evidence but no proof that a compact encoding of these decision problems is impossible. In the first part of this chapter, we discuss three such problems. They model important and complex decision questions. In the second part, we introduce three additional problems that represent even more complicated settings.

An instance of each of the three problems of the first part involves satisfiable CNF systems R and S. The two CNF systems may have any number of variables in common. Among these variables are $l \geq 1$ special variables $q_1, q_2, \ldots, q_l$. Due to frequent occurrence of these variables in this chapter, we introduce a space-saving notation where $q_1, q_2, \ldots, q_l$ is listed as $q_1, \ldots, q_l$. As a matter of consistency, we use analogous notation for all other indexed variables.

For each of the three problems, either one must establish the presence of certain solutions in R and S for all possible *True/False* values of $q_1, \ldots, q_l$, or one must exhibit a case of *True/False* values for $q_1, \ldots, q_l$ that precludes the existence of such solutions. We introduce precise definitions of the three problems.

We begin with definitions that link *True/False* values for $q_1, \ldots, q_l$ to satisfiability of R and S. Declare an assignment of *True/False* values

to $q_1, \ldots, q_l$ to be *R-acceptable* (resp. *R-unacceptable*) if the SAT instance of R with $q_1, \ldots, q_l$ fixed according to the given assignment is satisfiable (resp. unsatisfiable). Analogous definitions hold for the terms *S-acceptable* and *S-unacceptable*.

The first problem demands the following. Either conclude that all R-acceptable assignments are S-acceptable, or exhibit an R-acceptable assignment that is S-unacceptable. This problem is called Q-ALL SAT. The name is based on the following considerations. First, the letter Q could be viewed as the name of a set having the variables $q_1, \ldots, q_l$ as elements. Second, the use of the words ALL and SAT is motivated by the first conclusion of Q-ALL SAT, according to which all R-acceptable assignments for $q_1, \ldots, q_l$ are S-acceptable and thus are part of satisfying solutions of S.

The second and the third problems are generalizations of Q-ALL SAT that involve costs for the variables of R and S. For clarity, we define the costs of R to be *R-costs* and declare the costs of S to be *S-costs*.

The second problem demands the following. Either conclude that all R-acceptable assignments to $q_1, \ldots, q_l$ are S-acceptable; this part matches the first conclusion of Q-ALL SAT. Or, among the R-acceptable and S-unacceptable assignments, find one so that the optimal total cost of the MINSAT instance defined by R and the R-costs, with variables $q_1, \ldots, q_l$ fixed according to the assignment, is as small as possible. We call this problem Q-MIN UNSAT. The letter Q is interpreted as before. The word MIN indicates that a minimum cost solution of R is to be found. The word UNSAT tells that the assignment for $q_1, \ldots, q_l$ is to be S-unacceptable and thus to make S unsatisfiable.

The third problem calls for the following. Either exhibit an R-acceptable assignment for $q_1, \ldots, q_l$ that is S-unacceptable; this part agrees with the second conclusion of Q-ALL SAT. Or, among the R-acceptable assignments, find one so that the minimum total cost of the MINSAT instance defined by S and the S-costs, with variables $q_1, \ldots, q_l$ fixed according to the assignment, is as large as possible. This problem is named Q-MAX MINSAT. Here, the letter Q is interpreted as before. The word MAX tells that the minimum total cost is to be maximized by the assignment for $q_1, \ldots, q_l$. The word MINSAT records that MINSAT instances involving S are to be solved.

We reformulate Q-ALL SAT. Suppose that $q_1, \ldots, q_l$ and $x_1, \ldots, x_m$ are the variables of R, and that $q_1, \ldots, q_l$ and $y_1, \ldots, y_n$ are the variables of S. Recall that R and S may have common variables beyond $q_1, \ldots, q_l$. In that case, some variables x_i of R are actually the same as some variables y_j of S. The extent to which the x_i variables are equal to y_j variables is irrelevant for the discussion to follow. Hence, we leave that extent unspecified.

We introduce the notion of universal and existential quantification for

propositional formulas. In particular, the formula

$$\forall q_1 \ldots q_l \, [(\exists x_1 \ldots x_m \, R) \rightarrow (\exists y_1 \ldots y_n \, S)] \tag{4.1.1}$$

is defined to evaluate to *True* if the following condition holds for all possible *True/False* values of $q_1, \ldots, q_l$: If there exist *True/False* values for $x_1, \ldots, x_m$ such that the values for $q_1, \ldots, q_l$ and $x_1, \ldots, x_m$ satisfy R, then there exist *True/False* values for $y_1, \ldots, y_n$ such that the values for $q_1, \ldots, q_l$ and $y_1, \ldots, y_n$ satisfy S. On the other hand, if the above condition is not satisfied for some *True/False* values of $q_1, \ldots, q_l$, then we define that the formula of (4.1.1) evaluates to *False*.

Q-ALL SAT can now be restated as follows. Given R and S, we must evaluate the formula of (4.1.1). If the value of the formula is *False*, we must provide *True/False* values for $q_1, \ldots, q_l$ that demonstrate why the formula has the value *False*.

The formula of (4.1.1) is an example of the *quantified propositional formulas*, which are formulas constructed from propositional variables and the quantifiers $\forall$ and $\exists$ analogously to the construction of the formulas of first-order logic. Given the restatement of Q-ALL SAT in terms of the quantified propositional formula of (4.1.1), we may call Q-ALL SAT a *quantified* SAT *problem*. We may also rephrase the problems Q-MIN UNSAT and Q-MAX MINSAT in terms of the formula of (4.1.1) and therefore may call them *quantified* MINSAT *problems*.

In general, we may derive from any quantified propositional formula a number of quantified SAT and MINSAT problems. A discussion of all such SAT and MINSAT problems is beyond the scope of this book. Instead, we cover the three problems Q-ALL SAT, Q-MIN UNSAT, and Q-MAX MINSAT defined above and sketch three additional problems called P-EXIST Q-ALL SAT, P-MIN Q-ALL SAT, and P-MIN Q-MAX MINSAT that may be viewed as elaborations of Q-ALL SAT, Q-MIN UNSAT, and Q-MAX MINSAT.

For the reader who wants to explore quantified SAT and MINSAT problems further, the chapter contains references to relevant material and provides exercises that cover various aspects of the construction and evaluation of such problems.

The chapter proceeds as follows.

Section 4.2 deals with Q-ALL SAT.

Section 4.3 discusses Q-MIN UNSAT and a related problem called Q-MINFIX UNSAT.

Section 4.4 describes Q-MAX MINSAT.

Section 4.5 sketches the problems P-EXIST Q-ALL SAT, P-MIN Q-ALL SAT, and P-MIN Q-MAX MINSAT. It also covers a related problem called Q-MINFIX SAT.

Section 4.6 provides heuristic solution schemes for Q-ALL SAT, Q-MINFIX UNSAT, and Q-MINFIX SAT.

Section 4.7 lists further reading material.
Section 4.8 contains exercises.

4.2 Problem Q-ALL SAT

We introduce the problem via an example.

Example

Consider the following situation. There are three questions that can be answered by "yes" or "no." They are represented by variables q_1, q_2, and q_3. Answers "yes" and "no" to the questions correspond to the assignment of *True* and *False*, respectively, to the variables. In a slight abuse of terminology, we refer to each q_j as a question and consider *True* and *False* to be answers. The answers to the questions are not independent. In particular, at least one of the questions q_2 and q_3 is always answered by *True*. Correspondingly, we have the clause

$$q_2 \vee q_3 \tag{4.2.1}$$

Answers to the questions are useful for the identification of three defects represented by variables r, s, and t. If a defect is present (resp. absent), then the corresponding variable has the value *True* (resp. *False*). Questions and defects are linked by the following logic relationships.

$$\begin{aligned} &q_1 \dashrightarrow (r \vee t) \\ &q_2 \dashrightarrow (\neg r \vee \neg t) \\ &(q_1 \wedge q_2) \dashrightarrow \neg r \\ &q_3 \dashrightarrow (s \vee \neg t) \end{aligned} \tag{4.2.2}$$

Suppose identification of the defect represented by t is of particular importance. To find out whether that defect can be proved, we ask the first question and get the value *False* for q_1. We wonder if, given $q_1 = \mathit{False}$, we can possibly obtain answers for q_2 and q_3 such that t can be proved. If this is not the case, it would be useless to ask these two questions.

Solution of Example

We determine whether it is useless to obtain answers for q_2 and q_3. First, we add to the clause $q_2 \vee q_3$ of (4.2.1) the condition that q_1 has the value *False*, by adjoining the clause $\neg q_1$. Thus, we have

$$\begin{aligned} &q_2 \vee q_3 \\ &\neg q_1 \end{aligned} \tag{4.2.3}$$

Declare R to be the CNF system of (4.2.3). The satisfying solutions of R are precisely the cases of *True/False* values that $q_1, \ldots, q_3$ may take on, given the knowledge that $q_1 = \textit{False}$.

Second, we rewrite (4.2.2) in CNF form and add the condition that t has the value *False*. The resulting CNF system, which we call S, is given below.

$$\begin{aligned} &\neg q_1 \vee r \vee t \\ &\neg q_2 \vee \neg r \vee \neg t \\ &\neg q_1 \vee \neg q_2 \vee \neg r \\ &\neg q_3 \vee s \vee \neg t \\ &\neg t \end{aligned} \tag{4.2.4}$$

Suppose *True/False* values for $q_1, \ldots, q_3$ satisfy R. When these *True/False* values are enforced in (4.2.2) and in S of (4.2.4), then that version of (4.2.2) has t as a theorem if and only if the corresponding version of S is unsatisfiable. Hence, it is useless to ask for the *True/False* values of q_2 and q_3 if and only if, for any *True/False* values of $q_1, \ldots, q_3$ satisfying R, the version of S that has these *True/False* values enforced is satisfiable. We decide whether this case is at hand, as follows.

The satisfying solutions of the CNF system R of (4.2.3) are characterized by $q_1 = \textit{False}$ and the condition that at least one of q_2 and q_3 must have the value *True*. The first and third clauses of S of (4.2.4) are satisfied by $q_1 = \textit{False}$. The remaining clauses are satisfied by $t = \textit{False}$ and any value for r regardless of the values for q_2 and q_3. Thus, t cannot be proved regardless of the values assigned to q_2 and q_3, and it is useless to ask these two questions in the hope of proving t.

In contrast, consider the case where the answer to q_1 is *True*. The CNF system R is now

$$\begin{aligned} &q_2 \vee q_3 \\ &q_1 \end{aligned} \tag{4.2.5}$$

The satisfying solutions of (4.2.5) force q_1 and at least one of q_2 and q_3 to have the value *True*. We select from S of (4.2.4) the first, third, and last clauses, getting the following subsystem of S.

$$\begin{aligned} &\neg q_1 \vee r \vee t \\ &\neg q_1 \vee \neg q_2 \vee \neg r \\ &\neg t \end{aligned} \tag{4.2.6}$$

Using $q_1 = \textit{True}$, (4.2.6) can be simplified to

$$\begin{aligned} &r \\ &\neg q_2 \vee \neg r \end{aligned} \tag{4.2.7}$$

If $q_2 = True$, then the CNF system of (4.2.7) is unsatisfiable. Accordingly, $q_2 = True$ allows the proof that the defect t must hold. We conclude that, if $q_1 = True$ is known, then answers to q_2 and q_3 exist that establish the defect t. In particular, the answer $q_2 = True$ proves the defect t.

Problem Definition

Recall the definition of Section 4.1 according to which *True/False* values of $q_1, \ldots, q_l$ are R-acceptable (resp. R-unacceptable) if R has (resp. does not have) a satisfying solution when $q_1, \ldots, q_l$ take on the specified values. The terms S-acceptable and S-unacceptable are analogously defined. Using that terminology for the case at hand, it is useless to ask for the *True/False* values of q_2 and q_3 if and only if all R-acceptable *True/False* values for $q_1, \ldots, q_3$ are S-acceptable. We compare the latter task with that of problem Q-ALL SAT introduced in Section 4.1. To simplify the comparison, we include a compact statement of Q-ALL SAT.

(4.2.8) Q-ALL SAT

Instance: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables.

Solution: Either: "All R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-acceptable." Or: An R-acceptable assignment of *True/False* values for $q_1, \ldots, q_l$ that is S-unacceptable.

Evidently, if we let $q_1, \ldots, q_3$, R, and S of the example problem play the roles of $q_1, \ldots, q_l$, R, and S of Q-ALL SAT, then the solution of this instance of Q-ALL SAT answers the decision problem of the example. Exercises (4.8.2) and (4.8.3) ask the reader to link additional applications and problem Q-ALL SAT.

Solution Algorithm

It seems that Q-ALL SAT is more difficult than SAT. We say "seems" since at present no algorithm is known that reduces an arbitrary instance of Q-ALL SAT to SAT instances such that both the size of the SAT instances and the number of such instances is bounded by a polynomial in the size of the given Q-ALL SAT instance. At the same time, there is no proof that such a reduction is impossible. For references, see Section 4.7. Here, we describe a direct enumerative algorithm for Q-ALL SAT instances where l, the number of special variables, is small. A heuristic solution scheme for Q-ALL SAT instances with large l is given in Section 4.6.

(4.2.9) Algorithm SOLVE Q-ALL SAT. *Solves instance of* Q-ALL SAT.

Input: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables.

Output: Either: "All R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-acceptable." Or: An R-acceptable assignment of *True/False* values for $q_1, \ldots, q_l$ that is S-unacceptable.

Requires: Algorithm SOLVE SAT.

Procedure:

1. For each possible assignment of *True/False* values for $q_1, \ldots, q_l$, do the following: Use Algorithm SOLVE SAT to test if R with $q_1, \ldots, q_l$ fixed according to the assignment is satisfiable and if S with $q_1, \ldots, q_l$ fixed to the same values is unsatisfiable. If both tests are passed, output the given assignment as R-acceptable and S-unacceptable, and stop.
2. Declare that all R-acceptable assignments for $q_1, \ldots, q_l$ are also S-acceptable, and stop.

Next, we cover the problem Q-MIN UNSAT, which is a version of Q-ALL SAT involving costs for the variables of R.

4.3 Problem Q-MIN UNSAT

We motivate the problem by an example.

Example

We have a CNF system R given by

$$
\begin{aligned}
& q_1 \vee q_2 \vee q_3 \\
& \neg q_1 \vee \neg q_2 \\
& \neg q_2 \vee \neg q_3 \vee x
\end{aligned}
\tag{4.3.1}
$$

We also have the following R-costs.

(4.3.2)

Variable of R	Cost of *True*	Cost of *False*
q_1	3	0
q_2	9	0
q_3	5	0
x	25	0

R-costs for R of (4.3.1)

The variables $q_1, \ldots, q_3$ represent controls that can be set at will to any R-acceptable *True/False* values. Suppose we fix $q_1, \ldots, q_3$ to such *True/False* values and solve the MINSAT instance given by (4.3.1) and (4.3.2). We call the minimum total cost of that instance the *total R-cost*. One may view the total R-cost as the cost of selecting the given *True/False* values for the control variables $q_1, \ldots, q_3$.

We have a second CNF system given by

$$\begin{aligned} &\neg q_1 \vee r \vee t \\ &\neg q_2 \vee \neg r \vee \neg t \\ &\neg q_1 \vee \neg q_2 \vee \neg r \\ &\neg q_3 \vee \neg r \end{aligned} \tag{4.3.3}$$

The variable t of (4.3.3) is of particular interest. Specifically, we want t to become a theorem of (4.3.3) by fixing the control variables $q_1, \ldots, q_3$ of (4.3.3) to some *True/False* values that must be R-acceptable. Indeed, among the possible candidate values, we want to find ones for which the total R-cost is as small as possible.

We reformulate the task at hand. To this end, we add the clause $\neg t$ to (4.3.3), getting the following CNF system S.

$$\begin{aligned} &\neg q_1 \vee r \vee t \\ &\neg q_2 \vee \neg r \vee \neg t \\ &\neg q_1 \vee \neg q_2 \vee \neg r \\ &\neg q_3 \vee \neg r \\ &\neg t \end{aligned} \tag{4.3.4}$$

Now t is a theorem of (4.3.3) if and only if S is unsatisfiable, regardless how the variables $q_1, \ldots, q_3$ are fixed in both (4.3.3) and S of (4.3.4). Hence, our task is to select R-acceptable *True/False* values for $q_1, \ldots, q_3$ such that these values render S unsatisfiable. Subject to that requirement, the corresponding total R-cost is to be as small as possible.

Of course, R-acceptable *True/False* values for $q_1, \ldots, q_3$ that cause S to become unsatisfiable may not exist. In that case, it suffices that we conclude this fact.

Solution of Example

We determine the desired *True/False* values for $q_1, \ldots, q_3$ or conclude that they do not exist, as follows.

First, for all possible cases of *True/False* values of $q_1, \ldots, q_3$, we solve the related MINSAT instance of (4.3.1) and (4.3.2). That way, we get all

R-acceptable cases and the corresponding total R-cost. We sort these cases in order of increasing total R-cost. The reader should confirm that the following table displays the results of this step. To simplify the checking, the table includes for each case the optimal *True/False* value for the variable x.

(4.3.5)

q_1	q_2	q_3	x	Total R-Cost
True	*False*	*False*	*False*	3
False	*False*	*True*	*False*	5
True	*False*	*True*	*False*	8
False	*True*	*False*	*False*	9
False	*True*	*True*	*True*	39

Sorted MINSAT solutions of R of (4.3.1)

Next, we process the sets of *True/False* values for $q_1, \ldots, q_3$ displayed in the rows of the table in the given order. For each row, we assign the *True/False* values to $q_1, \ldots, q_3$ of S of (4.3.4) and test the resulting CNF system for satisfiability. As soon as we have a case of unsatisfiability, we stop; the *True/False* values for $q_1, \ldots, q_3$ of that case are the desired ones. On the other hand, if no row of *True/False* values leads to unsatisfiability of S of (4.3.4), then we conclude that, no matter how R-acceptable *True/False* values are selected, t cannot become a theorem.

The reader should verify that the first row of (4.3.5), which assigns $q_1 = \mathit{True}$ and $q_2 = q_3 = \mathit{False}$, results in satisfiability of S of (4.3.4). The second row of (4.3.5), with values $q_1 = q_2 = \mathit{False}$, and $q_3 = \mathit{True}$, produces the same conclusion. However, the third row, which specifies $q_1 = \mathit{True}$, $q_2 = \mathit{False}$, and $q_3 = \mathit{True}$, leads to unsatisfiability. We conclude that $q_1 = \mathit{True}$, $q_2 = \mathit{False}$, and $q_3 = \mathit{True}$ achieve unsatisfiability of S of (4.3.4). The related total R cost, 8, is the smallest possible.

Problem Definition

We may summarize the above example problem as follows. For R of (4.3.1) and S of (4.3.4), we are to find an R-acceptable assignment of *True/False* values for $q_1, \ldots, q_3$ that is S-unacceptable and that, subject to the latter requirement, has least total R-cost. In case every R-acceptable assignment is S-acceptable, we stop with that alternate conclusion. We compare this task with that of problem Q-MIN UNSAT introduced in Section 4.1. For ease of comparison, we include a compact statement of Q-MIN UNSAT.

(4.3.6) Q-MIN UNSAT

Instance: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables. R-costs for the variables of R.

Solution: Either: "All R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-acceptable." Or: An R-acceptable assignment of *True/False* values for $q_1, \ldots, q_l$ that is S-unacceptable and that, subject to that condition, has least total R-cost.

Evidently, if we view $q_1, \ldots, q_3$, R, and S of the example as $q_1, \ldots, q_l$, R, and S of Q-MIN UNSAT, then the solution of this instance of Q-MIN UNSAT answers the question posed in the example problem. Exercise (4.8.5) asks the reader to interpret another application problem as an instance of the problem Q-MIN UNSAT.

Solution Algorithm

Q-MIN UNSAT is a generalization of Q-ALL SAT and hence is at least as difficult to solve. For references, see Section 4.7. Below, we include a direct enumerative algorithm for Q-MIN UNSAT instances where l, the number of special variables, is small.

(4.3.7) Algorithm SOLVE Q-MIN UNSAT. *Solves instance of* Q-MIN UNSAT.

Input: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables. R-costs for the variables of R.

Output: Either: "All R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-acceptable." Or: An R-acceptable assignment of *True/False* values for $q_1, \ldots, q_l$ that is S-unacceptable and that, subject to that condition, has least total R-cost.

Requires: Algorithms SOLVE SAT and SOLVE MINSAT.

Procedure:

0. Initialize a list L as empty.
1. For each possible assignment of *True/False* values for $q_1, \ldots, q_l$, do the following: In R, fix $q_1, \ldots, q_l$ to the given values and use Algorithm SOLVE MINSAT to find a minimum R-cost satisfying solution or to determine unsatisfiability. If a minimum R-cost solution is found, collect the solution and its total R-cost in a record, and store that record in L.
2. Sort the records of L in increasing order of the total R-cost values.
3. Process the records of L in the given order. For each record, do the following: Use Algorithm SOLVE SAT to test if the given *True/False* values for $q_1, \ldots, q_l$ are S-acceptable. If the *True/False* values are S-unacceptable, output the values, and stop; these values are R-acceptable, S-unacceptable, and, subject to the latter condition, have least total R-cost.
4. Declare that all R-acceptable *True/False* values are S-acceptable, and stop.

We consider a variation of Q-MIN UNSAT called Q-MINFIX UNSAT.

Problem Q-MINFIX UNSAT

Given is a satisfiable CNF system S that has $q_1, \ldots, q_l$ among its variables. Also given is an S-unacceptable assignment for $q_1, \ldots, q_l$. A cost is associated with each q_j. It represents the cost of obtaining the given *True/False* value for q_j. Define any subset of the assignment to be a *partial assignment.* The total cost of a partial assignment is the sum of the costs associated with the q_j of that partial assignment. For solution of the problem, one must find a partial assignment that is S-unacceptable and that, subject to that condition, has minimum total cost. We summarize the problem.

(4.3.8) Q-MINFIX UNSAT

Instance: Satisfiable CNF system S containing $l \geq 1$ special variables $q_1, \ldots, q_l$. An S-unacceptable assignment. For each q_j, a cost of obtaining the given *True/False* value of q_j.

Solution: An S-unacceptable partial assignment with minimum total cost.

We discuss an application.

Application of Q-MINFIX UNSAT

The clauses of a CNF system S' link symptoms $q_1, \ldots, q_l$ with a disease t. For a patient, we have obtained a *True/False* value for each symptom q_j at a certain cost. Suppose this assignment for $q_1, \ldots, q_l$ renders $S = S' \wedge \neg t$ unsatisfiable. Thus, it has been proved that the patient has the disease. We would like to learn in hindsight how we could have established the disease at minimum total cost, by getting symptom values judiciously and only as needed.

We view the problem as an instance of Q-MINFIX UNSAT. Note that $S = S' \wedge \neg t$, without the assignment, is satisfiable since absence of the disease must be allowed in S'. An optimal solution of this instance of Q-MINFIX UNSAT provides an S-unacceptable assignment with minimum total cost. Thus, t can be proved just by using the *True/False* values of the partial assignment, and the total cost of acquiring these values is minimum.

Transformation of Q-MINFIX UNSAT to Q-MIN UNSAT

Problem Q-MINFIX UNSAT may be converted to a case of Q-MIN UNSAT as follows. Denote R, S, $q_1, \ldots, q_l$ of Q-MIN UNSAT by R', S', $q'_1, \ldots, q'_l$ to set them apart from S and $q_1, \ldots, q_l$ of Q-MINFIX UNSAT.

For the transformation from Q-MINFIX UNSAT to Q-MIN UNSAT, define R' to be the CNF system that has $q'_1, \ldots, q'_l$ as variables and that has no clauses. For each q'_j, define R'-costs by assigning the cost of obtaining the given *True/False* value for q_j as the cost of *True* and by declaring 0 to be the cost of *False*.

Derive S' of Q-MIN UNSAT from S of Q-MINFIX UNSAT in two steps. First, declare all clauses of S to be clauses of S'. Second, for each q_j, add to S' the clause $q'_j \to q_j$ (resp. $q'_j \to \neg q_j$) if the given assignment associates with q_j the value *True* (resp. *False*).

Suppose we solve the instance of Q-MIN UNSAT given by R', S', $q'_1, \ldots, q'_l$. For each j, we interpret an optimal value of *True* (resp. *False*) of q'_j in Q-MIN UNSAT as placing (resp. not placing) q_j and its value into the partial assignment. Exercise (4.8.6) asks the reader to prove that these choices for $q_1, \ldots, q_l$ constitute an optimal solution of the Q-MINFIX UNSAT instance.

Solution Algorithm

Due to the above transformation, Q-MINFIX UNSAT is readily solved by any algorithm for Q-MIN UNSAT. Thus, when l, the number of special variables, is small, Algorithm SOLVE Q-MIN UNSAT (4.3.7) can be used. For a slightly more elaborate case of Q-MINFIX UNSAT, with potentially large l, Section 4.6 contains a heuristic solution scheme.

We turn to the problem Q-MAX MINSAT, which is another version of Q-ALL SAT involving costs.

4.4 Problem Q-MAX MINSAT

We begin with an example.

Example

We have a CNF system R given by

$$\begin{aligned} & q_1 \vee q_2 \vee q_3 \\ & q_1 \vee \neg q_2 \vee q_3 \\ & q_1 \vee q_2 \vee \neg q_3 \end{aligned} \tag{4.4.1}$$

We also have a CNF system S specified by

$$\begin{aligned} & \neg q_1 \vee r \vee t \\ & \neg q_2 \vee \neg r \vee \neg t \\ & \neg q_1 \vee \neg q_3 \vee r \end{aligned} \tag{4.4.2}$$

For the variables of S, the following S-costs are given.

(4.4.3)

Variable of S	Cost of *True*	Cost of *False*
q_1	1	3
q_2	4	2
q_3	7	8
r	21	0
t	15	0

S-costs for S

The variables $q_1, \ldots, q_3$ represent certain events. When one of the variables has the value *True* (resp. *False*), then the corresponding event does (resp. does not) take place. The events are related, and the possible combinations of events are defined by the ways in which R-acceptable *True/False* values can be assigned to $q_1, \ldots, q_3$.

Let a combination of events be given by R-acceptable *True/False* values for $q_1, \ldots, q_3$. Fix the variables $q_1, \ldots, q_3$ of S to these values. Then the MINSAT instance defined by this version of S and the S-costs may be solved, and the resulting minimum cost solution may be viewed as a best-possible reaction to the events. In that interpretation, if the minimum total cost, which from now on we call *total S-cost*, is largest, then the MINSAT solution may be viewed as a *worst-case scenario*. We want to find R-acceptable *True/False* values for $q_1, \ldots, q_3$ that produce a worst-case scenario.

It is possible that, for some R-acceptable *True/False* values, the CNF system S becomes unsatisfiable. In that case, we stop with that conclusion and do not search for a worst-case scenario. Alternately, one could view that situation as a worst-case scenario with infinite total S-cost.

Solution of Example

We determine the desired values for $q_1, \ldots, q_3$.

First, we find the satisfying solutions of R of (4.4.1). The table below displays the possible cases.

(4.4.4)

q_1	q_2	q_3
True	*False*	*False*
True	*True*	*False*
False	*True*	*True*
True	*False*	*True*
True	*True*	*True*

Satisfying solutions of R of (4.4.1)

Next, we process the sets of *True/False* values of table (4.4.4) one by one. For each case, we solve the MINSAT instance given by S of (4.4.2) and by the S-costs of (4.4.3), with variables $q_1, \ldots, q_3$ fixed to the values of the given case. If such a MINSAT instance is unsatisfiable, we declare that, for the given R-acceptable values, the corresponding version of S is unsatisfiable and stop. If all MINSAT instances are satisfiable, we compare their total S-costs and select the case for which the total S-cost is largest.

It turns out that each case of (4.4.4) leads to a satisfiable MINSAT instance. For each such case, table (4.4.5) below contains a row listing the *True/False* values of the MINSAT solution and the total S-cost.

(4.4.5)

q_1	q_2	q_3	r	t	Total S-Cost
True	*False*	*False*	*False*	*True*	26
True	*True*	*False*	*False*	*True*	28
False	*True*	*True*	*False*	*False*	14
True	*False*	*True*	*True*	*False*	31
True	*True*	*True*	*True*	*False*	33

Minimum cost solutions of S of (4.4.3)

The last line of the table (4.4.5) contains the largest total S-cost, 33. That cost is achieved with $q_1 = q_2 = q_3 = \mathit{True}$. The values for the remaining variables of S are $r = \mathit{True}$ and $t = \mathit{False}$. Together, these values for the variables of S constitute a worst-case scenario.

Problem Definition

We summarize the above problem. For R of (4.4.1), we are to find an R-acceptable assignment of *True/False* values for $q_1, \ldots, q_3$ such that the following holds. For S of (4.4.2), either the assignment is S-unacceptable, or, if that case cannot be achieved, the total S-cost of the MINSAT instance defined by S and the S-costs of (4.4.3), is as large as possible. We compare this task with that of the problem Q-MAX MINSAT described in Section 4.1. For ease of comparison, we list a compact statement of Q-MAX MINSAT.

(4.4.6) Q-MAX MINSAT

Instance: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables. S-costs for the variables of S.

Solution: Either: An R-acceptable assignment of *True/False* values for $q_1, \ldots, q_l$ that is S-unacceptable. Or: An assignment of R-acceptable *True/False* values for $q_1, \ldots, q_l$ such that the total S-cost of the MINSAT

instance defined by S and by the S-costs, with $q_1, \ldots, q_l$ fixed according to the selected assignment, is as large as possible.

Clearly, if we view $q_1, \ldots, q_3$, R, S, and the S-costs of the example problem as $q_1, \ldots, q_l$, R, S, and the S-costs of Q-MAX MINSAT, then the solution of this instance of Q-MAX MINSAT also solves the example problem. Exercise (4.8.8) asks the reader to relate another application to Q-MAX MINSAT.

Solution Algorithm

Q-MAX MINSAT is a generalization of Q-ALL SAT and hence is at least as difficult to solve. For references, see Section 4.7. Below, we include a direct enumerative algorithm that is usable only if l, the number of special variables, is small.

(4.4.7) Algorithm SOLVE Q-MAX MINSAT. *Solves instance of* Q-MAX MINSAT.

Input: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables. S-costs for the variables of S.

Output: Either: An R-acceptable assignment of *True/False* values for $q_1, \ldots, q_l$ that is S-unacceptable. Or: An assignment of R-acceptable *True/False* values for $q_1, \ldots, q_l$ such that the total S-cost of the MINSAT instance defined by S and by the S-costs, with $q_1, \ldots, q_l$ fixed according to the selected assignment, is as large as possible.

Requires: Algorithms SOLVE SAT and SOLVE MINSAT.

Procedure:

0. Initialize $z^* = -\infty$.
1. For each possible assignment of *True/False* values for $q_1, \ldots, q_l$, do the following: In R, fix $q_1, \ldots, q_l$ to the given values and use Algorithm SOLVE SAT to determine satisfiability. If R is unsatisfiable, processing of the given assignment of values to $q_1, \ldots, q_l$ is completed. Use Algorithm SOLVE MINSAT to solve the MINSAT instance defined by S and its S-costs, with $q_1, \ldots, q_l$ fixed according to the given assignment. If S is unsatisfiable, output the *True/False* values of the assignment as being R-acceptable and S-unacceptable, and stop.
   Let z be the total S-cost determined by Algorithm SOLVE MINSAT. Update z^* to the maximum of z and z^*; if z^* is increased, retain the *True/False* values of the assignment.
2. The assignment retained with the current z^* is R-acceptable and results in the largest-possible total S-cost; the value of z^* is that total S-cost. Output the *True/False* values of that assignment, and stop.

This completes the discussion of the three problems Q-ALL SAT, Q-MIN UNSAT, and Q-MAX MINSAT. The next section extends the main ideas underlying the three problems and introduces three, more complicated, problems called P-EXIST Q-ALL SAT, P-MIN Q-ALL SAT, and P-MIN Q-MAX MINSAT.

4.5 More Complicated Quantified Problems

The three problems of this section are elaborations of the problems of Sections 4.2–4.4. They involve satisfiable CNF systems R_P, R_Q, and S instead of just R and S. The three CNF systems have $k \geq 1$ variables $p_1, \ldots, p_k$ among their common variables. In addition, R_Q and S have $l \geq 1$ variables $q_1, \ldots, q_l$ among their common variables.

Each of the three problems requires that either one exhibits *True/False* values for $p_1, \ldots, p_k$ such that a certain condition concerning R_P, R_Q, and S is satisfied, or one concludes that such *True/False* values do not exist. Loosely speaking, the condition demands that R_P, R_Q, and S with $p_1, \ldots, p_k$ fixed to the specified values have certain solutions for all possible *True/False* values of $q_1, \ldots, q_l$.

We need some definitions. Recall from Section 4.1 that an assignment of *True/False* values to $q_1, \ldots, q_l$ is *R-acceptable* (resp. *R-unacceptable*) if the SAT instance of R with $q_1, \ldots, q_l$ fixed according to the given assignment is satisfiable (resp. unsatisfiable). Here, we use analogous terminology for the CNF systems R_P, R_Q, and S when we have an assignment for $p_1, \ldots, p_k$ or an assignment for $q_1, \ldots, q_l$. Thus, we use the terms *R_P-acceptable*, *R_Q-acceptable*, and *S-acceptable*, as well as *R_P-unacceptable*, *R_Q-unacceptable*, and *S-unacceptable*. We employ the same terminology when assignments for $p_1, \ldots, p_k$ and $q_1, \ldots, q_l$ are simultaneously used to fix variables. In that case, we say that the assignments for $p_1, \ldots, p_k$ and $q_1, \ldots, q_l$ *together* are acceptable or unacceptable.

With these definitions, we can specify the above mentioned condition as follows. Let an assignment for $p_1, \ldots, p_k$ be given.

(4.5.1) The assignment for $p_1, \ldots, p_k$ must be R_P-acceptable. In addition, when that assignment and any assignment for $q_1, \ldots, q_l$ together are R_Q-acceptable, then these two assignments together must be S-acceptable.

Definition of Problem P-EXIST Q-ALL SAT

Problem P-EXIST Q-ALL SAT demands the following. Either one exhibits an assignment for $p_1, \ldots, p_k$ that obeys (4.5.1). Or one concludes that such

an assignment for $p_1, \ldots, p_k$ does not exist. In the name P-EXIST Q-ALL SAT, the letter P (resp. Q) may be considered to be the name of a set having the variables $p_1, \ldots, p_k$ (resp. $q_1, \ldots, q_l$) as elements. The terms EXIST, ALL, and SAT reflect the essence of the first conclusion, which requires the existence of an assignment for $p_1, \ldots, p_k$ so that for all assignments for $q_1, \ldots, q_l$ certain satisfiability requirements are met. We include a compact statement of the problem.

(4.5.2) P-EXIST Q-ALL SAT

Instance: Satisfiable CNF systems R_P, R_Q, and S. For $k \geq 1$, special variables $p_1, \ldots, p_k$ that occur in R_P, R_Q, and S. For $l \geq 1$, special variables $q_1, \ldots, q_l$ that occur in R_Q and S.

Solution: Either: An R_P-acceptable assignment for $p_1, \ldots, p_k$ such that the following holds. Whenever that assignment and any assignment for $q_1, \ldots, q_l$ together are R_Q-acceptable, then these two assignments together are S-acceptable. Or: "For any R_P-acceptable assignment for $p_1, \ldots, p_k$, there exists some assignment for $q_1, \ldots, q_l$ such that the assignments together are R_Q-acceptable and not S-acceptable."

Exercise (4.8.12)(a) asks the reader to construct a solution algorithm for P-EXIST Q-ALL SAT for the case of small k and l.

The problems P-MIN Q-ALL SAT and P-MIN Q-MAX MINSAT are cost minimization versions of P-EXIST Q-ALL SAT. Declare *R_P-costs* and *S-costs* to be costs associated with the variables of R_P (resp. S). For any R_P-acceptable assignment for $p_1, \ldots, p_k$, define the *total R_P-cost* to be the optimal total cost of the MINSAT instance defined by R_P and the R_P-costs, with $p_1, \ldots, p_k$ fixed according to the assignment.

Definition of Problem P-MIN Q-ALL SAT

Problem P-MIN Q-ALL SAT demands the following. Either one exhibits an assignment for $p_1, \ldots, p_k$ that obeys (4.5.1) and that, subject to that condition, has the total R_P-cost as small as possible. Or one concludes that such an assignment for $p_1, \ldots, p_k$ does not exist. The term MIN of P-MIN Q-ALL SAT refers to the selection of an assignment for $p_1, \ldots, p_k$ with total R_P-cost as small as possible. The remaining terms are interpreted as for P-EXIST Q-ALL SAT. Here is a compact statement of the problem.

(4.5.3) P-MIN Q-ALL SAT

Instance: Satisfiable CNF systems R_P, R_Q, and S. For $k \geq 1$, special variables $p_1, \ldots, p_k$ that occur in R_P, R_Q, and S. For $l \geq 1$, special variables $q_1, \ldots, q_l$ that occur in R_Q and S. R_P-costs for the variables of R_P.

Solution: Either: An R_P-acceptable assignment for $p_1, \ldots, p_k$ such that the following holds. Whenever that assignment and any assignment for

$q_1, \ldots, q_l$ together are R_Q-acceptable, then these two assignments together are S-acceptable. Subject to that condition, the assignment for $p_1, \ldots, p_k$ has least total R_P-cost. Or: "For any R_P-acceptable assignment for $p_1, \ldots, p_k$, there exists some assignment for $q_1, \ldots, q_l$ such that the assignments together are R_Q-acceptable and not S-acceptable."

Exercise (4.8.12)(b) calls for construction of a solution algorithm for P-MIN Q-ALL SAT for the case of small k and l.

We consider a variation of P-MIN Q-ALL SAT called Q-MINFIX SAT.

Definition of Problem Q-MINFIX SAT

Given are satisfiable CNF systems R and S that have $q_1, \ldots, q_l$ among their common variables. Also given is an assignment for $q_1, \ldots, q_l$ that is R-acceptable and S-acceptable. A cost is associated with each q_j. It is the cost of obtaining the given *True/False* value for q_j. Recall the following. A partial assignment is a subset of the given assignment. The total cost of a partial assignment is the sum of the costs associated with the q_j occurring in the partial assignment.

An *extension assignment* is obtained from a partial assignment by specifying *True/False* values for all q_j that are not given a value in the partial assignment. Note that the additional *True/False* values are allowed to be different from the values of the initial assignment.

One must find a partial assignment such that any extension assignment that is R-acceptable is also S-acceptable. Subject to that condition, the cost of the partial assignment is to be minimum. Here is a compact statement of the problem.

(4.5.4) Q-MINFIX SAT

Instance: Satisfiable CNF systems R and S having $l \geq 1$ variables $q_1, \ldots, q_l$ among their common variables. An assignment for $q_1, \ldots, q_l$ that is both R-acceptable and S-acceptable. For each q_j, a cost of obtaining the given *True/False* value for q_j.

Solution: A partial assignment such that each R-acceptable extension assignment is S-acceptable. Subject to that condition, the total cost of the partial assignment must be minimum.

We transform Q-MINFIX SAT to P-MIN Q-ALL SAT.

Transformation of Q-MINFIX SAT to P-MIN Q-ALL SAT

Problem Q-MINFIX SAT may be converted to a case of P-MIN Q-ALL SAT as follows. Denote R_P, R_Q, and S of P-MIN Q-ALL SAT by R'_P, R'_Q,

and S' to set these CNF systems apart from R and S of Q-MINFIX SAT. For the same reason, let $p'_1, \ldots, p'_k$ and $q'_1, \ldots, q'_l$ be the special variables of P-MIN Q-ALL SAT.

For the transformation, define $k = l$, and declare R'_P to have the variables $p'_1, \ldots, p'_l$ and no clauses. The cost of *True* for p'_j is the cost of obtaining the given *True/False* value for q_j in Q-MINFIX SAT.

Initialize R'_Q to be R with variables $q_1, \ldots, q_l$ replaced by $q'_1, \ldots, q'_l$. For $1 \leq j \leq l$, add one clause to R'_Q as follows. If in Q-MINFIX SAT the assignment has for q_j the value *True*, then add to R'_Q the clause $p'_j \rightarrow q'_j$; otherwise, add the clause $p'_j \rightarrow \neg q'_j$.

Define S' to be S with variables $q_1, \ldots, q_l$ replaced by $q'_1, \ldots, q'_l$.

Suppose we have solved the instance of P-MIN Q-ALL SAT specified above. For each j, declare q_j to be in the partial assignment of Q-MINFIX SAT if p'_j of P-MIN Q-ALL SAT has the solution value *True*.

Exercise (4.8.13) asks for proof that the solution for Q-MINFIX SAT obtained by the above process is indeed correct for Q-MINFIX SAT.

Solution Algorithm

Due to the above transformation, Q-MINFIX SAT is readily solved by any algorithm for P-MIN Q-ALL SAT. Thus, when l, the number of special variables, is small, the algorithm of Exercise (4.8.12)(b) for P-MIN Q-ALL SAT may be used. For a slightly more elaborate case of Q-MINFIX SAT, with potentially large l, Section 4.6 contains a heuristic solution scheme.

Next, we discuss an application of Q-MINFIX SAT.

Application of Q-MINFIX SAT

The application is similar to that discussed earlier in this chapter in connection with problem Q-MINFIX UNSAT (4.3.8). We have *True/False* values for symptoms $q_1, \ldots, q_l$. Using these values, we are not able to prove a disease t. For each q_j, the determination of the *True/False* value of q_j has entailed a certain cost. We would like to learn in hindsight how we could have established at minimum cost that the disease cannot be proved, by getting symptom values judiciously and only as needed. That is, we want to know a subset of the symptom values such that the disease t cannot be proved regardless of the values that the remaining symptoms may take on.

We formulate the application problem as an instance of Q-MINFIX SAT. Let the relationships between the symptoms $q_1, \ldots, q_l$ and the disease t be encoded in a CNF system S. We add to S the clause $\neg t$ and obtain the system $S_{t=False}$. The given values for the symptoms constitute an

assignment for $q_1, \ldots, q_l$. Since t cannot be proved, that assignment is $S_{t=False}$-acceptable. Below, we assume that the S-acceptable assignments for $q_1, \ldots, q_l$ constitute precisely the possible ways in which *True/False* values may occur for the symptoms.

With these definitions, we can restate the problem as follows. We want a partial assignment of minimum total cost such that any possible extension assignment, which is S-acceptable, is $S_{t=False}$-acceptable and thus does not allow a proof of t.

For the formulation of the problem as an instance of Q-MINFIX SAT, temporarily denote R and S of Q-MINFIX SAT by R' and S', and let these CNF systems have special variables $q'_1, \ldots, q'_l$.

We solve the Q-MINFIX SAT instance for which $R' = S$ and $S' = S_{t=False}$, and where the special variables $q'_1, \ldots, q'_l$ of Q-MINFIX SAT are the special variables $q_1, \ldots, q_l$ of the application problem. The cost of keeping q'_j fixed is the cost of determining the symptom value for q_j.

Once we have the solution of the Q-MINFIX SAT instance, we declare the value for symptom q_j to be in the sought-after subset if and only if q_j is kept fixed by the partial assignment of the solution. Exercise (4.8.14) calls for verification that this subset of symptom values rules out proof of the disease t and does so at minimum cost.

We turn to problem P-MIN Q-MAX MINSAT. It involves S-costs. Take any two assignments for $p_1, \ldots, p_k$ and $q_1, \ldots, q_l$ such that the assignment for $p_1, \ldots, p_k$ obeys (4.5.1) and such that the two assignments together are R_Q-acceptable. By (4.5.1), the two assignments together are then S-acceptable. Define the *total S-cost* to be the optimal total cost of the MINSAT instance defined by S and the S-costs, with $p_1, \ldots, p_k$ and $q_1, \ldots, q_l$ fixed according to the assignments. Keeping the assignment for $p_1, \ldots, p_k$ fixed but allowing the assignment for $q_1, \ldots, q_l$ to vary, define the *overall cost* to be the largest-possible total S-cost that can be achieved by suitable selection of the assignment for $q_1, \ldots, q_l$. Thus, the overall cost depends only on the assignment for $p_1, \ldots, p_k$.

Definition of Problem P-MIN Q-MAX MINSAT

Problem P-MIN Q-MAX MINSAT demands the following. Either one exhibits an assignment for $p_1, \ldots, p_k$ that obeys (4.5.1) and that, subject to that condition, has overall cost as small as possible. Or one concludes that such an assignment for $p_1, \ldots, p_k$ does not exist. The term MIN of P-MIN Q-MAX MINSAT points to the minimization of the overall cost by the assignment for $p_1, \ldots, p_k$. The term MAX indicates the maximization of the total S-cost by the assignment for $q_1, \ldots, q_l$. The term MINSAT tells that MINSAT instances of S are solved. A compact statement of the problem follows.

(4.5.5) P-MIN Q-MAX MINSAT

Instance: Satisfiable CNF systems R_P, R_Q, and S. For $k \geq 1$, special variables $p_1, \ldots, p_k$ that occur in R_P, R_Q, and S. For $l \geq 1$, special variables $q_1, \ldots, q_l$ that occur in R_Q and S. S-costs for the variables of S.

Solution: Either: An R_P-acceptable assignment for $p_1, \ldots, p_k$ such that the following holds. Whenever that assignment and any assignment for $q_1, \ldots, q_l$ together are R_Q-acceptable, then these two assignments together are S-acceptable. Subject to that condition, the assignment for $p_1, \ldots, p_k$ must have least overall cost. Or: "For any R_P-acceptable assignment for $p_1, \ldots, p_k$, there exists some assignment for $q_1, \ldots, q_l$ such that the assignments together are R_Q-acceptable and not S-acceptable."

Exercise (4.8.12)(c) requires construction of a solution algorithm for P-MIN Q-MAX MINSAT for the case of small k and l.

We interpret the three problems of this section using quantified propositional formulas.

Quantified Propositional Formulas

We first express P-EXIST Q-ALL SAT using a quantified propositional formula. Let R_P have, besides $p_1, \ldots, p_k$, the variables $w_1, \ldots, w_h$. Suppose that, besides $p_1, \ldots, p_k$ and $q_1, \ldots, q_l$, R_Q has the variables $x_1, \ldots, x_m$ and S has the variables $y_1, \ldots, y_n$. We allow commonality of variables for any two of R_P, R_Q, and S beyond that evident from the notation. For example, some of the variables w_i of R_Q may be variables q_j of R_Q and S. The extent of such commonality is irrelevant here. Hence, we leave that extent unspecified.

Consider the quantified propositional formula

$$\begin{aligned} &\exists p_1 \ldots p_k \, \{(\exists w_1 \ldots w_h \, R_P) \wedge \forall q_1 \ldots q_l \, [(\exists x_1 \ldots x_m \, R_Q) \rightarrow \\ &(\exists y_1 \ldots y_n \, S)]\} \end{aligned} \tag{4.5.6}$$

Declare the formula of (4.5.6) to evaluate to *True* if the following condition holds. There exist *True/False* for $p_1, \ldots, p_k$ such that, first, there exist *True/False* values for $w_1, \ldots, w_h$ such that R_P is satisfied, and such that, second, for all possible *True/False* values for $q_1, \ldots, q_l$, the following holds. Whenever there exist *True/False* values for $x_1, \ldots, x_m$ such that R_Q is satisfied, then there exist *True/False* values for $y_1, \ldots, y_n$ such that S is satisfied. We apologize for the long definition. We include it since it essentially is a verbalizing of the formula of (4.5.6). Using the terms R_P-, R_Q-, and S-acceptable, we shorten the definition as follows. The formula of (4.5.6) evaluates to *True* if there exists an assignment $p_1, \ldots, p_k$ such that the following holds. First, the assignment must be R_P-acceptable, and, second, whenever that assignment and any assignment for $q_1, \ldots,$

q_l together are R_Q-acceptable, then these two assignments together must also be S-acceptable. Evidently, the condition so imposed on $p_1, \ldots, p_k$ is (4.5.1). Thus, the formula of (4.5.6) evaluates to *True* if an assignment for $p_1, \ldots, p_k$ exists that obeys (4.5.1). Equivalently, the formula (4.5.6) evaluates to *True* if the first conclusion of P-EXIST Q-ALL SAT applies. Similarly, one may restate the optimization versions P-MIN Q-ALL SAT and P-MIN Q-MAX MINSAT. We leave it to the reader to fill in details.

The quantified propositional formula (4.1.1) as well that of (4.5.6) are members of a sequence of increasingly more complex formulas that encode ever more complex situations. These formulas are part of a multi-level structure called the *polynomial hierarchy* in the theory of computational complexity. We discuss that structure informally using the particular quantified propositional formulas introduced so far.

Let S be a propositional formula whose variables are $y_1, \ldots, y_n$. Then the formula

$$\exists y_1 \ldots y_n \, S \tag{4.5.7}$$

is at the first level of the polynomial hierarchy. That formula evaluates to *True* if there is an S-acceptable assignment for $y_1, \ldots, y_n$ or, equivalently, if S is satisfiable.

The formula (4.1.1), which is

$$\forall q_1 \ldots q_l \, [(\exists x_1 \ldots x_m \, R) \rightarrow (\exists y_1 \ldots y_n \, S)] \tag{4.5.8}$$

is at the second level of the polynomial hierarchy.

The formula (4.5.6), which is

$$\begin{aligned} &\exists p_1 \ldots p_k \, \{(\exists w_1 \ldots w_h \, R_P) \wedge \forall q_1 \ldots q_l \, [(\exists x_1 \ldots x_m \, R_Q) \rightarrow \\ &(\exists y_1 \ldots y_n \, S)]\} \end{aligned} \tag{4.5.9}$$

is at the third level of the polynomial hierarchy.

Exercise (4.8.15) asks the reader to extend the construction and to define formulas at any level of the polynomial hierarchy.

We point out an interpretation of the formulas (4.5.7)–(4.5.9) involving a competition. It concerns the satisfiability of S.

Consider that we want to achieve satisfiability of S, and that we have an opponent who wants to prevent us from achieving that goal.

At the first level, we have total control by selecting an assignment for $y_1, \ldots, y_n$ of (4.5.7). Our opponent cannot interfere with that selection.

At the second level, our opponent first chooses an R-acceptable assignment for $q_1 \ldots q_l$, and then we select an assignment for $y_1, \ldots, y_n$ of (4.5.8).

At the third level, we first select an R_P-acceptable assignment for $p_1 \ldots p_k$, then our opponent chooses an assignment for $q_1 \ldots q_l$ such that

the two assignments together are R_Q-acceptable, and finally we pick an assignment for $y_1 \ldots y_n$. Formulas at higher levels of the polynomial hierarchy repeat this pattern. That is, we and our opponent alternately make choices. If the level is odd, then we make the first decision. Otherwise, our opponent moves first.

The above interpretation is appropriate for a number of real-world situations. For example, the opponent may be a manifestation of Murphy's Law, which says "If it can go wrong, it will," and we try to outwit that law and find a satisfying solution for S regardless of the choices of the opponent.

In other applications, we trade places with the opponent. That is, we want to achieve unsatisfiability of S, while our opponent tries to obtain satisfiability. For example, we may obtain answers for some questions of a list of questions, while our opponent selects answers for the remaining questions of the list. We want to use the answers obtained by us to show unsatisfiability of S and thus to prove a certain conclusion. On the other hand, our opponent wants satisfiability of S to deny us the sought-after proof of the conclusion.

4.6 Heuristic Algorithms

This section describes heuristic solution algorithms for Q-ALL SAT and for slightly more elaborate versions of Q-MINFIX UNSAT and Q-MINFIX SAT. The algorithms are used in Chapter 10, where large problem instances with many special variables must be solved.

Q-ALL SAT Case

An instance of Q-ALL SAT (4.2.8) is defined by two satisfiable CNF formulas R and S with $q_1, \ldots, q_l$ among their common variables. One must decide whether all R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-acceptable.

Let an instance be given. In the heuristic algorithm for Q-ALL SAT, we delete $q_1, \ldots, q_l$ from S, getting a CNF system S', and test whether S' is satisfiable. If this is the case, then trivially all possible *True/False* values for $q_1, \ldots, q_l$ are S-acceptable. In particular, this is so for the case of R-acceptable values, so the Q-ALL SAT instance has been solved.

Suppose S' is unsatisfiable. We attempt to find R-acceptable values for $q_1, \ldots, q_l$. First, we apply Algorithm SOLVE MINCLS UNSAT (3.3.14) to S' to determine a minimal subsystem $\overline{S}'$ of unsatisfiable clauses. Collect in a CNF system S'' the clauses of S that correspond to the clauses of $\overline{S}'$.

Enlarge R to a CNF system R' by adding for each j for which q_j (resp. $\neg q_j$) occurs as literal in S'', the clause $\neg q_j$ (resp. q_j). For example, if q_2 occurs as literal in S'', then the clause $\neg q_2$ is added to R. The clause enforces $q = \mathit{False}$ in any satisfying solution of R' and thus causes all literals q_2 of S'' to evaluate to *False*.

We test if R' is satisfiable. Suppose it is. Take any satisfying solution. By the derivation of R', the values for $q_1, \ldots, q_l$ are R-acceptable and reduce S'' to the unsatisfiable $\overline{S}'$. Thus, the values for $q_1, \ldots, q_l$ are S-unacceptable and constitute a solution for the given Q-ALL SAT instance.

In the remaining case, R' is unsatisfiable, and we declare that the heuristic cannot solve the given Q-ALL SAT instance.

Example

The CNF system R is given by

$$q_1 \vee q_2 \tag{4.6.1}$$

The CNF system S has the clauses

$$\begin{aligned} & r \vee t \\ & \neg q_1 \vee \neg r \vee \neg t \\ & \neg q_2 \vee \neg s \vee \neg t \end{aligned} \tag{4.6.2}$$

The special variables are q_1 and q_2. Upon deletion of all literals of q_1 and q_2 from S , we have the CNF system S' as

$$\begin{aligned} & r \vee t \\ & \neg r \vee \neg t \\ & \neg s \vee \neg t \end{aligned} \tag{4.6.3}$$

The values $r = \mathit{True}$ and $s = t = \mathit{False}$ constitute a satisfying solution of S'. Thus, all R-acceptable assignment of *True/False* values for q_1 and q_2 are S-acceptable, and we stop with that conclusion.

In a second example, R is as before, but S is given by

$$\begin{aligned} & r \vee t \\ & \neg q_1 \vee \neg r \vee \neg t \\ & \neg q_2 \vee \neg r \\ & \neg q_3 \vee s \vee \neg t \\ & \neg t \end{aligned} \tag{4.6.4}$$

The special variables are $q_1, \ldots, q_3$. Their deletion from S produces the following S'.

$$
\begin{array}{l}
r \vee t \\
\neg r \vee \neg t \\
\neg r \\
s \vee \neg t \\
\neg t
\end{array}
\tag{4.6.5}
$$

It is easy to check that S' is unsatisfiable. Algorithm SOLVE MINCLS UNSAT (3.3.14) derives from S' the following minimal unsatisfiable subset of clauses, which constitute $\overline{S}'$.

$$
\begin{array}{l}
r \vee t \\
\neg r \\
\neg t
\end{array}
\tag{4.6.6}
$$

The corresponding subsystem S'' of S is

$$
\begin{array}{l}
r \vee t \\
\neg q_2 \vee \neg r \\
\neg t
\end{array}
\tag{4.6.7}
$$

Just one literal of $q_1, \ldots, q_3$ occurs in S''. It is $\neg q_2$ in the clause $\neg q_2 \vee \neg r$. Correspondingly, we add the clause q_2 to R and get the following R'.

$$
\begin{array}{l}
q_1 \vee q_2 \\
q_2
\end{array}
\tag{4.6.8}
$$

The values $q_1 = q_2 = \mathit{True}$ constitute a satisfying solution of R. Thus, these values are R-acceptable and S-unacceptable, and are a solution for the Q-ALL SAT instance.

On the other hand, if R' had been unsatisfiable, then the heuristic method would have been unable to draw a conclusion.

Heuristic Algorithm

The heuristic scheme essentially attempts to reduce each instance of Q-ALL SAT (4.2.8) to a few SAT instances. Below, we summarize the steps.

(4.6.9) Heuristic SOLVE Q-ALL SAT. *Attempts to solve instance of* Q-ALL SAT.

Input: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables.

Output: Either: "All R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-acceptable." Or: An R-acceptable assignment of *True/False* values for $q_1, \ldots, q_l$ that is S-unacceptable. Or: "The method cannot solve this instance of Q-ALL SAT."

Requires: Algorithms SOLVE MINCLS UNSAT (3.3.14) and SOLVE SAT.

Procedure:

1. Obtain S' from S by deleting the variables $q_1, \ldots, q_l$.
2. Test with Algorithm SOLVE SAT if S' is satisfiable. If this is the case, output "All R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-acceptable," and stop.
3. (S' is unsatisfiable.) Do Algorithm SOLVE MINCLS UNSAT (3.3.14) with S' as input to find a minimal subsystem $\overline{S}'$ of unsatisfiable clauses. Let S'' be the subsystem of S whose clauses correspond to those of $\overline{S}'$.
4. Enlarge R to a CNF system R' by adding for each j for which q_j (resp. $\neg q_j$) occurs as literal in S'', the clause $\neg q_j$ (resp. q_j). Apply Algorithm SOLVE SAT to R'. If a satisfying solution is found, output the *True/False* solution values of $q_1, \ldots, q_l$ as an R-acceptable and S-unacceptable assignment, and stop.
5. (R' is unsatisfiable.) Output "The method cannot solve this instance of Q-ALL SAT," and stop.

Heuristic methods invariably raise the question whether they are exact on some subclasses and, if this is the case, what these subclasses are. Typically, there are trivial subclasses with the desired feature. Harder to find are interesting subclasses. Exercise (4.8.21)(a) asks the reader to find subclasses of the latter kind for Heuristic SOLVE Q-ALL SAT (4.6.9).

Q-MINFIX UNSAT Case

We review problem Q-MINFIX UNSAT (4.3.8). An instance consists of the following: a satisfiable CNF system S containing variables $q_1, \ldots, q_l$; an S-unacceptable assignment for $q_1, \ldots, q_l$; and, for each q_j, a cost for obtaining the given *True/False* value for that variable. We must find an S-unacceptable partial assignment with minimum total cost.

We consider a slight generalization of the problem where, instead of a cost for obtaining the given *True/False* value for each variable q_j, we have costs for obtaining the *True/False* values of subsets of these variables. That case arises when values of $q_1, \ldots, q_l$ are determined by tests $T_1, \ldots, T_m$, and

when each test T_i produces a certain subset of these values. It is convenient that we view each test T_i also to be the set of indices j of the variables q_j for which the test produces the value. For example, $T_1 = \{1, 2, 4\}$ means that the test T_1 determines the values of the variables q_1, q_2, and q_4. The cost of carrying out test T_i is denoted by c_i. It is assumed that collectively the tests determine all values. Thus, $\bigcup_{i=1}^{m} T_i = \{1, 2, \ldots, l\}$. The solution of the problem consists of a collection of tests T_i such that the partial assignment obtained with these tests is S-unacceptable. Subject to that condition, the total cost of the collection of tests is to be minimum.

The case of Q-MINFIX UNSAT treated here is readily reduced to Q-MINFIX UNSAT (4.3.8). Exercise (4.8.22) asks for details of that reduction.

In the heuristic scheme for Q-MINFIX UNSAT with tests T_i, we fix in S the variables $q_1, \ldots, q_l$ according to the given assignment and delete all clauses now satisfied and all literals of $q_1, \ldots, q_l$. Let S' be the resulting CNF system. By assumption, S' is unsatisfiable. We apply algorithm MIN-CLS UNSAT (3.3.14) to S' to find a minimal subsystem $\overline{S}'$ of unsatisfiable clauses. Let S'' be the subsystem of S whose clauses correspond to those of $\overline{S}'$. For each variable q_j with at least one literal occurring in S'', let I_j be the set of indices i of the tests T_i for which $j \in T_i$. Thus, the tests T_i with $i \in I_j$ are precisely the tests that can determine the value of q_j. For example, if at least one literal of q_3 occurs in S'', and if the value of q_3 can be determined by tests T_1, T_4, or T_5, then $I_3 = \{1, 4, 5\}$.

We want to select enough tests so that the values of the q_j occurring in S'' are determined, and, subject to that condition, the total cost of the selected tests is minimum. We formulate the selection of the tests as a MINSAT instance. The variables are $choose(T_i)$, $1 \leq i \leq m$. *True* for $choose(T_i)$ means that test T_i is done, with associated cost c_i. The cost of *False* is 0. The clauses of the MINSAT instance are to assure that the selected tests determine the value for each variable q_j for which a literal occurs in S''. Thus, for each I_j defined above, we have the clause

$$\bigvee_{i \in I_j} choose(T_i) \tag{4.6.10}$$

For example, if $I_3 = \{1, 4, 5\}$, then (4.6.10) yields $\bigvee_{i \in I_3} choose(T_i) = choose(T_1) \vee choose(T_4) \vee choose(T_5)$.

We solve the MINSAT instance, thus getting a minimum-cost selection of tests that establish the *True/False* values of all q_j occurring in S''. These values reduce S'' to the unsatisfiable $\overline{S}'$ and thus correspond to an S-unacceptable partial assignment.

We carry out the computations for an example.

Example

The CNF system S is given by

$$\begin{aligned} &\neg q_1 \vee r \vee t \\ &\neg q_2 \vee r \vee \neg t \\ &\neg q_1 \vee \neg r \vee s \\ &\neg q_3 \vee s \vee \neg t \\ &\neg r \end{aligned} \tag{4.6.11}$$

The assignment $q_1 = q_2 = q_3 = \mathit{True}$ is given and renders S unsatisfiable. These values can be obtained by tests $T_1 = \{1\}$, $T_2 = \{1, 2\}$, $T_3 = \{2, 3\}$, and $T_4 = \{1, 3\}$, with associated costs $c_1 = 2$, $c_2 = 3$, $c_3 = 5$, and $c_4 = 3$, respectively.

Fixing $q_1, \ldots, q_3$ at the given values, we reduce S to the unsatisfiable S' given by

$$\begin{aligned} &r \vee t \\ &r \vee \neg t \\ &\neg r \vee s \\ &s \vee \neg t \\ &\neg r \end{aligned} \tag{4.6.12}$$

Algorithm SOLVE MINCLS UNSAT (3.3.14) deduces from S' a minimal unsatisfiable subsystem $\overline{S}'$ with clauses

$$\begin{aligned} &r \vee t \\ &r \vee \neg t \\ &\neg r \end{aligned} \tag{4.6.13}$$

The subsystem S'' of S corresponding to $\overline{S}'$ is

$$\begin{aligned} &\neg q_1 \vee r \vee t \\ &\neg q_2 \vee r \vee \neg t \\ &\neg r \end{aligned} \tag{4.6.14}$$

The variables q_1 and q_2 have literals in S''. Tests T_1, T_2, and T_4 can produce the value of q_1. Thus, $I_1 = \{1, 2, 4\}$. For q_2, we have $I_2 = \{2, 3\}$. For these two I_j sets, $\bigvee_{i \in I_j} \mathit{choose}(T_i)$ of (4.6.10) produces the clauses

$$\begin{aligned} &\mathit{choose}(T_1) \vee \mathit{choose}(T_2) \vee \mathit{choose}(T_4) \\ &\mathit{choose}(T_2) \vee \mathit{choose}(T_3) \end{aligned} \tag{4.6.15}$$

For each variable $choose(T_i)$, the cost of *True* is c_i. The cost of *False* is 0.

The MINSAT instance given by the variables $choose(T_i)$, the costs, and the clauses of (4.6.15) has the unique optimal solution $choose(T_1) = choose(T_3) = choose(T_4) = False$ and $choose(T_2) = True$, with total cost equal to 3. Thus, if test $T_2 = \{1, 2\}$ is done, the values of q_1 and q_2 are determined and produce an S-unacceptable partial assignment.

Heuristic Algorithm

The heuristic method can be summarized as follows.

(4.6.16) Heuristic SOLVE Q-MINFIX UNSAT. *Finds approximate solution for instance of* Q-MINFIX UNSAT *involving selection of tests.*

Input: Satisfiable CNF system S containing $l \geq 1$ special variables $q_1, \ldots, q_l$. An S-unacceptable assignment. For $1 \leq i \leq m$, test T_i for determining the values of a subset of the q_j at a cost c_i. The T_i collectively determine all values. (Thus, $\bigcup_{i=1}^{m} T_i = \{1, 2, \ldots, l\}$.)

Output: A collection of tests T_i that, at low total cost, determine the values of an S-unacceptable partial assignment.

Requires: Algorithms SOLVE MINCLS UNSAT (3.3.14) and SOLVE MINSAT.

Procedure:

1. In S, fix the variables $q_1, \ldots, q_l$ according to the given assignment, and reduce S to an unsatisfiable S' by deleting all satisfied clauses as well as all literals of $q_1, \ldots, q_l$.
2. Apply Algorithm SOLVE MINCLS UNSAT (3.3.14) to S' to find a minimal subsystem $\overline{S}'$ of unsatisfiable clauses. Let S'' be the subsystem of S whose clauses correspond to those of $\overline{S}'$. For each q_j with at least one literal in S'', define I_j to be the set of indices i of the tests T_i for which $j \in T_i$.
3. Define a MINSAT instance with variables $choose(T_i)$. The cost of *True* for $choose(T_i)$ is c_i, while the cost of *False* is 0. For each I_j, the MINSAT instance has the clause $\bigvee_{i \in I_j} choose(T_i)$ of (4.6.10). Solve the MINSAT instance. The variables $choose(T_i)$ with solution value *True* specify the desired collection of tests. Output that collection, and stop.

Exercise (4.8.21)(b) requests that the reader develop sufficient conditions guaranteeing that Heuristic SOLVE Q-MINFIX UNSAT (4.6.16) produces an optimal and not just low-cost collection of tests.

Q-MINFIX SAT Case

We review problem Q-MINFIX SAT (4.5.4). An instance consists of the following: satisfiable CNF systems R and S having $q_1, \ldots, q_l$ among their

common variables; an assignment for $q_1, \ldots, q_l$ that is both R-acceptable and S-acceptable; and, for each q_j, a cost for obtaining the given *True/False* value for that variable. Recall that an extension assignment is obtained from a partial assignment by specifying some *True/False* values for all q_j that are not given a value in the partial assignment. One must find a partial assignment such that every R-acceptable extension assignment is S-acceptable. Subject to that condition, the total cost of the partial assignment must be minimum.

We consider a slight generalization of the problem where, instead of a cost for obtaining the given *True/False* value for each variable q_j, we have costs for obtaining the *True/False* values for subsets of these variables. The subsets correspond to tests $T_1, \ldots, T_m$ involving costs $c_1, \ldots, c_m$, respectively. It is assumed that collectively the tests determine all values. Thus, $\bigcup_{i=1}^{m} T_i = \{1, 2, \ldots, l\}$. The solution of the problem consists of a collection of tests T_i such that the following holds for the partial assignment produced by the tests. Every R-acceptable extension assignment must be S-acceptable. Subject to that condition, the total cost of the collection of tests must be minimum.

The case of Q-MINFIX SAT treated here is readily reduced to Q-MINFIX SAT (4.5.4). Exercise (4.8.22) asks for details of that reduction.

The heuristic scheme solves the problem under the stronger requirement that S must remain satisfiable when the partial assignment is enforced and when all q_j which are not included in that assignment, are deleted. When this condition holds, we say that the partial assignment is *strongly S-acceptable*. The condition implies that each extension assignment of the given partial assignment is S-acceptable, regardless of whether the extension assignment is R-acceptable. Thus, the requirement imposed here is independent of R. Hence, we drop R and consider any instance to consist of S, and S-acceptable assignment, tests T_i, and costs c_i.

To summarize, we want a collection of tests such that the corresponding partial assignment is strongly S-acceptable. Subject to that condition, the collection of tests must have minimum total cost.

The heuristic scheme utilizes, for each q_j, the set I_j containing the indices i for which $j \in T_i$. The steps of the method are as follows.

We delete from S all literals of the q_j that under the given assignment evaluate to *False*. Let S' be the resulting CNF system. For each test T_i, we introduce a variable $choose(T_i)$ that has value *True* when test T_i is selected. We add to S' one clause for each q_j having a literal occurring in S'. If that literal is q_j, the clause is

$$q_j \rightarrow \bigvee_{i \in I_j} choose(T_i) \tag{4.6.17}$$

If the literal is $\neg q_j$, the clause is

$$\neg q_j \rightarrow \bigvee_{i \in I_j} choose(T_i) \tag{4.6.18}$$

Let S'' be the resulting CNF system.

We declare the cost of *True* for $choose(T_i)$ to be c_i, and define the cost of all other *True/False* values of the variables of S'' to be 0. We solve the MINSAT instance given by these costs and S''. We claim that the collection of tests implied by the variables $choose(T_i)$ with optimal solution value *True* is the desired one. The following two observations establish the claim.

First, assume that some *True/False* values for the $choose(T_i)$ are S''-acceptable. Using the definition of S' and S'', it is easily shown that the collection of tests defined via the *True* values of the $choose(T_i)$ supplies a partial assignment for $q_1, \ldots, q_l$ that is strongly S-acceptable.

Second, take any partial assignment of $q_1, \ldots, q_l$ that is strongly S-acceptable. Once more using the definition of S' and S'', one readily confirms that the partial assignment is S''-acceptable, and that any satisfying solution of S'' agreeing with that partial assignment has *True/False* values for the variables $choose(T_i)$ for which the following holds. The variables $choose(T_i)$ with value *True* specify a collection of tests that determine at least the partial assignment.

An example should clarify the method.

Example

The CNF system S has clauses

$$\begin{aligned} &\neg q_1 \vee r \vee t \\ &\neg q_2 \vee r \vee \neg t \\ &q_1 \vee \neg r \\ &q_3 \vee \neg t \end{aligned} \tag{4.6.19}$$

The S-acceptable assignment specifies $q_1 = q_2 = False$ and $q_3 = True$. Tests to obtain these values are $T_1 = \{1\}$, $T_2 = \{1, 2\}$, $T_3 = \{2, 3\}$, and $T_4 = \{1, 3\}$, with associated costs $c_1 = 2$, $c_2 = 3$, $c_3 = 5$, and $c_4 = 3$, respectively.

Since the assignment has $q_1 = False$ (resp. $q_2 = False$, $q_3 = True$), we delete from S all literals q_1 (resp. q_2, $\neg q_3$). The resulting CNF system S' has the clauses

$$\begin{aligned} &\neg q_1 \vee r \vee t \\ &\neg q_2 \vee r \vee \neg t \\ &\neg r \\ &q_3 \vee \neg t \end{aligned} \tag{4.6.20}$$

We have $I_1 = \{1,2,4\}$, $I_2 = \{2,3\}$, and $I_3 = \{3,4\}$. We add to S' the clauses $q_j \to \bigvee_{i \in I_j} choose(T_i)$ of (4.6.17) for $j = 3$ and the clause $\neg q_j \to \bigvee_{i \in I_j} choose(T_i)$ of (4.6.18) for $j = 1, 2$. The resulting S'' is

$$
\begin{aligned}
&\neg q_1 \vee r \vee t \\
&\neg q_2 \vee r \vee \neg t \\
&\neg r \\
&q_3 \vee \neg t \\
&\neg q_1 \to [choose(T_1) \vee choose(T_2) \vee choose(T_4)] \\
&\neg q_2 \to [choose(T_2) \vee choose(T_3)] \\
&q_3 \to [choose(T_3) \vee choose(T_4)]
\end{aligned}
\tag{4.6.21}
$$

We define the cost of *True* for $choose(T_i)$ to be c_i, and declare the costs of all other *True/False* values of the variables of S'' to be 0. We solve the MINSAT instance given by these costs and S''. The optimal solution assigns *True* to $choose(T_1)$ and *False* to all other $choose(T_i)$, and has total cost equal to 2. Thus, the collection of tests consists just of T_1, which determines the value *False* of q_1. That partial assignment is readily verified to be strongly S-acceptable, as it should.

Heuristic Algorithm

Here is the algorithm in compact form.

(4.6.22) Heuristic SOLVE Q-MINFIX SAT. *Finds approximate solution for instance of* Q-MINFIX SAT *involving selection of tests.*

Input: Satisfiable CNF system S containing $l \geq 1$ special variables $q_1, \ldots, q_l$. An S-acceptable assignment for $q_1, \ldots, q_l$. For $1 \leq i \leq m$, test T_i for determining the values of a subset of the q_j at a cost c_i. The T_i collectively determine all values. (Thus, $\bigcup_{i=1}^{m} T_i = \{1, 2, \ldots, l\}$.)

Output: A collection of tests T_i that, at low total cost, produce a partial assignment such that any extension assignment is S-acceptable. (Indeed, any extension assignment is strongly S-acceptable.)

Requires: Algorithm SOLVE MINSAT.

Procedure:

1. Delete from S all literals of $q_1, \ldots, q_l$ that evaluate to *False* for the given assignment. Let S' be the resulting CNF system.
2. For each q_j with at least one literal in S', define I_j to be the set of indices i for which $j \in T_i$. For each q_j having at least one literal q_j (resp. $\neg q_j$) in S', add to the S' the clause $q_j \to \bigvee_{i \in I_j} choose(T_i)$ of

(4.6.17) (resp. $\neg q_j \rightarrow \bigvee_{i \in I_j} choose(T_i)$ of (4.6.18)). Let S'' be the CNF system so derived from S'.

3. Define the cost of *True* of $choose(T_i)$ to be c_i, and declare the costs of all other *True/False* values of the variables of S'' to be 0. Solve the MINSAT instance given by these costs and S''. Output the collection of tests T_i for which $choose(T_i)$ has optimal value *True*, and stop.

Exercise (4.8.21)(c) requests that the reader develop sufficient conditions guaranteeing that Heuristic SOLVE Q-MINFIX SAT (4.6.22) produces an optimal and not just a low-cost selection of tests.

Looking Ahead

Chapter 10 makes extensive use of the heuristic methods of Section 4.6 in so-called question-and-answer processes.

4.7 Further Reading

A comprehensive discussion of quantified propositional formulas is given in Kleine Büning and Lettmann (1999).

The computational complexity of the problems covered in this chapter and of some special cases is established in Eiter and Gottlob (1995).

For a general discussion of the computational complexity of quantified propositional formulas, see Papadimitriou (1994).

Research on the problems of this chapter is proceeding rapidly. The reader should search the Internet for results, algorithms, and software using the keywords "quantified Boolean formula" or "QBF."

4.8 Exercises

The calculations required by some of the exercises may be accomplished with the software of Exercise (4.8.26).

(4.8.1) Solve the Q-ALL SAT instances given by (a) and (b) below.

(a) R is the CNF system

$$q_1 \vee q_2$$

S is the CNF system

$$\neg q_1 \vee q_2 \vee s$$
$$\neg q_1 \vee r \vee t$$
$$\neg q_2 \vee s \vee \neg t$$
$$\neg q_1 \vee \neg q_2 \vee \neg r$$
$$\neg s$$

(b) R is the CNF system

$$\neg q_1 \vee \neg q_2 \vee \neg q_3$$

S is the CNF system

$$\neg q_1 \vee r \vee s$$
$$\neg q_2 \vee s \vee t$$
$$q_3 \vee \neg r$$
$$q_3 \vee \neg t$$
$$\neg q_1 \vee q_2 \vee s \vee t$$
$$\neg s \vee \neg t$$

(4.8.2) Suppose that $q_1, \ldots, q_l$ constitute the variables of R and that they represent decisions. Assume that $q_1, \ldots, q_l$ and $y_1, \ldots, y_n$ are the variables of S. The variables $y_1, \ldots, y_n$ represent results. The clauses of R impose restrictions on the decisions. The clauses of S specify how the decisions influence the results.

(a) Suppose that, for some index j, we add the clause $\neg y_j$ to S. Interpret Q-ALL SAT in this setting.
(b) Same as (a) except that y_j is added to S.
(c) Interpret all combinations of outcomes possible for the Q-ALL SAT instances of (a) and (b).

(4.8.3) Let T be a CNF system. Among its variables are question variables $q_1, \ldots, q_l$. The possible answers for the questions are precisely the *True/False* values for which T with $q_1, \ldots, q_l$ fixed to these values is satisfiable. Let a variable t of T represent a conclusion.

(a) How would one answer the following question: Are there *True/False* values for $q_1, \ldots, q_l$ that allow t to be proved? (*Hint*: Take R to be the CNF system T. Derive S from T by adding a certain clause. Then examine this instance of Q-ALL SAT.)
(b) Define a given q_j of $q_1, \ldots, q_l$ to be *unrelated* to the conclusion t if the *True/False* value of q_j cannot influence whether t can be proved, regardless of the *True/False* values assigned to the remaining questions. Formulate the decision problem whether q_j is unrelated as an instance of Q-ALL SAT. (*Hint*: Construct a CNF system R having the following property. Whenever $q_1, \ldots, q_l$ are fixed to given *True/False* values, then R is satisfiable if and only if, first, T is satisfiable when the same *True/False* values are enforced, and, second, T is satisfiable when the same *True/False* values except for the value for q_j are enforced and when in addition $\neg t$ is enforced. Define S to be T with $\neg t$ enforced.)
(c) Declare the conclusion t to be *futile* if both t and $\neg t$ cannot be proved to be a theorem regardless how $q_1, \ldots, q_l$ are fixed to *True/False* values. Decide futility of t by solving two instances of Q-ALL SAT.

(d) Suppose the conclusion t represents an activity that, depending on circumstances, may be costly or profitable to carry out. The CNF system S encodes regulations regarding the activity. Interpret futility of t in this setting.

(4.8.4) Solve the Q-MIN UNSAT instances given by (a) and (b) below.

(a) R and S are the CNF systems of (4.8.1)(a). The R-costs are given by the following table.

Variable of R	Cost of *True*	Cost of *False*
q_1	5	0
q_2	7	0

(b) R and S are the CNF systems of (4.8.1)(b). The R-costs are given by the following table.

Variable of R	Cost of *True*	Cost of *False*
q_1	3	0
q_2	4	0
q_3	1	0

(4.8.5) Suppose that $q_1, \ldots, q_l$ constitute the variables of R and that they represent decisions. Assume that $q_1, \ldots, q_l$ and $y_1, \ldots, y_n$ are the variables of S. The variables $y_1, \ldots, y_n$ represent results, each of which is desirable or undesirable. The clauses of R impose restrictions on the decisions. The clauses of S specify how the decisions influence the results. For R, costs are associated with each decision.

(a) Suppose that, for some index j, we add the clause $\neg y_j$ to S. Interpret Q-MIN UNSAT in this setting while considering y_j to represent, first, a desirable result and, second, an undesirable result.

(b) Same as (a) except that y_j is added to S.

(4.8.6) Prove the validity of the transformation of problem Q-MINFIX UNSAT (4.3.8) to Q-MIN UNSAT (4.3.6) described in Section 4.3.

(4.8.7) Solve the Q-MAX MINSAT instances given by (a) and (b) below.

(a) R is the CNF system

$$q_1 \vee q_2$$

S is the CNF system

$$\neg q_1 \vee r \vee t$$
$$\neg q_2 \vee s \vee \neg t$$
$$\neg q_1 \vee \neg q_2 \vee r$$

The S-costs are as follows.

Variable of S	Cost of *True*	Cost of *False*
q_1	0	0
q_2	0	0
r	20	0
s	11	0
t	5	10

(b) R is the CNF system

$$\neg q_1 \vee \neg q_2 \vee \neg q_3$$

S is the CNF system

$$\neg q_1 \vee r \vee s$$
$$\neg q_2 \vee s \vee t$$
$$q_3 \vee \neg r$$
$$\neg q_1 \vee q_2 \vee s \vee t$$

The S-costs are as follows.

Variable of S	Cost of *True*	Cost of *False*
q_1	0	0
q_2	0	0
q_3	0	0
r	15	38
s	17	4
t	0	5

(4.8.8) Suppose that $q_1, \ldots, q_l$ constitute the variables of R and that they represent symptoms. Let $q_1, \ldots, q_l$ and $y_1, \ldots, y_n$ be the variables of S. The variables $y_1, \ldots, y_n$ represent defects as well as repair choices for defects. The clauses of R relate the symptoms to each other. The clauses of S specify how the symptoms imply defects and how defects can be remedied by appropriate repairs. In S, costs are associated with the repair variables. Given *True/False* values for the symptoms, a solution of the MINSAT instance defined by S and the S-costs provides the least-cost way to make repairs. Interpret Q-MAX MINSAT for this case. (*Hint*: For given *True/False* values of the symptoms, define the optimal solution of the MINSAT instance involving S and the S-costs to be a *scenario*. Rank the scenarios according to the total S-costs, and declare any scenario with largest-possible total S-cost to be a *worst-case scenario*.)

The next three exercises show that some quantified propositional logic problems actually are SAT or MINSAT problems.

(4.8.9) Consider the following variation of Q-ALL SAT called Q-ALL UNSAT.

Q-ALL UNSAT
Instance: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables.
Solution: Either: "All R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-unacceptable." Or: An R-acceptable assignment of *True/False* values for $q_1, \ldots, q_l$ that is S-acceptable.

Show that Q-ALL UNSAT is SAT in disguise.

(4.8.10) Consider the following variation of Q-MIN UNSAT called Q-MIN SAT.

Q-MIN SAT
Instance: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables. R-costs for the variables of R.
Solution: Either: "All R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-unacceptable." Or: An R-acceptable assignment of *True/False* values for $q_1, \ldots, q_l$ that is S-acceptable and that, subject to that condition, has least total R-cost.

Show that Q-MIN SAT is MINSAT in disguise.

(4.8.11) Consider the following variation of Q-MAX MINSAT called Q-MIN MINSAT.

Q-MIN MINSAT
Instance: Satisfiable CNF systems R and S that have $l \geq 1$ special variables $q_1, \ldots, q_l$ among their common variables. S-costs for the variables of S.
Solution: Either: "All R-acceptable assignments of *True/False* values for $q_1, \ldots, q_l$ are S-unacceptable." Or: An assignment of R-acceptable *True/False* values for $q_1, \ldots, q_l$ such that the total S-cost of the MINSAT instance defined by S and by the S-costs, with $q_1, \ldots, q_l$ fixed according to the selected assignment, is as small as possible.

Show that Q-MIN MINSAT is MINSAT in disguise.

(4.8.12) Construct straightforward enumerative solution algorithms for the problems below when the parameters k and l are small.
(a) Problem P-EXIST Q-ALL SAT (4.5.2).
(b) Problem P-MIN Q-ALL SAT (4.5.3).
(c) Problem P-MIN Q-MAX MINSAT (4.5.5).
(*Hint*: Use the algorithms of Sections 4.2–4.4 as subroutines.)

(4.8.13) Prove that the transformation of problem Q-MINFIX SAT (4.5.4) to problem P-MIN Q-ALL SAT (4.5.3) given in Section 4.5 is valid, by showing that any solution for the transformed problem yields a solution for the initial problem.

(4.8.14) Prove that the application of problem Q-MINFIX SAT described after (4.5.4) is solved by the process specified there.

(4.8.15) Based on the formulas of (4.5.7)–(4.5.9), construct a formula for the polynomial hierarchy at the 4th level, 5th level, and then at the nth level, for any $n \geq 6$. (*Hint*: Distinguish between the cases of odd and even n.)

The quantified propositional formulas introduced in this chapter have particular structure. Exercises (4.8.16)–(4.8.19) below deal with general quantified propositional formulas. For brevity, we refer to them just as quantified formulas.

(4.8.16) (Construction of quantified formula) Quantified formulas may be defined recursively as follows.

(a) Any propositional variable constitutes a formula. The variable is declared to be *free*.

(b) Let S and T be formulas. Then $\neg S$, $S \wedge T$, $S \vee T$, $S \rightarrow T$, $S \leftarrow T$, and $S \leftrightarrow T$ are all formulas. Any free variable of S is also free in $\neg S$, and any free variable of S or T is also free in $S \wedge T$, $S \vee T$, $S \rightarrow T$, $S \leftarrow T$, and $S \leftrightarrow T$.

(c) Let x be a free variable of a formula S. Then $\forall x\, S$ and $\exists x\, S$ are formulas, with free variables as in S except for x.

(d) The construction process is allowed to stop only if the formula at hand has no free variables.

Show details of the construction for the formulas of (4.5.7)–(4.5.9), using the abbreviated notation $\forall z_1 \ldots z_l$ for $\forall z_1 \forall z_2 \ldots \forall z_l$ and $\exists z_1 \ldots z_l$ for $\exists z_1 \exists z_2 \ldots \exists z_l$.

For Exercises (4.8.17)–(4.8.19), it is convenient to assume the following. First, it is supposed that quantified formulas do not use the abbreviated notation. Second, when S and T are combined to $S \wedge T$, $S \vee T$, $S \rightarrow T$, $S \leftarrow T$, or $S \leftrightarrow T$, then any variable x occurring in both S and T is assumed to be free in both S and T. If this is not the case for some x, we have a *coincidence of variable names*. Assume such a coincidence is at hand. Without loss of generality, assume that x is not free in S. Then we rename x in S to eliminate that coincidence.

(4.8.17) (Evaluation of quantified formulas–part one) Define U to be the set $\{t, f\}$. Let a quantified formula S be given. Do the following process.

(a) Replace each occurrence of $\forall x$ by $\bigwedge_{u_x \in U}$.

(b) Replace each occurrence of $\exists x$ by $\bigvee_{u_x \in U}$.
(c) Replace each occurrence of each variable x by $x(u_x)$.
(d) Let S' be the formula resulting from (a)–(c).
(e) Let S'' be the CNF system that for each variable x of the original formula S contains the two clauses $\bigvee_{u_x \in U} x(u_x)$ and $\bigvee_{u_x \in U} \neg x(u_x)$.
(f) Define P to be the propositional formula $S' \wedge S''$.
(g) Declare S to evaluate to *True* (resp. *False*) if P is satisfiable (resp. unsatisfiable).

Show that the evaluation of the quantified formula (4.1.1) as defined in Section 4.1 is consistent with the above definition via P.

(4.8.18) (Evaluation of quantified formulas–part two) Let S be a quantified formula. Process S as follows.
(a) If $S = \neg T$, process T recursively.
(b) If $S = \forall x\, T$ (resp. $S = \exists x\, T$), declare $S = T_{x=True} \wedge T_{x=False}$ (resp. $S = T_{x=True} \vee T_{x=False}$), and process $T_{x=True}$ and $T_{x=False}$ recursively. The formula $T_{x=True}$ (resp. $T_{x=False}$) is obtained from T by replacing each occurrence of x by *True* (resp. *False*) and by replacing each occurrence of $\neg x$ by *False* (resp. *True*).
(c) If S is equal to any one of $R \wedge T$, $R \vee T$, $R \leftarrow T$, $R \rightarrow T$, $R \leftrightarrow T$, then process R and T recursively.
(d) Stop the recursive processing when the formula on hand does not contain any variables.

Declare S to have the value *True* (resp. *False*) if the final formula has that value. Show that this evaluation of S is consistent with that of Exercise (4.8.17). (*Hint*: Use induction on the steps of construction sequences of quantified formulas as described in Exercise (4.8.16).)

(4.8.19) (Evaluation of quantified formulas–part three) Let S and T be quantified formulas.
(a) Show that $S \wedge T$ evaluates to *True* if and only if both S and T evaluate to *True*.
(b) Show that $S \vee T$ evaluates to *True* if and only if S or T evaluates to *True*.
(c) Using the conclusions of (a)–(c), devise rules for the evaluation of $S \rightarrow T$, $S \leftarrow T$, and $S \leftrightarrow T$.
(d) Show that $\neg(\forall x\, S)$ and $\exists x\, (\neg S)$ have the same *True/False* value.
(e) Show that $\neg(\exists x\, S)$ and $\forall x\, (\neg S)$ have the same *True/False* value.
(f) Show that S evaluates to *True* if and only if $\neg S$ evaluates to *False*.

(*Hint*: Use the evaluation process of Exercise (4.8.18).)

(4.8.20) Apply Heuristic SOLVE Q-ALL SAT (4.6.9) to the Q-ALL SAT instances of Exercise (4.8.1).

(4.8.21) Develop sufficient conditions that guarantee that the following heuristic methods are exact.

(a) Heuristic SOLVE Q-ALL SAT (4.6.9). (*Hint*: Let $\overline{S}'$ of Step 3 be unique.)
(b) Heuristic SOLVE Q-MINFIX UNSAT (4.6.16). (*Hint*: Let $\overline{S}'$ of Step 2 be unique.)
(c) Heuristic SOLVE Q-MINFIX SAT (4.6.22). (*Hint*: Assume that each q_j occurs in S either only nonnegated or only negated.)

(4.8.22) Reduce the more general cases of Q-MINFIX UNSAT and Q-MINFIX SAT involving tests and treated in Section 4.6, to problems Q-MINFIX UNSAT (4.3.8) and Q-MINFIX SAT (4.5.4), respectively.

(4.8.23) Apply Heuristic SOLVE Q-MINFIX UNSAT (4.6.16) to S given by

$$\neg q_1 \vee r \vee s$$
$$\neg q_2 \vee s \vee t$$
$$q_3 \vee \neg r$$
$$q_3 \vee \neg t$$
$$\neg s$$

The S-unacceptable assignment specifies $q_1 = q_2 = \mathit{True}$ and $q_3 = \mathit{False}$. The tests are $T_1 = \{1, 2\}$, $T_2 = \{2, 3\}$, and $T_3 = \{1, 3\}$, with costs $c_1 = 3$, $c_2 = 5$, and $c_3 = 9$, respectively.

(4.8.24) Apply Heuristic SOLVE Q-MINFIX SAT (4.6.22) to the case of Exercise (4.8.23), except that the assignment is $q_1 = q_2 = \mathit{False}$ and $q_3 = \mathit{True}$ and thus is S-acceptable.

(4.8.25)
(a) Prove that Heuristic SOLVE Q-MINFIX UNSAT (4.6.16) cannot output an empty collection of tests T_i.
(b) Prove that Heuristic SOLVE Q-MINFIX SAT (4.6.16) may output an empty collection of tests. How should that output be interpreted?

(4.8.26) (Optional programming project) There are two options if the reader wishes to avoid manual solution of some of the exercises. First, the reader may obtain commercially or publicly available software for at least some of the problems. Any search engine on the Internet will point to such software. Second, the user may opt to write programs that implement the algorithms of (a)–(f) below, and then use that software to solve the exercises. The task is easy if the reader has already created programs for SAT and MINSAT as described in Exercise (2.9.13).
(a) Algorithm SOLVE Q-ALL SAT (4.2.9).
(b) Algorithm SOLVE Q-MIN UNSAT (4.3.7).
(c) Algorithm SOLVE Q-MAX MINSAT (4.4.7).
(d) Heuristic SOLVE Q-ALL SAT (4.6.9).
(e) Heuristic SOLVE Q-MINFIX UNSAT (4.6.16).
(f) Heuristic SOLVE Q-MINFIX SAT (4.6.22).

Part II

Formulation of Logic Systems

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 5

Basic Formulation Techniques

5.1 Overview

The formulation of logic relationships using logic variables and clauses is partly art and partly science. The art portion is difficult to describe and is best acquired by looking at a number of examples. In contrast, the science portion lends itself to analysis and description. This chapter and Chapter 6 cover key elements of the science portion. Here, we describe basic results that are applicable to almost any setting.

Section 5.2 discusses elementary considerations concerning variables and clauses.

Sections 5.3–5.5 tell how one may identify flaws in a formulation. Section 5.3 covers the detection of redundant clauses. Section 5.4 deals with the identification of inconsistent clauses. Section 5.5 discusses the overall validation of logic formulations.

Section 5.6 introduces the decision pyramid concept, which is essential for the construction of compact formulations for complex situations.

Section 5.7 discusses how problem formulations using the decision pyramid concept simplify the problem of explaining decisions.

Section 5.8 shows how certain theorem-proving cases can be reduced by a MINSAT approach. The method is particularly useful when decision pyramids are used.

Section 5.9 deals with formulations arising from decision graphs, which are graphs derived from trees or acyclic graphs. Each of the graphs naturally leads to a decision pyramid.

Section 5.10 explores the limits of formulating situations in logic. The section also provides handy rules for compact representation of certain constraints.

Section 5.11 provides references for further reading.

Section 5.12 contains exercises.

5.2 Variables and Clauses

Usually, the text describing some situation or problem does not explicitly specify the logic variables that allow modeling of the relationships. Instead, one must guess appropriate logic variables, extract key facts from the description, and then attempt to express their relationships in clauses. We discuss the process using an example.

Example: Jobs Problem

Consider the following setting.

(5.2.1) Jobs Problem: Mary and Robert hold one job each. The jobs are nurse and teacher. The nursing job is held by a male person. What is Mary's job?

Determination of Variables and Sets

We select the following variables.

(5.2.2)

Variable	Meaning
$has\_job(x, y)$	x has job y
$is\_male(x)$	x is male

Variables for Jobs Problem (5.1.1)

Why have we selected these variables? We do not have a good justification, except to say that these variables seem a reasonable choice and suffice for the clauses to come. Of course, this flimsy explanation is not of much help. Better advice is: Define some variables and attempt to express the facts in clauses. If it can be done, good; if not, variables must be changed or added until the formulation is possible.

Using similar pragmatic reasoning, we define the following sets.

(5.2.3)
$$\begin{aligned} jobs &= \{nurse, teacher\} \\ persons &= \{Mary, Robert\} \end{aligned}$$

Determination of Clauses

How do we determine logic clauses that correctly depict the facts? A good way is to take the sentences one after another and express the logic content in clauses.

The first two sentences, which are "Mary and Robert hold one job each" and "The jobs are nurse and teacher," can be expressed in logic in several ways. Since there are exactly as many persons as there are jobs, we may rephrase the sentences as follows. First, each person has at most one of the jobs. Second, each job is done by at least one of the persons. The corresponding logic formulas are

$$\begin{aligned} &\forall x \in persons[\neg\, has\_job(x, nurse) \vee \neg\, has\_job(x, teacher] \\ &\forall y \in jobs[has\_job(Mary, y) \vee has\_job(Robert, y)] \end{aligned} \tag{5.2.4}$$

The third sentence, "The job of nurse is done by a male person," is represented by the formula

$$\forall x \in persons[has\_job(x, nurse) \rightarrow is\_male(x)] \tag{5.2.5}$$

Since we have processed all sentences describing the facts, we may think that all facts have been formulated as clauses. But there may be additional facts that are implicitly present and that we must still account for. Here, Mary is female and Robert is male. At least we would guess that from the names. Thus, we have the clause

$$\neg\, is\_male(Mary) \wedge is\_male(Robert) \tag{5.2.6}$$

Answer for Jobs Problem

The question "What is Mary's job?" is handled by the techniques for decision making described in Section 2.5. First, we convert the clauses (5.2.4)–(5.2.6) to CNF clauses. The resulting formulation is

$$\begin{aligned} &\neg\, has\_job(Mary, nurse) \vee \neg\, has\_job(Mary, teacher) \\ &\neg\, has\_job(Robert, nurse) \vee \neg\, has\_job(Robert, teacher) \\ &has\_job(Mary, nurse) \vee has\_job(Robert, nurse) \\ &has\_job(Mary, teacher) \vee has\_job(Robert, teacher) \\ &\neg\, has\_job(Mary, nurse) \vee is\_male(Mary) \\ &\neg\, has\_job(Robert, nurse) \vee is\_male(Robert) \\ &\neg\, is\_male(Mary) \\ &is\_male(Robert) \end{aligned} \tag{5.2.7}$$

Next, we compute a satisfying solution for (5.2.7). That solution provides $has\_job(Mary, teacher) = True$ and $has\_job(Mary, nurse) = False$. Thus, Mary is a teacher and not a nurse.

Exercises (5.12.1) and (5.12.2) ask the reader to carry out the calculations and to answer a related question.

As the complexity of problems grows, it becomes harder to design correct formulations. To keep the task manageable, we should begin with a simple formulation that only approximately models the situation. We check and correct that formulation as needed. Then we enrich the formulation, and check and improve that version. This recursive process continues until the entire problem has been correctly modeled. If the process is to succeed, the checking and correcting of formulations must be done quickly and effectively. The subsequent three sections tell how that goal can be achieved, by showing how redundant and inconsistent clauses are found, and how formulations can be validated. First, we deal with the identification of redundant clauses.

5.3 Redundant Clauses

When we derive a formulation from a given description of facts, we may inadvertently define clauses that actually are not needed.

Example

Implicit in the jobs problem (5.2.1) is the fact that each of the two persons has at least one of the two jobs. This fact is represented by the formula

$$\forall x \in persons[has\_job(x, teacher) \vee has\_job(x, nurse)] \tag{5.3.1}$$

Translation to CNF gives the following two clauses.

$$\begin{aligned} &has\_job(Mary, nurse) \vee has\_job(Mary, teacher) \\ &has\_job(Robert, nurse) \vee has\_job(Robert, teacher) \end{aligned} \tag{5.3.2}$$

Must the two clauses of (5.3.2) be added to the previous formulation (5.2.7), or are they not needed? Clearly, the two clauses must be added to (5.2.7) if and only if they eliminate at least one satisfying solution of (5.2.7). Thus, the two clauses are not needed if and only if every satisfying solution of (5.2.7) also satisfies the two clauses. In the terminology of theorem proving introduced in Section 2.5, the latter condition is equivalent to demanding that the two clauses are theorems of (5.2.7).

Exercise (5.12.4) calls for confirmation that the clauses of (5.3.2) are indeed theorems of (5.2.7). Hence, these clauses are not needed. We say that they are *redundant* for (5.2.7).

The general case of redundancy is defined and tested for as follows.

Redundancy

Let S and T be propositional formulas. Then T is *redundant for* S if S and $S \wedge T$ have the same set of satisfying solutions or, equivalently, if T is a theorem of S. Thus, we may test for redundancy by theorem proving. If T is redundant for S, then we also say that T is *redundant in* the formula $S \wedge T$.

Redundancy is maintained under several operations. The next theorem summarizes important cases.

(5.3.3) Theorem. *Let S and T be propositional formulas. Assume T is redundant for S. Suppose S' and T' are derived from S and T by any one of the operations of* (a)–(c) *below. Then T' is redundant for S'.*

(a) *Using an arbitrary propositional formula R, define $S' = S \wedge R$, and let $T' = T$ or $T' = T \wedge R$.*
(b) *Fix a given variable to a given True/False value either in S or in both S and T.*
(c) *Assume that S and T are CNF systems. Delete all occurrences of a given variable from both S and T.*

Proof. By assumption, T is a theorem of S. Thus, $S \wedge \neg T$ is unsatisfiable. For each case, we show that T' is a theorem of S', by showing $S' \wedge \neg T'$ to be unsatisfiable.

For the proof of (a), let $S' = S \wedge R$, and first consider the case $T' = T$. Since $S \wedge \neg T$ is unsatisfiable, $S' \wedge \neg T' = S \wedge R \wedge \neg T$ is unsatisfiable. For the case $T' = T \wedge R$, we have $S' \wedge \neg T' = S \wedge R \wedge \neg(T \wedge R) = (S \wedge R \wedge \neg T) \vee (S \wedge R \wedge \neg R)$. We have seen that $S \wedge R \wedge \neg T$ is unsatisfiable. Also, $S \wedge R \wedge \neg R$ is trivially unsatisfiable. Hence, $S' \wedge \neg T'$ is unsatisfiable.

If variable x has been fixed in (b), then use $R = x$ or $R = \neg x$ in (a) to prove (b).

We prove (c) by contradiction. Suppose that a variable x has been deleted from S and T, and that $S' \wedge \neg T'$ has a satisfying solution. We show that the solution plus an appropriate *True/False* value for x satisfy both $\neg T$ and S and thus $S \wedge \neg T$, a contradiction.

Since T and T' are CNF systems, $\neg T$ and $\neg T'$ are DNF systems. A DNF system is a disjunction of conjunctions. Hence, the DNF system $\neg T'$ evaluates to *True* if and only if at least one of its conjunctions evaluates to *True*. Select one such conjunction of $\neg T'$. The corresponding conjunction of $\neg T$ is either identical or contains x or $\neg x$. Regardless of the case, we can assign a *True/False* value to x such that this conjunction of $\neg T$ evaluates to *True*. Since S may be obtained from S' by adding $\neg x$ or x to some of the clauses of S', that extended solution satisfies S as well. □

When is testing for redundancy called for? This is a difficult question since redundant clauses in a CNF system may improve or worsen the

performance of a given satisfiability algorithm applied to the CNF system. Given that fact, we take a pragmatic viewpoint and suggest search for and removal of redundant clauses if a formulation seemingly contains too many clauses and has become unmanageable. In that case, apply the following process to each clause of S: Delete the clause from S, check if the clause is a theorem of the reduced S, and reinsert it into S if the clause is not a theorem. At termination of the process, the final S contains no redundant clauses. Note that the final S in general is not unique and depends on the order in which clauses are processed. Since we are taking a pragmatic approach to the problem of redundancy, we ignore that aspect and select clauses for processing in any convenient order.

The next section deals with the detection of inconsistencies in formulations.

5.4 Inconsistent Clauses

Suppose that we have a created a formulation that, according to the application, should be satisfiable, but that turns out to be unsatisfiable. Thus, the formulation contains some error. Assuming that the variables are properly selected, the error stems from incorrect clauses. How can we determine those clauses? We use the jobs problem (5.2.1) as an example to motivate and explain a solution approach.

Example

The jobs problem (5.2.1) is correctly encoded by the formulas of (5.2.4)–(5.2.6). Suppose we make a mistake when typing the formula of (5.2.6) and add the $\neg$ sign in front of $is\_male(Robert)$. We thus have the formula

$$\neg\, is\_male(Mary) \wedge \neg\, is\_male(Robert) \tag{5.4.1}$$

Due to that error, (5.2.7) becomes

$$\begin{aligned}
&\neg\, has_job(Mary, nurse) \vee \neg\, has_job(Mary, teacher)\\
&\neg\, has_job(Robert, nurse) \vee \neg\, has_job(Robert, teacher)\\
&has_job(Mary, nurse) \vee has_job(Robert, nurse)\\
&has_job(Mary, teacher) \vee has_job(Robert, teacher)\\
&\neg\, has_job(Mary, nurse) \vee is_male(Mary)\\
&\neg\, has_job(Robert, nurse) \vee is_male(Robert)\\
&\neg\, is_male(Mary)\\
&\neg\, is_male(Robert)
\end{aligned} \tag{5.4.2}$$

Exercise (5.12.5) asks for verification that (5.4.2) is unsatisfiable. This fact implies that (5.4.2) is in error. How can we localize the error? In general, this is a hard problem that has no easy solution. But we can try to reduce the search effort for the error by identifying a subset of clauses that, by themselves, are unsatisfiable. At least one of the clauses of any such subset must then be in error. One such unsatisfiable subsystem of (5.4.2) is

$$
\begin{array}{ll}
 & has\_job(Mary, nurse) \vee has\_job(Robert, nurse) \\
 & \neg\, has\_job(Mary, nurse) \vee is\_male(Mary) \\
(5.4.3) & \neg\, has\_job(Robert, nurse) \vee is\_male(Robert) \\
 & \neg\, is\_male(Mary) \\
 & \neg\, is\_male(Robert)
\end{array}
$$

Indeed, (5.4.3) is a minimal unsatisfiable subsystem of (5.4.2) since deletion of any clause from (5.4.3) results in a satisfiable subset. Exercise (5.12.5) calls for confirmation of this claim.

The clauses of (5.4.2) as well as of (5.4.3) are inconsistent in the sense that they are unsatisfiable when we expect them to be satisfiable. We now treat the general case of such inconsistency.

Inconsistency

Let S be a CNF system that is unsatisfiable but that, according to the application producing S, is expected to be satisfiable. We then say that the clauses of S are *inconsistent*. In the typical case, we also have a subsystem $\overline{S}$ defined by some clauses of S that we know to be correct. Thus, any errors must be in the clauses of S outside $\overline{S}$. To localize at least one such error, we want a minimal unsatisfiable subsystem obtained from S by the deletion of clauses not in $\overline{S}$. Such a subsystem is a *minimal inconsistent* subsystem of S relative to $\overline{S}$. Algorithm SOLVE MINCLS UNSAT (3.3.14) finds such a subsystem and thus helps localize at least one error.

Suppose we correct an error in S. If the new S is satisfiable, it is consistent, and we stop. Otherwise, other errors are still present, and we locate them by recursive application of the above process.

We have dealt with redundancy and inconsistency. Suppose we have a consistent CNF system S where redundancy of any clauses, if at all present, is acceptable. The next section shows how we may ascertain that S is a correct representation of the given problem domain. That step is called validation.

5.5 Validation

Checking whether a CNF system S represents a given problem domain correctly is a difficult task. We take a pragmatic viewpoint that leads to a workable checking process.

Define $\mathcal{T}$ to be the set of logic formulas that are defined from the variables of the problem domain and that are theorems of that domain. That is, for each set of *True/False* values that the logic variables may take on in the problem domain, each $T \in \mathcal{T}$ evaluates to *True*. Define $\mathcal{N}$ to be the logic formulas that are defined from the variables of the domain and that are not in $\mathcal{T}$. Hence, each $T \in \mathcal{N}$ is not a theorem of the domain and evaluates to *False* for some *True/False* values that the logic variables take on in the problem domain.

Define S to be *correct* for the problem domain if each $T \in \mathcal{T}$ is a theorem of S and if each $T \in \mathcal{N}$ is not a theorem of S. Equivalently, for each $T \in \mathcal{T}$ (resp. $T \in \mathcal{N}$), $S \wedge \neg T$ must be unsatisfiable (resp. satisfiable). *Validating* S means confirming that S is correct for the problem domain.

Suppose each T of $\mathcal{T}$ or $\mathcal{N}$ is a CNF clause. The vast majority of cases discussed in this book is of that type. Then each variable of the domain may occur nonnegated or negated in T, or may not occur at all. Hence, if the problem domain has n logic variables, we have 3^n possible statements T for $\mathcal{T} \vee \mathcal{N}$. Unless n is small, 3^n is a large or even huge number that precludes testing of each T to verify correctness of S.

Instead, we confine ourselves to testing subsets of statements T that we consider significant for the application. As symbols for these subsets we reuse $\mathcal{N}$ and $\mathcal{T}$. The selection of the subsets may rely on insight into the domain structure, prior lists of important statements T, random generation of statements T using relevant probability distributions, or some other process that generates important statements for the domain.

Validating S with respect to $\mathcal{T}$ and $\mathcal{N}$ means confirming that all statements of $\mathcal{T}$ (resp. $\mathcal{N}$) are (resp. are not) theorems of the problem domain.

It is mandatory that we construct the subsets $\mathcal{T}$ and $\mathcal{N}$ very carefully so that they do not contain any errors. If that condition is not met, the validation process becomes futile.

Let us look at an example.

Example

Consider the jobs problem (5.2.1), which we repeat here.

(5.5.1) Jobs Problem: Mary and Robert hold one job each. The jobs are nurse and teacher. The job of nurse is held by a male person. What is Mary's job?

From the discussion of Section 5.2, we know that Mary is a teacher and Robert is a nurse. Thus,

$$\begin{aligned} T_1 &= has\_job(Mary, teacher) \\ T_2 &= has\_job(Robert, nurse) \end{aligned} \tag{5.5.2}$$

are theorems of the problem domain. We declare $\mathcal{T}$ to consist of these two statements. We also know that Mary is not a nurse and Robert is not a teacher. Hence,

$$\begin{aligned} T_3 &= has\_job(Mary, nurse) \\ T_4 &= has\_job(Robert, teacher) \end{aligned} \tag{5.5.3}$$

are not theorems of the domain. These two statements make up $\mathcal{N}$.

Suppose somebody has given us the following formulation S of the jobs problem and wants us to validate it with respect to the sets $\mathcal{T}$ and $\mathcal{N}$ just defined.

$$\begin{aligned} &\neg\, has\_job(Mary, nurse) \vee \neg\, has\_job(Mary, teacher) \\ &\neg\, has\_job(Robert, nurse) \vee \neg\, has\_job(Robert, teacher) \\ &has\_job(Mary, nurse) \vee is\_male(Mary) \\ &has\_job(Robert, nurse) \vee is\_male(Robert) \\ &\neg\, is\_male(Mary) \\ &is\_male(Robert) \end{aligned} \tag{5.5.4}$$

As an aside, the formulation S may be derived from the correct formulation (5.2.7) by omitting the third and fourth clauses from (5.2.7) and dropping the negation $\neg$ occurring in the fifth and sixth clauses of (5.2.7).

As we shall see, S contains several errors. We find them while we attempt to validate S with respect to $\mathcal{T} = \{T_1, T_2\}$ and $\mathcal{N} = \{T_3, T_4\}$.

We first check for redundancy and consistency of the clauses of S. Exercise (5.12.6) demands verification that there are no redundant clauses and that the clauses are consistent. We proceed to validate S with respect to $\mathcal{T}$ and $\mathcal{N}$.

We begin by trying to confirm that T_3 and T_4 of $\mathcal{N}$ are not theorems of S. We detect that, with $\neg T_3 = \neg\, has\_job(Mary, nurse)$, $S \wedge \neg T_3$ is unsatisfiable. Thus, T_3 is a theorem of S while it should not be a theorem. How can we isolate the error of S producing this effect? Clearly, any unsatisfiable subset of clauses of $S \wedge \neg T_3$ contains an error. Hence, to narrow down the search for an error, we compute a minimal subset of unsatisfiable clauses using Algorithm SOLVE MINCLS UNSAT (3.3.14). The algorithm derives the following subset from (5.5.4).

$$\begin{aligned} &has\_job(Mary, nurse) \vee is\_male(Mary) \\ &\neg\, is\_male(Mary) \\ &\neg\, has\_job(Mary, nurse) \end{aligned} \tag{5.5.5}$$

We recognize that the first clause, which is taken from S and which supposedly represents "If Mary is a nurse, then Mary is male," is missing the negation $\neg$ in front of the term $has\_job(Mary, nurse)$.

We correct that error and also correct the related error in the clause $has\_job(Robert, nurse) \vee is\_male(Robert)$ of (5.5.4). These changes convert (5.5.4) to the following formulation S'.

$$
\begin{aligned}
&\neg\, has\_job(Mary, nurse) \vee \neg\, has\_job(Mary, teacher)\\
&\neg\, has\_job(Robert, nurse) \vee \neg\, has\_job(Robert, teacher)\\
&\neg\, has\_job(Mary, nurse) \vee is\_male(Mary)\\
&\neg\, has\_job(Robert, nurse) \vee is\_male(Robert)\\
&\neg\, is\_male(Mary)\\
&is\_male(Robert)
\end{aligned}
\tag{5.5.6}
$$

The change does not guarantee that S' is correct for T_3 or, for that matter, for T_4, since there may be additional errors that cause T_3 or T_4 to be theorems of S'. Hence, we repeat the above process to check for further errors. We see that $S \wedge \neg T_3$ and $S \wedge \neg T_4$ are satisfiable, so T_3 and T_4 are not theorems of S', as desired. We conclude that S' has been validated with respect to $\mathcal{N}$.

We proceed to the second part, where we check validity of S' with respect to $\mathcal{T} = \{T_1, T_2\}$. With $\neg T_1 = \neg\, has\_job(Mary, teacher\}$, we determine for $S' \wedge \neg T_1$ the following satisfying solution.

$$
\begin{aligned}
has\_job(Mary, nurse) &= False\\
has\_job(Mary, teacher) &= False\\
has\_job(Robert, nurse) &= True\\
has\_job(Robert, teacher) &= False\\
is\_male(Mary) &= False\\
is\_male(Robert) &= True
\end{aligned}
\tag{5.5.7}
$$

That solution proves T_1 to be not a theorem of S' and thus establishes S' to be in error. How can we eliminate the error? Since the solution values of (5.5.7) induce for T_1 a value of *False*, the situation depicted by that solution is not possible in the problem domain. Accordingly, we should change a clause of S' or add a clause to S' to rule out that solution.

We argue that we may restrict ourselves to adding a clause. Suppose a clause is incorrect, yet we do not change it and instead add a new, correct clause. Due to that change, the incorrect clause may become redundant, or it may cause a non-theorem $T \in \mathcal{N}$ to become a theorem. The first case has no effect on validity, while the second case is detected by the repeated validation of S with respect to $\mathcal{N}$ introduced in a moment. Hence, we can restrict ourselves to adding a clause.

We use a pragmatic approach to construct the clause to be added. We compare the solution values of (5.5.7) with the problem formulation (5.5.1) and look for a discrepancy. The clash of the solution values with a sentence or with a few sentences of the problem formulation gives a clue to the clause or clauses to be added. Here, the solution values $has\_job(Mary, teacher) = False$ and $has\_job(Robert, teacher) = False$ clash with the demand that each job is held by some person. Accordingly, we are justified to rule out that solution by adding to S' the clause

(5.5.8) $$has\_job(Mary, teacher) \vee has\_job(Robert, teacher)$$

Due to the symmetry, we are justified to also add the following clause.

(5.5.9) $$has\_job(Mary, nurse) \vee has\_job(Robert, nurse)$$

Let S'' be the resulting formulation. As discussed a moment ago, it is possible that for S'' some statements of $\mathcal{N}$ are theorems, in contrast to the situation for S'. In such a case, we must find and correct errors in S'' that eliminate all such occurrences. Once the required adjustments have been done, we return to $\mathcal{T}$ and check whether all $T \in \mathcal{T}$ are theorems. We proceed in this fashion, alternating between $\mathcal{N}$ and $\mathcal{T}$, to correct and add clauses, until no further changes are needed. The process terminates provided we do not introduce new errors when we correct or add clauses, since then each clause is corrected at most once and since only a finite number of new clauses can be added. On the other hand, if new errors are introduced, the process may go on forever. Hence, utmost care is needed to avoid errors as clauses are modified or added.

In our example case, S'' is the original, correct formulation of (5.2.7), for which all statements of $\mathcal{T}$ are theorems and for which all statements of $\mathcal{N}$ are non-theorems. Hence, S'' is validated with respect to $\mathcal{T}$ and $\mathcal{N}$.

The reader may be puzzled by our treatment of $\mathcal{T} = \{T_1, T_2\}$. We corrected S' so that T_1 became a theorem by adding the clauses (5.5.8) and (5.5.9). Why didn't we simply add $T_1 = has\_job(Mary, teacher)$ itself as clause? That criticism is well placed for the example problem. In general, though, we should take the fact that a $T \in \mathcal{T}$ is not a theorem to be an indicator of a possibly major formulation error. Thus, we should not just add T, but instead should use the satisfying solution found for $S' \wedge \neg T$ as a helpful tool when we search for and correct as many errors as we can detect. The above discussion reflects that idea.

We proceed to the general case of validating formulations.

Validation Algorithm

The following algorithm for the validation of formulations is based on the pragmatic approach taken above for the example problem.

(5.5.10) Algorithm VALIDATE CNF SYSTEM. *Validates* CNF *system S with respect to given sets $\mathcal{T}$ and $\mathcal{N}$.*

Input: CNF system S. Sets $\mathcal{T}$ and $\mathcal{N}$ of statements that must be theorems and non-theorems, respectively, of any correct formulation. It is advantageous but not required that S has been checked for and, if necessary, corrected to achieve consistency of the clauses and to reduce redundancy of clauses, if at all present, to an acceptable level.

Output: A revised CNF system S for which all statements of $\mathcal{T}$ (resp. $\mathcal{N}$) are theorems (resp. non-theorems).

Requires: Algorithm SOLVE MINCLS UNSAT(3.3.14).

Procedure:

1. For each $T \in \mathcal{N}$: Check if $S \wedge \neg T$ is unsatisfiable. If this is the case, derive a minimal unsatisfiable subset of the clauses of $S \wedge \neg T$ by Algorithm SOLVE MINCLS UNSAT (3.3.14), and use that subset to isolate and correct an error in S.
2. If during the most recent pass through Step 1 the CNF system S was changed, go to Step 1.
3. For each $T \in \mathcal{T}$: Check if $S \wedge \neg T$ is satisfiable. If this is the case, compare the related satisfying solution with the description of the problem domain to find a discrepancy, define a clause that rules out the discrepancy, add that clause to S, and go to Step 1.
4. If during the most recent pass through Step 3 the CNF system S was changed, go to Step 1.
5. Optionally, reduce redundancy of clauses, if at all present, to an acceptable level.
6. Output S as a CNF system that has been validated for $\mathcal{T}$ and $\mathcal{N}$.

In the next section, we discuss the structure of formulations.

5.6 Decision Pyramid

So far, we have defined variables and clauses freely and without considering any helpful framework or concepts. That approach is acceptable when the problem domain is small, but is not advisable when problems are complex. For such cases, we introduce a framework called *decision pyramid*. The main ideas and uses of decision pyramids are best explained using an example.

Example

Diagnosing a disease using symptoms is in principle a straightforward task, or so it seems. We construct a logic formulation that links symptoms and

diseases, collect *True/False* values for the symptoms, fix the symptom variables of the logic formulation at these values, and determine the applicable disease by theorem proving. This naïve viewpoint ignores several difficulties, among them the problem that a straightforward logic formulation may require so many clauses that a manual construction is not possible. How can we overcome that problem?

The clue for a compact formulation is provided by the way a physician derives diagnoses via intermediate conclusions. For example, symptoms reported by a patient may lead a psychiatrist to the intermediate conclusion that the patient had a series of panic attacks. Further discussion may result in a second intermediate conclusion that the patient suffers from severe depression. Continuing the exploration, the psychiatrist proves or rules out additional intermediate conclusions, each of which builds upon symptoms or conclusions already drawn or ruled out. Eventually, the reasoning process results in a final diagnosis of the mental illness.

Defining the Pyramid

We represent the above human reasoning approach using a pyramid. At the bottom of the pyramid we place all variables representing symptoms of the patient. We declare these variables to make up layer 0 of the pyramid. Immediately above layer 0 are variables for conclusions that can be deduced from the variables of layer 0. These variables constitute layer 1. Moving upward to layer 2, we have the variables of conclusions that can be expressed in terms of the variables of layers 0 and 1. We continue in this fashion for layers $i = 3, 4, \ldots$, and so on. Each variable of a layer represents a conclusion that can be deduced from the variables of the layers below. The highest layer consists of the variables that represent the final conclusions or diagnoses.

While we define the layers of the variables, we also express the relationships connecting the variables of each layer with the variables of the layers below. Thus, when we have processed the top layer, we have not only created all variables but also all clauses for the given problem.

Why should we use a decision pyramid? Why not just define the variables of the symptoms and of the final conclusions, and then link them directly by clauses? A short and incomplete answer is "Such a direct approach often requires an extraordinary number of clauses." A longer and more satisfactory answer requires a detour into the notion of *projection*, as follows.

Let S be a CNF system with variables $x_1, x_2, \ldots, x_m$ and $y_1, y_2, \ldots, y_n$. A CNF system S' with variables $x_1, x_2, \ldots, x_m$ is a *projection* of S if the following two conditions are satisfied. First, for every satisfying solution of S, the values assigned to $x_1, x_2, \ldots, x_m$ of S constitute a satisfying solution for the variables $x_1, x_2, \ldots, x_m$ of S'. Second, for every satisfying solution

for the variables $x_1, x_2, \ldots, x_m$ of S', there exist values for $y_1, y_2, \ldots, y_n$ such that the solution values for $x_1, x_2, \ldots, x_m$ and $y_1, y_2, \ldots, y_n$ together constitute a satisfying solution for S. When S' has been obtained by projection as defined, we also say that the variables $y_1, y_2, \ldots, y_n$ have been *projected out*.

The definition of projection directly implies the following. If S is a correct formulation, then S' obtained by projection is correct as well.

Consider the case where an S for medical diagnosis has been obtained via the decision pyramid approach. The variables $x_1, x_2, \ldots, x_m$ represent the symptoms of layer 0 and the final diagnoses of the top layer, and the variables $y_1, y_2, \ldots, y_n$ represent the intermediate conclusions of the layers in between. The projection S' contains just $x_1, x_2, \ldots, x_m$ and therefore has the variables of the symptoms and final diagnoses and has no variables of the intermediate conclusions. Hence, one may view S' to be a direct formulation that links symptoms and final conclusions without use of intermediate conclusions.

The *resolution process* projects out variables and thus converts a correct S to a correct S'. To explain the process, suppose instead of $y_1, y_2, \ldots, y_n$ in S we have just one variable y. Since clauses of S containing both y and $\neg y$ always evaluate to *True* and thus can always be deleted, we may assume that each clause of S either contains y, or contains $\neg y$, or does not contain y or $\neg y$. To represent these cases, we use C_i, D_j, and E_k to denote clauses that do not contain y or $\neg y$. We specify a typical clause of S by $C_i \vee y$ if it contains y, by $D_j \vee \neg y$ if it contains $\neg y$, and by E_k if it does not contain y or $\neg y$. Thus, S can be displayed as follows.

$$\begin{aligned} C_i \vee y, &\quad i = 1, 2, \ldots, l_1 \\ D_j \vee \neg y, &\quad j = 1, 2, \ldots, l_2 \\ E_k, &\quad k = 1, 2, \ldots, l_3 \end{aligned} \tag{5.6.1}$$

Take any satisfying solution for (5.6.1). Thus, all clauses of (5.6.1) evaluate to *True*. If $y = \mathit{True}$ in the solution, then, for every clause $D_j \vee \neg y$, D_j must evaluate to *True*. Similarly, if $y = \mathit{False}$, then, for every clause $C_i \vee y$, C_i must evaluate to *True*. We conclude that, regardless of the value of y, $C_i \vee D_j$ evaluates to *True*. Accordingly, the solution for (5.6.1) satisfies S' defined by the following clauses.

$$\begin{aligned} C_i \vee D_j, &\quad i = 1, 2, \ldots, l_1, \ j = 1, 2, \ldots, l_2 \\ E_k, &\quad k = 1, 2, \ldots, l_3 \end{aligned} \tag{5.6.2}$$

Conversely, suppose we have a solution for S' of (5.6.2). Then all C_i or all D_j must evaluate to *True*, since otherwise some $C_i \vee D_j$ evaluates to

False. If all C_i evaluate to *True*, we assign to y the value *False*. If all D_j evaluate to *True*, we assign to y the value *True*. If both cases apply, we arbitrarily assign to y the value *True* or *False*. Regardless of the case, the solution for S' of (5.6.2) plus the selected value for y constitute a solution for (5.6.1). Thus, S' is a projection of S.

Note that S' may have a lot more clauses than S. For example, if S has 10 clauses $C_i \vee y$ and 20 clauses $D_j \vee \neg y$, then these 30 clauses of (5.6.1) become $10 \cdot 20 = 200$ clauses $C_i \vee D_j$ of S'.

Suppose now we apply the resolution algorithm recursively to project out not just one variable y but, instead, we first project out y_1, then y_2, and so on. Then we may incur an explosive growth of the number of clauses even if the initial S has just a few clauses. Of course, some of the clauses of the final S' may be redundant and thus could be eliminated. But, in the general case, we cannot hope for much of a reduction since theoretical results and practical experimentation have shown that an S even with few clauses often produces an S' with a large or even huge number of clauses. These same results show that any other formulation that has the variables of S' and has the same set of satisfying solutions as S', often must have very many clauses. Thus, projection typically increases the number of clauses substantially, whether it is carried out by the resolution process or some other method.

We have reached the end of the detour and return to the case considered earlier, where a correct S is produced by a decision pyramid, and where a correct S' is a direct formulation. The above discussion has shown that, if S contains a number of variables for intermediate conclusions, then S', which omits these variables, often has far more clauses than S. Indeed, S' may have so many clauses that the formulation cannot be stored in a computer with huge storage capacity. These arguments make a compelling case for the decision pyramid approach and against direct formulations.

We skip an example construction using the decision pyramid approach since such an example is included in the next section. There, explanations of decisions are discussed. Additional construction examples are given in Section 5.9.

5.7 Explanations

Computer-made decisions are accepted by humans only if the rationale behind the decisions can be exhibited. This does not mean that every decision must be supplied with a justification. But it does mean that, if a human inquires about the reasons for a decision, then the system must be able to justify or explain the decision. In this section, we see how this can be done. We begin with an example.

Example: Engine Diagnosis

Suppose the engine of a car does not perform properly. We want to decide whether we should replace the engine, repair the engine, or replace auxiliary equipment.

For the diagnosis, two kinds of symptoms should be considered: symptoms that the driver of the car typically detects, and symptoms that require a quick check by a mechanic. The next table lists symptoms of the first kind.

(5.7.1)

Variable	Meaning
black_exhaust	Exhaust fumes are black
blue_exhaust	Exhaust fumes are blue
low_power	Engine has low power
overheat	Engine overheats
ping	Engine emits a pinging sound under load

Symptoms detectable by driver

Symptoms of the second kind are as follows.

(5.7.2)

Variable	Meaning
incorrect_timing	Ignition timing is incorrect
clogged_filter	Air filter is clogged
low_compression	Compression of engine is low

Symptoms established by mechanic

Together, the symptoms of (5.7.1) and (5.7.2) constitute the variables of layer 0 of the pyramid.

At layer 1, we have the following intermediate conclusions.

(5.7.3)

Variable	Meaning
carbon_deposits	Cylinders have carbon deposits
clogged_radiator	Radiator is clogged
defective_carburetor	Carburetor is defective
worn_rings	Piston rings are worn
worn_seals	Valve seals are worn

Intermediate conclusions

The top layer, which is layer 2, consists of the following variables of final decisions or diagnoses.

(5.7.4)

Variable	Meaning
replace_auxiliary	Replace auxiliary equipment
repair_engine	Repair engine
replace_engine	Replace engine

Final conclusions

The following facts relate variables of level 1 to variables of level 0.

(5.7.5)

(a) If the engine overheats and the ignition is correct, then the radiator is clogged.
(b) If the engine emits a pinging sound under load and the ignition timing is correct, then the cylinders have carbon deposits.
(c) If power output is low and the ignition timing is correct, then the piston rings are worn, or the carburetor is defective, or the air filter is clogged.
(d) If the exhaust fumes are black, then the carburetor is defective, or the air filter is clogged.
(e) If the exhaust fumes are blue, then the piston rings are worn, or the valve seals are worn.
(f) The compression is low if and only if the piston rings are worn.

The clauses corresponding to these facts are as follows.

(5.7.6)

$$
\begin{aligned}
&[overheat \wedge \neg incorrect\_timing] \rightarrow clogged\_radiator \\
&[ping \wedge \neg incorrect\_timing] \rightarrow carbon\_deposits \\
&[low\_power \wedge \neg incorrect\_timing] \rightarrow [worn\_rings \vee \\
&\quad defective\_carburetor \vee clogged\_filter] \\
&black\_exhaust \rightarrow [defective\_carburetor \vee clogged\_filter] \\
&blue\_exhaust \rightarrow [worn\_rings \vee worn\_seals] \\
&low\_compression \leftrightarrow worn\_rings
\end{aligned}
$$

The intermediate conclusions of layer 1 and the final decisions of layer 2 are linked by the following facts.

(5.7.7)

(a) If the piston rings are worn then the engine should be replaced.
(b) If carbon deposits are present in the cylinders or the carburetor is defective or valve seals are worn, then the engine should be repaired.
(c) If the air filter or radiator is clogged, then that auxiliary equipment should be replaced.

Here are the corresponding clauses.

$$
\begin{aligned}
&\mathit{worn\_rings} \rightarrow \mathit{replace\_engine} \\
&[\mathit{carbon\_deposits} \vee \mathit{defective\_carburetor} \vee \mathit{worn\_seals}] \rightarrow \\
&\quad \mathit{repair\_engine} \\
&[\mathit{clogged\_filter} \vee \mathit{clogged\_radiator}] \rightarrow \mathit{replace\_auxiliary}
\end{aligned}
\tag{5.7.8}
$$

Let S be a CNF system that is equivalent to the CNF system defined by (5.7.6) and (5.7.8). We use that system next.

Suppose the car owner complains that the engine overheats. Due to a recent engine check, it is known that the ignition timing is correct. What should be done to eliminate the problem?

We represent the given information in S by assigning the value *True* to the variable *overheat* and the value *False* to the variable *incorrect_timing*. To determine which of the final conclusions applies, we take each variable of layer 2, in turn, assign *False* to the variable, and check satisfiability of the resulting CNF system. Any case of unsatisfiability establishes that the corresponding final conclusion is a theorem.

We discover that *replace_auxiliary* is a theorem, while *repair_engine* and *replace_engine* are not. Accordingly, we tell the customer that an auxiliary component must be replaced. The customer's response is "What must be replaced, and why?"

To answer the question, we carry out the following process. As before, we enforce the given symptom values for *overheat* and *incorrect_timing*, and assign *False* to *repair_auxiliary*. We already know that the CNF system S becomes unsatisfiable due to these values. We apply Algorithm SOLVE MINCLS UNSAT (3.3.14) to obtain a minimal subset of unsatisfiable clauses. The CNF clauses of that subset are as follows.

$$
\begin{aligned}
&\neg \mathit{overheat} \vee \mathit{incorrect\_timing} \vee \mathit{clogged\_radiator} \\
&\neg \mathit{clogged\_filter} \vee \mathit{replace\_auxiliary} \\
&\neg \mathit{clogged\_radiator} \vee \mathit{replace\_auxiliary}
\end{aligned}
\tag{5.7.9}
$$

These two clauses were produced by the following original clauses.

$$
\begin{aligned}
&[\mathit{overheat} \wedge \neg \mathit{incorrect\_timing}] \rightarrow \mathit{clogged\_radiator} \\
&[\mathit{clogged\_filter} \vee \mathit{clogged\_radiator}] \rightarrow \mathit{replace\_auxiliary}
\end{aligned}
\tag{5.7.10}
$$

The latter clauses and the known symptom values support the following explanation for the final decision that some auxiliary equipment must be replaced.

Known Facts (= English version of symptom variables with given values)

1. Engine overheats.

2. Ignition timing is correct.

Reasoning Statements (= English version of the clauses of (5.7.10))

3. If the engine overheats and the ignition is correct, then the radiator is clogged.
4. If the air filter or radiator is clogged, then that auxiliary equipment should be replaced.

We could weave the known facts and the reasoning statements into a more polished explanation, but that is just an optional refinement.

At times, the explanation obtained via the known facts and the minimal unsatisfiable subset of clauses is a bit difficult to understand. For improved insight, we may use the intermediate conclusions of the decision pyramid to derive further explanations. In S, we assign values to the symptoms according to the known facts as before, take each variable of layers 1, 2, ... in turn, set that variable to *False*, and for each of these cases check satisfiability. Whenever an unsatisfiable instance is encountered, we record the theorem implied by that result and re-express it in English as part of the explanation for the final decision.

For the situation at hand, there is just one case of unsatisfiability. It establishes *clogged_radiator* to be a theorem. We add that insight to the explanation found earlier.

Additional Conclusions (= English version of intermediate conclusions proved to be theorems)

5. Radiator is clogged.

Explanation Algorithm

In the general case, we have a decision pyramid of variables and clauses. We also have equivalent English expressions and statements for the clauses and for each variable with assigned *True/False* value. The clauses need not be in CNF.

Suppose that values are known for some of the variables of layer 0, and that a final decision of the top layer has been proved to be a theorem. Algorithm EXPLAIN DECISION (5.7.11) below derives an explanation why that final decision is a theorem.

(5.7.11) Algorithm EXPLAIN DECISION. *Supplies English statements that explain a decision.*

Input: A decision pyramid of variables and clauses. Equivalent English expressions and statements. *True/False* values for some of the variables of layer 0. A final decision variable of the top layer that has been proved to be a theorem.

Output: English statements that cover the following: the known facts; a minimal set of clauses for proving the final decision; and the intermediate conclusions that can be proved.

Requires: Algorithm SOLVE MINCLS UNSAT.

Procedure:

1. Derive an equivalent CNF system S from the given clauses.
2. Use Algorithm SOLVE MINCLS UNSAT (3.3.14) to find a minimal unsatisfiable subset of the clauses of the system S' derived from S by assigning the known *True/False* values to variables of layer 0 and by assigning *False* to the given final decision variable.
3. (Known Facts) Output English statements covering the variables of layer 0 with given values.
   (Reasoning Statements) Identify the input clauses that give rise to the clauses of S' determined in Step 2. Output the English versions of those input clauses.
4. (Additional Conclusions) For each intermediate conclusion of the layers strictly between layer 0 and the top layer, determine whether that conclusion is a theorem of S when the given input values are assigned to variables of layer 0. If this is the case, output the equivalent English statement for that fact.

The derivation of final conclusions and the subsequent explanation process may involve numerous theorem-proving steps and thus may require substantial computing effort. The next section shows how that effort can be significantly reduced.

5.8 Accelerated Theorem Proving

When we determine which of the final conclusions of a decision pyramid are theorems, or when we derive explanations via Algorithm EXPLAIN DECISION (5.7.11), we may have to solve numerous satisfiability instances. We can shortcut the effort by judicious use of a MINSAT version derived from the given SAT instance. The next example demonstrates how this is done.

Example: Engine Diagnosis Revisited

We return to the engine diagnosis problem of Section 5.7. It involves the following variables.

(5.8.1)

Variable	Layer and Type
black_exhaust	
blue_exhaust	
low_power	
overheat	Layer 0
ping	Symptoms
incorrect_timing	
clogged_filter	
low_compression	
carbon_deposits	
clogged_radiator	Layer 1
defective_carburetor	Intermediate conclusions
worn_rings	
worn_seals	
replace_auxiliary	Layer 2
repair_engine	Final conclusions
replace_engine	

Variables for engine diagnosis

The clauses connecting the variables of the three layers are as follows.

(5.8.2)

Clauses connecting layer 0 and layer 1

$$[overheat \wedge \neg incorrect\_timing] \rightarrow clogged\_radiator$$

$$[ping \wedge \neg incorrect\_timing] \rightarrow carbon\_deposits$$

$$[low\_power \wedge \neg incorrect\_timing] \rightarrow [worn\_rings \vee defective\_carburetor \vee clogged\_filter]$$

$$black\_exhaust \rightarrow [defective\_carburetor \vee clogged\_filter]$$

$$blue\_exhaust \rightarrow [worn\_rings \vee worn\_seals]$$

$$low\_compression \leftrightarrow worn\_rings$$

Clauses connecting layers 1 and 2

$$worn\_rings \rightarrow replace\_engine$$

$$[carbon\_deposits \vee defective\_carburetor \vee worn\_seals] \rightarrow repair\_engine$$

$$[clogged\_filter \vee clogged\_radiator] \rightarrow replace\_auxiliary$$

Let S be a CNF system that is equivalent to (5.8.2). The satisfiability instances arising from engine diagnosis are all processed in the following way. We assign given *True/False* values to some symptom variables, and assign *False* to one intermediate or final conclusion variable. If S' so derived from S is unsatisfiable, the conclusion has been proved to be a theorem.

Otherwise, it cannot be proved. Since (5.8.1) has eight intermediate or final conclusions, eight versions of S' must be tested for satisfiability. How can we reduce that computing effort?

The key idea is a switch from the SAT instance S to a MINSAT instance S_{min}, by assigning to each conclusion variable a cost of 1 for the value *True* and a cost of 0 for the value *False*. The remaining variables are declared to have 0 cost for both *True* and *False*.

To demonstrate the use of S_{min}, let *True/False* values be given for some of the symptom variables as in Section 5.7. Thus, we have *overheat* = *True* and *incorrect_timing* = *False*. We assign these values in the MINSAT instance S_{min} and get an instance S'_{min}. We solve S'_{min}, getting as output a satisfying solution that minimizes total cost.

We focus on the solution values assigned to the conclusions of layers 1 and 2. Since the cost of *True* for these variables is 1 while the cost of *False* is 0, the solution avoids *True* values as much as possible for these variables. Indeed, the solution values are as follows.

$$\begin{aligned} \textit{carbon\_deposits} &= \textit{False} \\ \textit{clogged\_radiator} &= \textit{True} \\ \textit{defective\_carburetor} &= \textit{False} \\ \textit{worn\_rings} &= \textit{False} \\ \textit{worn\_seals} &= \textit{False} \\ \textit{replace\_auxiliary} &= \textit{True} \\ \textit{repair\_engine} &= \textit{False} \\ \textit{replace\_engine} &= \textit{False} \end{aligned} \tag{5.8.3}$$

Evidently, any conclusion with value *False* cannot be proved to be a theorem, while *clogged_radiator* = *True* and *replace_auxiliary* = *True* indicate that either one of the conclusions *clogged_radiator* and *replace_auxiliary* may be a theorem. To find out whether this is the case, we go back to S and assign *overheat* = *True* and *incorrect_timing* = *False*, getting S'. We carry out two satisfiability tests, assigning *clogged_radiator* = *False* for the first test, and assigning *replace_auxiliary* = *False* for the second one. Both cases result in unsatisfiability, so we have proved both *clogged_radiator* and *replace_auxiliary* to be theorems. Effectively, we have obtained the desired results by solving one MINSAT instance and two SAT instances, instead of the eight SAT instances used before. Which of the two approaches should be preferred? The discussion of the general case of accelerated theorem proving in the next subsection includes the answer.

Accelerated Proofs

In the general case of accelerated theorem proving, we have a CNF system S and *True/False* values for some of the variables, say, for $s_1, s_2, \ldots, s_n$.

Among the remaining variables are $t_1, t_2, \ldots, t_k$. We want to know which of the latter variables are theorems when the given values are assigned to $s_1, s_2, \ldots, s_n$.

Instead of solving k SAT instances where, in turn, each t_j is assigned the value *False*, we proceed as follows. For each t_j, we declare the cost of *True* to be 1 and the cost of *False* to be 0. All other variables of S have 0 cost for both *True* and *False*. Let S_{min} be the resulting MINSAT instance. We assign to $s_1, s_2, \ldots, s_n$ the given values, getting a MINSAT instance S'_{min}.

We solve the MINSAT instance S'_{min}, getting as output either a solution that minimizes total cost or the conclusion that S'_{min} is unsatisfiable. If the latter case occurs, each t_j is trivially a theorem. That interpretation is mathematically correct, but likely inappropriate. Indeed, unsatisfiability of S'_{min} usually means that the *True/False* values assigned to $s_1, s_2, \ldots, s_n$ cannot occur in the real world setting that is modeled by S, or that the clauses of S do not represent the real world setting correctly.

Suppose we obtain for S'_{min} a solution with minimum total cost. Since the cost of *True* for the variables $t_1, t_2, \ldots, t_k$ is 1 while the cost of *False* is 0, the solution avoids the value *True* for $t_1, t_2, \ldots, t_k$ as much as possible. Moreover, for each t_j = *False*, the solution establishes that t_j is not a theorem. We check the remaining t_j for being theorems via satisfiability tests. We assign in S the given *True/False* values to $s_1, s_2, \ldots, s_n$, getting S'. Taking each t_j for which the MINSAT solution has the value *True*, we assign the value *False* to t_j and check satisfiability. If that case is unsatisfiable, t_j is a theorem. Otherwise, it is not.

The next algorithm summarizes the procedure.

(5.8.4) Algorithm ACCELERATE THEOREM PROVING. *Accelerates theorem proving using* MINSAT.

Input: CNF system S. For $n \geq 0$ and $k \geq 1$, $s_1, s_2, \ldots, s_n$ and $t_1, t_2, \ldots, t_k$ are among the variables of S. The case $n = 0$ means that there are no variables s_i. If $n \geq 1$, *True/False* values are given for $s_1, s_2, \ldots, s_n$.

Output: For each t_j, a declaration whether the variable is or is not a theorem, assuming that $s_1, s_2, \ldots, s_n$ take on the given *True/False* values.

Requires: Algorithms SOLVE SAT and SOLVE MINSAT.

Procedure:

1. (Define MINSAT instance) For each t_j of S, declare the cost of *True* to be 1 and the cost of *False* to be 0. All other variables of S have 0 cost for both *True* and *False*. Let S_{min} be the resulting MINSAT instance. Assign to $s_1, s_2, \ldots, s_n$ the given values, getting a MINSAT instance S'_{min}.
2. (Solve MINSAT instance) Solve S'_{min}. If S'_{min} is unsatisfiable, declare each t_j to be a theorem, and stop.

3. (Interpret solution values of MINSAT instance) For each t_j with solution value *False*, declare that t_j is not a theorem. For each t_j with solution value *True*, proceed as follows. Assign in S the given *True/False* values to $s_1, s_2, \ldots, s_n$ and assign the value *False* to t_j. Check satisfiability of the resulting CNF system. If it is unsatisfiable, declare t_j to be a theorem; otherwise, declare that it is not.

When is it advantageous to use Algorithm ACCELERATE THEOREM PROVING (5.8.4)? The answer depends on the number k of theorems to be proved, the average solution time α for the SAT instances, the solution time α_{min} for the single MINSAT instance, and the number l of theorems to be proved once the MINSAT solution is at hand. Typically, we do not have exact values for α, α_{min}, and l, but we may be able to obtain reasonable estimates. We use that information as follows. If the estimated solution time for the single MINSAT instance and for the l SAT instances is less than the estimated solution time for solving the k SAT instances, that is, if $\alpha_{min} + l \cdot \alpha < k \cdot \alpha$, then the MINSAT approach is a good choice. Rearranging the inequality, we get the rule that accelerated theorem proving should be used if

$$\alpha_{min}/\alpha < k - l \tag{5.8.5}$$

For example, if $k = 100$, $l = 3$, $\alpha = 0.01$ sec, and $\alpha_{min} = 0.2$ sec, then $\alpha_{min}/\alpha = 0.2/0.01 = 20 < k - l = 100 - 3 = 97$, and accelerated theorem proving is advantageous.

The next section examines special cases that naturally lead to decision pyramids.

5.9 Decision Graphs

Some applications directly suggest the variables of the decision pyramid. In this section, we see some examples.

Example: Decision Tree

We consider situations where a number of choices must be made sequentially. Simple cases are modeled by a rooted tree where the root node A represents a required initial selection and where any other node represents a choice. For example, if Y and Z are descendant nodes of node X, then Y and Z are the two choices available if X has already been selected. As a matter of convenience, we reuse the node labels as the names of logic variables. The logic formula representing the choices involving the nodes

X, Y, and Z is then $(Y \vee Z) \to X$. An example tree is given by the following table. The column labeled "Layer" refers to the layer number of the subsequently formulated decision pyramid.

(5.9.1)

Node	Descendants	Layer
A (root)	B, C	2
B	D, E	1
C	F, G	
D	H, I	0
E	J, K	
F	L, M	
G	N, Q	

Decision tree

We reuse the node labels as logic variables as done before, and get the following logic clauses for this tree.

$$
\begin{aligned}
&(N \vee Q) \to G \\
&(L \vee M) \to F \\
&(J \vee K) \to E \\
&(H \vee I) \to D \\
&(F \vee G) \to C \\
&(D \vee E) \to B \\
&(B \vee C) \to A
\end{aligned}
\tag{5.9.2}
$$

Each choice X may involve some cost $c(X)$ if X is selected and a cost $d(X)$ if X is not selected. In the logic formulation, $c(X)$ (resp. $d(X)$) becomes the cost of *True* (resp. *False*) for variable X. A solution of the MINSAT instance defined by (5.9.2) and the costs translates to choices with least total cost.

The choices may be constrained by additional logic conditions. We then add the corresponding logic clauses to the formulation. The next example treats such a situation.

Example: Course Prerequisites

Prerequisites of university courses often are depicted by a decision tree plus some additional logic conditions. Consider the following case.

(5.9.3)

Course	Prerequisites	Layer
CS101	none	2
CS102	CS101	1
CS103	CS101	
CS210	CS103	0
CS220	CS103	
CS230	CS102, CS103	

Course prerequisites

We also have the following information. CS103 is a corequisite of CS210, which means that CS210 may be taken even if CS103 has not yet been taken, provided CS103 is taken together with CS210.

We use predicates *has_taken*() and *take*() for the logic formulation. Both predicates are defined on the set of courses and have the obvious interpretation. The information of Table (5.9.3) and the corequisite condition are represented by the following clauses.

$$
\begin{aligned}
&[take(\text{CS102}) \vee take(\text{CS103})] \rightarrow has\_taken(\text{CS101}) \\
&take(\text{CS210}) \rightarrow [has\_taken(\text{CS103}) \vee take(\text{CS103})] \\
&take(\text{CS220}) \rightarrow has\_taken(\text{CS103}) \\
&take(\text{CS230}) \rightarrow [has\_taken(\text{CS102}) \wedge has\_taken(\text{CS103})]
\end{aligned}
\tag{5.9.4}
$$

If the university rules out repeated taking of any course, we also must include, for each course X, the clause $take(X) \rightarrow \neg\, has\_taken(X)$.

Suppose we want to know which courses a student may take during the next semester. For each predicate instance of *take*(), we define a cost of 0 for *True* and a cost of 1 for *False*. We fix the instances of *has_taken*() to *True/False* values reflecting which courses the student has taken so far and solve the resulting MINSAT instance. The solution minimizes the number of *False* values for the instances of *take*() and thus maximizes the number of *True* values of those instances. Accordingly, the solution produces the widest possible range of choices for the next semester.

At times, the prerequisite conditions are depicted not by a tree but by an acyclic graph where each node is a course and the predecessors of any node X are the prerequisites for course X. The logic formulation is handled the same way as for the tree case.

Some competitive situations lead to so-called AND/OR trees. We see an example next.

Example: AND/OR Tree

Consider a game where we compete against an opponent. Suppose that the game is in configuration V. We are to make the next move to some other

configuration. Say, a move to w or x is possible, and we choose to move to w. The opponent now has the option of moving to some other configuration, say to Y or Z. If a tree is used to depict these relationships, then the configurations V, w, x, Y, and Z are represented by nodes. Specifically, w and x are descendants of V, and Y and Z are descendants of w. For reasons to become clear shortly, the nodes V, Y, and Z, where we make a move, are called OR nodes, and the nodes w and x, where the opponent makes a decision, are called AND nodes. The root node of the tree represents the initial configuration of the game and may be an AND or OR node. For clarity, we denote AND (resp. OR) nodes by lower (resp. upper) case letters.

As we move from the root node down the tree across AND and OR nodes, we eventually reach the *tip nodes* of the tree. Each tip node is labeled "win," "lose," or "draw," with the following interpretation. The label "win" means that we win, "lose" means that we lose, and "draw" means that the game ends without a winner. A tip node is not considered to be an AND or OR node since neither we nor the opponent makes a decision at that node.

The next table contains the data of a small AND/OR tree where the root node is an OR node. Thus, we make the first move.

(5.9.5)

Node	Type	Descendants	Layer
A (root)	OR	b, c	3
b	AND	D, E	2
c	AND	F, G	
D	win		1
E	OR	i, j	
F	OR	k, l	
G	win		
i	win		0
j	win		
k	draw		
l	lose		

AND/OR tree

We assume that the opponent is clever and will not let us win unless that event cannot be prevented. Thus, our goal is to make decisions so that, if at all possible, the game ends up at a "win" node regardless of the decisions of the opponent. A *best strategy* aims at achieving that goal. Specifically, a *best strategy* for an OR node X either states, "Proceeding from X, we will lose regardless of the decisions made by us," or it provides a choice from node X so that, if we subsequently make appropriate choices,

then we will win regardless of the decisions made by the opponent.

How can we determine a best strategy for a given OR node X? Suppose we can find a rooted subtree $T(X)$ of the AND/OR tree such that the following holds.

(5.9.6)
- (5.9.6.1) Node X is the root node of $T(X)$.
- (5.9.6.2) Every OR node Y of $T(X)$ has at least one descendant node.
- (5.9.6.3) Every AND node y of $T(X)$ has all descendant nodes that node y has in the original tree.
- (5.9.6.4) All tip nodes of $T(X)$ are labeled "win."

Suppose we are at node X and that we have a subtree $T(X)$ with these features. By (5.9.6.1) and (5.9.6.2), X has at least one descendant. If there is just one descendant, this is our choice. If there are several descendants, we arbitrarily select one of them. Say, the chosen descendant is y. By (5.9.6.3), $T(X)$ contains all descendants of y in the original tree. Hence, no matter which decision is made by the opponent, the corresponding descendant of y, say Z, is in $T(X)$. We proceed from Z analogously to the case of X, thus making our next choice. Then the opponent makes a decision, and so on, until a tip node of $T(X)$ is reached. By (5.9.6.4), all tip nodes of $T(X)$ are labeled "win," so at that point we have won.

We have seen that $T(X)$ guarantees a win for us. Conversely, suppose there exists a choice at X for us such that, no matter how subsequent decisions are made by the opponent, we can always win by suitable choices on our part. Then there is a subtree $T(X)$ satisfying (5.9.6.1)–(5.9.6.4). Exercise (5.12.17) asks the reader to supply a proof for this claim.

At this point, it should be clear why nodes where we make a choice are called OR nodes, while the nodes where the opponent makes a decision are AND nodes. Indeed, the subtree $T(X)$ of a best strategy must contain at least one descendant of X; in logic, this is an OR condition. Furthermore, the subtree must contain all descendants of any node y where the opponent makes a decision; in logic, this is an AND condition.

The following clauses express conditions (5.9.6.1)–(5.9.6.4) for $T(A)$ of the AND/OR tree of (5.9.5). For convenience, we reuse the node labels as logic variables, where the value *True* (resp. *False*) means that the node is (resp. is not) in $T(A)$.

(5.9.7)
$$
\begin{aligned}
&A \\
&A \to (b \vee c) \\
&b \to (D \wedge E) \\
&c \to (F \wedge G) \\
&E \to (i \vee j) \\
&F \to (k \vee l)
\end{aligned}
$$

We rule out selection of the "draw" node k and the "lose" node l for $T(A)$ by setting the variables k and l to *False*. On the other hand, we leave it open which of the "win" nodes D, G, and i become part of $T(A)$ by not fixing their *True/False* values. With the values of k and l fixed as described, the clauses of (5.9.7) have the following satisfying solution.

$$\begin{aligned} A &= \mathit{True} \\ b &= \mathit{True} \\ c &= \mathit{False} \\ D &= \mathit{True} \\ E &= \mathit{True} \\ F &= \mathit{False} \\ G &= \mathit{False} \\ i &= \mathit{True} \\ j &= \mathit{False} \\ k &= \mathit{False} \\ l &= \mathit{False} \end{aligned} \tag{5.9.8}$$

Accordingly, $T(A)$ consists of the subtree defined by the nodes A, b, D, E, and i, and that subtree is a best strategy for node A.

Some AND/OR trees have a slightly different form or may be accompanied by additional specifications that restrict choices. For example, the logic variables of the nodes are to observe additional logic constraints. Or each node label may be a literal such as x or $\neg x$, and the same label may be assigned to several nodes of the tree. Thus, the node labels may be viewed as literals of logic variables. In such a case, the tip nodes may not have "win," "lose," and "draw" labels, but may also be labeled by literals. If that is the case, a best strategy subtree must have the literals of all tip nodes evaluate to *True*. We formulate the satisfiability problem SAT for CNF systems S as the problem of finding such a subtree. Let S have variables $x_1, x_2, \ldots, x_n$ and m clauses. The root node r of the AND/OR tree is an AND node. Its descendants are OR nodes that correspond to the clauses of S, say using node labels $C_1, C_2, \ldots, C_m$. For $i = 1, 2, \ldots, m$, the descendants of C_i correspond to the literals in clause i. For example, if clause i is $x_1 \vee \neg x_2 \vee x_3$, then node C_i has three descendants labeled x_1, $\neg x_2$, and x_3. Exercise (5.12.20) calls for verification that a best strategy subtree where each tip node evaluates to *True* exists if and only if S is satisfiable.

For other extensions, the reader should refer to the references cited in Section 5.11.

Some constraints or conditions are difficult to formulate in propositional logic. In the next section, we see some examples and related formulation approaches.

5.10 Difficult Cases

A number of situations or settings cannot be expressed in propositional logic. For example, uncertainty of statements defies formulation in propositional logic. Other situations in principle can be expressed in propositional logic, but result in large formulas that are difficult to process computationally. Examples are numerical constraints or relationships. Suppose we have logic variables $t_1, t_2, \ldots, t_m$ and $x_1, x_2, \ldots, x_n$. We have the constraint that the number of *True* values assigned to $t_1, t_2, \ldots, t_m$ times the number of *True* values assigned to $x_1, x_2, \ldots, x_n$ does not exceed some given integer value. This fact can be expressed in propositional logic, but not in a small formula unless m and n are small.

Even much simpler numerical constraints can be cumbersome to handle. In this section, we examine a particular constraint that frequently arises from practical applications and that despite its simplicity can lead to a cumbersome formula unless properly expressed. Given $x_1, x_2, \ldots, x_n$, the constraint demands that at most one of the x_i is assigned the value *True*.

A direct but inefficient way to formulate the constraint uses the idea that at most one of the x_i has the value *True* if and only if, for any two x_i and x_j, at most one of x_i and x_j has the value *True*. That observation produces the following $n(n-1)/2$ clauses.

$$\neg x_i \vee \neg x_j, \quad \forall\ 1 \leq i < j \leq n \tag{5.10.1}$$

Now $n(n-1)/2$ grows rapidly when n moves beyond, say, 10. Accordingly, (5.10.1) is not a smart way to handle the constraint when n is large. There are better ways. We cover them below, together with some extensions.

Approach 1: Additional Variables

Besides $x_1, x_2, \ldots, x_n$, we use new variables $y_1, y_2, \ldots, y_n$. We define clauses that assure that the *True/False* values for $y_1, y_2, \ldots, y_n$ have the following structure. Either all y_i have the value *False*, or all y_i have the value *True*, or for some index k, $1 \leq k < n$, the y_i with $i \leq k$ have the value *True*, and the y_i with $i > k$ have the value *False*. With each of the three cases, we associate a *breakpoint*. In the first case, the breakpoint is 0, in the second case it is n, and in the third case it is the specified k. By this definition, there is a one-to-one mapping linking the breakpoints and the possible ways in which values may be assigned to the y_i.

The condition constraining the y_i values is compactly represented by the following $n-1$ clauses.

$$y_i \rightarrow y_{i-1}, \quad \forall\ 2 \leq i \leq n \tag{5.10.2}$$

We want any values assigned to $x_1, x_2, \ldots, x_n$ and any values assigned to $y_1, y_2, \ldots, y_n$ and satisfying (5.10.2) to observe the following. Let k be the breakpoint associated with the values for the y_i. If $k = 0$, then all x_i must have the value *False*. Otherwise, x_k is to have the value *True*, and the remaining x_i are to have the value *False*. The following clauses enforce that condition.

$$\begin{aligned} &x_i \rightarrow (y_i \wedge \neg y_{i+1}), \quad \forall\ 1 \leq i \leq n-1 \\ &x_n \rightarrow y_n \end{aligned} \tag{5.10.3}$$

When (5.10.2) and (5.10.3) are converted to CNF, $3n - 2$ clauses result. That number compares favorably with $n(n-1)/2$ of (5.10.1) when n exceeds 6. For example, for $n = 20$, $n(n-1)/2 = 190 > 3n - 2 = 58$.

The approach is readily extended to handle the condition that at most $k \geq 2$ x_i can take on the value *True*. Instead of $y_1, y_2, \ldots, y_n$, we introduce, for $l = 1, 2, \ldots, k$, new variables $y_1^l, y_2^l, \ldots, y_n^l$ and define clauses analogously to (5.10.2) for these new variables. Then (5.10.3) is replaced by

$$\begin{aligned} &x_i \rightarrow \bigvee_{l=1}^{k} (y_i^l \wedge \neg y_{i+1}^l), \quad \forall\ 1 \leq i \leq n-1 \\ &x_n \rightarrow \bigvee_{l=1}^{k} y_n^l \end{aligned} \tag{5.10.4}$$

Exercise (5.12.21) asks that the clauses of (5.10.4) are converted to CNF.

The next approach relies on minimization.

Approach 2: Logic Minimization

Suppose the condition that at most one x_i = *True* is to be added to a satisfiability instance S. We define a new variable y and use it in the following new clause.

$$x_1 \vee x_2 \vee \cdots x_n \vee y \tag{5.10.5}$$

Suppose we assign to each x_i and to y a cost of 1 for *True* and a cost of 0 for *False*. Then (5.10.5), by itself, forces the total cost to be at least 1. Moreover, if the total cost is equal to 1, then either y = *True* and all x_i = *False*, or y = *False* and exactly one x_i = *True*. Thus, a total cost of 1 represents that at most one x_i has the value *True*.

We add (5.10.5) to the clauses of S, getting S', assign *True/False* costs to the x_i and y as defined above, and declare the cost for *True* or *False* for any other variable to be 0.

When we solve the MINSAT instance defined by S' and the costs, we must have one of the following mutually exclusive cases as outcome.

(a) S' is unsatisfiable: Since S' differs from S by the clause of (5.10.5), and since the latter clause can always be satisfied by $y = \mathit{True}$, S must be unsatisfiable.

(b) S' is satisfiable and total cost is equal to 1: Using the above arguments concerning (5.10.5), we have a satisfying solution for S where at most one $x_i = \mathit{True}$, as desired.

(c) S' is satisfiable and total cost is greater than 1: Evidently, at least two variables of (5.10.5) have the value *True*. We claim that y is not among them. Indeed, if $y = \mathit{True}$, we change it to $y = \mathit{False}$, and total cost drops by 1. Since at least two variables of (5.10.5) originally had the value *True*, at least one variable does so now, and the new solution still satisfies S and (5.10.5), and thus satisfies S'. But then the original solution did not have minimum total cost, which is a contradiction. We conclude that $y = \mathit{False}$ as claimed. The same contradiction occurs if S', and thus S, have a solution where at most one x_i has the value *True*. Thus, no such solution exists, and the original problem is unsatisfiable.

We have proved that the solution of the MINSAT instance settles the original problem. Note the compact way in which this is accomplished. We add only one variable and one clause to S. Of course, we must solve a MINSAT instance instead of a SAT instance, which may or may not be substantially more difficult than solving the original SAT instance.

Exercise (5.12.23) covers an extension of the logic minimization approach where several subsets $X_1, X_2, \ldots, X_m$ of the set of variables of S are given and where for each j, at most one of the variables of X_j may have the value *True*.

Looking Ahead

The next chapter describes how uncertainty of facts can be handled in formulations.

5.11 Further Reading

A good starting point for additional reading are texts on expert systems such as Leondes (1993), Giarratano and Riley (1994), Poole, Mackworth, and Goebel (1998), and Hopgood (2001).

The resolution process is covered most thoroughly in texts on propositional or first-order logic; see, for example, Wos, Overbeek, Lusk, and Boyle (1992) or Kleine Büning and Lettmann (1999).

5.12 Exercises

The calculations required by some of the exercises may be accomplished with the software of Exercises (2.9.13) and (3.8.14).

(5.12.1) Carry out the calculations for the jobs problem (5.2.1) that decide Mary's job.

(5.12.2) Answer the question "Is it possible that Robert is a nurse?" for the jobs problem (5.2.1).

(5.12.3)

(a) Consider the following modified version of the jobs problem (5.2.1). There are four persons: Anne, John, Mary, and Robert. Each person has one job each. There are four jobs: engineer, nurse, programmer, and teacher. The jobs of engineer and teacher are done by female persons. Formulate the problem and answer the following question by satisfiability testing and theorem proving. What job may Robert have? What job must Robert have?

(b) Suppose that we have 20 persons, say with the names $N_1, \ldots, N_{20}$, and 20 jobs, say $J_1, \ldots, J_{20}$. Find logic clauses which express that each job is taken by just one person. (*Hint*: See Section 5.10.)

(5.12.4) Verify that the two clauses of (5.3.2) are theorems of (5.2.7).

(5.12.5) Verify that (5.4.3) is a minimal unsatisfiable subsystem of (5.4.2).

(5.12.6) Confirm that (5.5.4) has no redundant clauses and is consistent.

(5.12.7) Carry out all calculations of the validation process for the example S of (5.5.4) using Algorithm VALIDATE CNF SYSTEM (5.5.10).

(5.12.8) Formulate the control problem of a traffic light using the decision pyramid approach. (*Hint*: See Chapter 11.)

(5.12.9) Take a logic problem you are thoroughly familiar with, and formulate it using the decision pyramid approach. Also try the direct formulation approach. If the latter approach can be done at all, compare the two formulations.

(5.12.10) Derive an equivalent CNF system S for the clauses of (5.7.6) and (5.7.8).

(5.12.11) Suppose for the engine diagnosis problem of Section 5.7, the customer declares that the car produces blue exhaust fumes, and that the mechanic has established low compression.

(a) Determine the applicable final conclusion.

(b) Use Algorithm EXPLAIN DECISION (5.7.11) to construct an explanation for the final conclusion.

(5.12.12) Use the results of Sections 5.4 and 5.6 to compare finding of inconsistencies and computing explanations.

(5.12.13) Modify Algorithm EXPLAIN DECISION (5.7.11) so that in Step 4 the MINSAT approach of Section 5.8 is used.

(5.12.14) Same as (5.12.11), except that the MINSAT approach of Section 5.8 is to be used to determine the applicable final conclusion and to find an explanation for that decision.

(5.12.15) Solve the decision tree problem given by (5.9.1) and the following information: (1) At least one of the variables of the node subset $\{H, I, J, K, L, M, N, Q\}$ must have the value *True*; (2) The cost of *True* (resp. *False*) of each variable is equal to 1 (resp. 0).

(5.12.16)

(a) Using the conditions of course prerequisites given by (5.9.4), find out which courses a student may take now if he/she has already taken the courses CS101 and CS102.

(b) Can it be determined that a student has taken a course where he/she did not satisfy all prerequisites?

(c) Construct a case of prerequisites where a decision tree cannot be used and where one must employ an acyclic graph.

(5.12.17) Prove: If an OR node X of an AND/OR tree as start node results in our winning, provided we make appropriate choices, and if this is the case regardless of the decisions by the opponent, then there is a subtree $T(X)$ satisfying (5.9.6.1)–(5.9.6.4). (*Hint*: Use induction on the number of nodes that in the AND/OR tree are descendants of X.)

(5.12.18) Confirm that the subtree $T(A)$ defined by (5.9.8) satisfies (5.9.6.1)–(5.9.6.4).

(5.12.19) Do the clauses of (5.9.7) have a satisfying solution different from that of (5.9.8)? If yes, define the subtree $T(A)$ of such a solution.

(5.12.20) Prove that a best strategy subtree exists for the AND/OR tree representing a CNF system S if and only if S is satisfiable.

(5.12.21) Convert the clauses of (5.10.4) to CNF.

(5.12.22) Modify the two approaches of Section 5.10 so that the condition of exactly one x_i having the value *True* is handled. (*Hint*: Modify (5.10.5).)

(5.12.23) Let a satisfiability instance S with set X of variables be given. For each one of given nonempty subsets $X_1, X_2, \ldots, X_m$ of X, the following condition is to be enforced: At most one variable of the subset may have the value *True*.

(a) Handle these conditions for $X_1, X_2, \ldots, X_m$ by logic minimization. (*Hint*: For each set X_i, define analogously to (5.10.5) the new clause

$(\bigvee_{x_j \in X_i} x_j) \vee y_i$. Assign to each y_j a cost of 1 for *True* and a cost of 0 for *False*. For each variable $x \in X$: If x occurs in k of the subsets $X_1, X_2, \ldots, X_m$, then assign to x the value k as cost of *True* and 0 as cost of *False*.)

(b) Adapt the solution of (a) to the situation where, for some of the subsets X_i, exactly one variable must have the value *True*.

(c) Adapt the results for (a) and (b) to the case where the original problem is a MINSAT instance. (*Hint*: Estimate the maximum value α that may occur as total cost of the original problem. Select an arbitrary value $\beta > \alpha$. In the case of (a), define β to be the cost of *True* for each y_i, and add $k \cdot \beta$ to the cost of *True* of variables x where k is the number of subsets X_i containing x. Handle (b) analogously.)

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 6

Uncertainty

6.1 Overview

The logic relationships arising from a real-world application may only hold with some degree of likelihood and not with certainty. This is particularly so for relationships produced by abduction, where a valid logic implication is reversed. For example, the valid implication *infection* $\rightarrow$ *elevated_temperature* may become by abduction *elevated_temperature* $\rightarrow$ *infection frequently.* The term *frequently* tells the likelihood with which the implication holds. In general, one may assign a *likelihood value* to any logic formula. The value indicates how likely the formula is correct. We also call that value the *level* at which the formula holds. We use integer likelihood values ranging from 1 to 100, where 1 corresponds to the case where correctness is highly unlikely, while 100 indicates that correctness is certain. Throughout the book, we translate terms such as *frequently* to likelihood values using the following table.

(6.1.1)

Term	Value
always	100
very frequently	85
frequently	70
half the time	50
sometimes	25
rarely	15

Likelihood terms and values

The above definition of likelihoods and their values may be intuitively appealing but is not sufficiently precise for our purposes. We remedy that shortcoming.

Definition of Likelihood Value

Suppose we model a real-world application using logic variables $x_1, x_2, \ldots, x_n$. Define a *possible assignment* to be *True/False* values for $x_1, x_2, \ldots, x_n$ that correspond to some situation occurring in the application. A logic clause using some or all of the x_j as variables is *valid for the application* if it evaluates to *True* for all possible assignments. Suppose a clause is not valid. Say, for some integer α, it evaluates to *True* for $\alpha\%$ of all possible assignments. Then the clause is *valid at level* α. Likelihood terms such as *frequently* are nothing but terms that correspond to particular α values.

According to these definitions, likelihood values used in logic clauses have meaning only if the underlying application has been specified. Such specification may be explicitly given or implicit by the context.

Handling of Likelihood Values

A number of ways have been proposed for handling likelihoods in connection with logic processes such as consistency checking, validation, and theorem proving. First, one may view the values to be probability percentages and apply standard probability theory. Second, one may interpret the values to represent beliefs and handle them according to some *belief theory*. Third, one may declare the values to reflect a fuzziness of the statements and evaluate them using the methods of *fuzzy logic*. Section 6.7 contains references for further reading about the three approaches. Here, we just summarize fuzzy logic.

Fuzzy Logic

Fuzzy logic is widely used and has led to impressive results for *production rule systems*, which are related to logic as follows. A *production rule* is a statement of the form, "If some condition is satisfied, then some conclusion holds." For evaluation of an initial set of facts, production rule systems employ *forward chaining* recursively as follows. Given the facts on hand, forward chaining searches for a rule where the facts at hand imply that the condition of the rule is satisfied. When such a rule is found, the conclusion of the rule is added to the set of facts, and recursion is invoked. Forward chaining stops when no additional rule can be found whose condition holds. The output consists of the facts derived by the entire process. Fuzzy logic specifies how likelihood values are handled as part of the chaining process.

The theory is too complicated to be treated here. But we mention two rules. Let a fact A (resp. B) hold with likelihood α (resp. β). The first rule says that A and B hold simultaneously with likelihood $\min\{\alpha, \beta\}$. The second rule says that A or B holds with likelihood $\max\{\alpha, \beta\}$.

Fuzzy logic would be the theory of choice here, were it not for the fact that it does not apply to the processing of general propositional logic formulas involving likelihoods. However, we can adapt the first rule cited above so that the computational tasks imposed by the logic problems of interest can be carried out. In this book, that adapted version is the basic rule for the handling of likelihoods.

Section 6.2 describes that basic rule. Sections 6.3–6.5 use the rule for the processing of various logic problems.

Section 6.3 treats satisfiability.

Section 6.4 handles logic minimization.

Section 6.5 concerns quantified SAT and MINSAT.

Section 6.6 covers defuzzification, where one determines from several imprecise decisions one deterministic conclusion.

Section 6.7 lists references for further reading.

Section 6.8 provides exercises.

6.2 Basic Rule

This section covers the basic rule for handling uncertainty in logic formulas. We begin with an example.

Example: Medical Diagnosis

Let x and y be logic variables that represent two diseases, which for notational convenience we also denote by x and y. Suppose that when a symptom r is present, then x or y is frequently present. We also use r as logic variable for the symptom. The relationship between the variables r, x and y is therefore as follows.

(6.2.1) $$r \to (x \vee y) \quad \textit{frequently}$$

We have a second symptom s represented by variable s. If symptom s is present, then half the time disease y is absent. Accordingly, we have the following clause.

(6.2.2) $$s \to \neg y \quad \textit{half the time}$$

According to Table (6.1.1), the likelihoods or levels of *frequently* and *half the time* are 70 and 50, respectively. We list (6.2.1) and (6.2.2) using these values.

$$\begin{aligned} & r \to (x \vee y) \quad \textit{at level } 70 \\ & s \to \neg y \quad \textit{at level } 50 \end{aligned} \tag{6.2.3}$$

Suppose both symptoms r and s are present. What can we conclude about the diseases x and y? Since $r = s = \mathit{True}$, we deduce from (6.2.3) that $x \vee y$ holds at level 70, and that $\neg y$ holds at level 50. We want to combine these two facts. An exact way using probabilities would require more information than given by (6.2.3). Indeed, we would need the joint probability distribution of r, s, x, and y. Many times that joint distribution is not at hand, and data for estimating the distribution are not available. Even if these obstacles can be overcome, there remains the problem that computational treatment of probability theory in the realm of propositional logic or of quantified SAT and MINSAT is a very difficult task. Given these facts, we take a much simpler view of the situation.

Definition of Rule

We use a basic rule that is adapted from the rule of fuzzy logic that two events A and B with likelihoods α and β, respectively, occur simultaneously with likelihood $\min\{\alpha, \beta\}$. The adapted rule says that, if clauses with levels α_i, $i = 1, 2, \ldots$, are used in logic computations, then the outcome is considered valid with likelihood $\min_i\{\alpha_i\}$.

For application of the basic rule to the case at hand, we convert (6.2.3) to CNF.

$$\begin{aligned} & \neg r \vee x \vee y \quad \textit{at level } 70 \\ & \neg s \vee \neg y \quad \textit{at level } 50 \end{aligned} \tag{6.2.4}$$

The two clauses of (6.2.4) have level values 70 and 50. If both clauses are used to derive a conclusion, then the minimum of 70 and 50, which is 50, is the likelihood associated with the outcome.

Let $r = s = \mathit{True}$. We use both clauses of (6.2.4) to obtain conclusions. Since $s = \mathit{True}$, the clause $\neg s \vee \neg y$ is reduced to $\neg y$, and hence $y = \mathit{False}$. Since $r = \mathit{True}$ and $y = \mathit{False}$, the first clause $\neg r \vee x \vee y$ is reduced to x, and $x = \mathit{True}$. The result is interpreted as follows. If $r = s = \mathit{True}$, then with likelihood $\min\{50, 70\} = 50$ we have $x = \mathit{True}$ and $y = \mathit{False}$.

Justification of Rule

The reader familiar with the existing approaches to uncertainty in logic may object to the utter simplicity if not naïveness of the above approach.

The reader may even construct an example using probabilities where the probabilistic conclusion is very different from the result derived by the above rule. Similar objections were initially made to fuzzy logic, but later the approach proved to be eminently useful. As for the rule proposed here, we only claim that the computations using the rule are effective, that they are easily combined with other concepts and methods, and that they have produced reasonable and robust results in the application projects undertaken by us. For these reasons we employ the rule.

The subsequent sections provide details about the use of the rule in various settings. We begin with decisions based on satisfiability.

6.3 Satisfiability

Given is a CNF system S. Each of the clauses of S holds with a likelihood that may range from 1 to 100. Let n be the number of distinct likelihood values. We sort these values in decreasing order and denote them by $\alpha_1 > \alpha_2 > \cdots > \alpha_n$. We emphasize that any given α_i may be assigned to any number of clauses. We always add an additional value $\alpha_{n+1} = 0$ and sometimes add $\alpha_0 = 100$ to simplify the presentation.

Confidence Level of Solution

For $1 < i < n+1$, *True/False* values for the variables that satisfy all clauses with level greater than α_i and that do not satisfy at least one clause at level α_i, turn out to be of interest. We define such *True/False* values to constitute a *satisfying solution of S with confidence level* $\beta_i = 100 - \alpha_i$. We expand that definition to the cases $i = 1$ and $i = n + 1$ as follows. For $i = 1$, we declare any *True/False* values that do not satisfy at least one clause at level α_1, to be a *satisfying solution with confidence level* $\beta_1 = 100 - \alpha_1$. For $i = n+1$, any *True/False* values that satisfy all clauses of S are a *satisfying solution with confidence level* $\beta_{n+1} = 100 - \alpha_{n+1} = 100 - 0 = 100$. According to these definitions, the confidence level is, for some $1 \leq i \leq n + 1$, of the following form.

$$\beta_i = 100 - \alpha_i \tag{6.3.1}$$

Example: Medical Diagnosis, Revisited

For discussion of an example, consider the following S, which links symptoms r and s with diseases x and y using the following clauses.

$$\begin{aligned} &\neg r \vee x \vee y \quad \textit{at level } 70 \\ &r \vee \neg x \vee y \quad \textit{at level } 65 \\ &\neg s \vee \neg y \quad \textit{at level } 50 \end{aligned} \tag{6.3.2}$$

Evidently, $\alpha_1 = 70$, $\alpha_2 = 65$, and $\alpha_3 = 50$. Consider the assignment $r = y = \mathit{False}$ and $s = x = \mathit{True}$. It says that symptom r and disease y are absent, and that symptom s and disease x are present. The *True/False* values satisfy the clause $\neg r \vee x \vee y$, which is at level $\alpha_1 = 70$. But they do not satisfy the clause $r \vee \neg x \vee y$, which is at level $\alpha = 65$. Thus, the assignment is a satisfying solution with confidence level $\beta_2 = 100 - \alpha_2 = 35$. Note that the assignment satisfies the clause $\neg s \vee \neg y$, which has level $\alpha_3 = 50$. But that fact does not influence the confidence level.

Level of Unsatisfiability

Besides the confidence level of an assignment, we also have a likelihood of unsatisfiability. Specifically, we define the *highest likelihood value for unsatisfiability* of S, also called the *highest level for unsatisfiability* of S, as follows. If S is unsatisfiable, that value is the largest α_k for which the clauses of S with level greater than or equal to α_k are unsatisfiable. If S is satisfiable, that value is $\alpha_{n+1} = 0$. We associate an index with the highest likelihood value of unsatisfiability. If S is unsatisfiable, the index is k of the value α_k. If S is satisfiable, $k = n + 1$.

Unsatisfiability of the clauses of S with level $\alpha_1 = 100$ may or may not indicate a formulation error of S. Specifically, if S represents relationships of some real-world application, then unsatisfiability is caused by a formulation error. On the other hand, if S arises from the theorem-proving situation discussed later in this section, then the clauses with level 100 may indeed be unsatisfiable and turn out to prove a certain statement to be a theorem at level 100.

Satisfiability Problem UNCERTAIN SAT

Depending on the application, we may be interested in the highest level α_k of unsatisfiability or in a solution with confidence level $\beta_k = 100 - \alpha_k$. We define the problem of finding that level and solution next.

(6.3.3) UNCERTAIN SAT

Instance: CNF system S. Each clause of S holds with some likelihood. In sorted order, the levels are $\alpha_1 > \alpha_2 > \cdots > \alpha_n$. Let $\alpha_{n+1} = 0$.

Solution: The highest level of unsatisfiability of S, say α_k with $1 \leq k \leq n + 1$, and a satisfying solution of S with confidence level $\beta_k = 100 - \alpha_k$.

Solution Algorithm

The algorithm for solving problem UNCERTAIN SAT described below uses, for $i = 0, 1, \ldots, n+1$, certain CNF systems S_i that have the same variables

as S. The system S_0 has no clauses. For $i = 1, 2, \ldots, n$, S_i has all clauses of S with level greater than or equal to α_i. Finally, S_{n+1} has just one clause, which is empty. Evidently, any *True/False* values constitute a satisfying solution of S_0, while S_{n+1} is unsatisfiable.

Exercise (6.8.1) asks for verification of the following statements concerning the index k associated with the highest level of unsatisfiability of S. First, that index is equal to the smallest index i for which S_i is unsatisfiable. Second, any satisfying solution of S at confidence level $\beta_k = 100 - \alpha_k$ is a satisfying solution of S_{k-1}, and conversely. The proof of these statements validates the following algorithm.

(6.3.4) Algorithm SOLVE UNCERTAIN SAT. *Solves instance of* UNCERTAIN SAT.

Input: CNF system S, where each clause holds with some likelihood. The levels are $\alpha_1 > \alpha_2 > \cdots > \alpha_n$. Let $\alpha_{n+1} = 0$.

Output: The highest level of unsatisfiability of S, say α_k with $1 \leq k \leq n+1$, and a satisfying solution of S with confidence level $\beta_k = 100 - \alpha_k$.

Requires: Algorithm SOLVE SAT.

Procedure:

1. For $i = 1, 2, \ldots, n$, define S_i to be the CNF subsystem of S that has the same variables as S and whose clauses are the clauses of S with level greater than or equal to α_i. Define S_0 and S_{n+1} to be CNF systems that also have the same variables as S. The CNF system S_0 has no clauses, while S_{n+1} has just one clause, which is empty.
2. Do for $i = 0, 1, \ldots, n+1$: Use Algorithm SOLVE SAT to decide if S_i is satisfiable, and if this is the case, to obtain a satisfying solution. (If $i = 0$ or $i = n+1$, the test is trivial, since any *True/False* values satisfy S_0 and since S_{n+1} is unsatisfiable.) If S_i is found to be unsatisfiable, set $k = i$, and go to Step 3.
3. Output α_k as the highest level at which S is unsatisfiable. Also, output the satisfying solution found for S_{k-1} as the desired satisfying solution of S with confidence level $\beta_k = 100 - \alpha_k$. Stop.

Speed-up by Binary Search

If the clauses of S have $n > 2$ distinct levels, we may speed up the computation of Step 2 by a binary search where we first apply Algorithm SOLVE SAT to the case S_i with $i = \lfloor (n+1)/2 \rfloor$. If S_i is unsatisfiable, then we know that $S_{i+1}, S_{i+2}, \ldots, S_{n+1}$ are unsatisfiable as well, and we evaluate the remaining $S_0, S_1, \ldots, S_{i-1}$ recursively. On the other hand, if S_i is satisfiable, we know that $S_0, S_1, \ldots, S_{i-1}$ are satisfiable as well, and we evaluate the remaining $S_{i+1}, S_{i+2}, \ldots, S_{n+1}$ recursively. The binary search

reduces the number of applications of Algorithm SOLVE SAT from $O(n)$ to $O(\log n)$.

Theorem Proving

In the typical theorem-proving application, we have a CNF system S of axioms and a CNF clause T. Suppose each clause of S is given with a likelihood value. Define $S \wedge \neg T$ to be the following CNF system. The variables of $S \wedge \neg T$ are the variables occurring in S or T. The clauses of $S \wedge \neg T$ are the clauses of S with the given levels plus the clauses of $\neg T$, which are defined to have level 100. Let the likelihood values of $S \wedge \neg T$, in sorted order, be $\alpha_1 > \alpha_2 > \cdots > \alpha_n$.

We translate the concepts of the preceding section to the case at hand by letting $S \wedge \neg T$ play the role of S of that section. Recall from Chapter 2 that, in the absence of likelihood values, T is a theorem of S if and only if $S \wedge \neg T$ is unsatisfiable, and that any satisfying solution of $S \wedge \neg T$ is a counterexample that proves T not to be a theorem of S. Accordingly, for the case S with likelihood values, we define the *highest likelihood value* or *level at which T is a theorem of S* to be the highest level of unsatisfiability of $S \wedge \neg T$. Let α_k be that value. We declare the associated satisfying solution of $S \wedge \neg T$ with confidence level $\beta_k = 100 - \alpha_k$ to be a *counterexample of theorem T with confidence level β_k*. We demonstrate these definitions with an example.

Example: Medical Diagnosis, Once More

Let S have symptom variables r and s and disease variables x and y. The clauses are as follows.

$$\begin{aligned} &\neg r \vee x \vee y \quad \textit{at level } 70 \\ &r \vee \neg x \vee y \quad \textit{at level } 65 \\ &\neg s \vee \neg y \quad \textit{at level } 50 \end{aligned} \tag{6.3.5}$$

Suppose we know that symptom r is absent and that symptom s is present. We want to prove absence of x at the highest possible level. Accordingly, we define T to be the statement $(\neg r \wedge s) \rightarrow \neg x = r \vee \neg s \vee \neg x$. Thus, $\neg T = \neg r \wedge s \wedge x$.

In $S \wedge \neg T$, the levels of the clauses produced by S are 70, 65, and 50, and the levels of the three clauses of $\neg T$, which are $\neg r$, s, and x, are all equal to 100. Thus, $S \wedge \neg T$ has $n = 4$ likelihood values. In sorted order, they are $\alpha_1 = 100$, $\alpha_2 = 70$, $\alpha_3 = 65$, $\alpha_4 = 50$.

We carry out Algorithm SOLVE UNCERTAIN SAT (6.3.4), with $S \wedge \neg T$ playing the role of S in the algorithm. Exercise (6.8.2) calls for confirmation of the following results. The highest level of unsatisfiability of

$S \wedge \neg T$ is $\alpha_4 = 50$, and a satisfying solution of $S \wedge \neg T$ with confidence level $\beta_4 = 100 - \alpha_4 = 50$ is given by $r = \mathit{False}$ and $s = x = y = \mathit{True}$. Thus, the highest level at which T can be proved to be a theorem of S is $\alpha_4 = 50$, and the values $r = \mathit{False}$ and $s = x = y = \mathit{True}$ constitute a counterexample to the claim that T is a theorem, with confidence level $\beta_4 = 100 - \alpha_4 = 50$. We interpret these conclusions as follows. We have established absence of x at level 50. On the other hand, $r = \mathit{False}$ and $s = x = y = \mathit{True}$ describe a scenario where r is absent and where s, x, and y are present, and that scenario has confidence level 50.

Accelerated Theorem Proving

We have purposely chosen a representation of $S \wedge \neg T$ that displays how the clauses of $\neg T$ interact with those of S. Computationally, it is more efficient to use just the clauses of S in $S \wedge \neg T$ and to enforce the clauses of $\neg T$ by fixing of variables. The modification essentially does not affect the conclusions of Algorithm SOLVE UNCERTAIN SAT (6.3.4), but it reduces the size of the satisfiability problems. More importantly, the representation of $\neg T$ via fixing of variables allows application of Algorithm ACCELERATE THEOREM PROVING (5.8.4) when several theorems that are related as described in the input of that algorithm, are to be proved or disproved. Exercise (6.8.3) asks the reader to work out details for the use of Algorithm ACCELERATE THEOREM PROVING (5.8.4) in that setting.

Conflicting Clauses

The above theorem-proving approach does not produce reasonable results if S is unsatisfiable. In that case, any statement T can be proved to be a theorem at some level. Of course, the same difficulty can occur in the absence of likelihoods and is treated in Chapter 5 as part of validation. We mention it here since valid clauses with likelihoods may produce the same effect. For example, a clause may conclude x at some level, and a second clause may conclude $\neg x$ at some other level. If the two clauses cannot be satisfied simultaneously, then S is unsatisfiable. In general, this problem has a simple solution only if x is one of the final conclusions of the decision pyramid underlying the formulation. Let us assume we have the latter case. Since x is a final conclusion, we typically want to prove statements such as $T = x$ or $T = \neg x$ to be a theorem. To obtain such proofs, we replace in S and T the variable x by two variables x^+ and x^-, as follows. Wherever x (resp. $\neg x$) occurs in S, we replace it by x^+ (resp. x^-). Let S' and T' be the resulting CNF system and clause. Instead of proving T to be a theorem of S, we now prove T' to be a theorem of S'.

Example

An example demonstrates the process. We have a symptom r and a conclusion x. Let S be the CNF system defined by these variables and the following clauses.

$$\begin{aligned} &\neg r \vee x \quad \textit{at level } 70 \\ &\neg r \vee \neg x \quad \textit{at level } 25 \\ &r \quad \textit{at level } 100 \end{aligned} \tag{6.3.6}$$

Evidently, the three clauses of S cannot be satisfied. Replacing x by x^+ and $\neg x$ by x^-, we obtain the following system S', which is satisfiable.

$$\begin{aligned} &\neg r \vee x^+ \quad \textit{at level } 70 \\ &\neg r \vee x^- \quad \textit{at level } 25 \\ &r \quad \textit{at level } 100 \end{aligned} \tag{6.3.7}$$

From $T = x$ or $T = \neg x$, we get $T' = x^+$ or $T' = x^-$. Exercise (6.8.4) asks the reader to carry out theorem proving for S' and the two T' and to determine counterexamples.

If the variable x producing the unsatisfiability of S does not represent a final conclusion, the above remedy may not be appropriate. An alternative is the clause deletion approach mentioned next for yet another difficulty.

Facts Disproving Theorems

Suppose we compute T to be a theorem of S with some likelihood, but then obtain facts that establish T not to be a theorem with certainty. Evidently, S needs to be adjusted. We accomplish this by deleting certain clauses. We omit a detailed discussion here since that situation and related cases are covered in Chapter 9 as part of nonmonotonic reasoning.

The next section covers minimum cost satisfiability.

6.4 Minimum Cost Satisfiability

We have a CNF system S with likelihoods and also have cost values for *True* and *False* for each variable. Suppose that, roughly speaking, we want a minimum cost assignment of *True/False* values that satisfies all important clauses of S and that is allowed to violate unimportant ones. We make that vague statement precise using the concept of confidence level of satisfying solutions. We define a lower bound β_{min} on the confidence level, and demand a satisfying solution with confidence level $\beta_i \geq \beta_{min}$ such that total cost is minimum. In case several confidence levels produce the same minimum total cost, the selected β_i should be as large as possible.

Solution Approach

We can supply the desired solution for any β_{min} value if, for each β_i, either we have a satisfying solution with confidence level β_i and with minimum total cost among all such solutions, or we have confirmation that there is a satisfying solution with confidence level β_k, $\beta_k > \beta_i$, with total cost equal to the minimum total cost producible by the solutions with confidence level β_i. Indeed, given that information, we pick the largest $\beta_k > \beta_{min}$ for which we have a satisfying solution with confidence level β_k and with total cost matching the minimum total cost for the case of the smallest $\beta_i > \beta_{min}$. We declare that solution, with confidence level β_k, to be the answer.

There is one difficulty, though. Suppose the clauses of S are not satisfiable. Say, the clauses with levels $\alpha_1, \alpha_2, \ldots, \alpha_{i-1}$ are satisfiable, but when the clauses of α_i are added, unsatisfiability results. Thus, there cannot exist any satisfying solution with confidence level β_k, $k > i$. For that case, we declare that the minimum total cost for any $\beta_{min} \geq \beta_k$ is ∞.

We demonstrate the calculations and introduce an additional concept.

Example

A CNF system S is given with variables v, w, x, and y and with the following clauses.

$$\begin{aligned} & x \quad \textit{at level } 100 \\ & y \quad \textit{at level } 100 \\ & v \vee w \vee \neg x \quad \textit{at level } 70 \\ & w \vee \neg x \quad \textit{at level } 50 \\ & v \vee \neg y \quad \textit{at level } 25 \end{aligned} \tag{6.4.1}$$

For v, the cost of *True* is 4, and for w, that cost is 1. All other costs for *True* and *False* are 0.

We have $n = 4$. The levels are $\alpha_1 = 100$, $\alpha_2 = 70$, $\alpha_3 = 50$, $\alpha_4 = 25$, and $\alpha_5 = 0$. Using $\beta_i = 100 - \alpha_i$ of (6.3.1), we have the following values.

$$\begin{aligned} \beta_1 &= 100 - 100 = 0 \\ \beta_2 &= 100 - 70 = 30 \\ \beta_3 &= 100 - 50 = 50 \\ \beta_4 &= 100 - 25 = 75 \\ \beta_5 &= 100 - 0 = 100 \end{aligned} \tag{6.4.2}$$

We use the definition of S_i of Algorithm SOLVE UNCERTAIN SAT (6.3.4). That is, for $i = 1, 2, \ldots, n$, S_i is the subsystem of S whose clauses have

level greater than or equal to α_i. The subsystem S_0 has no clauses. We do not make use of S_{n+1}. For $i = 0, 1, \ldots, n$, let z_i denote the minimum total cost for the MINSAT instance defined by S_i.

It turns out to be advantageous that we process the β_i cases in reverse order. Thus, we start with $\beta_5 = 100$. Recall that a satisfying solution with confidence level β_i satisfies S_{i-1} but not S_i. Thus, for the case of β_5, we must solve the MINSAT instance involving S_4, which is equal to S. The unique optimal solution is as follows.

$$\begin{aligned} v &= \mathit{True} \\ w &= \mathit{True} \\ x &= \mathit{True} \\ y &= \mathit{True} \end{aligned} \tag{6.4.3}$$

Total cost is $z_4 = 5$.

Next is $\beta_4 = 75$, for which the subsystem S_3 has the following clauses.

$$\begin{aligned} &x \quad \mathit{at\ level}\ 100 \\ &y \quad \mathit{at\ level}\ 100 \\ &v \vee w \vee x \quad \mathit{at\ level}\ 70 \\ &w \vee \neg x \quad \mathit{at\ level}\ 50 \end{aligned} \tag{6.4.4}$$

The unique optimal solution is as follows.

$$\begin{aligned} v &= \mathit{False} \\ w &= \mathit{True} \\ x &= \mathit{True} \\ y &= \mathit{True} \end{aligned} \tag{6.4.5}$$

Total cost is $z_3 = 1$.

We proceed to $\beta_3 = 50$ and S_2 with the following clauses.

$$\begin{aligned} &x \quad \mathit{at\ level}\ 100 \\ &y \quad \mathit{at\ level}\ 100 \\ &v \vee w \vee \neg x \quad \mathit{at\ level}\ 70 \end{aligned} \tag{6.4.6}$$

The unique optimal solution matches that for β_4 and thus is not a satisfying solution with confidence level β_3. That conclusion is interesting, but more important is the fact that $z_2 = 1$ and thus $z_2 = z_3$. Hence, even if we had an optimal solution for S_2 that was a satisfying solution with confidence level β_3, we would never use it since we have a satisfying solution with the higher confidence level β_4 and same total cost. We also say that the case

$\beta_3 = 50$ is *dominated* by the case $\beta_4 = 75$. This fact is recorded later using an asterisk, by writing $\beta_3 = 50^*$.

We turn to $\beta_2 = 30$, with the following system S_1.

$$\begin{array}{l} x \quad \textit{at level } 100 \\ y \quad \textit{at level } 100 \end{array} \tag{6.4.7}$$

The unique optimal solution is as follows.

$$\begin{array}{l} v = \textit{False} \\ w = \textit{False} \\ x = \textit{True} \\ y = \textit{True} \end{array} \tag{6.4.8}$$

Total cost is $z_1 = 0$, which is less than z_2. We claim that this implies that we have a satisfying solution with confidence level β_2. Indeed, if this is not the case, then the solution of (6.4.8) does not violate any clauses at level α_3, and the optimal solution found earlier for $\beta_3 = 50$ must have cost $z_2 \leq z_1$. But we have $z_1 = 0 < z_2 = 1$, a contradiction.

For the final case β_0, the subsystem S_0 has no clauses. There are several optimal solutions. One of them is as follows.

$$\begin{array}{l} v = \textit{False} \\ w = \textit{False} \\ x = \textit{False} \\ y = \textit{False} \end{array} \tag{6.4.9}$$

Total cost is $z_0 = 0$, which matches $z_1 = 0$. The solution is satisfying with confidence level β_1 since it violates at least one clause at level α_1, but the fact that z_0 matches z_1 tells that the solution is dominated. As done before, we express this fact by an asterisk and write $\beta_1 = 0^*$.

We summarize the values for the β_i and the corresponding minimum total cost z_{i-1} for S_{i-1}.

(6.4.10)

Level	Min Cost
$\beta_1 = 0^*$	$z_0 = 0$
$\beta_2 = 30$	$z_1 = 0$
$\beta_3 = 50^*$	$z_2 = 1$
$\beta_4 = 75$	$z_3 = 1$
$\beta_5 = 100$	$z_4 = 5$

* indicates dominated case

Confidence levels and minimum total costs

Suppose β_{min} is given as 25. From (6.4.10), we choose the smallest undominated $\beta_i \geq \beta_{min}$ as $\beta_2 = 30$, with minimum total cost $z_1 = 0$. The associated solution given by (6.4.8) is the desired answer for $\beta_{min} = 25$. Now suppose $\beta_{min} = 45$. The smallest $\beta_i \geq \beta_{min}$ is $\beta_3 = 50$. But that case is dominated by $\beta_4 = 75$, so we choose the solution of (6.4.5) for β_4, with $z_3 = 1$.

In the example, satisfying solutions exist for all confidence levels. But if there is no such solution for some confidence level β_k, then we assign to z_k the value ∞. In fact, in such a case, for all β_i with $i > k$, the related S_{i-1} is unsatisfiable as well, and thus $z_{i-1} = \infty$.

Expected Min Cost

Imagine the following scenario. We have S with likelihoods and costs for *True/False* values. We randomly choose an integer value α, $1 \leq \alpha \leq 100$, select all clauses of S with level greater than or equal to α, solve the MINSAT instance defined by the selected clauses, and get a minimum total cost. Suppose we repeat the process indefinitely. What is the average of the minimum cost values? In this subsection, we answer that question.

Define $\alpha_0 = 100$. Then for $i = 0, 1, \ldots, n$, the selection rule implies that any α satisfying

$$\alpha_i \geq \alpha > \alpha_{i+1} \tag{6.4.11}$$

causes selection of the clauses of S_i. Note that S_0 is selected if and only if $\alpha > \alpha_1$, which is possible if and only if $\alpha_1 < 100$. The formula of (6.4.11) with $i = 0$ correctly accounts for this case.

Due to the random selection of α, the event of α satisfying (6.4.11) and thus of the clauses of S_i being selected occurs with the following probability p_i.

$$p_i = (\alpha_i - \alpha_{i+1})/100 \tag{6.4.12}$$

Let z_i be the total cost of an optimal solution for S_i. The average of the minimum total costs that are produced by the entire process is the following $\overline{z}$, which we call the *expected min cost*.

$$\begin{aligned} \overline{z} &= \sum_{i=0}^{n} z_i \cdot p_i \\ &= \sum_{i=0}^{n} z_i(\alpha_i - \alpha_{i+1})/100 \end{aligned} \tag{6.4.13}$$

If any S_i is unsatisfiable, then $z_i = \infty$, and we declare $\overline{z} = \infty$.

We want to rewrite (6.4.13) using β_i of (6.3.1), which is defined for $i = 1, 2, \ldots, n+1$. For this, we extend the formula of (6.3.1) to the case $i = 0$ by defining $\beta_0 = 100 - \alpha_0 = 0$. Since $\alpha_i - \alpha_{i+1} = (100 - \alpha_{i+1}) - (100 - \alpha_i) = \beta_{i+1} - \beta_i$, we obtain from (6.4.13) the following formula.

$$\overline{z} = \sum_{i=0}^{n} z_i(\beta_{i+1} - \beta_i)/100 \tag{6.4.14}$$

Example

We list again the results of (6.4.10) for S of (6.4.1) and add the just defined case of $\beta_0 = 0$, which has no corresponding minimum cost.

(6.4.15)

Level	Min Cost
$\beta_0 = 0$	undefined
$\beta_1 = 0^*$	$z_0 = 0$
$\beta_2 = 30$	$z_1 = 0$
$\beta_3 = 50^*$	$z_2 = 1$
$\beta_4 = 75$	$z_3 = 1$
$\beta_5 = 100$	$z_4 = 5$

* indicates dominated case

Confidence levels and minimum total costs

For the computation of the expected min cost, it does not matter that the cases $\beta_1 = 0$ and $\beta_3 = 50$ are dominated. Thus, the expected min cost is $\overline{z} = \sum_{i=0}^{n} z_i(\beta_{i+1} - \beta_i)/100 = [0(0-0)] + 0(30-0) + 1(50-30) + 1(75-50) + 5(100-75)]/100 = 1.65$.

Minimization Problem UNCERTAIN MINSAT

We combine the problem of finding minimum total cost solutions for β_1, $\beta_2, \ldots, \beta_{n+1}$ and of determining the expected min cost. The following problem results.

(6.4.16) UNCERTAIN MINSAT

Instance: CNF system S. For each variable of S, two rational cost values associated with the values *True* and *False* for that variable. Each clause of S holds with some likelihood. In sorted order, the levels are $\alpha_1 > \alpha_2 > \cdots > \alpha_n$. Let $\alpha_0 = 100$ and $\alpha_{n+1} = 0$.

Solution: For $i = 0, 1, \ldots, n$ and for the MINSAT instance defined by the clauses of S with level greater than or equal to α_i: The minimum total cost z_i and, if $z_i < \infty$, an optimal satisfying solution. (If $z_i < z_{i+1}$, the solution is undominated and has confidence level β_{i+1}.) The expected min cost $\overline{z}$.

Solution Algorithm

The earlier discussion validates the following algorithm for UNCERTAIN MINSAT.

(6.4.17) Algorithm SOLVE UNCERTAIN MINSAT. *Solves instance of* UNCERTAIN MINSAT.

Input: CNF system S. For each variable of S, two rational cost values associated with the values *True* and *False* for that variable. Each clause of S holds with some likelihood. In sorted order, the levels are $\alpha_1 > \alpha_2 > \cdots > \alpha_n$. Let $\alpha_0 = 100$ and $\alpha_{n+1} = 0$.

Output: For $i = 0, 1, \ldots, n$ and for the MINSAT instance defined by the clauses of S with level greater than or equal to α_i: The minimum total cost z_i and, if $z_i < \infty$, an optimal satisfying solution. (If $z_i < z + i + 1$, the solution is undominated, and the confidence level is β_{i+1}.) The expected min cost $\overline{z}$.

Requires: Algorithms SOLVE UNCERTAIN SAT (6.3.4) and SOLVE MINSAT.

Procedure:

1. For $i = 1, 2, \ldots, n$, define S_i to be the CNF subsystem of S that has the same variables as S and whose clauses are the clauses of S with level greater than or equal to α_i. Define S_0 and S_{n+1} to be CNF systems that also have the same variables as S. The CNF system S_0 has no clauses, while S_{n+1} has just one clause, which is empty.
2. (Determine the largest index k for which S_k is satisfiable.) Ignore the costs for *True/False* values and solve the UNCERTAIN SAT instance given by S, using Algorithm SOLVE UNCERTAIN SAT (6.3.4). Let k be the index for which the algorithm determines all S_i with $i \leq k$ (resp. $i > k$) to be satisfiable (resp. unsatisfiable). For all $k < i \leq n$, define $z_i = \infty$.
3. Do for $i = k, k-1, \ldots, 0$: Use Algorithm SOLVE MINSAT to solve the MINSAT instance defined by S_i and the given *True/False* costs. Let z_i be the minimum total cost. If $z_i < z_{i+1}$, the solution is undominated and has confidence level β_{i+1}.
4. Output the z_i values, the solutions of undominated cases, and the expected min cost $\overline{z} = \sum_{i=0}^{n} z_i(\beta_{i+1} - \beta_i)/100$. Stop.

6.5 Quantified SAT and MINSAT

One may extend the concepts of highest unsatisfiability at level α_k and of a satisfying solution with confidence level β_k to the quantified SAT and

MINSAT problems of Chapter 4. Due to space limitations, we cannot provide a detailed treatment of the extensions. Instead, we have included Exercises (6.8.7)–(6.8.13). Solving these problems fosters an understanding of the extensions.

6.6 Defuzzification

We have seen how a CNF system S involving likelihood values can be used to prove theorems at some level or to derive counterexamples with some confidence level. Sometimes, we are not interested in such conclusions and instead desire one deterministic decision. An example follows.

Example: Control of a Valve

A tank containing water is drained by a valve. How far the valve is opened depends on the pressure in the tank and the temperature of the water. The pressure can be low, normal, and high. The temperature is characterized using the same qualitative terms.

We have a rather vague description of the operation of the valve. For example, if the pressure is low, the valve is 1/4 open 90% of the time and is closed 10% of the time. That information and other cases are summarized in the following table.

(6.6.1)

Pressure	Temperature	Valve	Likelihood
low		1/4 open	90
low		closed	10
	low	1/4 open	80
	low	1/2 open	20
normal		1/2 open	70
normal		3/4 open	30
	normal	1/4 open	50
	normal	1/2 open	50
high		3/4 open	95
high		full open	5
	high	full open	100

Rules of valve operation

The information of the table is imprecise and inconsistent. For a demonstration, suppose that we have normal pressure and normal temperature. According to the table, normal pressure calls for the setting 1/2

open or 3/4 open, with likelihood values 70 and 30, while normal temperature requires the setting 1/4 open or 1/2 open, each with likelihood value 50. Real-world applications often involve such incomplete and inconsistent data. Defuzzification is one way of deriving reasonable deterministic operating decisions from such data.

We define two predicates *pressure*() and *temperature*() on the set $\{low, normal, high\}$ and introduce decision variables d_j, $j = 0, 1, \ldots, m$. If $d_j = True$, then the valve is $j/4$ open. Thus, $j = 0$ (resp. $j = 4$) corresponds to the case where the valve is closed (resp. full open). With these predicates and variables, Table (6.6.1) can be converted to a set of clauses. For example, the first line of the table, which declares that for low pressure the valve is 1/4 open 90% of the time, becomes the clause $pressure(low) \rightarrow d_1$ *at level* 90, or in CNF, $\neg\, pressure(low) \vee d_1$ *at level* 90. Below is the entire collection of clauses in CNF form, in decreasing order of likelihood values.

$$
\begin{aligned}
&\neg\, temperature(high) \vee d_4 \quad \textit{at level } 100\\
&\neg\, pressure(high) \vee d_4 \quad \textit{at level } 95\\
&\neg\, pressure(low) \vee d_1 \quad \textit{at level } 90\\
&\neg\, temperature(low) \vee d_1 \quad \textit{at level } 80\\
&\neg\, pressure(normal) \vee d_2 \quad \textit{at level } 70\\
&\neg\, temperature(normal) \vee d_1 \quad \textit{at level } 50\\
&\neg\, temperature(normal) \vee d_2 \quad \textit{at level } 50\\
&\neg\, pressure(normal) \vee d_3 \quad \textit{at level } 30\\
&\neg\, temperature(low) \vee d_0 \quad \textit{at level } 20\\
&\neg\, pressure(low) \vee d_0 \quad \textit{at level } 10\\
&\neg\, pressure(high) \vee d_3 \quad \textit{at level } 5
\end{aligned}
\tag{6.6.2}
$$

We have $n = 10$ different levels α_i. They are $\alpha_1 = 100$, $\alpha_2 = 95$, $\alpha_3 = 90$, $\alpha_4 = 80$, $\alpha_5 = 70$, $\alpha_6 = 50$, $\alpha_7 = 30$, $\alpha_8 = 20$, $\alpha_9 = 10$, and $\alpha_{10} = 5$.

Suppose we are told that the pressure is low and the temperature is normal. How far should the valve be opened? We record the situation by assigning *True* to *pressure*(*low*) and *temperature*(*normal*), and by assigning *False* to the remaining variables of the predicates *pressure*() and *temperature*(). Let S be the resulting CNF system. Before we go into the subsequent steps, we define the form of S for the general case.

General Case

We have a CNF system S with n levels $\alpha_1 > \alpha_2 > \cdots > \alpha_n$. As before, we define $\alpha_{n+1} = 0$. Among the logic variables of S are $m + 1$ conclusion variables $d_0, d_1, \ldots, d_m$. In addition, there are $m + 1$ rational vectors e^0,

$e^1, \ldots, e^m$ of equal length. Any linear combination $e = \sum_{j=0}^{m} \lambda_j \cdot e^j$, where each rational $\lambda_j \geq 0$ and where $\sum_{j=0}^{m} \lambda_j = 1$, corresponds to a possible decision. Due to the conditions imposed on the λ_j factors, the linear combination e is also called a *convex combination* of the e^j. Defuzzification is nothing but computation of e via theorem proving involving S and the d_j.

For the valve problem, we have $m = 4$. For $j = 0, 1, \ldots, 4$, e^j is the rational number $j/4$ and thus ranges from 0 to 1. Hence, any convex combination e of the e^j is a rational number ranging from 0 to 1. The case of $e = 0$ (resp. $e = 1$) corresponds to the valve being closed (resp. full open), and any value $0 < e < 1$ represents a partial opening of the valve.

Here are the details of the defuzzification process. For $j = 0, 1, \ldots, m$, we determine the highest level at which d_j can be proved to be a theorem. Let that level be $\alpha_{k(j)}$. If more than one $\alpha_{k(j)}$ is equal to 100, or if all $\alpha_{k(j)}$ are equal to 0, then S contains a formulation error, and we cannot proceed. If exactly one $\alpha_{k(j)} = 100$, the corresponding conclusion d_j is a theorem with certainty, and we select $e = e^j$ as decision. Otherwise, we compute e as a convex combination of the e^j using $\lambda_j = \alpha_{k(j)} / \sum_{l=0}^{m} \alpha_{k(l)}$. That is,

$$
\begin{aligned}
e &= \sum_{j=0}^{m} \lambda_j \cdot e^j \\
&= \sum_{j=0}^{m} \alpha_{k(j)} \cdot e^j / \sum_{l=0}^{m} \alpha_{k(l)}
\end{aligned}
\tag{6.6.3}
$$

We demonstrate the calculations using the valve example. The CNF system S is given by (6.6.2), by *pressure*(*low*) = *temperature*(*normal*) = *True*, and by *False* for all other variables of the predicates *pressure*() and *temperature*(). Exercise (6.8.14) asks for confirmation of the following theorem-proving results.

(6.6.4)

Conclusion	Proved at Level
d_0	$\alpha_{k(0)} = \alpha_9 = 10$
d_1	$\alpha_{k(1)} = \alpha_3 = 90$
d_2	$\alpha_{k(2)} = \alpha_6 = 50$
d_3	$\alpha_{k(3)} = \alpha_{11} = 0$
d_4	$\alpha_{k(4)} = \alpha_{11} = 0$

Proof levels for conclusions

No $\alpha_{k(j)}$ is equal to 100, and at least one $\alpha_{k(j)}$ is positive. Thus, there is no obvious formulation error. Using the equation (6.6.3), we have $e = \sum_{j=0}^{m} \alpha_{k(j)} \cdot e^j / \sum_{l=0}^{m} \alpha_{k(l)} = [10 \cdot 0 + 90 \cdot (1/4) + 50 \cdot (2/4)]/150 = 0.32$ as the fraction by which the valve should be opened.

Defuzzification Algorithm

The following algorithm summarizes the process.

(6.6.5) Algorithm DEFUZZIFY. *Computes decision by defuzzification.*

Input: CNF system S whose variables include, for some m, conclusion variables $d_0, d_1, \ldots, d_m$. Rational vectors $e^0, e^1, \ldots, e^m$ of equal length. Each clause of S holds at some level. The levels are $\alpha_1 > \alpha_2 > \cdots > \alpha_n$. Let $\alpha_{n+1} = 0$.

Output: Either: A rational decision vector e that is a convex combination of $e^0, e^1, \ldots, e^m$. Or: "S contains a formulation error."

Requires: Algorithm SOLVE UNCERTAIN SAT (6.3.4).

Procedure:

1. Do for $j = 0, 1, \ldots, m$: Using Algorithm SOLVE UNCERTAIN SAT (6.3.4), establish the highest level $\alpha_{k(j)}$ at which the conclusion d_j can be proved to be a theorem of S.
2. If more than one $\alpha_{k(j)}$ is equal to 100, or if all $\alpha_{k(j)}$ are equal to 0, then declare that S contains a formulation error, and stop.
3. Output the decision $e = (1/\sum_{l=0}^{m} \alpha_{k(l)}) \sum_{j=0}^{m} \alpha_{k(j)} \cdot e^j$, and stop.

Accelerated Defuzzification

When the number of conclusions d_j is large, the computational effort for Step 1 of Algorithm DEFUZZIFY (6.6.5) can often be drastically reduced by accelerated theorem proving. Exercise (6.8.15) calls for the development of the process.

6.7 Further Reading

For an overview of techniques for reasoning under uncertainty, see introductory artificial intelligence texts such as Nilsson (1998), Luger (2002), Negnevitsky (2002), and Russell and Norvig (2003). Nguyen and Walker (1997) is an introductory text for fuzzy logic.

The defuzzification process described in Section 6.6 is sufficient for the purposes of this book. Other approaches may be used to derive deterministic decisions from imprecise or uncertain information; see Kosko (1992) or Hopgood (2001).

6.8 Exercises

The calculations required by some of the exercises may be accomplished with the software of Exercise (6.8.16).

(6.8.1) Let $S_0, S_1, \ldots, S_{n+1}$ be the subsystems of S defined in Step 1 of Algorithm SOLVE UNCERTAIN SAT (6.3.4). Prove the following results for the index k associated with the highest level of unsatisfiability of S.
(a) The index k is equal to the smallest index i for which S_i is unsatisfiable.
(b) Any satisfying solution of S with confidence level $\beta_k = 100 - \alpha_k$ is a satisfying solution of S_{k-1}, and conversely,

(6.8.2) Confirm the following results for the medical diagnosis problem of Section 6.3. The highest level of unsatisfiability of $S \wedge \neg T$ is $\alpha_4 = 50$, and a satisfying solution of S with confidence level $\beta_4 = 100 - \alpha_4 = 50$ is given by $r = \mathit{False}$ and $s = x = y = \mathit{True}$.

(6.8.3) Provide an algorithm based on Algorithms ACCELERATE THEOREM PROVING (5.8.4) and SOLVE UNCERTAIN SAT (6.3.4) that accelerates theorem proving when many related theorems must be established.

(6.8.4) Define S' by (6.3.7), and let $T' = x^+$ or $T' = x^-$. For each of the two cases of T', answer (a) and (b) below.
(a) At what levels can T' be proved to be a theorem of S'?
(b) What is the confidence level of a counterexample for the statement that T' is a theorem of S'?

(6.8.5) Define S by the following clauses.

$$p \vee q \vee r \quad \mathit{at\ level}\ 100$$
$$\neg p \vee \neg q \vee \neg r \quad \mathit{at\ level}\ 85$$
$$p \vee \neg q \quad \mathit{at\ level}\ 70$$
$$\neg p \vee r \quad \mathit{at\ level}\ 60$$

Define T to be $r \rightarrow \neg q$.
(a) At what level is T a theorem of S?
(b) At what confidence level does a counterexample exist for the claim that T is a theorem of S?

(6.8.6) Let S be the CNF system of Exercise (6.8.5). For each variable of S, define the cost of *True* to be 1 and the cost of *False* to be 0. Solve this instance of UNCERTAIN MINSAT and compute the expected min cost.

Solution of Exercises (6.8.7)–(6.8.13) fosters an understanding of uncertainty in quantified SAT and MINSAT problems.

(6.8.7) Expand the notion of S-acceptable and S-unacceptable solutions to the case where the clauses of S have likelihood values assigned.

(6.8.8) Define a version of problem Q-ALL SAT (4.2.8) where the clauses CNF system S have likelihood values assigned. Call that version UNCERTAIN Q-ALL SAT.

(6.8.9) Define a solution algorithm for problem UNCERTAIN Q-ALL SAT of Exercise (6.8.8) for the case of few quantified variables $q_1, \ldots, q_l$. Use Algorithm SOLVE Q-ALL SAT (4.2.9) as a guide.

(6.8.10) Define a version of problem Q-MIN UNSAT (4.3.6) where the clauses of the CNF system S have likelihood values assigned. Call that version UNCERTAIN Q-MIN UNSAT.

(6.8.11) Define a solution algorithm for problem UNCERTAIN Q-MIN UNSAT of Exercise (6.8.10) for the case of few quantified variables $q_1, \ldots, q_l$. Use Algorithm SOLVE Q-MIN UNSAT (4.3.7) as a guide.

(6.8.12) Define a version of problem Q-MINFIX UNSAT (4.3.8) where the clauses of the CNF system S have likelihoods values assigned. Call that version UNCERTAIN Q-MINFIX UNSAT. Show that problem UNCERTAIN Q-MINFIX UNSAT is a special case of problem UNCERTAIN Q-MIN UNSAT of Exercise (6.8.10).

(6.8.13) Analogously to Exercises (6.8.8), (6.8.10), and (6.8.12), define uncertainty versions for problems Q-MAX MINSAT (4.4.6), P-EXIST Q-ALL SAT (4.5.2), P-MIN Q-ALL SAT (4.5.3), Q-MINFIX SAT (4.5.4), and P-MIN Q-MAX MINSAT (4.5.5).

(6.8.14) Confirm the levels $\alpha_{k(j)}$ of (6.6.4) for the valve problem, where S is defined by (6.6.2) and by the following *True/False* assignments. Both *pressure*(*low*) and *temperature*(*normal*) have the value *True*, and all other variables of the predicates *pressure*() and *temperature*() have the value *False*.

(6.8.15) Use the solution of Exercise (6.8.3) to devise an accelerated defuzzification process that replaces Step 1 of Algorithm DEFUZZIFY (6.6.5) by combined use of Algorithm SOLVE UNCERTAIN MINSAT (6.4.17) and Algorithm SOLVE UNCERTAIN SAT (6.3.4).

(6.8.16) (Optional programming project) There are two options if the reader wishes to avoid manual solution of some of the exercises. First, the reader may obtain commercially or publicly available software for at least some of the problems. Any search engine on the Internet will point to such software. Second, the reader may opt to write programs that implement the algorithms of (a)–(c) below, and then use that software to solve the exercises. The task is easy if the reader has already created programs for SAT and MINSAT as described in Exercise (2.9.13).

(a) Algorithm SOLVE UNCERTAIN SAT (6.3.4).
(b) Algorithm SOLVE UNCERTAIN MINSAT (6.4.17).
(c) Algorithm DEFUZZIFY (6.6.5).

Part III

Learning

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 7

Learning Formulas

7.1 Overview

Up to this point, all logic formulas have been manually derived. There are situations where that approach is economically infeasible or impossible. Consider a complex case of medical diagnosis, say of a rare disease. Suppose a procedure for early detection is not known, yet we want to build a diagnostic system for such detection. Can this be done? This chapter answers the question in the affirmative, provided we have patient data that are relevant for the disease. The main idea is to extract logic formulas from the patient data and to use these formulas in a diagnostic system to make decisions.

The extraction process is not just useful for decision problems in medicine, but applies generally when sufficient data exist that are relevant for the problem at hand. We call the extraction of logic formulas *learning logic.*

Learning logic is one of many methods of *data mining* and *machine learning.* These areas are concerned with the extraction of mathematical relationships from data and their use. Examples are statistical regression, clustering techniques, neural nets, and support vector machines. Section 7.9 contains references covering these and other methods. In this book, we favor learning logic, since we can combine learned logic formulas seamlessly with manually generated formulas and can process such combined formulas with the extensive computational machinery of logic.

Section 7.2 introduces the basic ideas for learning logic formulas that differentiate between two data sets. We call such differentiation a separation of the two sets.

Section 7.3 describes a general method for learning logic formulas that separate two data sets.

Sections 7.4 and 7.5 specialize the results of Section 7.3 and develop logic formulas that have a minimum or maximum number of literals or are optimized relative to certain costs.

Section 7.6 deals with the situation where additional logic conditions are imposed on learned formulas.

Section 7.7 describes variations of the learned formulas of Sections 7.3–7.6, obtained by a certain reversal of the roles of the given two data sets.

Section 7.8 introduces the concept of voting by several learned formulas. This material is used in Chapter 8, where vote totals of learned formulas are analyzed.

Section 7.9 suggests references for further reading.

Section 7.10 contains exercises.

7.2 Basic Concepts

We develop basic concepts of learning logic. We begin with an example.

Example: Medical Diagnosis

Suppose four symptoms e, f, g, and h are useful for diagnosis of a disease d. For patient 1, who has the disease, symptoms e, f, and g are present, while h is absent. We encode the information by declaring e, f, g, and h to be logic variables and by assigning *True* (resp. *False*) if the symptom is present (resp. absent). Accordingly, $e = f = g = \mathit{True}$ and $h = \mathit{False}$. Patient 2 also has the disease, and $e = \mathit{False}$ and $f = g = h = \mathit{True}$. Patients 3 and 4 do not have the disease. Patient 3 has $e = f = h = \mathit{True}$ and $g = \mathit{False}$, and patient 4 has $e = g = h = \mathit{True}$ and $f = \mathit{False}$.

Record

For each patient, we define the collection of symptom values to be a *record*. For example, the record of patient 1 is $(e = \mathit{True}, f = \mathit{True}, g = \mathit{True}, h = \mathit{False})$. We collect the records of patients 1 and 2, who have the disease, in a set A, and those of patients 3 and 4, who do not have the disease, in a set B. We consider the records of A to be randomly drawn from the *population*

$\mathcal{A}$ of records of patients with the disease. Analogously, the records of B come from the *population* $\mathcal{B}$ of records of patients without the disease. We shall derive from A and B a logic formula for estimating membership of records in $\mathcal{A}$ and $\mathcal{B}$. When combined with additional machinery, that formula may be used for diagnosis of the disease. The effectiveness of such diagnosis would depend on the relevance of the data used to estimate the formulas, that is, whether sets A and B are representative samples of $\mathcal{A}$ and $\mathcal{B}$.

Training Sets

The sets A and B are declared to be *training sets* containing *training records*. We summarize the two sets.

(7.2.1)

Set A	
$(e = True, f = True, g = True, h = False)$	(patient 1)
$(e = False, f = True, g = True, h = True)$	(patient 2)
Set B	
$(e = True, f = True, g = False, h = True)$	(patient 3)
$(e = True, f = False, g = True, h = True)$	(patient 4)

Training sets A and B

Format of Formula

We want a logic formula D involving the variables e, f, g, and h so that D has one *True/False* value for all records of A and has the opposite value for all records of B. Say, we desire the value *True* (resp. *False*) for the records of A (resp. B). We decide on the following format of D.

$$\begin{aligned} D = {} & (\text{at most one literal of } e)\wedge \\ & (\text{at most one literal of } f)\wedge \\ & (\text{at most one literal of } g)\wedge \\ & (\text{at most one literal of } h) \end{aligned} \tag{7.2.2}$$

Thus, D is a DNF clause. Since a literal of any variable x is equal to either x or $\neg x$, we can express the format of (7.2.2) also as

$$\begin{aligned} D = {} & (e \text{ or } \neg e \text{ or no literal of } e)\wedge \\ & (f \text{ or } \neg f \text{ or no literal of } f)\wedge \\ & (g \text{ or } \neg g \text{ or no literal of } g)\wedge \\ & (h \text{ or } \neg h \text{ or no literal of } h) \end{aligned} \tag{7.2.3}$$

Here, the case "no literal of e" means that the part of the formula containing that statement is not present. The corresponding cases for f, g, and h are interpreted analogously. Example formulas are $D = e \wedge \neg f \wedge g \wedge \neg h$, $D = \neg f \wedge g$, and $D = h$.

Logic Conditions

We develop conditions that the formula D must obey. For $x = e$, f, g, and h, we introduce logic variables $x(pos)$ and $x(neg)$. The *True/False* values of $x(pos)$ and $x(neg)$ decide if the formula contains one of the literals x or $\neg x$ or contains none of them. The rules are as follows.

$$\begin{aligned} &x(pos) = \mathit{True} \text{ and } x(neg) = \mathit{False}\text{: select literal } x \\ &x(pos) = \mathit{False} \text{ and } x(neg) = \mathit{True}\text{: select literal } \neg x \\ &x(pos) = x(neg) = \mathit{False}\text{: do not select literal } x \text{ or } \neg x \\ &x(pos) = x(neg) = \mathit{True}\text{: this case is not allowed} \end{aligned} \tag{7.2.4}$$

To prevent the fourth case $x(pos) = x(neg) = \mathit{True}$, we impose the condition

$$\neg x(pos) \vee \neg x(neg) \tag{7.2.5}$$

Since we have $x = e$, f, g, or h, we have altogether the variables $e(pos)$, $e(neg)$, $f(pos)$, $f(neg)$, $g(pos)$, $g(neg)$, $h(pos)$, and $h(neg)$. We develop a CNF system S for these variables so that any satisfying solution yields upon interpretation via (7.2.4) a logic formula D that evaluates to *True* for the records of A and to *False* for the records of B.

Consider the first record of B of (7.2.1), which says that $e = \mathit{True}$, $f = \mathit{True}$, $g = \mathit{False}$, $h = \mathit{True}$. Any formula D of the form (7.2.3) takes on the value *False* for that record only if at least one of the literals $\neg e$, $\neg f$, g, $\neg h$ occurs in D. By (7.2.4), the literal $\neg e$ occurs if and only if $e(neg) = \mathit{True}$, $\neg f$ occurs if and only if $f(neg) = \mathit{True}$, g occurs if and only if $g(pos) = \mathit{True}$, and $\neg h$ occurs if and only if $h(neg) = \mathit{True}$. At least one of these values is assured if we impose the condition

$$e(neg) \vee f(neg) \vee g(pos) \vee h(neg) \tag{7.2.6}$$

The second record of B of (7.2.1), which specifies $e = \mathit{True}$, $f = \mathit{False}$, $g = \mathit{True}$, and $h = \mathit{True}$, produces the constraint

$$e(neg) \vee f(pos) \vee g(neg) \vee h(neg) \tag{7.2.7}$$

The first record of A of (7.2.1), which declares $e = \mathit{True}$, $f = \mathit{True}$, $g = \mathit{True}$, $h = \mathit{False}$, is to result in the value *True* for D. Due to the format

(7.2.3), this is possible only if D has at least one literal and is of the form $D = (e \text{ or no literal of } e) \wedge (f \text{ or no literal of } f) \wedge (g \text{ or no literal of } g) \wedge (\neg h \text{ or no literal of } h)$. By (7.2.4), these conditions are equivalent to

$$\neg e(neg) \wedge \neg f(neg) \wedge \neg g(neg) \wedge \neg h(pos) \tag{7.2.8}$$

The second record of A, which says $e = False$, $f = True$, $g = True$, $h = True$, produces the condition

$$\neg e(pos) \wedge \neg f(neg) \wedge \neg g(neg) \wedge \neg h(neg) \tag{7.2.9}$$

Finally, using $x = e$, f, g, and h in (7.2.5), we have

$$\begin{aligned} &\neg e(pos) \vee \neg e(neg) \\ &\neg f(pos) \vee \neg f(neg) \\ &\neg g(pos) \vee \neg g(neg) \\ &\neg h(pos) \vee \neg h(neg) \end{aligned} \tag{7.2.10}$$

The conditions (7.2.6)–(7.2.10) are necessary for D. That is, they must be simultaneously satisfied if D is to have the desired properties. We summarize them for ready reference.

$$\begin{aligned} &e(neg) \vee f(neg) \vee g(pos) \vee h(neg) \\ &e(neg) \vee f(pos) \vee g(neg) \vee h(neg) \\ &\neg e(neg) \wedge \neg f(neg) \wedge \neg g(neg) \wedge \neg h(pos) \\ &\neg e(pos) \wedge \neg f(neg) \wedge \neg g(neg) \wedge \neg h(neg) \\ &\neg e(pos) \vee \neg e(neg) \\ &\neg f(pos) \vee \neg f(neg) \\ &\neg g(pos) \vee \neg g(neg) \\ &\neg h(pos) \vee \neg h(neg) \end{aligned} \tag{7.2.11}$$

Formula for Diagnosis Problem

It is easily checked that the CNF system (7.2.11) has a unique solution that assigns $True$ to $f(pos)$ and $g(pos)$ and $False$ to the remaining variables. According to (7.2.4), that solution implies $D = f \wedge g$. Evaluating D for each record of (7.2.1), we confirm that D evaluates to $True$ (resp. $False$) for all records of A (resp. B).

The discussion has skipped over the problem of the existence of D. We cover that aspect later when we treat the general case of a DNF formula D that evaluates to $True$ for the records of A and to $False$ for the records of B. We say that such D *separates* the records of A from the records of B, or, shorter, that D *separates* A from B. Before we establish how such D is computed, we introduce the general format of records.

Format of Records

A *record* is a collection of values for given variables $x_1, x_2, \ldots, x_n$. The possible values for any x_j are *True*, *False*, *Unavailable*, and *Absent*. The value *Unavailable* means that a *True/False* value cannot be obtained, while *Absent* tells that we do not know whether *True* or *False* applies. It may seem curious that we view *Absent* as a value since it refers to a state of ignorance about the *True/False* value. But that convention is useful for the description of algorithms and results.

An example record is $(x_1 = \mathit{True}, x_2 = \mathit{False}, x_3 = \mathit{False}, x_4 = \mathit{Absent}, x_5 = \mathit{Unavailable})$, with the obvious interpretation.

Uniqueness of Records

Formally, we consider each record to have a unique identifier that differentiates it from other records. As a consequence, if two records have identical values, we nevertheless consider the records to be distinct. Thus, when two records with identical values are placed into a set, then the records constitute two distinct elements of the set.

The classification of unknown *True/False* values into *Unavailable* and *Absent* is not as straightforward as it might seem. We look into this aspect, beginning with an example.

Example: Medical Diagnosis

Consider the situation where the records of A (resp. B) represent patients with (resp. without) a certain disease. Part of each patient record is information about a measurement involving the left foot of the patient. Say, the variable x_1 is assigned *True* (resp. *False*) if the measured value is above (resp. equal to or below) a given threshold. One of the patients has lost the left foot in an accident. Evidently, the measurement cannot be done, and the value for x_1 must be declared to be unknown. The situation is different from that of another patient who has both feet, but where for some reason, say, due to carelessness of a technician, the measurement has not been properly done or recorded.

Other Unknown Value Cases

Additional situations resulting in unknown values are possible. For example, the value may exist in principle, but cannot be obtained or is almost impossible to obtain. Consider a medical test that normally is easily done, but that in the presence of a certain disease may be life-threatening and

thus cannot be performed. For a patient with that disease, the test effectively may not be performed even though in principle it could be done.

In yet another situation, some *True/False* entries of records are purposely omitted because they are considered irrelevant or unimportant. For example, a physician examines a patient and, by direct observation or tests, obtains *True/False* values for some symptoms and not for others. Evidently, the physician considers any symptoms for which he/she does not obtain *True/False* to be irrelevant or not needed for the diagnosis. We could say that the physician assigns the value *Unimportant* to the disregarded symptoms.

Types of Unknown Values

We approximate the various types of unknown values by the two types *Unavailable* and *Absent* introduced above. As we have seen, there are in-between situations for which we could define additional terms. Instead, we are pragmatic and force classification of cases using the two cited terms.

For a demonstration, we force classification of the above four cases. For the patient without a left foot, the value for x_1 is *Unavailable* since the value cannot possibly be obtained. For the patient with both feet, the value of x_1 is *Absent* since a repetition of the measurement can be done. More subtle are the two cases of patients where a record entry is not obtained due to life-threatening implications or due to the fact that the physician considers the entry to be irrelevant for the diagnosis. The two cases have in common that the entry is unavailable due to some restriction or decision. For this reason, we declare in both cases that the entry has the value *Unavailable*. A corollary of the argument is that, for the purposes of separating sets by logic formulas, we consider the value *Unimportant* to be equivalent to the value *Unavailable*. We list this rule for future reference.

(7.2.12) Rule. *In the context of separating logic formulas, the value Unimportant is represented in records by the value Unavailable.*

Evaluation of Formulas

We refine the evaluation of separating DNF formulas so that the two types of unknown values are reasonably accommodated. Let D_1 be a DNF clause, and define r to be a record involving variables $x_1, \ldots, x_n$. The value of each variable is *True*, *False*, *Unavailable*, or *Absent*. We declare that D_1 evaluates to *True* if the *True/False* values of the record cause all literals of D_1 to evaluate to *True*. We define D_1 to evaluate to *False* if, for each literal of D_1, the *True/False* value of the corresponding variable is known and causes the literal to evaluate to *False*, or if there is a literal of D_1 for which the corresponding variable has the value *Unavailable*. If D_1 does not

evaluate to *True* or *False* according to these definitions, then it evaluates to *Undecided*. Note that the last case is not possible if the record r does not contain any *Absent* entries. Thus, such a record r always produces *True* or *False*.

The evaluation of a DNF formula is as expected. That is, the value of D is *True* if at least one clause has value *True*, the value is *False* if all clauses have value *False*, and the value is *Undecided* otherwise.

We explain the definitions using intuitive arguments based on the above example where a measurement of the left foot determines the value of x_1. For simplicity, we suppose that *True/False* values are known for x_2, $x_3, \ldots, x_n$. Suppose a clause D_1 contains a literal of x_1. According to the above definition, D_1 can become *True* only if the value of x_1 is known and thus if the measurement has been taken. On the other hand, for the patient without a left foot, the value of x_1 is *Unavailable*, and D_1 evaluates to *False* regardless of the *True/False* values of the other variables. Finally, the case where the measurement was not properly performed on a patient with both feet, the value of x_1 is *Absent*, and D_1 evaluates to *Undecided*. These conclusions agree with the argument that, for the patient without a left foot, D_1 can never become *True* since x_1 cannot be ascertained, while, for a patient with both feet, D_1 can possibly become *True*.

Logic Data, Rational Data, and Set Data

In most settings, the record entries of the training sets A and B are not confined to *True/False* values, but can be rational numbers or may be *nominal* entries. An entry of the latter kind is an element or subset of some finite set. To differentiate among the cases, we call *True/False* entries *logic data*, rational number entries *rational data*, and nominal entries *set data*. For application of the methods of this chapter, we must transform all rational and set data of the training records into logic data. The same transformations are used subsequently when records are classified by the learned formulas.

In the remainder of this section, we sketch several transformations. We assume that, for fixed j, the jth entries of the training records are of the same type or unknown. The latter case is indicated by the value *Absent* or *Unavailable* as discussed earlier.

Processing of Absent and Unavailable Entries

We first strip out all *Absent* and *Unavailable* entries and establish transformations for the reduced data. Suppose a transformation replaces each jth entry by *True/False* values of k logic variables. We then encode each *Absent* or *Unavailable* entry in jth position in the original data by assigning

the same value, *Absent* or *Unavailable*, to each of the k variables. Due to this process, the transformations described below need not consider *Absent* and *Unavailable* entries. We begin with transformations for set data.

Transformation of Set Data

Let W be a finite set, and suppose that the jth entries of the records of A and B are elements or subsets of W. We do not allow a mixture of elements and subsets, so either all entries in jth position are elements or all entries are subsets. The transformation to logic data depends on which of the two cases is at hand.

Case of Element Entries

Suppose the set W has m elements, say $W = \{w_1, w_2, \ldots, w_m\}$. Thus, each entry in jth position is some $w_i \in W$.

If m is small, say $m \leq 5$, we introduce $k = m$ logic variables $x_1, x_2, \ldots, x_k$ and encode occurrence of entry w_i by

$$x_l = \begin{cases} \textit{True} & \text{if } i = l \\ \textit{False} & \text{otherwise} \end{cases} \tag{7.2.13}$$

For example, let $W = \{w_1, w_2, w_3\}$. Then $k = m = 3$, and w_1 is encoded by $x_1 = \textit{True}$, $x_2 = \textit{False}$, $x_3 = \textit{False}$, w_2 by $x_1 = \textit{False}$, $x_2 = \textit{True}$, $x_3 = \textit{False}$, and w_3 by $x_1 = \textit{False}$, $x_2 = \textit{False}$, $x_3 = \textit{True}$.

If m is large, we select an integer $f \geq 1$ and derive from W two sets W_A and W_B where W_A (resp. W_B) contains the $w_i \in W$ that occur at least f times as jth entry in records of A (resp. B) and never as jth entry in records of B (resp. A). We use a single logic variable x and encode w_i by

$$x = \begin{cases} \textit{True} & \text{if } w_i \in W_A \\ \textit{False} & \text{if } w_i \in W_B \\ \textit{Unavailable} & \text{otherwise} \end{cases} \tag{7.2.14}$$

For example, let $W = \{w_1, w_2, \ldots, w_{50}\}$ and $f = 5$. Suppose w_4, w_5, w_{10} are the $w_i \in W$ that occur at least f times as jth entry in records of A and that never occur as jth entry in records of B. Thus, $W_A = \{w_4, w_5, w_{10}\}$. Assume that, similarly, $W_B = \{w_7, w_9, w_{17}, w_{35}\}$. We encode three example cases w_5, w_{17}, and w_{31} using these sets. The entry w_5 is in W_A, so it is represented by $x = \textit{True}$. The entry w_{17} is in W_B and is encoded by $x = \textit{False}$. The entry w_{31} is not in W_A or W_B, and is encoded by $x = \textit{Unavailable}$.

An alternate method for the case of large m combines the ideas of the above two schemes, as follows. We define a small integer $k \geq 1$ and

collect in a set W' the k elements of W that occur most frequently as jth entry in the records of $A \cup B$. Suppose $W' = \{w_1, w_2, \ldots, w_k\}$. Let $W'' = W - W'$. If $w_i \in W'$, we represent w_i by *True/False* values of variables $x_1, x_2, \ldots, x_k$ as described above for small m. We also assign the value *Unavailable* to an additional variable x_{k+1}. If $w \in W''$, we define $x_1 = x_2 = \cdots = x_k =$ *Unavailable* and assign a *True/False/Unavailable* value to x_{k+1} according to the above method for large m; that is, the set W'' and the variable x_{k+1} are the input set and variable of the latter method.

For example, let $W = \{w_1, w_2, \ldots, w_{50}\}$ and $k = 3$. Suppose the k most frequently occurring entries in jth position are w_1, w_2, w_3. Then $W' = \{w_1, w_2, w_3\}$ and $W'' = W - W' = \{w_4, w_5, \ldots, w_{50}\}$. We show the encoding for example cases $w_2 \in W'$ and $w_8 \in W''$. Using the above scheme for small m, the encoding for $w_2 \in W'$ is $x_1 =$ *False*, $x_2 =$ *True*, $x_3 =$ *False*. We add to that encoding $x_{k+1} = x_4 =$ *Unavailable*. For the encoding of $w_8 \in W''$, suppose that the above method for large m determines from W'' the sets $W_A = \{w_4, w_5, w_{10}\}$ and $W_B = \{w_7, w_9, w_{17}, w_{35}\}$. Then $w_8 \in W''$ is encoded by $x_4 =$ *True* plus $x_1 = x_2 = x_3 =$ *Unavailable*.

Case of Set Entries

Suppose any one of given sets $V_1, V_2, \ldots, V_m$ may occur as jth entry. We could define $W = \{V_1, V_2, \ldots, V_m\}$ and use one of the methods of the preceding subsection to encode each V_i. That approach ignores possible relationships among the V_i and thus may be unsatisfactory. The following method recognizes such relationships and uses them for the encoding.

Define W_A (resp. W_B) to be the union of the V_i occurring as jth entry in records of A (resp. B) and never in records of B (resp. A). We define a *strength of membership* value $s(V_i)$ by

$$s(V_i) = \frac{|V_i \cap W_A| - |V_i \cap W_B|}{|V_i \cap (W_A \cup W_B)|} \tag{7.2.15}$$

Since $|V_i \cap W_A|$ and $|V_i \cap W_B|$ cannot be larger than $|V_i \cap (W_A \cup W_B)|$, we have $-1 \leq s(V_i) \leq 1$. In the records of A and B, we now replace each V_i occurring as jth entry by the rational value $s(V_i)$.

For example, let $W_A = \{w_1, w_2, \ldots, w_{10}\}$ and $W_B = \{w_{11}, w_{12}, \ldots, w_{30}\}$. If the jth entry of a record is $V_i = \{w_2, w_7, w_{10}, w_{14}, w_{36}, w_{54}\}$, then $V_i \cap W_A = \{w_2, w_7, w_{10}\}$, $V_i \cap W_B = \{w_{14}\}$, and $V_i \cap (W_A \cup W_B) = \{w_2, w_7, w_{10}, w_{14}\}$. Thus, $s(V_i) = (|V_i \cap W_A| - |V_i \cap W_B|)/|V_i \cap (W_A \cup W_B)| = (3-1)/4 = 0.5$.

Once all jth entries V_i have been transformed to rational entries $s(V_i)$, we replace those rational data by logic data using the techniques of the next subsection.

Transformation of Rational Data

We denote the rational numbers in jth position by $z_1, z_2, \ldots, z_m$. The transformation of the z_i to logic data uses $k \geq 1$ *markers* $p_1 < p_2 < \cdots < p_k$. These markers divide the rational line into $k+1$ intervals. The first interval is $(-\infty, p_1]$, the next $k-1$ intervals are $(p_l, p_{l+1}]$, $l = 1, 2, \ldots, k-1$, and the last interval is $(p_k, \infty]$.

The transformation replaces each z_i by *True/False* values of k logic variables $x_1, x_2, \ldots, x_k$ according to the interval into which z_i falls, as follows. If $z_i \in (-\infty, p_1]$, then $x_1 = x_2 = \cdots = x_k = \textit{False}$. If $z_i \in (p_l, p_{l+1}]$, then $x_1 = x_2 = \cdots = x_l = \textit{True}$ and $x_{l+1} = x_{l+2} = \cdots = x_k = \textit{False}$. If $z_i \in (p_k, \infty]$, then $x_1 = x_2 = \cdots = x_k = \textit{True}$. A compact representation of the rule is, for given z_i and for $l = 1, 2, \ldots, k$,

$$x_l = \begin{cases} \textit{True} & \text{if } z_i > p_l \\ \textit{False} & \text{otherwise} \end{cases} \tag{7.2.16}$$

For example, if we have $k = 2$ markers $p_1 = -17$ and $p_2 = 35$, then for $z_i = 3$ we have $z_i > p_1$ and $z_i < p_2$. Thus, z_i is represented by $x_1 = \textit{True}$, $x_2 = \textit{False}$.

The markers must be selected in such a way that the resulting logic data can be separated. The separation conditions are discussed in the next section. One may select the desired markers by statistical techniques, heuristics that minimize description length, clustering techniques, or methods that search for abrupt pattern changes. In the typical application of these methods, one determines an initial set of markers, checks if the separation conditions are satisfied and, if this is not the case, recursively adds markers, one at a time, until that goal is achieved. Due to space constraints, we cannot include a detailed discussion. However, that material is covered in detail elsewhere. References are included in Section 7.9.

In the remainder of the chapter, we assume that the training sets A and B contain logic data only. We are ready to compute separating DNF formulas for such A and B.

7.3 Separation of Two Sets

We are given two sets A and B of records. We are looking for a DNF clause D_1 that evaluates to *True* for the records of a maximum subset A_1 of A, and that evaluates to *False* for all records of B. Note that we do not allow D_1 to evaluate to *Undecided* for any record of A or B. That value may surface later when we use a learned DNF formula to evaluate records that were not used in the training.

If $A_1 = A$, then $D = D_1$ is a DNF formula that separates A from B. If $A_1 \subset A$, then we later invoke recursion to find additional DNF clauses that, together with D_1, constitute a DNF formula D that separates A from B. For the moment, we ignore this aspect and focus on the derivation of D_1.

The general form of D_1 is

$$D_1 = \bigwedge_{j=1}^{n} (x_j \text{ or } \neg x_j \text{ or no literal of } x_j) \tag{7.3.1}$$

For $j = 1, 2, \ldots, n$, we define two logic variables $x_j(pos)$ and $x_j(neg)$. *True/False* values for these variables induce a selection of literals for D_1 according to the rule

$$\begin{aligned} &x_j(pos) = \mathit{True} \text{ and } x_j(neg) = \mathit{False}\text{: select literal } x \\ &x_j(pos) = \mathit{False} \text{ and } x_j(neg) = \mathit{True}\text{: select literal } \neg x \\ &x_j(pos) = x_j(neg) = \mathit{False}\text{: do not select literal } x \text{ or } \neg x \\ &x_j(pos) = x_j(neg) = \mathit{True}\text{: this case is not allowed} \end{aligned} \tag{7.3.2}$$

We rule out the case $x_j(pos) = x_j(neg) = \mathit{True}$ of (7.3.2) by

$$\neg x_j(pos) \vee \neg x_j(neg), \quad j = 1, 2, \ldots, n \tag{7.3.3}$$

Let r be a record of A or B. Define $J_+^r = \{j \mid x_j = \mathit{True} \text{ in record } r\}$, $J_-^r = \{j \mid x_j = \mathit{False} \text{ in record } r\}$, $J_u^r = \{j \mid x_j = \mathit{Unavailable} \text{ in record } r\}$, and $J_a^r = \{j \mid x_j = \mathit{Absent} \text{ in record } r\}$. Thus, the record r can be described by

$$\begin{aligned} &x_j = \mathit{True}, \quad \forall j \in J_+^r \\ &x_j = \mathit{False}, \quad \forall j \in J_-^r \\ &x_j = \mathit{Unavailable}, \quad \forall j \in J_u^r \\ &x_j = \mathit{Absent}, \quad \forall j \in J_a^r \end{aligned} \tag{7.3.4}$$

Logic Conditions

The clause D_1 of (7.3.1) is to evaluate to *False* for all records $s \in B$. Thus, some literal x_j or $\neg x_j$ of D_1 must evaluate to *False* due to some record entry. This happens if and only if there is an index j such that D_1 contains the literal x_j or $\neg x_j$ and the record s has $x_j = \mathit{Unavailable}$, or D_1 contains the literal x_j and s has $x_j = \mathit{False}$, or D_1 contains the literal $\neg x_j$ and s has $x_j = \mathit{True}$. Due to (7.3.2), we may demand equivalently that there is

an index j such that $x_j(pos)$ has value *True* and $j \in (J_-^s \cup J_u^s)$, or $x_j(neg)$ has value *True* and $j \in (J_+^s \cup J_u^s)$. In logic notation, the condition becomes

$$\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \tag{7.3.5}$$

The clause D_1 is to evaluate to *True* for all records of the subset $A_1 \subseteq A$. Thus, for each literal of D_1, each record $r \in A_1$ must contain a *True/False* value such that the literal evaluates to *True*. Equivalently, for $j \in (J_+^r \cup J_a^r \cup J_u^r)$, $x_j(neg)$ must be *False*, and for $j \in (J_-^r \cup J_a^r \cup J_u^r)$, $x_j(pos)$ must be *False*. The condition is captured by

$$\bigwedge_{j \in (J_+^r \cup J_a^r \cup J_u^r)} \neg x_j(neg) \wedge \bigwedge_{j \in (J_-^r \cup J_a^r \cup J_u^r)} \neg x_j(pos) \tag{7.3.6}$$

We model membership of r in A_1 by a logic variable $select(r)$ which has value *True* if r is in A_1 and thus is separated from B. Consider

(7.3.7)

$$select(r) \rightarrow \Big[\bigwedge_{j \in (J_+^r \cup J_a^r \cup J_u^r)} \neg x_j(neg) \wedge \bigwedge_{j \in (J_-^r \cup J_a^r \cup J_u^r)} \neg x_j(pos) \Big], \quad \forall r \in A$$

Take any $r \in A$. If $select(r) = False$ and thus $r \notin A_1$, then the condition of (7.3.7) for r is trivially satisfied. On the other hand, if $select(r) = True$ and thus $r \in A_1$, then the condition of (7.3.7) for r becomes (7.3.6), which enforces D_1 to have the value *True* for r. We conclude that (7.3.7) correctly models the condition that D_1 must evaluate to *True* for all records of A_1.

We reconsider (7.3.3), which, for $j = 1, 2, \ldots, n$, requires $\neg x_j(pos) \vee \neg x_j(neg)$. First, we assume that some $select(r)$ has the value *True*. Then for that r, (7.3.7) becomes (7.3.6). The latter condition implies that, for $j = 1, 2, \ldots, n$, $x_j(pos) = False$ or $x_j(neg) = False$. Thus the solution satisfies $\neg x_j(pos) \vee \neg x_j(neg)$, and (7.3.3) has become redundant.

Second, we consider the remaining case, where all $select(r)$ have the value *False*. This means that we do not select any record $r \in A$ for A_1. But then there is no need or justification to interpret *True/False* values for $x_j(pos)$ or $x_j(neg)$, and it is irrelevant whether their values satisfy the constraint $\neg x_j(pos) \vee \neg x_j(neg)$. Hence, we conclude once more that the condition (7.3.3) can be eliminated.

Summary of Conditions

When we convert (7.3.7) to CNF and combine the resulting clauses with the CNF clauses of (7.3.5), we get the CNF system

$$\begin{aligned} &\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \\ &\neg x_j(neg) \vee \neg\, select(r), \quad \forall j \in (J_+^r \cup J_a^r \cup J_u^r), \ \forall r \in A \\ &\neg x_j(pos) \vee \neg\, select(r), \quad \forall j \in (J_-^r \cup J_a^r \cup J_u^r), \ \forall r \in A \end{aligned} \tag{7.3.8}$$

These clauses imply two important facts. First, the distinction between *Unavailable* and *Absent* entries in record $r \in A$ is irrelevant for the separation. Second, that distinction is important for the records of B.

Determination of Clause

We find D_1 separating a maximum subset A_1 of A from B as follows. We assign to each $select(r)$, $r \in A$, a cost of 1 for *False* and declare the costs of all other *True/False* cases of the variables of (7.3.8) to be 0. We solve the MINSAT instance given by these costs and (7.3.8). Due to the cost values, the MINSAT solution avoids assignment of *False* to the $select(r)$ variables as much as possible. Effectively, the solution enforces, for a maximum subset A_1 of A,

$$\bigwedge_{j \in (J_+^r \cup J_a^r \cup J_u^r)} \neg x_j(neg) \wedge \bigwedge_{j \in (J_-^r \cup J_a^r \cup J_u^r)} \neg x_j(pos), \quad \forall r \in A_1 \tag{7.3.9}$$

The DNF clause D_1 defined by that solution evaluates to *True* on A_1 and to *False* on B.

Recursive Process

Suppose A_1 is nonempty. We repeat the process recursively with the following changes. We use $A - A_1$ instead of A, declare the resulting subset of $A - A_1$ to be A_2, and define the resulting DNF clause to be D_2. By these definitions, D_2 evaluates to *True* on A_2 and to *False* on B. We conclude that the DNF formula $D = D_1 \vee D_2$ evaluates to *True* on $A_1 \cup A_2$ and to *False* on B. Suppose A_2 is nonempty. If $A_1 \cup A_2 = A$, we stop with that D. Otherwise, we repeat the above process with suitably reduced A to get D_3 and A_3, then D_4 and A_4, and so on. Suppose the successively found A_k are all nonempty. Then in a finite number of iterations, say l, we have $A = \bigcup_{k=1}^{l} A_k$, and the DNF formula $D = \bigvee_{k=1}^{l} D_k$ evaluates to *True* on $A = \bigcup_{k=1}^{l} A_k$ and to *False* on B.

The above arguments assume that each A_k is nonempty. We look at a simple case to understand why A_k may be empty. From that insight, we derive necessary and sufficient conditions for each A_k to be nonempty.

Empty Subset

Suppose that we have three variables x_1, x_2, x_3. The set A contains the record

$$(x_1 = True, x_2 = False, x_3 = Unavailable) \tag{7.3.10}$$

The set B contains the record

$$(x_1 = True, x_2 = Absent, x_3 = True) \tag{7.3.11}$$

For that record of B, the condition (7.3.5) becomes

$$x_1(neg) \vee x_3(neg) \tag{7.3.12}$$

For the A record, the condition (7.3.6) is

$$\neg x_1(neg) \wedge \neg x_2(pos) \wedge \neg x_3(pos) \wedge \neg x_3(neg) \tag{7.3.13}$$

The constraint (7.3.13) forces $x_1(neg) = x_2(pos) = x_3(pos) = x_3(neg) = False$. When these values are used in (7.3.12), that condition becomes unsatisfiable. Thus, the two conditions (7.3.12) and (7.3.13) cannot be simultaneously satisfied, and the A record (7.3.10) cannot be separated from the B record (7.3.11). The unsatisfiability is caused by the fact that, for each x_j for which the A record has a *True/False* value, the B record has the same *True/False* value or *Absent*. We may rephrase the situation as follows. In the B record, we can replace *Absent* values by *True/False* values such that all *True/False* values of A occur in the modified B record. When this is the case, we say that the A record is *weakly nested* in the B record.

Separation and Weak Nestedness

The next theorem shows that exclusion of weak nestedness is not only necessary but also sufficient for the existence of separations.

(7.3.14) Theorem. *Let A and B be sets of records. Then there is* a DNF *formula D that separates A from B if and only if no record $r \in A$ is weakly nested in any record $s \in B$.*

Proof. Suppose a DNF formula D separates A from B. Take any record $r \in A$. For that record, D evaluates to *True*. Thus, D has a clause D_k so that, for each literal x_j (resp. $\neg x_j$) of D_k, the record r specifies $x_j = True$ (resp. $x_j = False$). For any B record s, D_k must evaluate to *False*. Therefore, for some literal of D_k, say, with index j, the record s has $x_j = Unavailable$, or the literal is x_j and s has $x_j = False$, or the literal is $\neg x_j$ and s has $x_j = True$. These facts prove that r is not weakly nested in s.

Conversely, suppose that no $r \in A$ record is weakly nested in any $s \in B$. Define D to be the DNF formula where each $r \in A$ defines a DNF clause, say D_r, as follows. For each $x_j = True$ (resp. $x_j = False$) of r, the clause D_r contains the literal x_j (resp. $\neg x_j$). By this construction, D_r

evaluates to *True* for r. Take any $s \in B$. By assumption, the record r is not weakly nested in s. Hence, record r has a *True/False* value for some x_j for which record s has the opposite *True/False* value or has the value *Unavailable*. Regardless of the case, the clause D_r evaluates to *False* for record s. We conclude that the DNF formula D, which is composed of the clauses D_r, $r \in A$, evaluates to *True* (resp. *False*) for all records of A (resp. B). □

Algorithm for Separating DNF Formula

We summarize the above process for finding a DNF formula separating two given sets of records.

(7.3.15) Algorithm SEPARATING DNF FORMULA. *Determines a DNF formula for separating sets.*

Input: Nonempty sets A and B whose records involve variables $x_1, x_2, \ldots, x_n$. No A record is weakly nested in any B record. (If this condition does not hold, there is no DNF formula that separates A from B.)

Output: A DNF formula D that evaluates to *True* (resp. *False*) for each A record (resp. B record). Thus, D separates A from B.

Requires: Algorithm SOLVE MINSAT.

Procedure:

0. Initialize $k = 1$.
1. (Compute DNF clause D_k.) Solve the MINSAT instance with CNF clauses

$$
\begin{aligned}
&\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \\
&\neg x_j(neg) \vee \neg\, select(r), \quad \forall j \in (J_+^r \cup J_a^r \cup J_u^r), \ \forall r \in A \\
&\neg x_j(pos) \vee \neg\, select(r), \quad \forall j \in (J_-^r \cup J_a^r \cup J_u^r), \ \forall r \in A
\end{aligned}
\tag{7.3.16}
$$

   Each variables $select(r)$ has cost 1 for the value *False*. All other costs are 0. Define the DNF clause D_k from the MINSAT solution using the following rule. For $j = 1, 2, \ldots, n$, if $x_j(pos)$ (resp. $x_j(neg)$) has the value *True*, then declare x_j (resp. $\neg x_j$) to be a literal of D_k. Define $A_k = \{r \in A \mid select(r) = True\}$.
2. Update A by removing the records of A_k. If the new A is empty, output $D = D_1 \vee D_2 \vee \cdots \vee D_k$ as the desired DNF formula, and stop. Otherwise, increment k by 1, and go to Step 1.

Proof of Validity. The algorithm is a restatement of the iterative process discussed earlier. Theorem (7.3.14) establishes that exclusion of weak nestedness is necessary and sufficient for the existence of the separating DNF formula. □

Exclusion of Weak Nestedness

Algorithm SEPARATING DNF FORMULA (7.3.15) requires that no A record is weakly nested in any B record. Suppose A and B do not satisfy that condition, yet we want to use the sets for training after suitable modification. There are several ways to achieve absence of weak nestedness of A records in B records. We describe a pragmatic approach that in applications has worked well.

Reduction of Sets

Define a record $r \in A$ to be *nested* in a record $s \in B$ if the *True/False* values of r also occur in s. Evidently, nestedness is a special case of weak nestedness.

If the training data originally contained rational data or set data that were replaced by logic data, then we check if a change of the parameters of the transformations can eliminate the nestedness. In the case of set data where elements are replaced by *True/False* values, the changes are limited to modification of the parameters f and, possibly, k. For details about these parameters, see Section 7.2. In the case of set data where subsets are transformed to logic data, the transformation first converts the set data to rational data, and then converts these rational data to logic data. The latter transformation can be improved by using additional markers. The same modification applies generally when rational data are converted to logic data. Due to space constraints, we cannot include details of the refinement process. However, that material is covered elsewhere. References are provided in Section 7.9.

Suppose that we cannot eliminate the nestedness by more precise transformations, or that the original entries are already logic data. We delete some records of A or B to achieve nonnestedness in a two-stage process.

In the first stage, we delete from A any record r that is nested in any record $s \in B$. The removal of r is appropriate since every *True/False* value of r occurs in s and thus cannot possibly be sufficient to prove membership of $r \in A$.

In the second stage, we delete any $r \in A$ and any $s \in B$ for which r is weakly nested in s. We cannot argue that removal of both records is mandatory. We prefer that reduction to any remedy where just one of the two records is removed, since such a remedy may introduce a bias into the training sets.

Unseen Record

Define a record r of the two populations $\mathcal{A}$ and $\mathcal{B}$ that is not part of the training sets, that is, $r \in [(\mathcal{A} - A) \cup (\mathcal{B} - B)]$, to be *unseen in training*, for

short *unseen*. The DNF formula D computed by Algorithm SEPARATING DNF FORMULA (7.3.15) evaluates to *True* (resp. *False*) for each record of A (resp. B). When D is applied to an unseen record r, the value may be *True*, *False*, or *Undecided*. For the third case, recall that no clause of D evaluates to *True*, and that at least one clause has value *Undecided*. The value *Undecided* is at hand for a clause, say D_k, if all literals of D_k for which the record has a *True/False* value evaluate to *True*, and if for each remaining literal of D_k the corresponding value of the record is *Absent*.

Undecided Value of Formula

We examine the *Undecided* case for the DNF formula D. Evidently, we can replace the *Absent* values of r by *True/False* so that at least one of the clauses, and thus D, evaluate to *True*. However, we may not be able to select such a replacement of values for r so that all clauses, and thus D, evaluate to *False*. An example demonstrates the latter situation. Suppose that $D = (x_1 \wedge \neg x_2) \vee x_2$ and that $r = (x_1 = \mathit{True}, x_2 = \mathit{Absent})$. Then for both $x_2 = \mathit{True}$ and $x_2 = \mathit{False}$, D evaluates to *True*. Thus, no value for x_2 can lead to the value *False* for D. In general, one may test whether replacement of *Absent* values by *True/False* values can produce *False* for D by the following process. First, delete all clauses from D that evaluate to *False* for record r. Since the value of D is *Undecided*, all remaining clauses have value *Undecided*, and at least one such clause must remain.

From each remaining clause, delete all literals that due to *True/False* values of r evaluate to *True*. The second step cannot eliminate any clause entirely since otherwise D would have evaluated to *True*. Let D' be the reduced DNF formula. All variables defining literals in D' must have the value *Absent* in r, since otherwise at least one clause of D' evaluates to *False*, a contradiction.

We want to know if replacement of *Absent* values by *True/False* in r can produce *False* for D', or equivalently, *True* for $\neg D'$. Since D' is a DNF formula, $S = \neg D'$ is a CNF formula, and the question about the possibility of *False* for D' has "yes" as answer if and only if S is satisfiable. The above example has $D = (x_1 \wedge \neg x_2) \vee x_2$ and $x_1 = \mathit{True}$. Thus, $D' = \neg x_2 \vee x_2$, and $S = \neg D' = x_2 \wedge \neg x_2$, which is unsatisfiable. This result confirms the earlier conclusion that no value for x_2 can result in *False* for D. Thus, we are justified to replace the *Undecided* value by *True*.

So far, we have not paid any attention to the particular form of the learned DNF formula D. In the next two sections, we examine that aspect.

7.4 Min and Max Formulas

Each iteration through Step 2 of Algorithm SEPARATING DNF FORMULA (7.3.15) produces a DNF clause D_k for the separating formula D. Indeed, the solution of the MINSAT instance defined by (7.3.16) and the accompanying costs defines a subset $A_k = \{r \in A \mid select(r) = True\}$ of A that D_k separates from B. If our goal is just separation of A from B, the particular form of D_k does not matter. However, if we intend to apply D to classify unseen records r, that is, $r \in [(\mathcal{A} - A) \cup (\mathcal{B} - B)]$, then the form of each D_k matters. In this section, we look at two particular forms of the clauses D_k that are important for the processing of unseen records. In the first (resp. second) case, D_k has a minimum (resp. maximum) number of literals. We see later in this section why such D_k are of interest. In the next section, we encounter yet another form of D_k that is important for the classification of unseen records. There, the literals are optimized relative to certain tests and costs.

Formulation for Min/Max Formulas

For the set A_k at hand, we compute a D_k with minimum or maximum number of literals as follows. Assume we have already obtained via Algorithm SEPARATING DNF FORMULA (7.3.15) the set A_k that D_k is to separate from B. We simplify (7.3.16) by deleting the variables $select(r)$ and by restricting the conditions previously imposed on all $r \in A$ to the $r \in A_k$. The resulting CNF clauses enforce that the records of A_k are separated from B, as follows.

$$
\begin{aligned}
&\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \\
&\neg x_j(neg), \quad \forall j \in (J_+^r \cup J_a^r \cup J_u^r), \ \forall r \in A_k \\
&\neg x_j(pos), \quad \forall j \in (J_-^r \cup J_a^r \cup J_u^r), \ \forall r \in A_k
\end{aligned} \tag{7.4.1}
$$

Min Case

For the case of minimum number of literals of D_k, we define the cost of *True* (resp. *False*) for each $x_j(pos)$ and $x_j(neg)$ to be 1 (resp. 0). The MINSAT instance defined by (7.4.1) and these costs effectively demand that the value *True* for $x_j(pos)$ and $x_j(neg)$ is to be avoided as much as possible. By (7.3.2), *True* for $x_j(pos)$ (resp. $x_j(neg)$) results in the literal x_j (resp. $\neg x_j$) for D_k. Hence, the MINSAT solution produces a D_k with minimum number of literals. We say that D_k is a *min* DNF *clause* and denote it by D_k^{min}. The DNF formula produced this way is a *min* DNF *formula*, denoted by D^{min}.

Max Case

For the case of maximum number of literals of D_k, we define the cost of *True* (resp. *False*) of $x_j(pos)$ and $x_j(neg)$ to be 0 (resp. 1). Correspondingly, the solution has as many *True* values as possible, and D_k has as many literals as possible. We call such D_k a *max* DNF *clause* and denote it by D_k^{max}. The resulting DNF formula is a *max* DNF *formula*, denoted by D^{max}.

Computational Aspects

We take a closer look at the two MINSAT instances. For both of them, the CNF clauses of (7.4.1) arising from the records $r \in A_k$ amount to a fixing of some of the $x_j(pos)$ and $x_j(neg)$ variables to *False*. In the min case, we then must select a minimum number of *True* values for the remaining $x_j(pos)$ and $x_j(neg)$ so that the clauses arising from the records $s \in B$ are satisfied. The problem essentially is the so-called MINIMUM COVER problem, where one must cover a given set with a least number of given subsets. Exercise (7.10.5) asks the reader to work out the precise relationships. MINIMUM COVER is known to be hard. Thus, the min case, which essentially is the problem MINIMUM COVER, is difficult to solve.

On the other hand, the max case requires that we select a maximum number of *True* values. That is readily achieved by assigning *True* to each $x_j(pos)$ and $x_j(neg)$ that has not been fixed by some record of A_k. Thus, the max case is easily solved.

Link between Min and Max Cases

The above discussion implies that each $x_j(pos)$ or $x_j(neg)$ with value *True* in the min case is also assigned *True* in the max case. The next theorem restates that observation.

(7.4.2) Theorem. *Each literal of each clause D_k^{min} is also a literal of the corresponding clause D_k^{max}.*

Proof. We have seen that each $x_j(pos)$ and $x_j(neg)$ with value *True* in the min case also has that value in the max case. Thus, each literal of D_k^{min} must also occur in D_k^{max}. □

The ease of computing D^{max} vanishes when we consider additional logic relationships connecting the variables $x_j(pos)$ and $x_j(neg)$, $j = 1, 2, \ldots, n$. This case is treated in Section 7.6.

Algorithm for Min/Max DNF Formulas

The following algorithm for computation of D^{min} and D^{max} relies on Step 1 of Algorithm SEPARATING DNF FORMULA (7.3.15). Validity of the algorithm follows from the above discussion.

(7.4.3) Algorithm MIN/MAX DNF FORMULAS. *Determines min and max* DNF *formulas for separating sets.*

Input: Nonempty sets A and B whose records involve variables $x_1, x_2, \ldots, x_n$. No A record is weakly nested in any B record. (If this condition does not hold, there is no DNF formula that separates A from B.)

Output: DNF formulas D^{min} and D^{max} that evaluate to *True* (resp. *False*) for each A record (resp. B record). Thus, both D^{min} and D^{max} separate A from B.

Requires: Algorithm SOLVE MINSAT.

Procedure:

0. Initialize $k = 1$.
1. (Compute A_k.) Solve the MINSAT instance with CNF clauses

$$
\begin{aligned}
&\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \\
&\neg x_j(neg) \vee \neg\, select(r), \quad \forall j \in (J_+^r \cup J_a^r \cup J_u^r), \ \forall r \in A \\
&\neg x_j(pos) \vee \neg\, select(r), \quad \forall j \in (J_-^r \cup J_a^r \cup J_u^r), \ \forall r \in A
\end{aligned} \tag{7.4.4}
$$

   Each variable $select(r)$ has cost 1 for the value *False*. All other costs are 0. Using the MINSAT solution, define $A_k = \{r \in A \mid select(r) = True\}$.
2. (Compute clauses D_k^{min} and D_k^{max}.) Solve the MINSAT instance with CNF clauses

$$
\begin{aligned}
&\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \\
&\neg x_j(neg), \quad \forall j \in (J_+^r \cup J_a^r \cup J_u^r), \ \forall r \in A_k \\
&\neg x_j(pos), \quad \forall j \in (J_-^r \cup J_a^r \cup J_u^r), \ \forall r \in A_k
\end{aligned} \tag{7.4.5}
$$

   Define the cost of *True* (resp. *False*) for each $x_j(pos)$ and $x_j(neg)$ to be 1 (resp. 0). Define DNF clause D_k^{min} from the MINSAT solution using the following rule. For $j = 1, 2, \ldots, n$, if $x_j(pos)$ (resp. $x_j(neg)$) has the value *True*, then declare x_j (resp. $\neg x_j$) to be a literal of D_k^{min}. Define D_k^{max} to have the following literals. For $j = 1, 2, \ldots, n$, if the

clauses of (7.4.5) involving A_k do not force $x_j(pos)$ (resp. $x_j(neg)$) to have the value *False*, then x_j (resp. $\neg x_j$) is a literal of D_k^{max}.

3. Update A by removing the records of A_k. If the new A is empty, output as the desired min and max DNF formulas the DNF formula D^{min} composed of $D_1^{min}, D_2^{min}, \ldots, D_k^{min}$ and the DNF formula D^{max} composed of $D_1^{max}, D_2^{max}, \ldots, D_k^{max}$, and stop. Otherwise, increment k by 1, and go to Step 1.

Bias of Min/Max Formulas

Let us examine the behavior of min formula D^{min} and max formula D^{max} when an unseen record is evaluated. If D^{max} evaluates to *True*, then some clause D_k^{max} evaluates to *True*. By Theorem (7.4.2), D_k^{max} contains all literals of D_k^{min}, so D_k^{min} evaluates to *True* as well. Put differently, if D^{max} classifies the record as being in $\mathcal{A}$, then D^{min} does so as well. On the other hand, D_k^{min} may evaluate to *True* and D_k^{max} to *False* or *Undecided*, so D^{min} may classify the record as being in $\mathcal{A}$, while D^{max} declares it to be in $\mathcal{B}$ or *Undecided*. We thus may say that D^{min} *favors* classification into $\mathcal{A}$ when compared with D^{max}. Analogous arguments involving $\mathcal{B}$ show that D^{max} *favors* classification into $\mathcal{B}$ when compared with D^{min}. This *bias* of D^{min} and D^{max} is useful in Section 7.8 and in Chapter 8.

So far, the values of training records were always given. The next section covers the case where these values can only be obtained via tests at some cost.

7.5 Optimized Formulas

As before, we assume that records involve variables $x_1, x_2, \ldots, x_m$. Suppose that the values of records can only be obtained by certain tests, and that each test involves a given cost. Note that this ignorance of values is conceptual during the training process. That is, we already have the values of the record, but we are not allowed to use these values for learning logic formulas unless we obtain the values, conceptually, by tests.

Test and Cost

In general, a test T may produce the values for the variables indexed by a subset of $\{1, 2, \ldots, n\}$. For notational convenience, we denote that subset also by T. Thus, the test T determines the values of the variables x_j with $j \in T$. As an example, if $T = \{1, 2, 4\}$, then the test T determines the values for the variables x_1, x_2, and x_4.

Let T determine the value of x_j. Obviously, *True* and *False* are possible values produced by T for x_j. But we also consider *Unavailable* to be a test outcome. It corresponds to the event where T somehow fails to produce a *True/False* value for x_j.

We emphasize that T cannot produce the value *Absent* for x_j. Indeed, one may declare that T converts the value *Absent* of x_j to *True*, *False*, or *Unavailable*.

We introduce an exception to the above rule for training records. We have the values for these records already, but act as if we did not have them and obtain them, conceptually, by tests. Suppose a training record r has $x_j = Absent$. When conceptually we carry out test T to determine the value of x_j, then we cannot assign *True*, *False*, or *Unavailable* as test outcome to x_j since the record does not provide that information. Thus, we allow an exception and declare that the test leaves the value of x_j as *Absent*. Alternately, we could bar the use of test T for obtaining values for the training record r. But that restriction would unnecessarily complicate the algorithms of this section, and we prefer to adopt the exception. We emphasize that the exception applies to training records only and not to unseen records, where we truly do not have any entries and must obtain them by tests.

Suppose that, for some $m \geq 1$, tests $T_1, T_2, \ldots, T_m$ are possible. For $i = 1, 2, \ldots, m$, we denote the cost of carrying out test T_i by c_i. For each variable, we assume that there is at least one test that determines the value of the variable.

Sequential Classification Process

We consider the following sequential process for the classification of an unseen record. At the outset, we do not know any information about the record. Accordingly, we assign *Absent* as value to all x_j. We select a test, obtain *True/False/Unavailable* values for the corresponding variables, and check if the record can be classified as being in $\mathcal{A}$ or $\mathcal{B}$. In the affirmative case, we are done. Otherwise, we proceed recursively until the affirmative case is at hand. Note that we cannot have an *Undecided* outcome, since collectively the tests determine *True/False/Unavailable* values for all variables.

Selection of Tests

We want to compute a DNF formula D and select the tests so that the expected total cost of classifying a randomly selected record $r \in (\mathcal{A} \cup \mathcal{B})$ is minimum. Exact solution of the problem is not easy. It requires the joint probability distribution of the *True/False/Unavailable* values of records. That information often is not available with sufficient precision.

Instead of searching for an exact solution, we break down the problem into four subproblems, which we solve in sequence. The solutions of the subproblems support a testing and classification scheme that can be reasonably expected to classify a record r at low total cost.

Summary of Subproblems

In Subproblem 1, we determine, for each record $r \in A$, a collection of tests T_i so that the values produced by these tests allow separation of r from all records of B at minimum cost. We denote that minimum cost for r by d_r^*.

In Subproblem 2, we find tests so that each $r \in A$ can be separated from B at a cost close to d_r^*. Let K_r^* be the set of indices of the tests so selected for r. The reader may wonder why we consider testing cost that are only close to d_r^*, given that we already have the minimum testing cost d_r^*. The reason is that each distinct cost value later requires at least one DNF clause in the yet to be found separating DNF formula. By allowing the cost of tests to be close to d_r^* instead of equal to d_r^*, we typically hold down the number of distinct costs and keep the number of DNF clauses reasonable.

As part of Subproblem 2, we derive an *optimized record* r^* from each $r \in A$ by replacing each *True/False* value of r not obtainable by any test of K_r^* by *Unavailable*. A more descriptive substitution value would be *Unimportant*. The use of *Unavailable* is justified by Rule (7.2.12), according to which *Unimportant* can be replaced by *Unavailable*.

We collect the optimized records r^* in a set A^*.

Define $K^* = \{K_r^* \mid r \in A\}$, which thus contains all test sets for A. We call K^* the *optimized test set*.

In Subproblem 3, we separate A^* from B analogously to the separation of A from B discussed in Section 7.4, using Algorithm MIN/MAX DNF FORMULAS (7.4.3). We relabel the DNF formulas D^{min} and D^{max} produced by the algorithm as D^{*min} and D^{*max} and declare them to be *optimized* DNF *formulas*.

When the roles of A and B in Subproblems 1–3 are reversed, we derive, for each record $s \in B$, an optimized record s^* with minimum cost e_s^* and corresponding tests specified by a set L_s^*. We also obtain optimized DNF formulas E^{*min} and E^{*max}. Let $L^* = \{L_s^* \mid s \in B\}$, which is a second optimized test set.

In Subproblem 4, we use A^*, B^*, K^*, L^*, D^{*min}, D^{*max}, E^{*min}, and E^{*max} to classify records of $\mathcal{A} \cup \mathcal{B}$ by sequentially selecting tests so that expected total classification cost is low.

Subproblems 1–3 are solved next. Subproblem 4 is treated in Section 10.12 of Chapter 10 as part of question-and-answer processes. The material is sufficiently complex to defy a summarizing description by a few sentences. Thus, we do not attempt such a summary here, and instead

advise the reader interested in Subproblem 4 to skip ahead to Chapter 10 and, after reading that material, to return to this point.

Subproblem 1: Minimum Separation Cost

For each record $r \in A$, we determine tests so that the values produced by the tests allow separation of r from all records of B at minimum cost. Let $T_1, T_2, \ldots, T_m$ be the possible tests, with costs $c_1, c_2, \ldots, c_m$, respectively. Recall that, if a training record r has *Absent* entries, then no test can produce *True/False/Unavailable* values for such entries.

Any DNF formula that separates r from B is just a DNF clause, say D_r, that satisfies (7.4.1) when we temporarily consider A_k of (7.4.1) to contain just r. Thus, we have the following conditions.

$$
\begin{aligned}
&\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \\
&\neg x_j(neg), \quad \forall j \in (J_+^r \cup J_a^r \cup J_u^r) \\
&\neg x_j(pos), \quad \forall j \in (J_-^r \cup J_a^r \cup J_u^r)
\end{aligned} \tag{7.5.1}
$$

For $i = 1, 2, \ldots, m$, let $choose(T_i)$ be the logic variable that is *True* if test T_i is done.

Recall that $x_j(pos) = True$ (resp. $x_j(neg) = True$) effectively places x_j (resp. $\neg x_j$) into D_r. In turn, if the literal x_j or $\neg x_j$ occurs in D_r, then, for the classification of r, at least one test T_i must be performed that can produce the value of x_j. These facts lead to the following condition, where $I_j = \{i \mid T_i \text{ determines the } x_j \text{ value}\}$.

$$
[x_j(pos) \vee x_j(neg)] \rightarrow \bigvee_{i \in I_j} choose(T_i), \quad j = 1, 2, \ldots, n \tag{7.5.2}
$$

For $i = 1, 2, \ldots, m$, let the cost c_i of carrying out test T_i be the cost of *True* for $choose(T_i)$. All other costs are 0. We solve the MINSAT instance given by these costs and the clauses of (7.5.1) and (7.5.2), and thus obtain the desired minimum cost d_r^* for separating r from B.

We include an example.

Example: Minimum Separation Cost

Let the record r and the set B be given by (7.5.3) below. There are two tests for obtaining values of the variables. Test T_1 determines the values of x_1, x_2, and x_3 at cost \$15, while test T_2 determines the values of x_3 and x_4 at cost \$7.

(7.5.3)

Record r
$(x_1 = Absent, x_2 = True, x_3 = True, x_4 = False)$
Set B
$(x_1 = True, x_2 = True, x_3 = False, x_4 = True)$
$(x_1 = True, x_2 = False, x_3 = True, x_4 = Absent)$

Record r of A and training set B

From (7.5.1) and (7.5.2), we deduce the following conditions for the case at hand.

(7.5.4)

$$
\begin{aligned}
&x_1(neg) \vee x_2(neg) \vee x_3(pos) \vee x_4(neg)\\
&x_1(neg) \vee x_2(pos) \vee x_3(neg)\\
&\neg x_1(neg) \wedge \neg x_2(neg) \wedge \neg x_3(neg)\\
&\neg x_1(pos) \wedge \neg x_4(pos)\\
&[x_1(pos) \vee x_1(neg)] \rightarrow choose(T_1)\\
&[x_2(pos) \vee x_2(neg)] \rightarrow choose(T_1)\\
&[x_3(pos) \vee x_3(neg)] \rightarrow [choose(T_1) \vee choose(T_2)]\\
&[x_4(pos) \vee x_4(neg)] \rightarrow choose(T_2)
\end{aligned}
$$

For the value *True*, $choose(T_1)$ has cost \$15, and $choose(T_2)$ has cost \$7. All other costs are 0.

The optimal solution for the MINSAT instance has $choose(T_1) = True$ and $choose(T_2) = False$. Thus, the minimum separation cost is $d_r^* = 15$, and $I_r^* = \{1\}$. Since r has $x_1 = Absent$, the test T_1, though designed to determine x_1, x_2, and x_3, fails to produce a value for x_1 in this case. Thus, T_1 obtains the values $x_2 = x_3 = True$ of r, and these values separate r from B at minimum separation cost.

Suppose the record r has $x_1 = Unavailable$ instead of $x_1 = Absent$. It is easy to see that this change is of no consequence for (7.5.4) and thus does not alter the optimal selection of tests. On the other hand, suppose that the second record of B has $x_4 = Unavailable$ instead of $x_4 = Absent$. Then the clause $x_1(neg) \vee x_2(pos) \vee x_3(neg)$ of (7.5.4) changes to $x_1(neg) \vee x_2(pos) \vee x_3(neg) \vee x_4(pos) \vee x_4(neg)$, and the optimal solution of the modified MINSAT instance becomes $choose(T_1) = False$ and $choose(T_2) = True$, with minimum total cost $d_r^* = 7$ instead of $d_r^* = 15$. Thus, it is important that T_2 determines $x_4 = Unavailable$ in the second B record. These conclusions are in agreement with the two facts following (7.3.8), according to which *Unavailable* in A (resp. B) records are irrelevant (resp. important) for separation of A from B.

Algorithm for Minimum Separation Cost

We summarize the computation of minimum separation cost.

(7.5.5) Algorithm MINIMUM SEPARATION COST. *Determines minimum cost for separating a given record from all records of a given set.*

Input: Record r and nonempty set B involving variables $x_1, x_2, \ldots, x_n$. Record r is not weakly nested in any B record. (If this condition does not hold, there is no DNF formula that separates r from B.) Tests $T_1, T_2, \ldots, T_m$ with costs $c_1, c_2, \ldots, c_m$, respectively. For each variable x_j, there is at least one test that provides the value of the variable.

Output: The minimum cost d_r^* required for tests that allow separation of r from B. An index set I_r^* of tests that achieve that result.

Requires: Algorithm SOLVE MINSAT.

Procedure:

0. For $j = 1, 2, \ldots, n$, define $I_j = \{i \mid T_i \text{ determines the } x_j \text{ value}\}$.
1. Solve the MINSAT instance given by the following conditions and costs.

$$
\begin{aligned}
&\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \\
&\neg x_j(neg), \quad \forall j \in (J_+^r \cup J_a^r \cup J_u^r) \\
&\neg x_j(pos), \quad \forall j \in (J_-^r \cup J_a^r \cup J_u^r) \\
&[x_j(pos) \vee x_j(neg)] \rightarrow \bigvee_{i \in I_j} choose(T_i), \quad j = 1, 2, \ldots, n
\end{aligned}
\tag{7.5.6}
$$

   For $i = 1, 2, \ldots, m$, the cost of *True* for $choose(T_i)$ is c_i. All other costs are 0.
2. Using the MINSAT solution, define d_r^* to be the minimum total cost and $I_r^* = \{i \mid choose(T_i) = True\}$. Output d_r^* and I_r^*, and stop.

   We are ready to tackle Subproblem 2.

Subproblem 2: Optimized Records and Test Set

We want to obtain the set A^* of optimized records r^* and the optimized test set $K^* = \{K_r^* \mid r \in A\}$. Each K_r^* specifies tests that have total cost close to the minimum cost d_r^* computed by Subproblem 1.

Parameter λ

How close the total cost is to d_r^* depends on a rational parameter $\lambda \geq 1.0$. Indeed, the total cost of the tests is guaranteed to lie in the interval $[d_r^*, \lambda \cdot d_r^*]$. Why not always choose $\lambda = 1.0$? As we shall see, decreasing values of λ tend to increase the number of clauses of the yet to be found DNF formula that separates A^* from B, and $\lambda = 1.0$ may result in an unacceptably large number of clauses. Practically speaking, we should select λ as the smallest value for which the separating DNF formula has a reasonable number of clauses. Later, we determine that value by trial and error with the algorithm for Subproblem 3.

Computation of A^* and K^*

Given λ, the desired A^* and K^* is found as follows. Remove from A a record r_0 having the smallest d_r^* value, and initialize a set $A' = \{r_0\}$. By the derivation of d_r^*, one can separate r_0 from B using tests with total costs equal to $d_{r_0}^*$. From Subproblem 1, recall that $I_{r_0}^*$ is the set of indices of tests determined for r_0. Initialize a set $I^* = I_{r_0}^*$.

Considering the records $r \in A$ one at a time, we attempt to enlarge A' so that the new A' can be separated from B by one DNF clause and the total cost of obtaining *True/False* values for the variables of that clause does not exceed $\lambda \cdot d_{r_0}^*$. Thus, if a record r of A has $d_r^* > \lambda \cdot d_{r_0}^*$, then it cannot be a candidate for addition to A'. Accordingly, we only consider $r \in A$ with $d_r^* \leq \lambda \cdot d_{r_0}^*$.

We tentatively remove r from A and add it to A'. We check if one clause can separate A' from B by solving a MINSAT problem. Analogously to (7.5.6), the following conditions enforce that requirement.

$$
\begin{aligned}
&\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \\
&\neg x_j(neg), \quad \forall j \in (J_+^{r'} \cup J_a^{r'} \cup J_u^{r'}), \ \forall r' \in A' \\
&\neg x_j(pos), \quad \forall j \in (J_-^{r'} \cup J_a^{r'} \cup J_u^{r'}), \ \forall r' \in A' \\
&[x_j(pos) \vee x_j(neg)] \rightarrow \bigvee_{i \in I_j} choose(T_i), \quad j = 1, 2, \ldots, n
\end{aligned}
\tag{7.5.7}
$$

For $i = 1, 2, \ldots, m$, the cost of *True* for $choose(T_i)$ is the cost c_i of carrying out test T_i. All other costs are 0.

We solve the MINSAT instance. If (7.5.7) is unsatisfiable, or if the total cost of the solution exceeds $\lambda \cdot d_{r_0}^*$, we remove r from A' and place it back into A. Otherwise, we leave r in A' and update the set I^* so that it contains the indices of the tests T_i for which $choose(T_i) = True$.

The iterative process stops when all records $r \in A$ with $d_r^* \leq \lambda \cdot d_{r_0}^*$ have been tried for transfer to A'. At that time, we carry out the following steps. For each $r \in A'$, we define $K_r^* = I^*$ and place into A^* a record r^* that is derived from r by replacing the *True/False* values that are not determined by any test with index in K_r^*, by *Unavailable*. Then we apply the above process recursively. That is, we find another r_0 of the current A whose cost $d_{r_0}^*$ is minimum among the d_r^*, $r \in A$, initialize $A' = \{r_0\}$ and $I^* = I_r^*$, and so on. The recursive process stops when the reduced A has become empty.

We skip example calculations since they are quite tedious, yet in spirit similar to those of Subproblem 1 for d_r^*.

Algorithm for Optimized Records

The solution process for Subproblem 2 is captured in the following algorithm.

(7.5.8) Algorithm OPTIMIZED RECORDS. *Determines set of optimized records and related test sets.*

Input: Nonempty sets A and B whose records involve variables $x_1, x_2, \ldots, x_n$. No A record is weakly nested in any B record. (If this condition does not hold, there is no DNF formula that separates A from B.) Tests $T_1, T_2, \ldots, T_m$ with costs $c_1, c_2, \ldots, c_m$, respectively. For each variable x_j, there is at least one test that provides the value of the variable. For all $r \in A$, the minimum cost d_r^* required for tests whose results allow separation of r from B, and the index set I_r^* of tests that achieve that result. Rational parameter $\lambda \geq 1.0$.

Output: A set A^* of optimized A records and a corresponding optimized test set $K^* = \{K_r^* \mid r \in A\}$.

Requires: Algorithm SOLVE MINSAT.

Procedure:

0. For $j = 1, 2, \ldots, n$, define $I_j = \{i \mid T_i \text{ determines the } x_j \text{ value}\}$. Define A^* as the empty set. Sort the set A in increasing order of minimum cost values d_r^*.
1. (Initialize A' and I^*.) Remove from A the record r_0 having smallest cost value. Initialize $A' = \{r_0\}$ and $I^* = I_{r_0}^*$. Declare each record $r \in A$ with $d_r^* \leq \lambda \cdot d_{r_0}^*$ to be *open*. All other records of A are *closed*.
2. (Termination test.) If A is empty or has no open records, then, for each $r \in A'$, do the following. Define $K_r^* = I^*$; derive from r an optimized record r^* by replacing any *True/False* value which is not obtainable by any test of K_r^*, by *Unavailable*; and place the record r^* into A^*.
   If A is empty, output A^* and $K^* = \{K_r^* \mid r \in A\}$, and stop.

If A has no open record, go to Step 1.

3. (Increase A'.) Let r be the open record of A with smallest cost. Add r to A'. Solve the MINSAT instance with the conditions

$$
\begin{aligned}
&\bigvee_{j\in(J_+^s\cup J_u^s)} x_j(neg) \vee \bigvee_{j\in(J_-^s\cup J_u^s)} x_j(pos), \quad \forall s \in B \\
&\neg x_j(neg), \quad \forall j \in (J_+^{r'} \cup J_a^{r'} \cup J_u^{r'}), \ \forall r' \in A' \\
&\neg x_j(pos), \quad \forall j \in (J_-^{r'} \cup J_a^{r'} \cup J_u^{r'}), \ \forall r' \in A' \\
&[x_j(pos) \vee x_j(neg)] \rightarrow \bigvee_{i\in I_j} choose(T_i), \quad j = 1, 2, \ldots, n
\end{aligned}
\tag{7.5.9}
$$

For $i = 1, 2, \ldots, m$, the cost of *True* for $choose(T_i)$ is c_i. All other costs are 0. If (7.5.9) is unsatisfiable or if the total cost of the solution exceeds $\lambda \cdot d^*_{r_0}$, remove r from A'; otherwise, remove r from A, and update the set I^* so that it now contains the indices of the tests T_i for which $choose(T_i) = True$. Go to Step 2.

Exclusion of Weak Nestedness

The following result is important for Subproblem 3 to come.

(7.5.10) Theorem. *No optimized record $r^* \in A^*$ is weakly nested in any record $s \in B$.*

Proof. The solution values for the $x_j(pos)$ and $x_j(neg)$ variables computed in Step 3 of Algorithm OPTIMIZED RECORDS (7.5.8) implicitly define a DNF formula, say D', that separates A' from B. By the conditions (7.5.9) for the MINSAT instance, D' only uses values of the records in A' that are determined by the tests indexed by I^*. Precisely those tests are used in Step 2 to define the values in the records of A^* corresponding to A'. Thus, D' separates the records of A^* corresponding to A' from B. By Theorem (7.3.14), no A^* record is weakly nested in any B record. □

We make use of A^* and K^* produced by Algorithm OPTIMIZED RECORDS (7.5.8) next.

Subproblem 3: Optimized Formulas

We obtain min/max DNF formulas that separate A^* from B with Algorithm MIN/MAX DNF FORMULAS (7.4.3) by specifying A^* and B as input instead of A and B. To indicate that the resulting DNF formulas D^{min} and D^{max} are based on optimized records, we relabel these DNF

formulas as D^{*min} and D^{*max}, respectively, and call them *optimized* DNF *formulas.*

Since each record $r^* \in A^*$ is obtained from the corresponding record $r \in A$ by replacing some *True/False* values by *Unavailable*, D^{*min} and D^{*max} separate A^* as well as A from B. The two formulas achieve that separation of A from B in an economical way. Specifically, for $r \in A$, the tests specified by the index set K_r^* produce the *True/False/Unavailable* values of the corresponding $r^* \in A^*$ with total cost not exceeding $\lambda \cdot d_r^*$, where d_r^* is the minimum cost for separating r from B.

The next algorithm summarizes the processes of Subproblems 1–3 and thus produces D^{*min} and D^{*max} from A and B, using the tests $T_1, T_2, \ldots, T_m$ with costs $c_1, c_2, \ldots, c_m$, respectively. The algorithm requires the parameter $\lambda \geq 1.0$ introduced in Subproblem 2.

Selection of Parameter λ

The appropriate value of λ is selected by repeated application of the algorithm, beginning with the value $\lambda = 1.0$. If, for that value of λ, D^{*min} and D^{*max} do not have an excessive number of clauses, then they are the desired optimized DNF formulas. Otherwise, a value $\lambda > 1.0$ is needed. In general, increasing (resp. decreasing) λ tends to reduce (resp. increase) the number of clauses of D^{*min} and D^{*max}, but also tends to increase (resp. reduce) the cost of tests specified by the sets K_r^*. Thus, the desired λ is a compromise that, on one hand, results in D^{*min} and D^{*max} with reasonable number of clauses and, on the other hand, leads to tests of K_r^* with costs reasonably close to the minimum d_r^*. The search for an appropriate λ can be speeded up using dichotomous search. The initial search interval is $[1.0, \lambda_{max}]$, where λ_{max} is large enough that any $\lambda > \lambda_{max}$ cannot possibly be an appropriate choice. In applications, we often have used $\lambda = 1.2$.

Algorithm for Optimized DNF Formulas

Here is the algorithm for Subproblem 3.

(7.5.11) Algorithm OPTIMIZED DNF FORMULAS. *Determines optimized* DNF *formulas for separating sets.*

Input: Nonempty sets A and B whose records involve variables $x_1, x_2, \ldots, x_n$. No A record is weakly nested in any B record. (If this condition does not hold, there is no DNF formula that separates A from B.) Tests $T_1, T_2, \ldots, T_m$ with costs $c_1, c_2, \ldots, c_m$, respectively. For each variable x_j, there is at least one test that provides the value of the variable. Rational parameter $\lambda \geq 1.0$.

Output: For each $r \in A$, the minimum separation cost d_r^*. A set A^* of optimized A records and a corresponding optimized test set $K^* = \{K_r^* \mid r \in A\}$. For each $r \in A$, the total cost of the tests specified by the index set K_r^* does not exceed $\lambda \cdot d_r^*$. Optimized DNF formulas D^{*min} and D^{*max} that separate A^* and A from B.

Requires: Algorithms MINIMUM SEPARATION COST (7.5.5), OPTIMIZED RECORDS (7.5.8), and MIN/MAX DNF FORMULAS (7.4.3).

Procedure:

1. (Minimum test costs.) For each $r \in A$, carry out Algorithm MINIMUM SEPARATION COST (7.5.5) to obtain the minimum test cost d_r^* and the related index set I_r^* of tests. Output the d_r^* values.
2. (Optimized record set.) Using Algorithm OPTIMIZED RECORDS (7.5.8), obtain the set A^* of optimized records and the corresponding optimized test set $K^* = \{K_r^* \mid r \in A\}$. Output A^* and K^*.
3. (Optimized DNF formulas.) Carry out Algorithm MIN/MAX DNF FORMULAS (7.4.3) with input sets A^* and B instead of A and B. Output the DNF formulas D^{min} and D^{max} produced by the algorithm as D^{*min} and D^{*max}, and stop.

Proof of Validity. Given the validity of the algorithms used in Steps 1–3, we only need to establish that, in Step 3, no $r^* \in A^*$ is weakly nested in any $s \in B$. This holds by Theorem (7.5.10). □

The constraint of (7.5.2) enforces that the *True/False/Unavailable* value of x_j in any record must be obtained by at least one test T_i with $i \in I_j$. There may be other requirements that one would like to impose. The next section shows how this can be done.

7.6 Additional Logic Constraints

Consider an additional requirement that must be satisfied when the separating DNF formula is constructed from the training data. If the requirement can be expressed by logic conditions using the variables $x_j(pos)$, $x_j(neg)$, $choose(T_i)$, and possibly additional variables, then we can define appropriate clauses and add them to the MINSAT formulations used in the various algorithms of the preceding sections.

For example, suppose that test T_1 can be performed only if test T_2 is not being done. Then we add the clause $choose(T_1) \rightarrow \neg\, choose(T_2)$ to the conditions of the relevant MINSAT instances.

Negative Consequences

When clauses are added in the described manner, the existence of a separating DNF formula may no longer be guaranteed by exclusion of weak

nestedness, and thus Theorems (7.3.14) and (7.5.10) may no longer hold. Moreover, it generally is no longer possible to determine the max DNF formula D^{max} by trivial fixing of variables, and Theorem (7.4.2) need not hold. Exercise (7.10.10) covers exceptions to these negative results.

The unfortunate consequences of added logic conditions are mitigated if we add a checking step.

Checking Step

The step tests whether each record of A can be separated from B while the additional conditions are enforced. If that check is passed, the algorithms of the preceding sections perform correctly, except that D^{max} and D^{*max} must be obtained via solution of MINSAT instances. On the other hand, if some A record cannot be separated from B, then a change of at least one of the sets or of at least one of the added logic conditions is needed. To test whether a record $r \in A$ can be separated from B under the added conditions, one decides satisfiability of the logic formula S composed of the added logic conditions and (7.5.1), which demands

$$\begin{aligned} &\bigvee_{j \in (J_+^s \cup J_u^s)} x_j(neg) \vee \bigvee_{j \in (J_-^s \cup J_u^s)} x_j(pos), \quad \forall s \in B \\ &\neg x_j(neg), \quad \forall j \in (J_+^r \cup J_a^r \cup J_u^r) \\ &\neg x_j(pos), \quad \forall j \in (J_-^r \cup J_a^r \cup J_u^r) \end{aligned} \tag{7.6.1}$$

In case optimized DNF formulas are to be computed, one also adds to S the condition of (7.5.2), which states

$$[x_j(pos) \vee x_j(neg)] \rightarrow \bigvee_{i \in I_j} choose(T_i), \quad j = 1, 2, \ldots, n \tag{7.6.2}$$

The record $r \in A$ can be separated from B if and only if S is satisfiable. In the case of unsatisfiability, the techniques of Section 5.4 help to isolate the cause of the unsatisfiability. The remedy may be removal of one or more records from A or B, or a change of the added logic conditions.

The DNF formulas obtained so far in this chapter separate A from B. In the next section, we reverse the roles of A and B and determine DNF formulas that separate B from A.

7.7 Reversing the Roles of Sets

It may seem redundant that we reverse the roles of A and B and compute DNF formulas that separate B from A. We justify that option once we have introduced notation for the new DNF formulas and related concepts.

Definitions

Define E^{min} and E^{max} to be min and max DNF formulas that separate B from A analogously to the way D^{min} and D^{max} separate A from B. For the case of optimization of test costs, define e_s^* to be the minimum cost for separating record $s \in B$ from A, and let L_s^* be the index set of the corresponding tests. We collect the index sets L_s^* in the optimized test set $L^* = \{L_s^* \mid s \in B\}$. Finally, let E^{*min} and E^{*max} be the optimized DNF formulas that separate B^* and B from A.

For future reference, we define two algorithms that compute these DNF formulas.

Algorithm for Four Min/Max DNF Formulas

The first algorithm determines the min and max DNF formulas.

(7.7.1) Algorithm FOUR MIN/MAX DNF FORMULAS. *Determines four min and max* DNF *formulas for separating sets.*

Input: Nonempty sets A and B whose records involve variables $x_1, x_2, \ldots, x_n$. No A (resp. B) record is weakly nested in any B (resp. A) record. (If the condition does not hold, A cannot be separated from B, or B cannot be separated from A.)

Output: DNF formulas D^{min} and D^{max} (resp. E^{min} and E^{max}) that evaluate to *True* for each A (resp. B) record and to *False* for each B (resp. A) record. Thus, D^{min} and D^{max} (resp. E^{min} and E^{max}) separate A from B (resp. B from A).

Requires: Algorithm MIN/MAX DNF FORMULAS (7.4.3).

Procedure:

1. Use Algorithm MIN/MAX DNF FORMULAS (7.4.3) twice to obtain the desired DNF formulas separating A from B and B from A. Output these DNF formulas, and stop.

Algorithm for Four Optimized DNF Formulas

The optimization case is handled by the following scheme.

(7.7.2) Algorithm FOUR OPTIMIZED DNF FORMULAS. *Determines four optimized* DNF *formulas for separating sets.*

Input: Nonempty sets A and B whose records involve variables $x_1, x_2, \ldots, x_n$. No A (resp. B) record is weakly nested in any B (resp. A) record. (If the condition does not hold, A cannot be separated from B, or B cannot

be separated from A.) Tests $T_1, T_2, \ldots, T_m$ with costs $c_1, c_2, \ldots, c_m$, respectively. For each variable x_j, there is at least one test that provides the value of the variable. Rational parameters $\lambda_A \geq 1.0$ and $\lambda_B \geq 1.0$.

Output: For each $r \in A$ (resp. $s \in B$), minimum separation cost d_r^* (resp. e_s^*). Set A^* (resp. B^*) of optimized A (resp. B) records and corresponding optimized test set $K^* = \{K_r^* \mid r \in A\}$ (resp. $L^* = \{L_s^* \mid s \in B\}$). For each $r \in A$ (resp. $s \in B$), the tests specified by the index set K_r^* (resp. L_s^*) produce the *True/False* values of the corresponding record $r^* \in A^*$ (resp. $s^* \in B^*$), with total test cost not exceeding $\lambda_A \cdot d_r^*$ (resp. $\lambda_B \cdot e_s^*$). Optimized DNF formulas D^{*min} and D^{*max} (resp. E^{*min} and E^{*max}) that separate A^* and A from B (resp. B^* and B from A).

Requires: Algorithm OPTIMIZED DNF FORMULAS (7.5.11).

Procedure:

1. Use Algorithm OPTIMIZED DNF FORMULAS (7.5.11) twice to obtain the desired optimized DNF formulas separating A from B and B from A, plus the related information about minimum costs and tests. Output these results, and stop.

Reasons for Role Reversal

One reason for computing DNF formulas that separate B from A is as follows. Suppose that instead of a DNF formula that evaluates to *True* on A and to *False* on B, we want CNF formulas with that feature. Now E^{min} and E^{max} are DNF formulas that evaluate to *True* on B and to *False* on A. By negating them, we get CNF formulas $\neg E^{min}$ and $\neg E^{max}$ with the desired property. Indeed, if we define the min/max property for CNF formulas analogously to the definition of the min/max property for DNF formulas in Section 7.4, then $\neg E^{min}$ and $\neg E^{max}$ are *min* and *max* CNF *formulas*. The evaluation of such CNF formulas follows the usual rules unless some of the x_j values take on the value *Unavailable*. Recall from Section 7.2 that x_j = *Unavailable* causes any learned DNF clause with a literal of x_j to evaluate to *False*. Correspondingly, x_j = *Unavailable* causes any CNF clause that is the negation of a learned DNF clause and that contains a literal of x_j, to evaluate to *True*. Due to this rule, care must be taken when CNF clauses derived from learned DNF clauses are inserted into CNF systems. Section 10.12 of Chapter 10 covers the process.

Another reason for computing DNF formulas that separate B from A is given in the next section, where values of DNF formulas are viewed as votes for membership in A or B.

7.8 Voting

Section 7.4 establishes that the DNF formulas D^{min} and D^{max} exhibit a bias when applied to unseen records of $\mathcal{A} \cup \mathcal{B}$. That is, D^{min} (resp. D^{max}) favors classification of unseen records into $\mathcal{A}$ (resp. $\mathcal{B}$). Analogously, E^{min} (resp. E^{max}) favors classification of unseen records into $\mathcal{B}$ (resp. $\mathcal{A}$). Similar arguments apply to the optimized versions D^{*min}, D^{*max}, E^{*min} and E^{*max}.

We exploit the bias of formulas to achieve various classification goals. The main idea is to select formulas with appropriate bias and to classify records using a consensus decision of the selected formulas.

Depending on the selection, the bias inherent in the formulas is canceled out or reinforced in the consensus decision. For example, if the four formulas D^{min}, D^{max}, E^{min} and E^{max} are selected, then the bias of D^{min} (resp. E^{min}) is the opposite of that of D^{max} (resp. E^{max}), and the consensus decision can be expected to be neutral. On the other hand, if D^{min} and E^{max} are selected, then the bias of D^{min} is reinforced by that of E^{max}, and the consensus decision has a tendency to favor classification of unseen records into $\mathcal{A}$.

The consensus decision is based on votes by the selected formulas. We define the vote concept next.

Vote

If the value of a formula for a given record indicates membership in $\mathcal{A}$ (resp. $\mathcal{B}$), then the *vote* of the formula is 1 (resp. -1). Thus, *True* (resp. *False*) for D^{min} or D^{max} and *False* (resp. *True*) for E^{min} or E^{max} produce a vote of 1 (resp. -1).

What is the vote for *Undecided*? Recall from Section 7.3 that the value *Undecided* may possibly be determined to be *True* by a satisfiability test. In many applications, it is difficult to incorporate that satisfiability test into the computing process. For example, if the votes are computed as part of an optimization process, then it would be difficult to include the satisfiability problem for *Undecided* cases. Instead, we ignore here this subtle aspect of *Undecided* and simply associate with *Undecided* a vote equal to 0.

Vote Total

Let the *vote total* be the integer sum of the $\{0, \pm 1\}$ votes produced by the formulas. If the number of formulas is k, then the vote total may range from $-k$ to k. A positive (resp. negative) vote total may be interpreted as evidence that the record is in $\mathcal{A}$ (resp. $\mathcal{B}$) with a likelihood that increases

with the absolute value of the vote total. A vote total of 0 means that we do not have a clue whether the record is in $\mathcal{A}$ or $\mathcal{B}$.

In a bold move, we could associate numerical likelihood levels with the vote total. For example, suppose the four formulas D^{min}, D^{max}, E^{min}, and E^{max} have been selected. We could declare that the likelihood of a record being in $\mathcal{A}$ is 25% (resp. 50%, 75%, 100%) if the vote total v is equal to 1 (resp. 2, 3, 4). For negative values of v, analogous claims could be made for the likelihood of membership in $\mathcal{B}$. But such an interpretation is not based on any evidence and thus should not be done. The next chapter shows that likelihood evidence can be obtained from the training data when a more elaborate voting system is employed.

7.9 Further Reading

For an overview of machine learning techniques, see introductory artificial intelligence texts such as Nilsson (1998), Hopgood (2001), Luger (2002), Negnevitsky (2002), or Russell and Norvig (2003). However, these texts contain little or no information about support vector machines. An introductory text for these methods is Cristianini and Shawe-Taylor (2000). Various data mining methods are described in Hand, Mannila, and Smyth (2001). Research on learning logic is rapidly advancing. A good starting point for further reading is Triantaphyllou and Felici (2004).

For transformations of rational data to logic data, see the machine learning text by Mitchell (1997). In applications, we have used the cutpoint method of Bartnikowski, Granberry, Mugan, and Truemper (2004), where the markers of the transformations correspond to abrupt pattern changes.

7.10 Exercises

The calculations required by some of the exercises may be accomplished with the software of Exercise (7.10.12).

(7.10.1) Consider the following training sets.

Set A	
$(e = Absent, f = True, g = True, h = False)$	(patient 1)
$(e = False, f = True, g = True, h = Absent)$	(patient 2)
Set B	
$(e = True, f = True, g = False, h = Absent)$	(patient 3)
$(e = True, f = False, g = True, h = Unavailable)$	(patient 4)

(a) Formulate the CNF system (7.3.8).
(b) Determine a separating DNF formula using Algorithm SEPARATING DNF FORMULA (7.3.15).

(7.10.2) Consider the following training sets.

Set A
$(x = True, y = True, z = False)$
$(x = True, y = Unavailable, z = True)$
$(x = False, y = False, z = Absent)$
Set B
$(x = False, y = True, z = Unavailable)$
$(x = True, y = False, z = Absent)$

(a) Formulate the CNF system (7.3.8).
(b) Determine a separating DNF formula using Algorithm SEPARATING DNF FORMULA (7.3.15).

(7.10.3) Suppose for the separating DNF formula $D = (x \wedge y) \vee (\neg x \wedge z) \vee (\neg y \wedge \neg z)$ we have a record with $x = True$ and $y = z = Absent$.
(a) Demonstrate that D has the value *Undecided*.
(b) Decide if *True/False* values exist for y and z so that D evaluates to *False*, using the satisfiability test of Section 7.3.

(7.10.4) Let training sets A and B have the following records.

Set A
$(x_1 = False, x_2 = False, x_3 = True, x_4 = False, x_5 = Absent)$
$(x_1 = True, x_2 = False, x_3 = Absent, x_4 = False, x_5 = False)$
$(x_1 = False, x_2 = Unavailable, x_3 = True, x_4 = False, x_5 = False)$
Set B
$(x_1 = Absent, x_2 = True, x_3 = True, x_4 = True, x_5 = True)$
$(x_1 = False, x_2 = True, x_3 = True, x_4 = True, x_5 = Absent)$
$(x_1 = True, x_2 = True, x_3 = Absent, x_4 = True, x_5 = Absent)$

Use Algorithm MIN/MAX DNF FORMULAS (7.4.3) to determine D^{min} and D^{max} that separate A from B.

(7.10.5) The MINIMUM COVER problem is specified by a set P and subsets $Q_1, Q_2, \ldots, Q_n$ of P, for some $n \geq 1$. One must find a minimum number N of the subsets whose union is equal to P. The version of MINIMUM COVER where one must determine if the minimum N is below a given constant K is $\mathcal{NP}$-complete.

Show that the problem of finding a min DNF clause D_k^{min} essentially is the MINIMUM COVER problem. (*Hint*: The set P is defined by the records of B.)

(7.10.6) Define A and B of Exercise (7.10.4) to be the given training sets. For the records of these sets, let three tests T_1, T_2, and T_3 determine variable values as follows. The test T_1 determines the values of x_1, x_2, and x_3, the test T_2 finds the values of x_2, x_3, and x_4, and the test T_3 provides the values of x_3, x_4, and x_5. The costs of T_1, T_2, and T_3 are $c_1 = 10$, $c_2 = 7$, and $c_3 = 21$, respectively.

(a) Use Algorithm MINIMUM SEPARATION COST (7.5.5) to find, for the records $r \in A$, the minimum costs d_r^* and the related index sets I_r^* of tests.

(b) Use Algorithm OPTIMIZED RECORDS (7.5.8) to determine the set A^* of optimized records and the corresponding optimized test set K^*.

(c) Use Algorithm MIN/MAX DNF FORMULAS (7.4.3) with A^* and B as input to obtain optimized DNF formulas D^{*min} and D^{*max}.

(7.10.7) Reverse the roles of A and B in Exercise (7.10.4) and thus obtain the min/max DNF formulas E^{min} and E^{max}.

(7.10.8) Reverse the roles of A and B in Exercise (7.10.6) and thus obtain minimum costs e_s^*, set B^* of optimized records, optimized test set L^*, and optimized DNF formulas E^{*min} and E^{*max}.

(7.10.9) Suppose test T_1 can only be performed if test T_2 is not done. For A and B of (7.10.4), check if A can be separated from B under that condition.

(7.10.10) Suppose additional logic conditions concerning the selection of tests are such that (a) or (b) below holds. For each of the two cases, prove that A can be separated from B under the added logic conditions if and only if no A record is weakly nested in any B record.

(a) The added conditions permit the assignment $choose(T_i) = True$, $i = 1, 2, \ldots, m$.

(b) The added conditions permit an assignment of *True/False* values to the $choose(T_i)$ variables such that the tests T_i for which $choose(T_i) = True$ collectively determine all *True/False* values for $x_1, x_2, \ldots, x_n$.

(7.10.11) Using D^{min} and D^{max} of Exercise (7.10.4) and E^{min} and E^{max} of Exercise (7.10.7), compute vote totals for the following records and classify them accordingly as being in $\mathcal{A}$ or $\mathcal{B}$.

$(x_1 = True, x_2 = False, x_3 = True, x_4 = False, x_5 = Absent)$
$(x_1 = True, x_2 = True, x_3 = False, x_4 = True, x_5 = True)$
$(x_1 = False, x_2 = True, x_3 = True, x_4 = False, x_5 = True)$
$(x_1 = False, x_2 = Absent, x_3 = True, x_4 = False, x_5 = False)$

(7.10.12) (Optional programming project) Some of the exercises defy manual solution. For these cases, the reader should obtain commercially

or publicly available software. Any search engine on the Internet will point to such software. Alternately, the reader may opt to write programs that implement the algorithms of (a)–(g) below, and then use that software to solve the exercises. The task is easy if the reader has already created a program MINSAT as described in Exercise (2.9.13).

(a) Algorithm SEPARATING DNF FORMULA (7.3.15).
(b) Algorithm MIN/MAX DNF FORMULAS (7.4.3).
(c) Algorithm MINIMUM SEPARATION COST (7.5.5).
(d) Algorithm OPTIMIZED RECORDS (7.5.8).
(e) Algorithm OPTIMIZED DNF FORMULAS (7.5.11).
(f) Algorithm FOUR MIN/MAX DNF FORMULAS (7.7.1).
(g) Algorithm FOUR OPTIMIZED DNF FORMULAS (7.7.2).

The reader may also want to create or obtain via the Internet software for transforming rational data or set data into logic data. Several methods are sketched in Section 7.2. References are included in Section 7.9.

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 8

Accuracy of Learned Formulas

8.1 Overview

This chapter expands upon the learning of logic formulas discussed in Chapter 7. Specifically, we extract logic formulas in particular ways from the given data, view the *True/False* values produced by them as votes, combine the votes to vote totals, and analyze the behavior of the vote totals to obtain certain probability distributions. We use these distributions to evaluate and control the accuracy with which the formulas classify unseen data.

We review key concepts of Chapter 7. Given records of training sets A and B that were randomly selected from two populations $\mathcal{A}$ and $\mathcal{B}$, respectively, one may compute several DNF formulas that evaluate to one *True/False* value on the records of A and to the opposite value on the records of B. In particular, the DNF formulas D^{min} and D^{max} (resp. E^{min} and E^{max}) and the optimized versions D^{*min} and D^{*max} (resp. E^{*min} and E^{*max}) evaluate to *True* (resp. *False*) on A and to *False* (resp. *True*) on B.

For unseen records, that is, for records of $(\mathcal{A}-A)\cup(\mathcal{B}-B)$, each of the cited formulas may evaluate to *True*, *False*, or *Undecided*. These values are interpreted in the obvious way. Thus, *True* (resp. *False*) for D^{min}, D^{max}, D^{*min}, or D^{*max} and *False* (resp. *True*) for E^{min}, E^{max} E^{*min}, or E^{*max} classify an unseen record as being in $\mathcal{A}$ (resp. $\mathcal{B}$). The value *Undecided* means that the formula does not classify the record.

Each *True*, *False*, and *Undecided* value may also be viewed as an integer *vote*. If the value of the formula classifies the record as being in

$\mathcal{A}$ (resp. $\mathcal{B}$), then the vote is defined to be 1 (resp. -1). If the value is *Undecided*, the vote is 0.

If several formulas produce votes, then the sum of the votes is the *vote total*. Chapter 7 leaves open the interpretation of the vote total beyond the qualitative observation that a large positive (resp. negative) vote total makes it likely that the unseen record is in $\mathcal{A}$ (resp. $\mathcal{B}$). In this chapter, we refine the learning of logic formulas and the voting process so that, for unseen records, we not only derive the vote total, but also obtain an assessment of the likelihood that a positive (resp. negative) vote total correctly indicates membership in $\mathcal{A}$ (resp. $\mathcal{B}$).

To be sure, there is a simple way to make such an assessment. First, we split the training set A (resp. B) into two sets A_1 and $\overline{A}_1$ (resp. B_1 and $\overline{B}_1$). Second, we obtain logic formulas using A_1 and B_1. Finally, we test the accuracy of the vote totals on $\overline{A}_1$ and $\overline{B}_1$ using standard statistical techniques.

This approach works fine when one has large training sets A and B. Often, this is not the case, and one would like to use all records of A and B for the computation of the logic formulas. This chapter shows how that requirement can be met and how, simultaneously, the accuracy can be determined with which the vote total of the logic formulas classifies unseen records.

The basic idea is to learn formulas for a number of subsets of A and B, and to evaluate the accuracy of the formulas on the complements of these subsets. That approach uses all training data for learning formulas and simultaneously allows a reasonably precise assessment of the likelihood that a positive (resp. negative) vote total correctly indicates membership in $\mathcal{A}$ (resp. $\mathcal{B}$).

Section 8.2 describes the subsets of the training sets that are used for the learning of formulas and the evaluation of the accuracy of formulas.

Section 8.3 covers the computation of the formulas. This step uses Algorithm MIN/MAX DNF FORMULAS (7.4.3) or OPTIMIZED DNF FORMULAS (7.5.11) of Chapter 7 and derives 40 min/max or optimized formulas, respectively. These 40 formulas, or possibly a subset selected from them, produce a vote total for each record.

Sections 8.4 and 8.5 reuse the subsets of the training data to evaluate the accuracy with which the vote total classifies additional records. For the evaluation, it is assumed that a record is classified to be in one of the two populations if the vote total exceeds a specified threshold, and to be in the other population otherwise. The main result consists of two conditional distributions of the vote total, where the condition is membership of records in a specified population.

Section 8.6 assumes that, besides the two conditional distributions, one has prior probabilities of membership in each of the two populations. With that information, the unconditional probability of erroneous classifi-

cation, as a function of the threshold, is computed. Also treated is the case where each classification error results in some cost, and where one desires a threshold that minimizes the expected misclassification cost.

Section 8.7 extends the results to the situation where the records of more than two populations are to be separated.

Section 8.8 cites references for further reading.

Section 8.9 contains exercises.

8.2 Subsets of Training Data

We begin by defining various subsets of the training data.

Partitions

Assume that each of the training sets A and B has at least 10 records. We partition A into 10 subsets $a_1, a_2, \ldots, a_{10}$ of approximately equal size, as follows. If A has k records, then let $l = \lfloor k/10 \rfloor$. For $i = 1, 2, \ldots, 10$, place l records of A into a_i. This leaves $k' = k - 10l$ records of A unassigned. If $k' > 0$, we distribute the leftover records among the subsets $a_1, a_2, \ldots, a_{k'}$. Thus, each of these subsets receives exactly one of the leftover records. We view $a_1, a_2, \ldots, a_{10}$ as a circular list where the successor of a_{10} is a_1.

Overlapping Subsets

From the list $a_1, a_2, \ldots, a_{10}$, we derive 10 sets $A_1, A_2, \ldots, A_{10}$ where A_i is the union of a_i and of the next 5 a_j. Thus, A_1 is the union of $a_1, a_2, \ldots, a_6$, and A_{10} is the union of $a_{10}, a_1, \ldots. a_5$. We define $\overline{A}_i$ to be the complement of A_i in A, so $\overline{A}_i = A - A_i$.

Analogous definitions apply to B. Thus, we partition B into 10 subsets $b_1, b_2, \ldots, b_{10}$ of essentially equal size and from these define $B_1, B_2, \ldots, B_{10}$ and their complements $\overline{B}_1, \overline{B}_2, \ldots, \overline{B}_{10}$.

The sets A_i and B_i are used later for training, while $\overline{A}_i$ and $\overline{B}_i$ are used for statistical testing.

The reader may wonder why we select 10 subsets $a_1, a_2, \ldots, a_{10}$ of A (resp. $b_1, b_2, \ldots, b_{10}$ of B), and why each A_i (resp. B_i) consists of 6 of these subsets. Here are the reasons.

Suppose that, for some $k > 1$, k formulas are learned from each of the 10 subset pairs A_i and B_i. Then the resulting $10k$ formulas produce $10k$ votes that are combined to the vote total. In principle, the larger the number of subsets, the higher the classification accuracy of the vote total, and the higher the effort to compute the formulas. According to

experiments, 10 subsets give good classification accuracy while requiring reasonable computing effort to determine the formulas.

The selection of 6 subsets of $a_1, a_2, \ldots, a_{10}$ (resp. $b_1, b_2, \ldots, b_{10}$) for each A_i (resp. B_i) is based on the following two requirements. We want the vote total to be positive (resp. negative) for the records of A (resp. B). Subject to that condition, we want to estimate the accuracy of the vote total on unseen records as precisely as possible. It turns out that the first requirement is met if each A_i (resp. B_i) contains more than half of the subsets $a_1, a_2, \ldots, a_{10}$ (resp. $b_1, b_2, \ldots, b_{10}$). The second requirement is fulfilled if each complement $\overline{A}_i$ (resp. $\overline{B}_i$) contains as many of the subsets $a_1, a_2, \ldots, a_{10}$ (resp. $b_1, b_2, \ldots, b_{10}$) as possible. The selection of 6 subsets for each A_i and B_i satisfies both requirements.

Case of Optimized Records

When optimized formulas are to produce votes, we also define sets A_1^*, $A_2^*, \ldots, A_{10}^*$ and $B_1^*, B_2^*, \ldots, B_{10}^*$ in two steps. First, we carry out Algorithms MINIMUM SEPARATION COST (7.5.5) and OPTIMIZED RECORDS (7.5.8) to obtain optimized record sets A^* and B^*. Second, for $i = 1, 2, \ldots, 10$, we define A_i^* (resp. B_i^*) to consist of the optimized records of A^* (resp. B^*) that correspond to the records of A_i (resp. B_i).

Expansion of Sets

In the discussion so far, we have assumed that A and B have at least 10 training records each. Suppose that A has $k < 10$ records. We adjust the set so that the revised version has at least 10 records, using a scheme that preserves the relative frequency with which data entries of records occur. Let l be the smallest integer so that $l \cdot k \geq 10$. Thus, $l = \lceil 10/k \rceil$. We replace each record of A by l copies, which we view as distinct. The revised set A has at least 10 records, and the relative frequencies of the data entries of records is preserved. The same process applies to a set B with less than 10 records.

As an example, suppose A has 7 records. We replace each record of A by $l = \lceil 10/7 \rceil = 2$ copies. The expanded A has $2 \cdot 7 = 14$ records.

Algorithm for Expansion of Set

We summarize the expansion process.

(8.2.1) Algorithm EXPANSION OF SET. *Enlarges a given set by replacing each record by several copies.*

Input: Nonempty set A with less than 10 records.

Output: An enlarged set A with at least 10 records. The expansion preserves relative frequency of data entries.

Requires: No additional algorithm needed.

Procedure:

1. Define k to be the number of records of A, and let $l = \lceil 10/k \rceil$. Replace each record of A by l copies.

Due to the expansion process, the algorithms to follow need only assume that the sets A and B are nonempty. However, we shall not do so since that assumption leads to undesirable notational complexity where we must distinguish between an original set A or B and its expanded version. To eliminate that complication, we enforce that A and B have at least 10 records each and view handling of cases with less than 10 records as preprocessing that need not be accounted for in the description of the algorithms.

Algorithm for Training Subsets

We summarize the construction of the training subsets.

(8.2.2) Algorithm TRAINING SUBSETS. *Determines training subsets from given training data. Optionally, determines optimized training subsets as well.*

Input: Sets A and B whose records involve variables $x_1, x_2, \ldots, x_n$. The sets have at least 10 records each. No A (resp. B) record is weakly nested in any B (resp. A) record. (The condition assures that the subsets derived from A and B can be separated by DNF formulas.) If optimized training subsets are to be computed, optimized sets A^* and B^*.

Output: Subsets $A_1, A_2, \ldots, A_{10}$ of A and subsets $B_1, B_2, \ldots, B_{10}$ of B. Optionally, the corresponding optimized record subsets $A_1^*, A_2^*, \ldots, A_{10}^*$ and $B_1^*, B_2^*, \ldots, B_{10}^*$.

Requires: No additional algorithm needed.

Procedure:

1. Let A have k records, and define $l = \lfloor k/10 \rfloor$. Partition A into subsets $a_1, a_2, \ldots, a_{10}$ as follows. Place l records of A into each subset a_i. If $k' = k - 10l$ is positive, distribute the as yet unassigned k' records among the subsets $a_1, a_2, \ldots, a_{k'}$. Thus, each of these subsets receives exactly one record. View $a_1, a_2, \ldots, a_{10}$ as a circular list where the successor of a_{10} is a_1. For $i = 1, 2, \ldots, 10$, output A_i as the union of a_i and the next 5 a_j.
   Carry out the analogous process for B, and output the resulting subsets $B_1, B_2, \ldots, B_{10}$.

2. If optimized sets A^* and B^* are provided as part of the input, define, for $i = 1, 2, \ldots, 10$, A_i^* (resp. B_i^*) to be the subset of A^* (resp. B^*) whose records correspond to those of A_i (resp. B_i). Output these subsets of optimized records, and stop.

We examine the property of weak nestedness for the subsets derived by Algorithm TRAINING SUBSETS (8.2.2).

Weak Nestedness

Section 7.3 defines a record r to be weakly nested in a record s if one can replace *Absent* values of s by *True/False* values such that all *True/False* values of r occur in the modified s. Theorem (7.3.14) states that there is a DNF formula D that separates A from B if and only if no record $r \in A$ is weakly nested in any record $s \in B$. In the next section, we compute, for $i = 1, 2, \ldots, 10$, formulas that separate A_i or A_i^* from B_i, and B_i or B_i^* from A_i. According to Theorem (7.3.14), this requires that no record of A_i or A_i^* (resp. B_i or B_i^*) is weakly nested in any record of B_i (resp. A_i). The next theorem assures that this is the case if weak nestedness is ruled out for A and B.

(8.2.3) Theorem. *Suppose that no record of A (resp. B) is weakly nested in any record of B (resp. A). Then, for $i = 1, 2, \ldots, 10$, no record of A_i or A_i^* (resp. B_i or B_i^*) is weakly nested in any record of B_i (resp. A_i).*

Proof. Each A_i and B_i is a subset of A and B, respectively, and absence of weak nestedness holds for the two subsets by definition of the property. By Theorem (7.5.10), no optimized record $r^* \in A^*$ is weakly nested in any record $s \in B$. Using once more the subset argument, we conclude that no record of A_i^* is weakly nested in any record of B_i. By symmetry, no record of B_i^* is weakly nested in any record of A_i. □

We are ready to compute DNF formulas that separate A_i or A_i^* from B_i, or B_i or B_i^* from A_i.

8.3 Logic Formulas for Subsets

For $i = 1, 2, \ldots, 10$, we learn logic formulas from the training sets A_i and B_i by applying Algorithm MIN/MAX DNF FORMULAS (7.4.3) twice to separate A_i from B_i and B_i from A_i, getting 4 logic formulas D_i^{min}, D_i^{max}, E_i^{min}, and E_i^{max}. Thus, a total of $10 \cdot 4 = 40$ formulas are computed.

In the optimization case, we use the same algorithm twice to separate A_i^* from B_i and B_i^* from A_i, getting 4 optimized formulas D_i^{*min}, D_i^{*max}, E_i^{*min}, and E_i^{*max}. Here, too, a total of $10 \cdot 4 = 40$ formulas are computed.

Computation of Formulas

We summarize the computations in the next two algorithms. To simplify application of the schemes, they include the derivation of the subsets A_i and B_i of A and B and, if applicable, computation of A^* and B^* and derivation of the subsets A_i^* and B_i^*.

Algorithm for Forty Min/Max DNF Formulas

Here is the scheme for the min/max case.

(8.3.1) Algorithm FORTY MIN/MAX DNF FORMULAS. *Determines* 40 *min and max* DNF *formulas for separating subsets of given sets.*

Input: Sets A and B whose records involve variables $x_1, x_2, \ldots, x_n$. The sets have at least 10 records each. No A (resp. B) record is weakly nested in any B (resp. A) record. (The condition assures that the subsets derived from A and B can be separated by DNF formulas.)

Output: For $i = 1, 2, \ldots, 10$, subsets A_i and B_i, and DNF formulas D_i^{min} and D_i^{max} (resp. E_i^{min} and E_i^{max}) that evaluate to *True* for each A_i (resp. B_i) record and to *False* for each B_i (resp. A_i) record. Thus, D_i^{min} and D_i^{max} (resp. E_i^{min} and E_i^{max}) separate A_i from B_i (resp. B_i from A_i).

Requires: Algorithms TRAINING SUBSETS (8.2.2) and MIN/MAX DNF FORMULAS (7.4.3).

Procedure:

1. Use Algorithm TRAINING SUBSETS (8.2.2) to determine, for $i = 1, 2, \ldots, 10$, subsets A_i and B_i. Output these sets.
2. For $i = 1, 2, \ldots, 10$, use Algorithm MIN/MAX DNF FORMULAS (7.4.3) twice to obtain the desired 4 DNF formulas separating A_i from B_i and B_i from A_i. Output the 40 DNF formulas, and stop.

Algorithm for Forty Optimized DNF Formulas

The optimized case is handled analogously.

(8.3.2) Algorithm FORTY OPTIMIZED DNF FORMULAS. *Determines* 40 *optimized* DNF *formulas for separating subsets of given sets.*

Input: Sets A and B whose records involve variables $x_1, x_2, \ldots, x_n$. The sets have at least 10 records each. No A (resp. B) record is weakly nested in any B (resp. A) record. (The condition assures that the subsets derived from A and B can be separated by DNF formulas.) Tests $T_1, T_2, \ldots, T_m$

with costs $c_1, c_2, \ldots, c_m$, respectively. For each variable x_j, there is at least one test that provides the value of the variable. Rational parameters $\lambda_A \geq 1.0$ and $\lambda_B \geq 1.0$.

Output: For each $r \in A$ (resp. $s \in B$), the minimum separation cost d_r^* (resp. e_s^*). A set A^* (resp. B^*) of optimized A (resp. B) records and a corresponding optimized test set $K^* = \{K_r^* \mid r \in A\}$ (resp. $L^* = \{L_s^* \mid s \in B\}$). For each $r \in A$ (resp. $s \in B$), the tests specified by the index set K_r^* (resp. L_s^*) produce the *True/False* values of the corresponding record $r^* \in A^*$ (resp. $s^* \in B^*$), with total test cost not exceeding $\lambda_A \cdot d_r^*$ (resp. $\lambda_B \cdot e_s^*$). For $i = 1, 2, \ldots, 10$, subsets A_i and B_i, optimized versions A_i^* and B_i^*, and optimized DNF formulas D_i^{*min} and D_i^{*max} (resp. E_i^{*min} E_i^{*max}) that separate A_i^* and A_i from B_i (resp. B_i^* and B_i from A_i).

Requires: Algorithms OPTIMIZED DNF FORMULAS (7.5.11), TRAINING SUBSETS (8.2.2), and MIN/MAX DNF FORMULAS (7.4.3).

Procedure:

1. Carry out Steps 1 and 2 of Algorithm OPTIMIZED DNF FORMULAS (7.5.11) twice to obtain A^* and B^*, plus the related information about minimum costs and tests. Output these results.
2. Use Algorithm TRAINING SUBSETS (8.2.2) to determine, for $i = 1, 2, \ldots, 10$, subsets A_i, B_i, A_i^*, and B_i^*. Output these sets.
3. For $i = 1, 2, \ldots, 10$, use Algorithm MIN/MAX DNF FORMULAS (7.4.3) twice to obtain the desired 4 DNF formulas separating A_i^* from B_i, and B_i^* from A_i. Output the 40 DNF formulas, and stop.

Proof of Validity. Step 2 of Algorithm FORTY MIN/MAX DNF FORMULAS (8.3.1) and Step 3 of Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2) invoke Algorithm MIN/MAX DNF FORMULAS (7.4.3), which requires absence of weak nestedness for the input sets. Such absence is assured in both cases by Theorem (8.2.3). □

In the next several sections, we estimate the accuracy with which the DNF formulas produced by the two algorithms classify unseen records.

8.4 Classification Errors

In applications, we may use all DNF formulas produced by Algorithm FORTY MIN/MAX DNF FORMULAS (8.3.1) or by Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2), or we may use just a certain subset. The discussion below allows for the selection of such subsets.

Selection of Formulas

For the discussion of classification accuracy, suppose we have chosen some k, $1 \leq k \leq 4$. Then, for $i = 1, 2, \ldots, 10$, we select k DNF formulas from

the set $\{D_i^{min}, D_i^{max}, E_i^{min}, E_i^{max}\}$ in the min/max case and from the set $\{D_i^{*min}, D_i^{*max}, E_i^{*min}, E_i^{*max}\}$ in the optimization case. Thus, we have a total of $10k$ selected formulas.

Application of Formulas

In the min/max (resp. optimization) case, we apply the $10k$ formulas to records of $\mathcal{A}$ and $\mathcal{B}$ where all record entries are given (resp. determined by sequential tests). Specifically, in the optimization case, one chooses a test and uses it to obtain some entries of the given record. If the entries now on hand allow classification into $\mathcal{A}$ or $\mathcal{B}$ by each of the $10k$ optimized formulas, we are done. Otherwise, we choose another test, get additional entries, and so on. The reader may want to verify that this process is consistent with that of Subproblem 4 in Section 7.5 of Chapter 7. Details of the test selection are described in Section 10.12 of Chapter 10, where the selection is carried out by a certain question-and-answer process.

For the determination of the accuracy of formulas, it does not matter whether record entries are given or obtained by tests. Thus, for both min/max and optimized formulas, we ascertain their accuracy by evaluating their performance on unseen records of $\mathcal{A}$ and $\mathcal{B}$. By "unseen" we mean that the record is not used in training directly or after reduction to an optimized record.

We assume that the selected $10k$ formulas are applied to randomly drawn unseen records of $\mathcal{A}$ and $\mathcal{B}$. For each such record, the sum of the $10k$ votes may be viewed as the value of a random variable Z. The possible values of Z range from $-10k$ to $10k$.

Threshold

Let z be an integer *threshold value* for the classification of records. We declare an unseen record r to be in $\mathcal{A}$ if the vote total Z satisfies $Z > z$, and to be in $\mathcal{B}$ otherwise. For the moment, the specific value of z is not important. However, in Section 8.6, we choose z so that certain misclassification costs are minimized.

Conditional Probabilities

Assume that record r is in $\mathcal{A}$. Then r is misclassified by the rule if and only if $Z \leq z$. The probability of that event, $P_{err|\mathcal{A}}(z)$, is given by the conditional distribution $\mathcal{F}_{\mathcal{A}}(z) = \text{Prob}[Z \leq z \mid \text{unseen } \mathcal{A} \text{ record}]$. That is, $P_{err|\mathcal{A}}(z) = \mathcal{F}_{\mathcal{A}}(z)$.

Using the power function $\mathcal{G}_{\mathcal{A}}(z) = 1 - \mathcal{F}_{\mathcal{A}}(z)$, the probability of correct classification is $P_{cor|\mathcal{A}}(z) = \mathcal{G}_{\mathcal{A}}(z)$.

Analogous equations hold for unseen records of $\mathcal{B}$. Let $\mathcal{F}_{\mathcal{B}}(z) = \text{Prob}[Z \leq z \mid \text{unseen } \mathcal{B} \text{ record}]$ and $\mathcal{G}_{\mathcal{B}}(z) = 1 - \mathcal{F}_{\mathcal{B}}(z)$. The probability of misclassification is $P_{err|\mathcal{B}}(z) = \mathcal{G}_{\mathcal{B}}(z)$, while the probability of correct classification is $P_{cor|\mathcal{B}}(z) = \mathcal{F}_{\mathcal{B}}(z)$.

We summarize these equations.

$$
\begin{aligned}
\mathcal{F}_{\mathcal{Q}}(z) &= \text{Prob}[Z \leq z \mid \text{unseen } \mathcal{Q} \text{ record}], \quad \mathcal{Q} = \mathcal{A},\ \mathcal{B} \\
\mathcal{G}_{\mathcal{Q}}(z) &= 1 - \mathcal{F}_{\mathcal{Q}}(z), \quad \mathcal{Q} = \mathcal{A},\ \mathcal{B} \\
P_{err|\mathcal{A}}(z) &= \mathcal{F}_{\mathcal{A}}(z) \\
P_{cor|\mathcal{A}}(z) &= \mathcal{G}_{\mathcal{A}}(z) \\
P_{err|\mathcal{B}}(z) &= \mathcal{G}_{\mathcal{B}}(z) \\
P_{cor|\mathcal{B}}(z) &= \mathcal{F}_{\mathcal{B}}(z)
\end{aligned}
\tag{8.4.1}
$$

In the next section, we estimate the conditional distributions $\mathcal{F}_{\mathcal{A}}(z)$ and $\mathcal{F}_{\mathcal{B}}(z)$.

8.5 Vote Distributions

We estimate distributions of the vote total.

Sets of Unseen Records

Recall that, for $i = 1, 2, \ldots, 10$, Algorithm FORTY MIN/MAX FORMULAS (8.3.1) produces the formulas D_i^{min}, D_i^{max}, E_i^{min}, and E_i^{max} from A_i and B_i. Thus, for D_i^{min}, D_i^{max}, E_i^{min}, and E_i^{max}, the records of the sets $\overline{A}_i = A - A_i$ and $\overline{B}_i = B - B_i$ are unseen.

Similarly, Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2) derives D_i^{*min} and D_i^{*max} from A_i^* and B_i, and E_i^{*min} and E_i^{*max} from A_i and B_i^*. Thus, for D_i^{*min}, D_i^{*max}, E_i^{*min}, and E_i^{*max}, the records of the sets $\overline{A}_i = A - A_i$ and $\overline{B}_i = B - B_i$ are unseen.

We conclude that, for any one of the min/max or optimized formulas with index i, and thus for any k formulas selected from them, the records of $\overline{A}_i$ and $\overline{B}_i$ are unseen and therefore may be used to estimate classification accuracy on unseen records in general.

Vote Total for Unseen Record

Recall that the subsets $a_1, a_2, \ldots, a_{10}$ of A form a partition of A and that each A_i is the union of a_i and the next 5 subsets.

We arbitrarily index the $10k$ selected formulas by $j = 1, 2, \ldots, 10k$. Each formula j has been computed from two of the sets A_i, B_i, A_i^*, B_i^*. Let us denote the index i of those two sets by $i(j)$.

For the discussion to follow, let r be any record of A. Define $J(r)$ to be the set of indices j such that record r is not in $A_{i(j)}$. Thus, record r is unseen for the formulas $j \in J(r)$.

What is the cardinality of $J(r)$? The record r is in some subset a_l of A. Let $I(l)$ be the set of indices of the four subsets a_i that immediately follow a_l in the circular list $a_1, a_2, \ldots, a_{10}$. Since each A_i is the union of a_i and the next five subsets, the four sets A_i, $i \in I(l)$, are precisely the A_i that do not contain r. Accordingly,

$$J(r) = \{j \mid i(j) \in I(l)\} \tag{8.5.1}$$

and

$$|J(r)| = 4k \tag{8.5.2}$$

Denote by v_{rj} the vote of formula j for record r. Let v_r be the vote total of the v_{rj} for which $j \in J(r)$. Thus,

$$v_r = \sum_{j \in J(r)} v_{rj} \tag{8.5.3}$$

$|J(r)| = 4k$ implies that

$$-4k \leq v_r \leq 4k \tag{8.5.4}$$

There is a simpler way to compute v_r. The equation $|J(r)| = 4k$ of (8.5.2) implies that r is unseen for $4k$ formulas. Thus, $10k - 4k = 6k$ formulas have been computed using r or an optimized version of that record. Regardless of the case, these $6k$ formulas classify r correctly, produce a vote equal to 1 each, and therefore contribute $\delta_r = 6k$ to the vote total of the $10k$ formulas. When we remove that contribution from the vote total, we get v_r. Thus,

$$v_r = \sum_{j=1}^{10k} v_{rj} - \delta_r \tag{8.5.5}$$

Suppose we scale up v_r by a factor $5/2$, getting a value w_r. Since $-4k \leq v_r \leq 4k$, we have $(5/2)(-4k) \leq (5/2)v_r = w_r \leq (5/2)(4k)$, which yields

$$-10k \leq w_r \leq 10k \tag{8.5.6}$$

Thus, w_r looks a bit like a vote total of $10k$ formulas. We say "looks a bit like," since w_r may be fractional, while any vote total is integer.

Empirical tests have verified that the w_r values approximately represent vote totals produced by the entire set of selected $10k$ formulas on unseen records $r \in A$. Section 8.8 contains references. We use this empirical result as follows. First, we round down the values w_r, $r \in A$, to integer values $z_r = \lfloor w_r \rfloor$. Second, we treat these integer values as if they were samples of the vote total Z of the selected $10k$ formulas, and from these samples estimate $\mathcal{F}_{\mathcal{A}}(z)$ by standard statistical techniques. The estimated distribution has a staircase shape where jumps occur at less than half of the integers of the interval $[-10k, 10k]$. If desired, one may smooth the estimated distribution using linear interpolation so that a jump may occur at each integer between $-10k$ and $10k$.

We summarize the above steps.

Algorithm for Vote Distribution

The algorithm treats the case where the conditional distribution $\mathcal{F}_{\mathcal{A}}(z)$ for the $10k$ selected formulas is to be estimated from the training set A. The algorithm may also be used to compute $\mathcal{F}_{\mathcal{B}}(z)$ by letting the set B play the role of A and by defining δ_r of Step 2 as $\delta_r = -6k$.

(8.5.7) Algorithm VOTE DISTRIBUTION. *Estimates conditional distribution of vote total produced by selected formulas.*

Input: Set A with at least 10 records randomly taken from population $\mathcal{A}$. Subsets $a_1, a_2, \ldots, a_{10}$ and $A_1, A_2, \ldots, A_{10}$ constructed by Algorithm TRAINING SUBSETS (8.2.2). $10k$ DNF formulas selected from a set of 40 formulas, each derived from two sets of A, B, A^*, and B^*.

Output: An estimate $\hat{\mathcal{F}}_{\mathcal{A}}(z)$ of the conditional distribution $\mathcal{F}_{\mathcal{A}}(z) = \text{Prob}[Z \leq z \mid \text{unseen } \mathcal{A} \text{ record}]$.

Requires: No additional algorithm needed.

Procedure:

1. Index the $10k$ input formulas arbitrarily by $j = 1, 2, \ldots, 10k$. For each index j and each record $r \in A$, let v_{rj} be the vote by formula j for r.
2. For each $r \in A$, define $v_r = (\sum_{j=1}^{10k} v_{rj}) - \delta_r$ where $\delta_r = 6k$, $w_r = (5/2)v_r$, and $z_r = \lfloor w_r \rfloor$.
3. Construct an estimate $\hat{\mathcal{F}}_{\mathcal{A}}(z)$ of the conditional distribution $\mathcal{F}_{\mathcal{A}}(z)$ of random variable Z by viewing the z_r values as samples of Z. If desired, one may smooth the estimated distribution using linear interpolation so that a jump may occur at each integer between $-10k$ and $10k$. Output the estimated distribution $\hat{\mathcal{F}}_{\mathcal{A}}(z)$, and stop.

Example

We demonstrate the calculations of Algorithm VOTE DISTRIBUTION (8.5.7). Suppose $k = 4$, so there are $10k = 40$ selected formulas in total. Take any $r \in A$. That record is unseen for $|J(r)| = 4k = 16$ formulas. Thus, the vote total of those 16 formulas ranges from -16 to 16.

Suppose that A has 15 records r, and that the respective 15 vote totals v_r, each produced by 16 formulas $j \in J(r)$, are in increasing order 4(1), 8(3), 9(2), 10(1), 12(3), 14(1), 15(2), 16(2), where $x(y)$ means that value x occurs y times.

When we scale up the vote totals v_r by 5/2, we get 15 w_r values 10(1), 20(3), 22.5(2), 25(1), 30(3), 35(1), 37.5(2), 40(2).

Using $z_r = \lfloor w_r \rfloor$, the 15 sorted values of z_r are 10(1), 20(3), 22(2), 25(1), 30(3), 35(1), 37(2), 40(2). From these values, $\hat{\mathcal{F}}_{\mathcal{A}}(z)$ is derived. For example, 6 out of the 15 values of z_r are less than 25, 1 value is equal to 25, and there are no values equal to 26, 27, 28, or 29. Thus, for $25 \le z < 30$, $\hat{\mathcal{F}}_{\mathcal{A}}(z) = 6/15 = 0.40$. Here is the entire function.

$$\hat{\mathcal{F}}_{\mathcal{A}}(z) = \begin{cases} 0.00 & \text{if } z < 10 \\ 0.07 & \text{if } 10 \le z < 20 \\ 0.33 & \text{if } 20 \le z < 22 \\ 0.40 & \text{if } 22 \le z < 25 \\ 0.47 & \text{if } 25 \le z < 30 \\ 0.67 & \text{if } 30 \le z < 35 \\ 0.73 & \text{if } 35 \le z < 37 \\ 0.87 & \text{if } 37 \le z < 40 \\ 1.00 & \text{if } z \ge 40 \end{cases} \tag{8.5.8}$$

If desired, one may smooth the function by linear interpolation. We skip that simple step.

So far, we have looked at the probability of misclassification under the condition that the record is in a certain population. In the next section, we compute the probability of misclassification without that condition and show how classification errors can be controlled.

8.6 Classification Control

Suppose we want to separate records of $\mathcal{A}$ from those of $\mathcal{B}$. Assume that we have $\mathcal{F}_{\mathcal{A}}(z)$ and $\mathcal{F}_{\mathcal{B}}(z)$, the conditional distributions of the vote total on unseen records of $\mathcal{A}$ and $\mathcal{B}$, respectively. As before, let $\mathcal{G}_{\mathcal{A}}(z) = 1 - \mathcal{F}_{\mathcal{A}}(z)$ and $\mathcal{G}_{\mathcal{B}}(z) = 1 - \mathcal{F}_{\mathcal{B}}(z)$. By (8.4.4), the conditional probabilities of erroneous (resp. correct) classification are $P_{err|\mathcal{A}}(z) = \mathcal{F}_{\mathcal{A}}(z)$ and $P_{err|\mathcal{B}}(z) = \mathcal{G}_{\mathcal{B}}(z)$ (resp. $P_{cor|\mathcal{A}}(z) = \mathcal{G}_{\mathcal{A}}(z)$ and $P_{cor|\mathcal{B}}(z) = \mathcal{F}_{\mathcal{B}}(z)$).

Prior Probabilities

Let $P_{\mathcal{A}}$ (resp. $P_{\mathcal{B}}$) be the prior probability that a randomly chosen, unseen record is in $\mathcal{A}$ (resp. $\mathcal{B}$). We assume here that any record must be in $\mathcal{A}$ or $\mathcal{B}$, so

$$P_{\mathcal{A}} + P_{\mathcal{B}} = 1 \tag{8.6.1}$$

Error Probabilities

We denote the unconditional probability of erroneous (resp. correct) classification of an unseen record of $\mathcal{A} \cup \mathcal{B}$ by $P_{err}(z)$ (resp. $P_{cor}(z)$). We have

$$\begin{aligned} P_{err}(z) &= P_{err|\mathcal{A}}(z) \cdot P_{\mathcal{A}} + P_{err|\mathcal{B}}(z) \cdot P_{\mathcal{B}} \\ &= \mathcal{F}_{\mathcal{A}}(z) \cdot P_{\mathcal{A}} + \mathcal{G}_{\mathcal{B}}(z) \cdot P_{\mathcal{B}} \\ P_{cor}(z) &= 1 - P_{err}(z) \\ &= \mathcal{G}_{\mathcal{A}}(z) \cdot P_{\mathcal{A}} + \mathcal{F}_{\mathcal{B}}(z) \cdot P_{\mathcal{B}} \end{aligned} \tag{8.6.2}$$

Expected Misclassification Cost

Let $c_{err|\mathcal{A}}$ (resp. $c_{err|\mathcal{B}}$) be the cost of misclassifying an $\mathcal{A}$ (resp. $\mathcal{B}$) record. The expected total classification cost, $C_{err}(z)$, is

$$C_{err}(z) = c_{err|\mathcal{A}} \cdot P_{err|\mathcal{A}}(z) \cdot P_{\mathcal{A}} + c_{err|\mathcal{B}} \cdot P_{err|\mathcal{B}}(z) \cdot P_{\mathcal{B}} \tag{8.6.3}$$

Optimal Threshold

The vote total may range from $-10k$ to $10k$. Since vote totals greater than the threshold z trigger classification in $\mathcal{A}$, we only need to consider thresholds z ranging from $-10k - 1$ to $10k$. Thus, we find a threshold z^* minimizing the expected total misclassification cost by solving

$$\min_{-10k-1 \le z \le 10k} C_{err}(z) \tag{8.6.4}$$

If z^* is not unique, we break the tie using any suitable criterion. For example, we may choose the z^* closest to 0, with a secondary tie broken by a random choice.

Algorithm for Classification Control

We summarize the above steps.

(8.6.5) Algorithm CLASSIFICATION CONTROL. *Determines misclassification probability and threshold minimizing total misclassification costs.*

Input: Conditional distributions $\mathcal{F}_{\mathcal{A}}(z)$ and $\mathcal{F}_{\mathcal{B}}(z)$ of the vote total produced by $10k$ selected formulas. Prior probability $P_{\mathcal{A}}$ (resp. $P_{\mathcal{B}}$) that a randomly selected record is in $\mathcal{A}$ (resp. $\mathcal{B}$). Cost $c_{err|\mathcal{A}}$ (resp. $c_{err|\mathcal{B}}$) of misclassifying a record of $\mathcal{A}$ (resp. $\mathcal{B}$).

Output: The misclassification probability $P_{err}(z)$ when threshold z is used. The expected misclassification cost $C_{err}(z)$. A threshold z^* that minimizes total misclassification cost.

Requires: No additional algorithm needed.

Procedure:

1. Do for $-10k-1 \le z \le 10k$: Using $P_{err|\mathcal{A}}(z) = \mathcal{F}_{\mathcal{A}}(z)$ and $P_{err|\mathcal{B}}(z) = 1 - \mathcal{F}_{\mathcal{B}}(z)$, compute $P_{err}(z) = P_{err|\mathcal{A}}(z) \cdot P_{\mathcal{A}} + P_{err|\mathcal{B}}(z) \cdot P_{\mathcal{B}}$ and $C_{err}(z) = c_{err|\mathcal{A}} \cdot P_{err|\mathcal{A}}(z) \cdot P_{\mathcal{A}} + c_{err|\mathcal{B}} \cdot P_{err|\mathcal{B}}(z) \cdot P_{\mathcal{B}}$.
2. Find an optimal threshold z^* solving $\min_{-10k-1 \le z \le 10k} C_{err}(z)$. Use any suitable secondary criterion to break ties.
3. Output, for $-10k-1 \le z \le 10k$, $P_{err}(z)$ and $C_{err}(z)$. Also output z^*, and stop.

Correct Classification of Training Records

How accurately does a minimum-cost threshold z^* or, more generally, any threshold z classify the records $r \in (A \cup B)$, which have been used for the computation of the formulas? The next theorem gives a sufficient condition guaranteeing correct classification.

(8.6.6) Theorem. *Each record $r \in (A \cup B)$ is correctly classified with threshold z if $-2k \le z \le 2k-1$.*

Proof. The equation $|J(r)| = 4k$ of (8.5.2) implies that any $r \in (A \cup B)$ is unseen for exactly $4k$ of the total of $10k$ formulas. Suppose $r \in A$. In the worst case, the $4k$ formulas may produce a wrong vote of -1 each, while the remaining $6k$ votes are guaranteed to produce a vote of 1. Thus, the vote total is greater than or equal to $6k - 4k = 2k$. Analogously, if $r \in B$, then the vote total is less than or equal $-2k$. Since a vote total greater than z results in classification into $\mathcal{A}$, any threshold z satisfying $-2k \le z \le 2k-1$ handles the worst cases, and thus every case, correctly. □

We apply Algorithm CLASSIFICATION CONTROL (8.6.5) to an example case.

Example

Given are $P_{\mathcal{A}} = 0.1$, $P_{\mathcal{B}} = 0.9$, $c_{err|\mathcal{A}} = 100$, and $c_{err|\mathcal{B}} = 10$. Table (8.6.7) below shows a portion of the given $\mathcal{F}_{\mathcal{A}}(z)$ and $\mathcal{F}_{\mathcal{B}}(z)$ values. For those values, Algorithm CLASSIFICATION CONTROL (8.6.5) outputs the $P_{err}(z)$ and $C_{err}(z)$ values displayed in the table.

(8.6.7)

z	$\mathcal{F}_{\mathcal{A}}(z)$	$\mathcal{F}_{\mathcal{B}}(z)$	$P_{err}(z)$	$C_{err}(z)$
.	.	.	.	.
.	.	.	.	.
−4	0.00	0.31	0.62	6.2
−3	0.20	0.61	0.37	5.5
−2	0.20	0.79	0.21	3.9
−1	0.25	0.79	0.21	4.4
0	0.25	0.85	0.16	3.9
1	0.37	0.85	0.17	5.1
2	0.37	0.88	0.15	4.8
3	0.55	0.88	0.16	6.6
4	0.55	1.00	0.06	5.5
.	.	.	.	.
.	.	.	.	.

$\mathcal{F}_{\mathcal{A}}(z)$, $\mathcal{F}_{\mathcal{B}}(z)$, $P_{err}(z)$, and $C_{err}(z)$ of example

For example, for $z = 2$, we have $\mathcal{F}_{\mathcal{A}}(2) = 0.37$ and $\mathcal{F}_{\mathcal{B}}(2) = 0.88$. Then $P_{err|\mathcal{A}}(2) = \mathcal{F}_{\mathcal{A}}(2) = 0.37$, $P_{err|\mathcal{B}}(2) = 1 - \mathcal{F}_{\mathcal{B}}(2) = 1 - 0.88 = 0.12$, $P_{err}(2) = \mathcal{F}_{\mathcal{A}}(2) \cdot P_{\mathcal{A}} + \mathcal{G}_{\mathcal{B}}(2) \cdot P_{\mathcal{B}} = (0.37)(0.1) + (0.12)(0.9) = 0.15$, and $C_{err}(2) = c_{err|\mathcal{A}} \cdot P_{err|\mathcal{A}}(2) \cdot P_{\mathcal{A}} + c_{err|\mathcal{B}} \cdot P_{err|\mathcal{B}}(2) \cdot P_{\mathcal{B}} = (100)(0.37)(0.1) + (10)(0.12)(0.9) = 4.8$.

According to Table (8.6.7), both $z^* = -2$ and $z^* = 0$ minimize $C_{err}(z)$, with cost $C_{err}(z^*) = 3.9$. We select minimization of the probability of misclassification as the tie-breaking criterion. We have $P_{err}(-2) = 0.21$ and $P_{err}(0) = 0.16$. The smallest probability, 0.16, is achieved by $z^* = 0$. Hence, we select that threshold. Accordingly, a record of $\mathcal{A} \cup \mathcal{B}$ is declared to be in $\mathcal{A}$ if the vote total exceeds $z^* = 0$, and to be in $\mathcal{B}$ otherwise. For unseen records, the classification is incorrect with probability $P_{err}(0) = 0.16$, and the average misclassification cost is $C_{err}(0) = 3.9$. Since $-2k = 8 \leq z^* = 0 \leq 2k - 1 = 7$, Theorem (8.6.6) guarantees that the selected threshold results in correct classification of every $r \in (A \cup B)$.

In applications, we estimate $\mathcal{F}_{\mathcal{A}}(z)$ and $\mathcal{F}_{\mathcal{B}}(z)$ by Algorithm VOTE DISTRIBUTIONS (8.5.7) and estimate $P_{\mathcal{A}}$ and $P_{\mathcal{B}}$ from some sample data. Indeed, if the ratio $|A|/|B|$ reasonably represents the relative occurrence of records of $\mathcal{A}$ versus records of $\mathcal{B}$, then we can estimate $P_{\mathcal{A}}$ by $|A|/(|A| + |B|)$ and $P_{\mathcal{B}}$ by $|B|/(|A| + |B|)$. These estimates may be used

in Algorithm CLASSIFICATION CONTROL (8.6.5) instead of the exact values to obtain estimates of the misclassification probability $P_{err}(z)$ and of the minimum-cost threshold z^*.

8.7 Multipopulation Classification

We adapt the results of the preceding sections to the case where, instead of two populations $\mathcal{A}$ and $\mathcal{B}$, we have $q \geq 3$ populations $\mathcal{A}^1, \mathcal{A}^2, \ldots, \mathcal{A}^q$. Formally, the results also apply to the situation of $q = 2$ populations. For that case, the results effectively become those of the preceding sections.

Training Sets and Populations

For $p = 1, 2, \ldots, q$, the following definitions and assumptions hold. $\mathcal{B}^p$ is the union of the populations $\mathcal{A}^l$ with index $l \neq p$. That is,

$$\mathcal{B}^p = \bigcup_{l \neq p} \mathcal{A}^l \tag{8.7.1}$$

Let A^p be a training set whose records have been randomly chosen from population $\mathcal{A}^p$. Define a training set B^p of $\mathcal{B}^p$ by

$$B^p = \bigcup_{l \neq p} A^l \tag{8.7.2}$$

It is assumed that A^p has at least 10 records and that no record of A^p is weakly nested in any record of any A^l, $l \neq p$. This implies that no record of A^p is weakly nested in any record of B^p and *vice versa*.

Optimized Case

If tests to determine record entries and related costs are given, we compute via Algorithms MINIMUM SEPARATION (7.5.5) and OPTIMIZED RECORDS (7.5.8), with input sets $A = A^p$ and $B = B^p$, a set A^{*p} of optimized records and the related optimized test set K^{*p}.

Unions of Training Sets and Populations

Let $\mathcal{R}$ be the union of all populations, and define R to be the union of the training data. Thus,

$$\begin{aligned} \mathcal{R} &= \bigcup_{p=1}^{q} \mathcal{A}^p \\ R &= \bigcup_{p=1}^{q} A^p \end{aligned} \tag{8.7.3}$$

In the optimization case, we also have

$$R^* = \bigcup_{p=1}^{q} A^{*p} \tag{8.7.4}$$

Various classification problems may be of interest.

Special Cases

In the simplest case, one wants to separate each population $\mathcal{A}^p$ from $\mathcal{B}^p$. That case is directly handled by the algorithms of the preceding sections, using $\mathcal{A} = \mathcal{A}^p$, $\mathcal{B} = \mathcal{B}^p$, $A = A^p$, and $B = B^p$, and, in the case of optimized sets, $A^* = A^{*p}$ and $B^* = R^* - A^{*p}$.

In a more complicated case that also can be reduced to two populations, each population A^p is small and has a *representative record* r^p, and every other record of A^p is a variation of r^p. For each p, we take each record $r \in R$ and construct a *difference record* δ_r^p that encodes the differences between r^p and r. Let A^δ (resp. B^δ) be the set of all possible difference vectors δ_r^p for which r^p and r are from the same training set (different training sets). Let $\mathcal{A}^\delta$ and $\mathcal{B}^\delta$ be the populations underlying A^δ and B^δ, respectively.

We compute $10k$ logic formulas for separating A^δ and B^δ and classify unseen records r as follows. For each p, we calculate the difference vector δ_r^p and use the vote total of the $10k$ formulas to classify δ_r^p into $\mathcal{A}^\delta$ or $\mathcal{B}^\delta$; in the case of $\mathcal{A}^\delta$ (resp. $\mathcal{B}^\delta$), we declare that r is in $\mathcal{A}^p$ (resp. $\mathcal{B}^p$). If this method does not classify r into exactly one $\mathcal{A}^p$, the conflicting conclusions are resolved using the vote totals and the accuracy computed for them, as described in Sections 8.5 and 8.6. The approach is readily adapted to optimized difference records and optimized formulas. Exercise (8.9.5) asks for details of the procedure.

General Case

In the general case, one may carry out the classification of records of $\mathcal{R}$ via a sequence of two population cases. To reduce the complexity of the discussion, we do not consider the use of optimized records and formulas.

The following scheme is likely to come to mind first. We define a rooted binary decision tree where a set containing some of the training sets A^p is assigned to each node. Specifically, the set of all A^p is assigned to the root node, a set containing one A^p is associated with each tip node, and the sets associated with the two descendants of any non-tip node form a partition of the set associated with that non-tip node.

For each non-tip node, we compute logic formulas that separate the union of the A^p associated with one descendant from the union of the A^p of the second descendants. Classification is done with these formulas in the obvious way, by moving from the root node to some end node, making the branching decision at each non-tip node with the logic formulas at hand for that node. The scheme may seem attractive, but has drawbacks. First, a secondary method is needed that determines the training sets affiliated with each node of the binary tree. Second, cost control of classification errors is rather complicated. Instead, we propose another scheme which does not require another, secondary method, and for which cost control of errors is of manageable complexity.

Training Subsets

We first derive from each A^p the sets A_1^p, $A_2^p, \ldots,$ A_{10}^p using Algorithm TRAINING SUBSETS (8.2.2) and define, for $i = 1, 2, \ldots, 10$,

$$B_i^p = \bigcup_{l \neq p} A_i^l \tag{8.7.5}$$

Computation of Formulas

For each population p, we let the sets A^p and B^p, the training subsets A_1^p, $A_2^p, \ldots, A_{10}^p$ and $B_1^p, B_2^p, \ldots, B_{10}^p$, and the populations $\mathcal{A}^p$ and $\mathcal{B}^p$ play the role of A, B, $A_1, A_2, \ldots, A_{10}, B_1, B_2, \ldots, B_{10}, \mathcal{A}$, and $\mathcal{B}$ of the preceding sections. Thus, for each p, we obtain 40 min/max formulas. Analogously to the two population case, we are allowed to select, for each p, $10k$ of the 40 formulas; those selected formulas make up a set $\mathcal{H}^p$. The selection rule must be independent of p. We consider the $10k$ formulas of each $\mathcal{H}^p$ indexed by $j = 1, 2, \ldots, 10k$ in the same way. That is, for each $\mathcal{H}^p$, the index j refers to the same type of formula. We call such indexing *consistent*.

We want to predict the accuracy of the formulas of $\mathcal{H}^p$ for records not used in the computation of the formulas. Such records are *unseen*.

Vote Total for Unseen Record

Let $r \in R$, which is the union of the training sets. Thus, r is in some A^p, say $A^{p(r)}$. The set $A^{p(r)}$ and the corresponding set $B^{p(r)}$ are the pair of training sets that have been used to compute the formulas of $\mathcal{H}^{p(r)}$.

We summarize facts implied by the consistent indexing of the formulas, using the approach of Section 8.5. Each formula j of each $\mathcal{H}^p$ has been computed from some A_i^p and B_i^p, where the index i depends on j and not on p. Thus, we may denote that index i by $i(j)$. Define $J(r)$ to be the set

of indices j such that record r, which by definition of $p(r)$ is in $A^{p(r)}$, is not in $A^{p(r)}_{i(j)}$. Thus, record r is unseen for each formula $j \in J(r)$ of $\mathcal{H}^{p(r)}$. By the definition of B^p_i in (8.7.5), for all $p \neq p(r)$ and all i, $A^{p(r)}_i$ is a subset of B^p_i. Hence, we could equivalently demand in the definition of $J(r)$, for any $p \neq p(r)$, that r is not in $B^p_{i(j)}$. We conclude that, for each formula $j \in J(r)$ of each $\mathcal{H}^p$ with $p \neq p(r)$, record r is unseen.

Continuing with the approach of Section 8.5, we derive results for $J(r)$. Thus, $A^{p(r)}$ is partitioned by sets $a^{p(r)}_1, a^{p(r)}_2, \ldots, a^{p(r)}_{10}$, r is in some subset $a^{p(r)}_l$ of $A^{p(r)}$, $I(l)$ is the set of indices of the four subsets $a^{p(r)}_i$ that immediately follow $a^{p(r)}_l$ in the circular list $a^{p(r)}_1, a^{p(r)}_2, \ldots, a^{p(r)}_{10}$, and, finally, $J(r) = \{j \mid i(j) \in I(l)\}$ and $|J(r)| = 4k$.

For each p, denote by v^p_{rj} the vote of formula j of $\mathcal{H}^p$ for record r. Let v^p_r be the vote total of the v^p_{rj} for which $j \in J(r)$. Thus,

$$v^p_r = \sum_{j \in J(r)} v^p_{rj} \tag{8.7.6}$$

Since $|J(r)| = 4k$, we have

$$-4k \leq v^p_r \leq 4k \tag{8.7.7}$$

Arguing as for (8.5.5), we also have

$$v^p_r = \sum_{j=1}^{10k} v^p_{rj} - \delta^p_r \tag{8.7.8}$$

where the term δ^p_r accounts for the contribution of the $6k$ formulas whose computation made use of r. If $p = p(r)$ (resp. $p \neq p(r)$), then $r \in A^{p(r)}$ (resp. $r \in B^{p(r)}$), and the contribution is $\delta^p_r = 6k$ (resp. $\delta^p_r = -6k$). Hence,

$$\delta^p_r = \begin{cases} 6k & \text{if } p = p(r) \\ -6k & \text{otherwise} \end{cases} \tag{8.7.9}$$

Analogously to the approach of Section 8.5, we scale up the v^p_r values by the factor 5/2, getting w^p_r. In turn, we derive from each w^p_r the integer $z^p_r = \lfloor w^p_r \rfloor$. Finally, for each p, we treat the latter integer values as if they were samples of the vote total Z^p of the $10k$ formulas of $\mathcal{H}^p$ for unseen records of $\mathcal{R}$. Using the decision strategy described next, we use the samples to estimate the probability of misclassification and to control misclassification costs.

Decision Strategy

The *decision strategy* consists of integer decision parameters $d^1, d^2, \ldots, d^q$ that are used as follows. For each p, let z_r^p be the vote total produced by the $10k$ formulas of $\mathcal{H}^p$ for an unseen record of $\mathcal{R}$, and define

$$e_r^p = z_r^p + d^p \tag{8.7.10}$$

Let p^* be an index for which e_r^p is maximum. Thus, p^* solves

$$\max_{1 \le p \le q} e_r^p \tag{8.7.11}$$

We then classify r as being in A^{p^*}.

Tie Breaker

If several p^* solve (8.7.11), we apply a suitable criterion T to break the tie. For the discussion to follow, we select a T based on the prior probabilities $P_{\mathcal{A}^p}$ that a randomly selected, unseen record of $\mathcal{R}$ is in $\mathcal{A}^p$. That is, T chooses p^* for which that prior probability, or an estimate of that probability, is largest. If that rule produces another tie, preference is given to the largest index value. The selected T induces a total ordering of the p indices where a higher ranking means preference by T. Without risk of confusion, we denote the ranking of p by $T(p)$. Thus, for two given indices p and p', $T(p) > T(p')$ means that, in case of a tie, T prefers p over p'.

Conditional Probabilities

For a given decision strategy $d^1, d^2, \ldots, d^q$ and population $\mathcal{A}^p$, we denote by $P_{err|\mathcal{A}^p}(d^1, \ldots, d^q)$ the probability that the formulas of the $\mathcal{H}^p$ sets and the tie breaker T erroneously classify an unseen record of $\mathcal{A}^p$. That probability can be estimated for all p as follows.

For all p, initialize integer variables $m_{err|\mathcal{A}^p} = 0$. When the subsequent process stops, $m_{err|\mathcal{A}^p}$ is the number of records of A^p that have been misclassified. Take each record $r \in R$. Let $p(r)$ be the index of the training set containing r; thus, $r \in \mathcal{A}^{p(r)}$. For each p, compute the vote total v_r^p via (8.7.8), and define $w_r^p = (5/2)v_r^p$, $z_r^p = \lfloor w_r^p \rfloor$ and $e_r^p = z_r^p + d^p$. Let p^* solve $\max_{1 \le p \le q} e_r^p$, possibly after application of the tie breaker T. If $p^* \neq p(r)$, then declare this to be an instance of misclassification of a record of $A^{p(r)}$, and increment $m_{err|\mathcal{A}^{p(r)}}$ by 1.

When all records of $r \in R$ have been processed, $P_{err|\mathcal{A}^p}(d^1, \ldots, d^q)$ is estimated by

$$\hat{P}_{err|\mathcal{A}^p}(d^1, \ldots, d^q) = m_{err|\mathcal{A}^p}/|A^p| \tag{8.7.12}$$

Correspondingly, the probability $P_{cor|\mathcal{A}^p}(d^1, \ldots, d^q)$ of correct classification of unseen records of $\mathcal{A}^p$ is estimated by $\hat{P}_{cor|\mathcal{A}^p}(d^1, \ldots, d^q) = 1 - \hat{P}_{err|\mathcal{A}^p}(d^1, \ldots, d^q)$.

Algorithm for Multiclassification Probability

We assemble the above steps to the following algorithm.

(8.7.13) Algorithm MULTICLASSIFICATION PROBABILITY. *Estimates the probabilities of erroneous classification for a given decision strategy.*

Input: Training sets A^1, $A^2, \ldots$, A^q randomly chosen from populations $\mathcal{A}^1$, $\mathcal{A}^2, \ldots, \mathcal{A}^q$, respectively. For each population p, a set $\mathcal{H}^p$ of $10k$ logic formulas that separate A^p from $B^p = \cup_{l \neq p} A^l$. Decision integers d^1, $d^2, \ldots$, d^q and tie breaker T for resolving ties of $\max_{1 \leq p \leq q} e_r^p$.

Output: For $1 \leq p \leq q$, an estimate $\hat{P}_{err|\mathcal{A}^p}(d^1, \ldots, d^q)$ of the probability $P_{err|\mathcal{A}^p}(d^1, \ldots, d^q)$ that an unseen record of $\mathcal{A}^p$ is misclassified.

Requires: No additional algorithm needed.

Procedure:

0. For $1 \leq p \leq q$, initialize $m_{err|\mathcal{A}^p} = 0$.
1. Do for each record $r \in R$: Determine $p(r)$ such that $r \in A^{p(r)}$. For each p, compute v_r^p via (8.7.8), and define $w_r^p = (5/2)v_r^p$, $z_r^p = \lfloor w_r^p \rfloor$, and $e_r^p = z_r^p + d^p$. If $\max_{1 \leq p \leq q} e_r^p$ has $p^* \neq p(r)$ as solution, increment $m_{err|\mathcal{A}^{p(r)}}$ by 1.
2. For $1 \leq p \leq q$, output $\hat{P}_{err|\mathcal{A}^p}(d^1, \ldots, d^q) = m_{err|\mathcal{A}^p}/|A^p|$, and stop.

Error Probabilities

Let $P_{err}(d^1, \ldots, d^q)$ (resp. $P_{cor}(d^1, \ldots, d^q)$) be the probability that an unseen record of $\mathcal{R}$ is erroneously (resp. correctly) classified. Using $P_{\mathcal{A}^p}$, the prior probability that a randomly selected, unseen record of $\mathcal{R}$ is in population $\mathcal{A}^p$, we have

$$P_{err}(d^1, \ldots, d^q) = \sum_{p=1}^{q} P_{err|\mathcal{A}^p}(d^1, \ldots, d^q) \cdot P_{\mathcal{A}^p}$$

(8.7.14)
$$\begin{aligned} P_{cor}(d^1, \ldots, d^q) &= 1 - P_{err}(d^1, \ldots, d^q) \\ &= \sum_{p=1}^{q} P_{cor|\mathcal{A}^p}(d^1, \ldots, d^q) \cdot P_{\mathcal{A}^p} \end{aligned}$$

Expected Misclassification Cost

For $1 \leq p \leq q$, let $c_{err|\mathcal{A}^p}$ be the rational cost incurred when a record of population $\mathcal{A}^p$ is erroneously classified. The expected total classification cost, $C(d^1, \ldots, d^q)$, is given by

$$C(d^1, \ldots, d^q) = \sum_{p=1}^{q} c_{err|\mathcal{A}^p} \cdot P_{err|\mathcal{A}^p}(d^1, \ldots, d^q) \cdot P_{\mathcal{A}^p} \tag{8.7.15}$$

An attractive goal is minimization of the expected total classification cost by a suitable choice of the decisions d^1, $d^2, \ldots,$ d^q. Thus, we would like to solve

$$\min_{d^1,\ldots,d^q} C(d^1, \ldots, d^q) \tag{8.7.16}$$

Another problem of interest is minimization of the probability of erroneous classification, that is,

$$\min_{d^1,\ldots,d^q} P_{err}(d^1, \ldots, d^q) \tag{8.7.17}$$

Since $P_{cor}(d^1, \ldots, d^q) = 1 - P_{err}(d^1, \ldots, d^q)$, the latter problem is equivalent to the problem of maximization of $P_{cor}(d^1, \ldots, d^q)$, that is,

$$\max_{d^1,\ldots,d^q} P_{cor}(d^1, \ldots, d^q) \tag{8.7.18}$$

Suppose we define, for $1 \leq p \leq q$, $c_{err|\mathcal{A}^p} = 1$. Then the formula (8.7.15) for $C(d^1, \ldots, d^q)$ becomes

$$\begin{aligned} C(d^1, \ldots, d^q) &= \sum_{p=1}^{q} c_{err|\mathcal{A}^p} \cdot P_{err|\mathcal{A}^p}(d^1, \ldots, d^q) \cdot P_{\mathcal{A}^p} \\ &= \sum_{p=1}^{q} P_{err|\mathcal{A}^p}(d^1, \ldots, d^q) \cdot P_{\mathcal{A}^p} \\ &= P_{err}(d^1, \ldots, d^q) \end{aligned} \tag{8.7.19}$$

We conclude that the minimization problem (8.7.17) and thus the maximization problem (8.7.18) are special cases of (8.7.16).

Estimation of Probabilities and Cost

Typically, we do not know the probabilities $P_{err|\mathcal{A}^p}(d^1, \ldots, d^q)$, and direct solution of (8.7.16), (8.7.17), or (8.7.18) is not possible. But using Algorithm MULTICLASSIFICATION PROBABILITY (8.7.13), we can compute estimates $\hat{P}_{err|\mathcal{A}^p}(d^1, \ldots, d^q)$. If, in addition, we have estimates $\hat{P}_{\mathcal{A}^p}$ for the probabilities $P_{\mathcal{A}^p}$, then we have the following estimates of $P_{err}(d^1, \ldots, d^q)$ and $P_{cor}(d^1, \ldots, d^q)$.

$$\begin{aligned} \hat{P}_{err}(d^1, \ldots, d^q) &= \sum_{p=1}^{q} \hat{P}_{err|\mathcal{A}^p}(d^1, \ldots, d^q) \cdot \hat{P}_{\mathcal{A}^p} \\ \hat{P}_{cor}(d^1, \ldots, d^q) &= 1 - \hat{P}_{err}(d^1, \ldots, d^q) \end{aligned} \tag{8.7.20}$$

We also have the following estimate of $C(d^1, \ldots, d^q)$.

$$\hat{C}(d^1, \ldots, d^q) = \sum_{p=1}^{q} c_{err|\mathcal{A}^p} \cdot \hat{P}_{err|\mathcal{A}^p}(d^1, \ldots, d^q) \cdot \hat{P}_{\mathcal{A}^p} \tag{8.7.21}$$

The corresponding optimization problems are

$$\begin{aligned} &\min_{d^1,\ldots,d^q} \hat{P}_{err}(d^1, \ldots, d^q) \\ &\max_{d^1,\ldots,d^q} \hat{P}_{cor}(d^1, \ldots, d^q) \end{aligned} \tag{8.7.22}$$

and

$$\min_{d^1,\ldots,d^q} \hat{C}(d^1, \ldots, d^q) \tag{8.7.23}$$

Arguing as before, the problems of (8.7.22) are special cases of (8.7.23).

Bounds for Decision Strategies

How can we solve (8.7.23)? We claim that in the solution of that problem, we only need to consider d^p values in the range $0 \leq d^p \leq 20k + 1$ and can demand the smallest d^p value to be 0. For a proof, let the largest (resp. smallest) d^p value be achieved by an index $p = p_{max}$ (resp. $p = p_{min}$). Since the solution of $\max_{1 \leq p \leq q} e_r^p$ of (8.7.11) is not affected if an arbitrary constant is added to each d^p, we may assume that $d^{p_{min}} = 0$, as claimed. We prove validity of the restriction $0 \leq d^p \leq 20k + 1$ by contradiction. Hence, suppose $d^{p_{max}} > 20k+1$. For each p, define $d'^p = d^p - d^{p_{max}} + 20k+1$ and $d''^p = \max\{d'^p, 0\}$. These formulas imply $d'^{p_{max}} = d''^{p_{max}} = 20k + 1$. Since the d^p and d'^p values differ by the constant $-d^{p_{max}} + 20k + 1$, they produce the same p^*. We show that the same conclusion holds for the d''^p values, by proving that any p for which d'^p and d''^p differ, that is, for which $d'^p < 0$ and $d''^p = 0$, the index p is not selected as p^*, regardless of whether d'^p or d''^p is used to decide the case. For all p, $-10k \leq z_r^p \leq 10k$, so $z_r^{p_{max}} + d''^{p_{max}} = z_r^{p_{max}} + d'^{p_{max}} \geq -10k + 20k + 1 = 10k + 1 > 10k + 0 \geq z_r^p + d''^p > z_r^p + d'^p$, which rules out p as candidate for p^*, no matter whether d'^p or d''^p is used for the decision.

We no longer need an index p_{min} for the smallest d^p value, and from now on will indicate that value by d^{min}. Let D be the set of d^1, d^2, ..., d^q cases that, according to the above argument, need to be considered. Hence,

$$D = \{(d^1, \ldots, d^q) \mid 0 \leq d^p \leq 20k + 1,\ d^{min} = 0\} \tag{8.7.24}$$

The problem $\min_{(d^1,\ldots,d^q)} \hat{C}(d^1, \ldots, d^q)$ of (8.7.23) has been reduced to

$$\min_{(d^1,\ldots,d^q) \in D} \hat{C}(d^1, \ldots, d^q) \tag{8.7.25}$$

but generally still appears to be difficult. An intuitive explanation is that $\hat{C}(d^1, \ldots, d^q)$ may have local minima that are not global. This is possible even when $q = 2$. Exercise (8.9.10) asks for a demonstration of this claim. When $q \leq 3$, local minima are no obstacle to rapid solution of (8.7.25), since one may enumerate all possible choices of $(d^1, \ldots, d^q) \in D$. Indeed, Exercise (8.9.13) asks for a proof that $|D| = (20k + 2)^q - (20k + 1)^q$. Assume the largest possible value for k, which is $k = 4$. For $q = 3$, we then have a reasonable $|D| = 19,927$, while $q = 4$ already produces a large $|D| = 2,165,455$. Evidently, complete enumeration is computationally infeasible beyond $q = 3$. We are not aware of any efficient technique that implicitly skips most of the cases even when the number of populations is modest, say, $q \leq 20$. Hence, we propose a heuristic technique that is based on linear programming, network flow techniques, and rounding, and that quickly produces a good if not optimal solution for any value of q. The discussion below assumes knowledge of those techniques. For references about these techniques, see Section 8.8.

Approximate Solution

A *linear program* (LP) requires minimization of a linear objective function where the variables are constrained by linear inequalities. An *integer program* (IP) is an LP with the additional constraint that the variables are integer. We formulate the selection of the optimal strategy as an IP. The variables are the nonnegative integer variables d^p, $1 \leq p \leq q$, representing the strategy decisions, and $\{0, \pm 1\}$ variables x_r, $r \in A^p$, telling by the value 1 (resp. 0) whether r is erroneously (resp. correctly) classified.

For any $r \in R$, let $A^{p(r)}$ be the training set containing r. For $1 \leq p \leq q$ and $r \in R$, define z_r^p to be the integer value computed via $z_r^p = \lfloor w_r^p \rfloor$ and $w_r^p = (5/2)v_r^p$, where v_r^p is determined by (8.7.6) or (8.7.8).

Since $m_{err|\mathcal{A}^p}$ is the number of misclassified cases computed for A^p, and since each x_r indicates such misclassification by the value 1, we have

$$m_{err|\mathcal{A}^p} = \sum_{r \in A^p} x_r \tag{8.7.26}$$

Define

$$c^p = c_{err|\mathcal{A}^p} \cdot \hat{P}_{\mathcal{A}^p} / |A^p| \tag{8.7.27}$$

Using (8.7.26), (8.7.27), and $\hat{P}_{err|\mathcal{A}^p}(d^1, \ldots, d^q) = m_{err|\mathcal{A}^p}/|A^p|$ of (8.7.12) in the equation of (8.7.21) for $\hat{C}(d^1, \ldots, d^q)$, we have

$$
\begin{aligned}
\hat{C}(d^1, \ldots, d^q) &= \sum_{p=1}^{q} c_{err|\mathcal{A}^p} \cdot \hat{P}_{err|\mathcal{A}^p}(d^1, \ldots, d^q) \cdot \hat{P}_{\mathcal{A}^p} \\
&= \sum_{p=1}^{q} c_{err|\mathcal{A}^p}(m_{err|\mathcal{A}^p}/|A^p|)\hat{P}_{\mathcal{A}^p} \\
&= \sum_{p=1}^{q} (c_{err|\mathcal{A}^p} \cdot \hat{P}_{\mathcal{A}^p}/|A^p|) \sum_{r \in A^p} x_r \qquad (8.7.28) \\
&= \sum_{p=1}^{q} c^p \cdot \sum_{r \in A^p} x_r \\
&= \sum_{r \in R} c^{p(r)} \cdot x_r
\end{aligned}
$$

Inequality Constraints

Let r be any record of R. As before, let $p(r)$ be the index of the training set containing r. Thus, $r \in A^{p(r)}$. Define K to be the constant $40k + 2$. Consider the following inequality system with added integrality constraints. Recall that $T(p) > T(p')$ means that the tie breaker T prefers p over p'.

(8.7.29)

$$
\begin{aligned}
& d^{p(r)} + z_r^{p(r)} \geq d^p + z_r^p - K \cdot x_r, \quad \forall p \neq p(r),\ T(p(r)) > T(p) \\
& d^{p(r)} + z_r^{p(r)} \geq d^p + z_r^p + 1 - K \cdot x_r, \quad \forall p \neq p(r),\ T(p) > T(p(r)) \\
& 0 \leq d^p \leq 20k + 1, \quad \forall p \\
& 0 \leq x_r \leq 1 \\
& d^p \text{ integer}, \quad \forall p \\
& x_r \text{ integer}
\end{aligned}
$$

Suppose that each d^p of a decision strategy satisfies $0 \leq d^p \leq 20k + 1$, and that r is correctly classified into $A^{p(r)}$. We prove that (8.7.29) holds with $x_r = 0$. Since $p(r)$ solves $\max_{1 \leq p \leq q} e_r^p$, we have $e^{p(r)} \geq e^p$ (resp. $e^{p(r)} \geq e^p + 1$) for all $p \neq p(r)$ such that the tie breaker T prefers $p(r)$ to p (resp. p to $p(r)$). Since, for all p, $e_r^p = d^p + z_r^p$, these inequalities prove that (8.7.29) is satisfied with $x_r = 0$.

On the other hand, suppose that the given d^p values result in misclassification of r, so $p(r)$ does not solve $\max_{1 \leq p \leq q} e_r^p$. Thus, for some p, $d^{p(r)} + z_r^{p(r)} < d^p + z_r^p$ and T prefers $p(r)$ to p, or $d^{p(r)} + z_r^{p(r)} < d^p + z_r^p + 1$ and T prefers p to $p(r)$. Regardless of the case, the inequality of (8.7.29)

for that p is violated if $x_r = 0$. We show that $x_r = 1$ eliminates all such violations.

We examine the first two inequalities of (8.7.29). The smallest possible value for the left-hand side of either inequality is produced by $d^{p(r)} = 0$ and $z_r^{p(r)} = -10k$ and thus is equal to $-10k$, while the largest right-hand side value occurs in the second inequality when $d^p = 20k+1$ and $z_r^p = 10k$. Indeed, that right-hand side value, with $x_r = 1$, is $d^p + z_r^p + 1 - K \cdot x_r = 20k+1+10k+1-(40k+2) = -10k$, which is equal to the smallest possible left-hand side value. We conclude that the inequalities of (8.7.29) hold for the misclassification case when $x_r = 1$.

IP Formulation

For $r \in R$ and all $p \neq p(r)$, define

$$b_r^p = \begin{cases} z_r^p - z_r^{p(r)} & \text{if } T(p(r)) > T(p) \\ z_r^p - z_r^{p(r)} + 1 & \text{otherwise} \end{cases} \tag{8.7.30}$$

We use b_r^p in (8.7.29) for the following compact formulation of an IP that represents the minimization problem $\min_{(d^1,\ldots,d^q)\in D} \hat{C}(d^1,\ldots,d^q)$. Let "s. t." stand for "subject to." The IP is

$$\begin{array}{ll} \min & \sum_{r\in R} c^{p(r)} \cdot x_r \\ \text{s. t.} & d^{p(r)} - d^p + K \cdot x_r \geq b_r^p, \quad \forall r \in R,\ p \neq p(r) \\ & 0 \leq d^p \leq 20k+1, \quad \forall p \\ & 0 \leq x_r \leq 1, \quad \forall r \in R \\ & d^p \text{ integer}, \quad \forall p \\ & x_r \text{ integer}, \quad \forall r \in R \end{array} \tag{8.7.31}$$

We solve this IP approximately, and thus get a decision strategy that approximately minimizes the expected total misclassification cost, by repeatedly solving variations of the LP of (8.7.32) below and by rounding fractional solution values.

LP Formulation

We drop the integrality conditions from (8.7.31) and rewrite the inequalities $d^p \leq 20k+1$ and $x_r \leq 1$ as $-d^p \geq -(20k+1)$ and $-x_r \geq -1$, to get the LP

$$\begin{array}{ll} \min & \sum_{r\in R} c^{p(r)} \cdot x_r \\ \text{s. t.} & d^{p(r)} - d^p + K \cdot x_r \geq b_r^p, \quad \forall r \in R,\ p \neq p(r) \\ & -d^p \geq -(20k+1), \quad \forall p \\ & -x_r \geq -1, \quad \forall r \in R \\ & d^p \geq 0, \quad \forall p \\ & x_r \geq 0, \quad \forall r \in R \end{array} \tag{8.7.32}$$

Solution Process

Details of the solution process are as follows. We begin by computing an optimal extreme point solution of the LP of (8.7.32). Let x_r^* denote the optimal values for the x_r variables. There are two cases, depending on whether each x_r^* is integer or not.

Suppose each x_r^* is integer, and thus equal to 0 or 1. We prove that all optimal values for the d^p variables are integer as well; thus, we have the desired good if not optimal decision strategy. First, all right-hand side values and coefficients of the LP constraints are integer. Second, each constraint involving d^p variables has at most two of these variables, and if two variables occur, the coefficient of one variable is $+1$, while the coefficient of the second variable is -1. This implies that the coefficient submatrix corresponding to the d^p variables is totally unimodular. Third, the combination of integer data and total unimodularity forces all extreme point solutions where each x_r is equal to 0 or 1, to be integer. For references about total unimodularity and the above arguments, see Section 8.8.

Suppose some of the x_r^* values are fractional. Among them, we select an index r for which x_r is closest to $1/2$; that is $|x_r - 1/2|$ is minimum. Ties are resolved arbitrarily. Let r^* be the selected index. We fix x_{r^*} to 0 and solve the LP again, and then fix x_{r^*} to 1 and solve the LP once more. Thus, we get two optimal objective function values. We permanently fix the variable x_{r^*} to the 0 or 1 that produces the smaller of the two objective function values. In case of a tie, we prefer $x_{r^*} = 0$. Due to the inequalities $d^{p(r)} - d^p + K \cdot x_r \geq b_r^p$ of (8.7.32), the LP may become infeasible when $x_{r^*} = 0$. In that case, we assign ∞ as the objective function value. Once x_{r^*} has been permanently fixed in the LP, we invoke recursion. That is, we declare the revised LP to be the current one, and carry out the above process again.

Solution of the LP

We want to devise a fast solution algorithm for the LP of (8.7.32). To this end, we first replace the variables x_r of the LP by y_r using the rule $x_r = y_r/K$. This converts the LP of (8.7.32) to

$$
\begin{array}{lll}
\min & \sum_{r \in R} (c^{p(r)}/K) \cdot y_r & \\
\text{s. t.} & d^{p(r)} - d^p + y_r \geq b_r^p, & \forall r \in R,\ p \neq p(r) \\
& -d^p \geq -(20k+1), & \forall p \\
& -y_r \geq -K, & \forall r \in R \\
& d^p \geq 0, & \forall p \\
& y_r \geq 0, & \forall r \in R
\end{array}
\tag{8.7.33}
$$

Matrix Formulation of the LP and its Dual

Let E be the constraint coefficient matrix of the inequalities of (8.7.33) save the nonnegativity conditions of variables. Define column vectors s, g, and h to contain the variables of (8.7.33), the objective function coefficients, and the right side values of the inequalities except for the nonnegativity conditions, respectively. Using these arrays, we rewrite the LP of (8.7.33) as follows. The superscripted "t" denotes transpose.

$$\begin{array}{ll} \min & g^t \cdot s \\ \text{s. t.} & E \cdot s \geq h \\ & s \geq 0 \end{array} \tag{8.7.34}$$

The dual of this LP, with column vector f of dual variables, is

$$\begin{array}{ll} \max & h^t \cdot f \\ \text{s. t.} & E^t \cdot f \leq g \\ & f \geq 0 \end{array} \tag{8.7.35}$$

We use a column vector u of nonnegative slack variables in the inequality $E^t \cdot f \leq g$ to convert (8.3.35) to the equivalent problem

$$\begin{array}{ll} \max & h^t \cdot f \\ \text{s. t.} & E^t \cdot f + u = g \\ & f \geq 0 \\ & u \geq 0 \end{array} \tag{8.7.36}$$

The constraint coefficient matrix of (8.7.36) is $[E^t|I]$, where I is an identity matrix of appropriate size.

Conversion to Network Flow Problem

Let each column of E be indexed by the associated variable d^p or y_r. The corresponding indexing holds for the rows of E^t. From (8.7.33), it is evident that each row of E has one or three nonzeros, all of which are ± 1s. In the case of three nonzeros, exactly two of them are 1s, and they occur in columns $d^{p(r)}$ and y_r, for some $r \in R$. Thus, each column of E^t has one or three ± 1s, and in the case of three ± 1s, exactly two of them are 1s occurring in rows $d^{p(r)}$ and y_r, for some $r \in R$.

For each $r \in R$, we multiply row y_r of $[E^t|I]$ by -1 and add it to row $d^{p(r)}$ of that matrix. The resulting matrix has in each column one or two nonzeros, which are ± 1s. In the case of two nonzeros, one is 1, and the other one is -1. The row operations can be viewed as premultiplication of

$[E^t|I]$ by a nonsingular matrix Q that is derived from an identity matrix by inserting a -1 into certain rows. We use Q in (8.7.36) to get the LP

$$\begin{array}{ll} \max & h^t \cdot f \\ \text{s. t.} & Q \cdot E^t \cdot f + Q \cdot u = Q \cdot g \\ & f \geq 0 \\ & u \geq 0 \end{array} \tag{8.7.37}$$

with coefficient matrix $[Q \cdot E^t|Q] = Q[E^t|I]$. Due to the above described structure of $Q[E^t|I]$, the LP of (8.7.37) is a network flow optimization problem for which an optimal dual solution can be rapidly calculated. For references, see Section 8.8. We premultiply the optimal dual solution of (8.7.37) by Q^t and get an optimal dual solution for (8.7.35) and thus an optimal primal solution for (8.7.34) and (8.7.33). Straightforward LP techniques convert that primal solution to the desired extreme point solution for the rounding process.

Algorithm for Multiclassification Control

We summarize the solution process in the following algorithm.

(8.7.38) Heuristic MULTICLASSIFICATION CONTROL. *Determines a good if not optimal decision strategy for minimization of the expected multipopulation classification cost.*

Input: Training sets $A^1, A^2, \ldots, A^q$ randomly chosen from populations $\mathcal{A}^1, \mathcal{A}^2, \ldots, \mathcal{A}^q$, respectively. For each population p, a set $\mathcal{H}^p$ of $10k$ logic formulas that separate A^p from $B^p = \cup_{l \neq p} A^l$. For each p, cost $c_{err|\mathcal{A}^p}$ of misclassifying a record of $\mathcal{A}^p$.

Output: A good if not optimal decision strategy $d^1, d^2, \ldots, d^q$ for minimization of the expected multipopulation classification cost.

Requires: Algorithm for network flow optimization problem.

Procedure:

1. Do for each record $r \in R$: Determine $p(r)$ such that $r \in A^{p(r)}$. For each p, compute v_r^p via (8.7.8), define $w_r^p = (5/2)v_r^p$ and $z_r^p = \lfloor w_r^p \rfloor$, and compute b_r^p of (8.7.30). Use these results and $K = 40k + 2$ to define the IP of (8.7.31) and the LP of (8.7.32). In the latter problem, replace x_r by y_r/K to get the LP of (8.7.33), which in matrix notation is given by (8.7.34). Take the dual of (8.7.34), getting (8.7.35), and convert that dual via the slack vector u and the transformation matrix Q to the network flow problem (8.7.37).
2. Solve the IP of (8.7.31) approximately by rounding, using the following step recursively.
   Compute an optimal extreme point solution of the LP of (8.7.32) via a network flow method applied to (8.7.37). Let x_r^* denote the optimal

values of the x_r variables of (8.7.32). If all x_r^* are 0 or 1, go with the optimal values for the d^p variables, which are integer, to Step 3. Otherwise, select an index r for which $|x_r - 1/2|$ is minimum. Break ties arbitrarily. Let r^* be the selected index. Solve (8.7.32) twice, once with $x_{r^*} = 0$ and once with $x_{r^*} = 1$. Permanently fix x_{r^*} to the 0 or 1 that produces the smaller of the two optimal objective function values. In case of a tie, the value $x_{r^*} = 0$ is preferred.

3. Output the optimal values of the d^p variables as a good if not optimal decision strategy, and stop.

Correct Classification of Training Sets

Analogously to Theorem (8.6.6), one may guarantee correct classification of training records of A^1, $A^2, \ldots,$ A^q by restricting the values of the decision strategy computed by Heuristic MULTICLASSIFICATION CONTROL (8.7.38).

(8.7.39) Theorem. *Each record r of $R = \cup_{p=1}^{q} A^p$ is correctly classified by the decision strategy computed by Heuristic* MULTICLASSIFICATION CONTROL (8.7.38) *if, in the* IP *of* (8.7.31) *and the related* LPs*, each d^p is restricted to $d^p \leq 4k - 1$.*

Proof. We argue as in the proof of Theorem (8.6.6). Take any $r \in R$. The smallest vote total $\sum_{j=1}^{10k} v_{rj}^{p(r)}$ produced by the $10k$ formulas separating $A^{p(r)}$ and $B^{p(r)}$ is $2k$, while, for any $p \neq p(r)$, the largest vote total $\sum_{j=1}^{10k} v_{rj}^{p}$ by the $10k$ formulas separating A^p and B^p is $-2k$. Then for any $p \neq p(r)$, $\sum_{j=1}^{10k} v_{rj}^{p(r)} + d^{p(r)} \geq 2k + 0 > 2k - 1 = -2k + 4k - 1 \geq \sum_{j=1}^{10k} v_{rj}^{p} + d^p$, which implies that r is classified into $A^{p(r)}$, as desired. □

Looking Ahead

Section 10.12 of Chapter 10 describes how learned formulas can be inserted into CNF systems. Sections 11.6–11.8 of Chapter 11 demonstrate the use of learned formulas in applications.

8.8 Further Reading

In data mining, classifiers are frequently combined to improve the accuracy of predictions; see Hand, Mannila, and Smyth (2001). The voting process described in this chapter is one way to make use of that general idea. The process is based on Sun (1998) and Felici, Sun, and Truemper (2004).

Various other methods are described in Mendelson and Smola (2003); see also Russell and Norvig (2003).

A scheme called 10-*fold cross-validation* is often used to compare the accuracy of algorithms. The training data are partitioned into 10 subsets containing 10% each. Training is done on 90% of the data and accuracy is tested on the remaining 10%. This process is carried out in the 10 possible ways. From these results, the average accuracy is computed. Details are provided in Hand, Mannila, and Smyth (2001).

Cross-validation may seem similar to the voting approach taken here, where 10 times 60% of the training data are used for training. But this similarity is only superficial, since the process used here allows reliable estimation of distributions of the vote total.

Linear programming and network flow techniques are covered in Chvátal (1983) and Ahuja et al. (1993), respectively.

A discussion of total unimodularity and related properties is included in Schrijver (1986).

The rounding method used in Section 8.7 can be improved; see, for example, Chapter 8 of Truemper (1998).

8.9 Exercises

Most exercises rely on the training sets and subsets constructed in Exercise (8.9.2). Alternately, the reader may download data sets from the Internet. A good web site to begin the search is the Repository of Machine Learning Databases and Domain Theories of the University of California at Irvine.

The calculations required by some of the exercises may be accomplished with the software of Exercise (8.9.14).

(8.9.1) Apply Algorithm EXPANSION OF SET (8.2.1) to the following set A.

Set A
$(x_1 = x_2 = \mathit{False}, x_3 = \mathit{True}, x_4 = x_5 = x_6 = \mathit{Absent})$
$(x_1 = \mathit{True}, x_2 = \mathit{False}, x_3 = \mathit{Absent}, x_4 = \mathit{True}, x_5 = x_6 = \mathit{False})$
$(x_1 = \mathit{False}, x_2 = \mathit{Unavailable}, x_3 = x_4 = \mathit{True}, x_5 = x_6 = \mathit{Absent})$

(8.9.2)

(a) Construct training sets A^1, A^2, and A^3 as follows. Each record of each of the three sets contains values for six variables $x_1, x_2, \ldots, x_6$. For each set, construct 20 records. Each record has *True/False* values for a subset of the variables as specified below, and has arbitrarily

chosen *True*, *False*, *Unavailable*, and *Absent* values for the remaining variables.
Set A^1:
For any record, either $x_1 = x_2 = x_3 = True$, or $x_1 = True$ and $x_2 = x_3 = False$. The choice is made randomly.
Set A^2:
For any record, either $x_1 = x_3 = x_5 = False$, or $x_1 = False$ and $x_3 = x_5 = True$. The choice is made randomly.
Set A^3:
For any record, either $x_2 = x_3 = False$ and $x_5 = True$, or $x_2 = x_3 = True$ and $x_5 = False$. The choice is made randomly.

(b) Use Algorithm TRAINING SUBSETS (8.2.2) to get, for $p = 1$, 2, and 3, the subsets A_1^p, $A_2^p \ldots$, A_{10}^p.

(8.9.3) Carry out Algorithm FORTY MIN/MAX DNF FORMULAS (8.3.1) using as sets A and B the sets A^1 and A^2 of Exercise (8.9.2).

(8.9.4) Carry out Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2) using as sets A and B the sets A^1 and A^2 of Exercise (8.9.2). There are three tests T_1, T_2, and T_3, with costs $c_1 = 10$, $c_2 = 24$, and $c_3 = 15$. Test T_1 determines the values of x_1, x_2, and x_3; test T_2 finds the values of x_2,x_4, x_5, and x_6; and test T_3 provides the values of x_1, x_3, and x_5.

(8.9.5) Give details of the multipopulation case of Section 8.7 where difference vectors produce a case with two populations that is handled by optimized formulas.

(8.9.6) Carry out Algorithm VOTE DISTRIBUTION (8.5.7) for the following cases.

(a) The set A of Exercise (8.9.3) and the 40 min/max DNF formulas produced by that exercise.
(b) The set B of Exercise (8.9.3) and the 40 min/max DNF formulas produced by that exercise.
(c) The set A^* produced by Exercise (8.9.4) and the 20 optimized DNF formulas computed in that exercise for separating A^* from B.

(8.9.7) Carry out Algorithm CLASSIFICATION CONTROL (8.6.5) with $P_{\mathcal{A}} = 0.4$, $P_{\mathcal{B}} = 0.6$, $c_{err|\mathcal{A}} = 11$, and $c_{err|\mathcal{B}} = 17$. Instead of $\mathcal{F}_{\mathcal{A}}(z)$ and $\mathcal{F}_{\mathcal{B}}(z)$, use estimated distributions $\hat{\mathcal{F}}_{\mathcal{A}}(z)$ and $\hat{\mathcal{F}}_{\mathcal{B}}(z)$ generated by Exercise (8.9.6)(a) and (b).

(8.9.8) Prove the following claim for the multipopulation case. If, for all p, no record of training set A^p is weakly nested in any record of any training set A^l, $l \neq p$, then the min/max formulas of each $\mathcal{H}^p$ can always be computed. (*Hint*: Use Theorems (7.3.14) and (7.5.10), and examine the construction of the various subsets A_i^p and B_i^p.)

(8.9.9) Theorem (8.2.3) roughly says that absence of weak nestedness involving training sets induces the same property for certain training subsets. Does the converse hold? (*Hint*: Examine the construction of the subsets.)

(8.9.10) Show that $C(d^1, \ldots, d^q)$ of (8.7.15) can have local minima. (*Hint*: Prove that $C(d^1, \ldots, d^q)$ with $q = 2$ effectively is $C_{err}(z)$ of (8.6.3), and cite the example case (8.6.7)).

(8.9.11)

(a) Take A^1, A^2, and A^3 of Exercise (8.9.2) and the sets A_i^p computed by that exercise.

(b) Using the equations of Section 8.7, compute, for $p = 1, 2, 3$, the set B^p and, for $i = 1, 2, \ldots, 10$, the set B_i^p.

(c) For each p, use Algorithm FORTY MIN/MAX DNF FORMULAS (8.3.1) to compute 40 min/max DNF formulas that separate A^p from B^p, and place them into a set $\mathcal{H}^p$.

(d) Carry out Algorithm MULTICLASSIFICATION PROBABILITY (8.7.13), with input given by A^1, A^2, A^3, $\mathcal{H}^1$, $\mathcal{H}^2$, $\mathcal{H}^3$ defined above and $d^1 = d^2 = d^3 = 0$. The tie breaker prefers the optimal p with minimum index.

(e) Let the prior probabilities be $P_{\mathcal{A}^1} = 0.2$, $P_{\mathcal{A}^2} = 0.3$, and $P_{\mathcal{A}^3} = 0.5$. Define the costs of misclassification to be $c_{err|\mathcal{A}^1} = 10$, $c_{err|\mathcal{A}^2} = 25$, and $c_{err|\mathcal{A}^3} = 8$. Using the probabilities estimated in (d) above, estimate $P_{err}(d^1, d^2, d^3)$ and $C(d^1, d^2, d^3)$.

(f) Carry out Heuristic MULTICLASSIFICATION CONTROL (8.7.38) with the input data computed above.

(8.9.12) Let D be the set $D = \{(d^1, \ldots, d^q) \mid 0 \leq d^p \leq 20k+1,\ d^{min} = 0\}$ of (8.7.24). Prove that $|D| = (20k+2)^q - (20k+1)^q$. (*Hint*: Determine in how many ways m of q values $d^1, d^2, \ldots, d^q$ can be equal to 0. Determine the number of ways in which $q - m$ of q values $d^1, d^2, \ldots, d^q$ can satisfy $1 \leq d^p \leq 20k+1$. Combine the two cases, sum over the possible m values, and use the binomial theorem for $(a+b)^q$.)

(8.9.13) Extend the treatment of classification costs in Section 8.7 to the situation where, for each $1 \leq p \leq q$, $1 \leq t \leq q$, and $p \neq t$, a cost $c_{\mathcal{A}^t|\mathcal{A}^p}$ is given for misclassifying a record of $\mathcal{A}^p$ into $\mathcal{A}^t$. (*Hint*: Introduce the probability $P_{\mathcal{A}^t|\mathcal{A}^p}(d^1, \ldots, d^q)$ of misclassifying an unseen record of $\mathcal{A}^p$ into $\mathcal{A}^t$. Estimate the expected total misclassification cost $C_{\mathcal{A}^t|\mathcal{A}^p}(d^1, \ldots, d^q)$. Extend Heuristic MULTICLASSIFICATION CONTROL (8.7.38) to find a good if not optimal decision strategy.)

(8.9.14) (Optional programming project) Some of the exercises defy manual solution. For these cases, the reader should obtain commercially or publicly available software. Any search engine on the Internet will point to such software. Alternately, the reader may opt to write programs that implement the algorithms of (a)–(h) below, and then use that software

to solve the exercises. The task is easy if the reader has already created programs for the algorithms of Chapter 7.

(a) Algorithm EXPANSION OF SET (8.2.1).
(b) Algorithm TRAINING SUBSETS (8.2.2).
(c) Algorithm FORTY MIN/MAX DNF FORMULAS (8.3.1).
(d) Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2).
(e) Algorithm VOTE DISTRIBUTION (8.5.7).
(f) Algorithm CLASSIFICATION CONTROL (8.6.5).
(g) Algorithm MULTICLASSIFICATION PROBABILITY (8.7.13).
(h) Heuristic MULTICLASSIFICATION CONTROL (8.7.38).

Part IV

Advanced Reasoning

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 9

Nonmonotonic and Incomplete Reasoning

9.1 Overview

Let us pause and look at the use of logic in mathematics. There, proofs are based on deduction using the axioms of first-order logic and suitable extensions. For example, if we have the two axioms x and $x \rightarrow y$, then we can deduce y. Proofs by examples are not permitted; just exhibiting many situations where a certain claim holds, is not good enough to prove the claim in general. Nevertheless, over time it became obvious that proofs by sufficiently general examples should be allowed in some form. To integrate that notion formally into the deductive framework, the *induction axiom* was introduced: If a statement holds for some integer n_0, where typically $n_0 = 0$ or $n_0 = 1$, and if, for arbitrary $n \geq n_0$, it holds for $n+1$ provided it holds for n, then we are permitted to conclude that it holds for all $n \geq n_0$. There are additional concepts that introduce flexibility into mathematics, such as the axiom of choice, which, given any collection of nonempty sets, allows one to identify, simultaneously, one element from each set.

Until the 20th century, it seemed that deduction was the only tool needed to prove all results holding for given axioms. But then *Gödel's incompleteness theorem* established that notion to be wrong. The theorem roughly says that deduction cannot derive all true statements holding for a given set of mathematical axioms unless the set of axioms is very simple. More precisely, the incompleteness case occurs already if the axioms constitute a consistent extension of a certain number theory. That theory

specifies the concept of zero, successor, addition, and multiplication of binary numbers, and does not even call for induction. As an aside, deduction is sufficient to prove all results if the axioms are those of first-order logic.

Despite the incompleteness defect of deduction established by Gödel, there seems to be general agreement that the body of mathematics erected with deduction over the past 4000 years is the most elaborate intellectual achievement of mankind. This fact, and the multitude of results achieved by mathematical deduction in the sciences, may convince one to make deduction the core method for the design of intelligent systems. But, if one were to do so, one would commit a major error. We discuss two example situations to support that claim.

In the first example, we enter a lit room and turn off the light. Another person, sitting already in the room and not having seen us operate the switch, concludes "Somebody turned off the light." The laws of electricity say that, if the light switch goes into the off position, then the light goes out. Defining variables x and y by $x = \mathit{True}$ if the light switch goes off and $y = \mathit{True}$ if the light goes out, the law says $x \rightarrow y$. When the light goes out, the person realizes that $y = \mathit{True}$, and then uses the axiom $x \rightarrow y$ to conclude $x = \mathit{True}$, which means that the switch has been turned off. This reasoning is abduction, treated in Chapter 6. Abduction is not permitted in mathematics. Yet, when one reviews the decisions made by humans every day, it becomes evident that the vast majority of them are based on abduction.

In the second example, we learn to drive a car. There are the steering wheel and two, three, or even four pedals, and a number of switches. We do not learn to drive by studying differential equations that model steering, engine, and brake performance of the car. Instead, we learn by trying out how the car responds as we take various actions under the supervision of an instructor. From the limited number of situations encountered that way, the brain constructs a reasoning mechanism that allows us to drive the car. Such learning is covered in Chapters 7 and 8. It falls outside mathematical deduction, and thus is not permissible there. But, much of a person's ability to meet the challenges of everyday life is based on such learning, and thus it should be part of the tools for the construction of intelligent systems.

The reader may object to the above claims that mathematics does not permit abduction or reasoning from examples. The argument is likely to invoke probability theory and statistics, which, analogously to the earlier induction case, convert the various ways of reasoning to deduction. We also strive for such a conversion, but do not use tools such as probability theory and statistics. Instead, we handle abduction by attaching likelihoods to clauses, and learn from examples by extracting logic formulas. Indeed, our goal is to handle all cases more or less directly by logic formulas.

The approach used here has the advantage that the various logic for-

mulas can be seamlessly merged. But it also has a significant drawback. When additional information becomes available, earlier derived conclusions may become invalid. For example, conclusions by abduction may turn out to be wrong, or logic formulas learned from data sets may not cover all possible cases and thus may produce wrong results. In each case, the conclusion is proved to be erroneous by additional information or facts. Any logic system admitting such events is called *nonmonotonic*. The logic used in the construction of mathematics does not allow such occurrences and thus is called *monotonic*. That feature is essential for mathematics. If it were absent, we would not know if previously derived mathematical results were still valid upon addition of new axioms. In contrast, our approach does introduce the problem of nonmonotonicity.

There is a situation that, in some sense, is the opposite of nonmonotonicity. We are given the information that a certain logic result is always true, yet we cannot prove that fact using the logic axioms at hand. We hasten to add that this is not the situation treated by Gödel's incompleteness theorem, but is of the far simpler variety where a given set of axioms formulated in propositional logic is simply not sufficient to prove all logic results of interest. We call this a case of *incompleteness*. In the context of human reasoning and intelligent systems, incompleteness is a frequently encountered situation. Indeed, we typically proceed with information that we know to be partial, and add axioms as needed.

There is an interplay between nonmonotonicity and incompleteness. When we discover nonmonotonicity, we change or remove axioms so that results can no longer be proved. When we encounter incompleteness, we change or add axioms so that certain results can now be proved. In this book, we avoid changes of axioms and confine ourselves in the case of nonmonotonicity (resp. incompleteness) to removal (resp. addition) of axioms. It is even possible that an axiom is removed at some point in time due to nonmonotonicity, added later due to incompleteness, and removed subsequently once more due to nonmonotonicity. In mathematics, this would be objectionable. But in the context of intelligent systems that adapt to a changing environment, such repeated revisions are acceptable.

The situation becomes more complicated when clauses involve likelihoods. In the case of nonmonotonicity (resp. incompleteness), a conclusion can (resp. cannot) be proved from the axioms at a certain likelihood level, but according to available data, the conclusion should not be (resp. should be) provable at that level. For example, the axioms may support proof of a conclusion at level 50, but data show that the conclusion should not be provable at that level.

There is a naïve way to deal with nonmonotonicity and incompleteness. When new facts invalidate previously derived results, or when facts cannot but should be proved, we manually go over the logic formulation, isolate the inconsistency or establish the missing clauses, and make suit-

able adjustments. Chapter 5 tells how this can be done. Except for rare cases, such manual correction is impractical. Thus, we need corrective algorithms that remove or add axioms without human assistance. This chapter describes such algorithms.

Section 9.2 gives an algorithm for the extreme situation of nonmonotonicity where proof of a conclusion at any positive level is to be eliminated.

Section 9.3 deals with the corresponding extreme case of incompleteness, where proof of a conclusion at level 100 is to be achieved.

Section 9.4 provides an algorithm for the general case of nonmonotonicity, where a conclusion can be proved from the axioms at a certain likelihood level, but, according to available data, should not be provable at that level. The section also deals with the corresponding general case of incompleteness, where a conclusion cannot be proved from the axioms at a certain likelihood level, but should be provable at that level. We call these cases *uncertain nonmonotonicity* and *uncertain incompleteness* and refer to the particular subcases of Sections 9.2 and 9.3 simply as *nonmonotonicity* and *incompleteness*, respectively. Put differently, nonmonotonicity is uncertain nonmonotonicity where proof at any positive level is to be eliminated, and incompleteness is uncertain incompleteness where proof at level 100 is to be made possible.

Section 9.5 lists references for further reading.

Section 9.6 contains exercises.

Review of Uncertainty

In Chapter 6, the different likelihood values associated with the clauses of a CNF system S are denoted by $100 \geq \alpha_1 > \alpha_2 > \cdots > \alpha_n \geq 1$. Thus, any number of clauses may have the same likelihood value α_i. In this chapter, we use a slightly different definition where the likelihood value associated with each clause C^k of S is denoted by α_k. Due to this notational change, we no longer can tell how many different likelihood values are used in S. But that information is not important here.

For any level α, $1 \leq \alpha \leq 100$, define S_α to be the subsystem of S having the clauses C^k with likelihood value $\alpha_k \geq \alpha$. Let Y be any formula. According to Section 6.3, if $S_\alpha \wedge \neg Y$ is unsatisfiable, then Y can be proved to be a theorem of S at or above level α. If $S_\alpha \wedge \neg Y$ is satisfiable, then such a proof is not possible.

9.2 Nonmonotonicity

We begin the treatment of nonmonotonicity with an example arising from abduction.

Example: Incorrect Clauses due to Abduction

A CNF system S has variables r, s, x, and y and the following clauses.

$$\begin{aligned} &r \vee s \vee \neg x \quad \textit{at level } 70 \\ &r \vee \neg x \quad \textit{at level } 50 \\ &s \vee \neg y \quad \textit{at level } 25 \end{aligned} \tag{9.2.1}$$

The variables x and y represent symptoms, while r and s represent defects of some equipment. Given $x = \textit{True}$, we want to prove that defect r is present. We set $r = \textit{False}$ and apply Algorithm SOLVE UNCERTAIN SAT (6.3.4). The scheme determines unsatisfiability at level 50. Thus, defect r has been proved at level 50. Suppose we check whether the defect is really present in the equipment, and discover that it is not. Thus, we know that $r = \textit{False}$ holds.

By the very process that proved r at level 50, the known values $x = \textit{True}$ and $r = \textit{False}$ cannot be extended to a satisfying solution of S. Yet, such a solution must exist, since otherwise r can be proved at some positive level. This means that the clauses of S must be modified. By inspection, we see that deletion of the clause $r \vee \neg x \quad \textit{at level } 50$ is sufficient to produce the desired effect. Hence, we may define a new S by

$$\begin{aligned} &r \vee s \vee \neg x \quad \textit{at level } 70 \\ &s \vee \neg y \quad \textit{at level } 25 \end{aligned} \tag{9.2.2}$$

We discuss a second example. Here, the choice of clauses to be deleted is more difficult.

Example: Reduction Possibilities

A CNF system S is defined by variables x, y, z, r, and s, where x, y, and z are input variables of some system and r and s are output variables. The clauses are

$$\begin{aligned} &y \vee \neg x \\ &s \vee \neg y \\ &z \vee \neg y \\ &r \vee \neg z \end{aligned} \tag{9.2.3}$$

We are given the input value $x = \textit{True}$. We want to prove that output r must have the value $\textit{True}$. Since $x = \textit{True}$ and $r = \textit{False}$ produce unsatisfiability, the output value provably must be $r = \textit{True}$. Suppose that in the real world the output $r = \textit{True}$ does not occur when $x = \textit{True}$. Thus,

S must be changed so that $x = \mathit{True}$ and $r = \mathit{False}$ can be extended to a satisfying solution of the modified S. A simple check shows that deletion of any clause from S except $s \vee \neg y$ achieves that goal. But deletion of the clause $y \vee \neg x$ no longer would allow us to prove that $s = \mathit{True}$ when $x = \mathit{True}$, while deletion of $r \vee \neg z$ retains that feature. Thus, deletion of the latter clause is preferable, and the new S is

$$\begin{array}{l} y \vee \neg x \\ s \vee \neg y \\ z \vee \neg y \end{array} \tag{9.2.4}$$

The two examples have the following features in common. A given *True/False* value for variable x depicts a certain setting. We did not show it above, but in both cases, fixing x to the given value leaves the CNF system S satisfiable. A given *True/False* value for variable r represents the negation of a conclusion or an output value. When the value of r is enforced together with that of x, S becomes unsatisfiable. Yet, according to the real world, the case represented by the values is possible, and S should remain satisfiable. To achieve satisfiability, a clause is removed from S.

General Case of Nonmonotonicity

For the description of the general case, we assume the following situation. A real-world setting is partially described by *True/False* values for some variables of a CNF system S, say for $x_1, x_2, \ldots, x_n$. We say that the x_i values *exist* in the real world. The CNF system S is satisfiable when these values are enforced. *True/False* values for additional variables, say for r_1, $r_2, \ldots, r_p$ are also given. When the values of the r_i and x_j are assigned to S, that CNF system becomes unsatisfiable. However, it turns out that, in the real world, the values of the r_i and x_j exist. Thus, S does not model the real world correctly.

To simplify the subsequent discussion, we define any *True/False* values of variables of S to be *S-acceptable* (resp. *S-unacceptable*) if the values can (resp. cannot) be extended to a satisfying solution of S. With this terminology, the *True/False* values of the x_j are S-acceptable, while the values of the r_i and x_j are S-unacceptable. We want to remove some clauses from S so that, for the resulting CNF system S', the r_i and x_j values are S'-acceptable. Thus, S' models correctly that the r_i and x_j values exist in the real world.

Let us take a different viewpoint. Define I_+, I_-, J_+, and J_- to be index sets so that the *True/False* values for the r_i and x_j are defined by

$$(9.2.5) \qquad r_i = \begin{cases} \mathit{True} & \text{if } i \in I_+ \\ \mathit{False} & \text{if } i \in I_- \end{cases} \qquad x_j = \begin{cases} \mathit{True} & \text{if } j \in J_+ \\ \mathit{False} & \text{if } j \in J_- \end{cases}$$

These values uniquely satisfy the DNF clauses

$$(9.2.6) \qquad \begin{aligned} G &= \bigwedge_{j \in J_+} x_j \wedge \bigwedge_{j \in J_-} \neg x_j \\ H &= \bigwedge_{i \in I_+} r_i \wedge \bigwedge_{i \in I_-} \neg r_i \wedge \bigwedge_{j \in J_+} x_j \wedge \bigwedge_{j \in J_-} \neg x_j \end{aligned}$$

Negation of the DNF clauses G and H produces the following CNF clauses X and Y, respectively.

$$(9.2.7) \qquad \begin{aligned} X &= \bigvee_{j \in J_+} \neg x_j \vee \bigvee_{j \in J_-} x_j \\ Y &= \bigvee_{i \in I_+} \neg r_i \vee \bigvee_{i \in I_-} r_i \vee \bigvee_{j \in J_+} \neg x_j \vee \bigvee_{j \in J_-} x_j \end{aligned}$$

The CNF clause Y can be rewritten as the implication

$$(9.2.8) \qquad \Big(\bigwedge_{j \in J_+} x_j \wedge \bigwedge_{j \in J_-} \neg x_j \Big) \rightarrow \Big(\bigvee_{i \in I_+} \neg r_i \vee \bigvee_{i \in I_-} r_i \Big)$$

We allow for the special case where there are only r_i variables and no x_j variables. In that situation, the variables x_j and their values are omitted from (9.2.5), and G, H, X, and Y of (9.2.6)–(9.2.7) become $G = \mathit{True}$, $H = (\bigwedge_{i \in I_+} r_i) \wedge (\bigwedge_{i \in I_-} \neg r_i)$, $X = \mathit{False}$, and $Y = (\bigvee_{i \in I_+} \neg r_i) \vee (\bigvee_{i \in I_-} r_i)$. In the sequel, we omit discussion of this special case since the required changes follow in a straightforward manner from the revised definitions of G, H, X, and Y.

The fact that the x_j values are S-acceptable and that the r_i and x_j values are S-unacceptable can be restated as $S \wedge \neg X$ being satisfiable and $S \wedge \neg Y$ being unsatisfiable. Thus, X is not a theorem of S, whereas Y is a theorem. If some clauses of S have associated likelihood values, then X is not a theorem at any positive level, and Y is a theorem at some positive level. For the remainder of this section, we assume that this more general case is present. We supposed earlier that the values of the r_i and x_j exist in the real world. Since these values are precisely the *True/False* values implied by $\neg Y$, the conclusion that Y is a theorem of S at some positive level clashes with the real world. We also say that Y *is not* a theorem of the

real world at any positive level. The subsystem S' introduced above resolves this clash. Indeed, since the values of the r_i and x_j are S'-acceptable, $S' \wedge \neg Y$ is satisfiable or, equivalently, Y is not a theorem of S' at any positive level. Note that $S' \wedge \neg X$ is satisfiable since $S \wedge \neg X$ is satisfiable. Thus, X is not a theorem of S' at any positive level.

We want to select an S' that, intuitively speaking, still models the real world as well as possible. There are various ways to make this vague statement precise. We opt for an approach that reduces the selection of S' to solution of a MINSAT instance. To this end, define K to be the set of indices k of the clauses C^k of S that are considered for deletion. The set K must be sufficiently large so that deletion of all clauses C^k with $k \in K$ from S produces an S' for which the r_i and x_j values are S'-acceptable. We call this requirement the *satisfiability condition* of K.

For each $k \in K$, we represent the undesirability of removing clause C^k from S by a positive rational cost d_k. That cost is incurred if clause C^k is removed from S. The desired S' is to entail minimum total cost of the removed clauses. We find such an S' as follows.

First, we modify the clauses of S. For each $k \in K$, we define a new variable u_k and redefine C^k to be $C^k \vee u_k$. Second, we remove from each clause C^k of S, even from those with index k not in K, any associated likelihood level. The resulting clauses, denoted by $\tilde{C}^k$, define a CNF system $\tilde{S}$. For each $k \in K$, we assign the cost d_k to the value *True* of u_k. All other *True/False* values of the variables of $\tilde{S}$ have associated cost 0. If $u_k = \mathit{True}$, then clause $\tilde{C}^k$ of $\tilde{S}$ is satisfied regardless of the values assigned to the other variables. Thus, $u_k = \mathit{True}$ effectively removes the constraints imposed by clause $\tilde{C}^k$. Accordingly, assignment of *True* to all u_k with $k \in K$ corresponds to removal of all clauses $\tilde{C}^k$ with $k \in K$ from $\tilde{S}$, producing, say, $\tilde{S}'$. We assumed above that the corresponding removal of clauses C^k with $k \in K$ from S produces an S' for which the r_i and x_j values are S'-acceptable. Hence, the r_i and x_j values are $\tilde{S}'$-acceptable. We conclude that $\tilde{S}$ is satisfiable.

We solve the MINSAT instance given by $\tilde{S}$ and the assigned costs while enforcing the values of the r_i and x_j. We define the desired S' from S by deleting from S each clause C^k with $k \in K$ for which u_k has optimal value *True*.

The discussion has sidestepped the problem of specifying K and the costs d_k. We describe a reasonable selection scheme. It defines K as

$$K = \{k \mid \text{clause } C^k \text{ has at least one literal of } r_1, r_2, \ldots, r_p\} \tag{9.2.9}$$

and is based on the intuitive argument that restricting the removal of clauses C^k to $k \in K$, is more likely to produce an S' that still represents the real world in a reasonable way. The next theorem states that this K fulfills the satisfiability condition.

(9.2.10) Theorem. *Let S' be obtained from S by deletion of all clauses C^k with $k \in K$ of* (9.2.9). *Then the r_i and x_j values are S'-acceptable.*

Proof. By assumption, the x_j values are S-acceptable and, thus, S'-acceptable. By the definition of K, S' has no clause involving any variable r_i. Thus, the values of the r_i and x_j values are S'-acceptable. □

We turn to the definition of the costs d_k for the variables u_k with $k \in K$. Since clauses C^k with high (resp. low) likelihood values are more (resp. less) important, the associated cost d_k should be relatively high (resp. low). Also, clauses with large (resp. few) number of literals are likely to represent relatively loose (resp. tight) constraints, and the associated cost should be relatively low (resp. high). We combine the two considerations. For each $k \in K$, let α_k be the likelihood associated with clause C^k, and define l_k to be the number of literals of that clause. Using two rational parameters $\gamma \geq 0$ and $\delta \geq 0$, we define the cost d_k by

$$d_k = \gamma \cdot \alpha_k - \delta \cdot l_k, \quad k \in K \tag{9.2.11}$$

A reasonable choice for γ and δ seems to be $\gamma = \delta = 1$, but particular situations may call for different values.

We carry out example calculations for the abduction case of (9.2.1).

Example: Abduction Case, Once More

We rewrite the clauses of (9.2.1) to indicate each C^k.

$$\begin{aligned} C^1 &= r \vee s \vee \neg x \quad \textit{at level } 70 \\ C^2 &= r \vee \neg x \quad \textit{at level } 50 \\ C^3 &= s \vee \neg y \quad \textit{at level } 25 \end{aligned} \tag{9.2.12}$$

There is only one r_i, namely r, and only one x_j, namely x. We have $r =$ *False* and $x =$ *True*. Since literals of r occur in both clauses C^1 and C^2, but not in C^3, we have $K = \{k \mid \text{clause } C^k \text{ has at least one literal of the } r_i\}$ of (9.2.9) as $K = \{1, 2\}$. We use $d_k = \gamma \cdot \alpha_k - \delta \cdot l_k$ of (9.2.11) with $\gamma = \delta = 1$. For $k = 1$, we have likelihood level $\alpha_1 = 70$ and $l_1 = 3$ literals, so $d_1 = 1 \cdot 70 - 1 \cdot 3 = 67$. For $k = 2$, we have likelihood level $\alpha_2 = 50$ and $l_2 = 2$ literals, so $d_2 = 1 \cdot 50 - 1 \cdot 2 = 48$.

The clauses of the CNF system $\tilde{S}$ are obtained from those of S by first adding appropriate literals u_k and by deleting likelihood levels. With $K = \{1, 2\}$, the clauses of $\tilde{S}$ are

$$\begin{aligned} \tilde{C}^1 &= r \vee s \vee \neg x \vee u_1 \\ \tilde{C}^2 &= r \vee \neg x \vee u_2 \\ \tilde{C}^3 &= s \vee \neg y \end{aligned} \tag{9.2.13}$$

These clauses and the above d_k values define the MINSAT instance. The optimal solution of that instance is unique and has $u_1 = \mathit{False}$ and $u_2 = \mathit{True}$. Thus, we delete C^2 from S to obtain the desired S', which thus has the clauses

$$\begin{array}{l} r \vee s \vee \neg x \quad \mathit{at\ level}\ 70 \\ s \vee \neg y \quad \mathit{at\ level}\ 25 \end{array} \tag{9.2.14}$$

Algorithm for Nonmonotonicity

We summarize the method.

(9.2.15) Algorithm NONMONOTONICITY. *Finds clauses whose removal eliminates nonmonotonicity.*

Input: CNF system S, with $r_1, r_2, \ldots, r_p$ and $x_1, x_2, \ldots, x_n$ among its variables. *True/False* values for the r_i and x_j given by (9.2.5), for some index sets I_+, I_-, J_+, and J_-; equivalently, X and Y of (9.2.7) may be given. The x_j values are S-acceptable, while the r_i and x_j values are S-unacceptable; equivalently, X is not a theorem of S at any positive level, while Y is a theorem at some positive level.
Optional: Index set K of clauses C^k of S that are to be considered for deletion. The set K must be sufficiently large so that deletion of all clauses C^k with $k \in K$ from S produces an S' for which the values of the r_i and x_j are S'-acceptable. Costs d_k, $k \in K$, expressing the undesirability of deletion. (A large cost indicates that deletion is highly undesirable.)
If K and the d_k are not supplied, the algorithm selects K and uses rational parameters $\gamma \geq 0$ and $\delta \geq 0$ to compute costs d_k. The parameters may be given as part of the input, or the algorithm selects default values.

Output: An index set $K' \subseteq K$ and the CNF system S' derived from S by deletion of the clauses C^k with $k \in K'$. The r_i and x_j values are S'-acceptable; equivalently, Y is not a theorem of S' at any positive level. (Trivially, the x_j values are S'-acceptable, and X is not a theorem of S' at any level. In terms of the costs d_k of removing clauses, S' is the best subsystem of S with the desired features.)

Requires: Algorithm SOLVE MINSAT.

Procedure:

1. If the index set K and the costs d_k are not supplied: Using (9.2.9), (9.2.11), and parameters γ and δ, compute set K and costs d_k. If the parameters are not supplied, use default values $\gamma = \delta = 1$.
2. Derive the following MINSAT instance from S. For each $k \in K$, define a new variable u_k, and replace C^k by $C^k \vee u_k$. Remove from each clause C^k of S, even from those with index k not in K, any associated likelihood level. The resulting clauses define a CNF system $\tilde{S}$. For

each $k \in K$, assign the cost d_k to the value *True* of u_k. All other costs of *True/False* values are 0.

3. Solve the MINSAT instance given by the CNF system $\tilde{S}$ and the associated costs while enforcing the r_i and x_j values. Define K' to be the set of $k \in K$ for which u_k has optimal value *True*. Remove the clauses C^k with $k \in K'$ from S. Let S' be the resulting CNF system. Output K' and S', and stop.

Sometimes, nonmonotonicity involves clauses whose likelihood exceeds a specified level. We refer to this case as *uncertain nonmonotonicity*. It is covered in Section 9.4.

Algorithm for Unsatisfiability

The ideas leading to Algorithm NONMONOTONICITY (9.2.15) can also be used to derive from an unsatisfiable CNF system S a satisfiable subsystem S' by removal of clauses. Indeed, we only need to omit the references to the r_i and x_j variables and to X and Y from Algorithm NONMONOTONICITY (9.2.15). Here is the method.

(9.2.16) Algorithm UNSATISFIABILITY. *Finds clauses whose removal eliminates unsatisfiability.*

Input: Unsatisfiable CNF system S.
Optional: Index set K of clauses C^k of S that are to be considered for deletion. The set K must be sufficiently large so that deletion of all clauses C^k with $k \in K$ from S produces a satisfiable CNF system. Costs d_k, $k \in K$, expressing the undesirability of deletion. (A large cost indicates that deletion is highly undesirable.)
If K and the d_k are not supplied, the algorithm selects K and uses rational parameters $\gamma \geq 0$ and $\delta \geq 0$ to compute costs d_k. The parameters may be given as part of the input, or the algorithm selects default values.

Output: An index set $K' \subseteq K$ and the satisfiable CNF system S' derived from S by deletion of the clauses C^k with $k \in K'$. (In terms of the costs d_k of removing clauses, S' is the best subsystem of S with the desired feature.)

Requires: Algorithm SOLVE MINSAT.

Procedure:

1. If the index set K and the costs d_k are not supplied: Define K as the index set of all clauses of S, and use (9.2.11) and parameters γ and δ to compute costs d_k. If the parameters are not supplied, use default values $\gamma = \delta = 1$.
2. Derive the following MINSAT instance from S. For each $k \in K$, define a new variable u_k, and add the literal u_k to C^k. Second, remove from each clause C^k of S, even from those with index k not in K, any

associated likelihood level. The resulting clauses define a CNF system $\tilde{S}$. For each $k \in K$, assign the cost d_k to the value *True* of u_k. All other costs of *True/False* values are 0.

3. Solve the MINSAT instance given by the CNF system $\tilde{S}$ and the associated costs. Define K' to be the set of $k \in K$ for which u_k has optimal value *True*. Remove the clauses C^k with $k \in K'$ from S. Let S' be the resulting CNF system. Output K' and S', and stop.

We turn to the problem of incompleteness of axioms.

9.3 Incompleteness

We start with an example involving likelihoods.

Example: Missing Abduction Clause

A CNF system S is given by

$$\begin{aligned} &r \vee s \vee \neg x \quad \textit{at level } 70 \\ &s \vee \neg y \quad \textit{at level } 25 \end{aligned} \tag{9.3.1}$$

The variables x and y represent symptoms, and r and s represent defects. We know that symptom x is present, so $x = \textit{True}$. It is easily checked that defect r cannot be proved at any positive level. Suppose that, in the real world, defect r is present. We want to add this conclusion to S. Evidently, if we add the clause $r \vee \neg x$, then r can be proved when $x = \textit{True}$. We add the clause $r \vee \neg x$ to S and thus get an expanded CNF system S' with clauses

$$\begin{aligned} &r \vee s \vee \neg x \quad \textit{at level } 70 \\ &s \vee \neg y \quad \textit{at level } 25 \\ &r \vee \neg x \end{aligned} \tag{9.3.2}$$

Note that the new clause has no associated likelihood level since it is valid without likelihood qualification. In Section 9.4, we do assign a likelihood level to the added clause, in a situation called *uncertain incompleteness*.

We discuss a second example. It demonstrates that the added clause may cause undesirable unsatisfiability for a certain case.

Example: Undesirable Unsatisfiability

A CNF system S has input variables x, y, and z and output variables r and s. The clauses are

$$\begin{aligned} &y \vee \neg x \\ &r \vee s \vee \neg y \\ &z \vee \neg y \\ &\neg r \vee \neg x \vee \neg y \end{aligned} \tag{9.3.3}$$

We are given $x = y = \mathit{True}$ and want to prove $r = \mathit{True}$. Thus, we enforce $x = y = \mathit{True}$ and $r = \mathit{False}$ in (9.3.3) and check satisfiability. It is readily checked that a satisfying solution exists, so $r = \mathit{True}$ cannot be proved.

The DNF clause

$$x \wedge y \wedge \neg r \tag{9.3.4}$$

evaluates to *True* precisely when $x = y = \mathit{True}$ and $r = \mathit{False}$. Negation of the DNF clause produces the CNF clause

$$Y = r \vee \neg x \vee \neg y \tag{9.3.5}$$

which is equivalent to the implication

$$(x \wedge y) \rightarrow r \tag{9.3.6}$$

By the derivation of Y and of the equivalent implication, these formulas are not theorems of S, but should be, according to the real world. We remedy that defect of S by adding Y. The resulting CNF system S' is given by

$$\begin{aligned} &y \vee \neg x \\ &r \vee s \vee \neg y \\ &z \vee \neg y \\ &\neg r \vee \neg x \vee \neg y \\ &r \vee \neg x \vee \neg y \end{aligned} \tag{9.3.7}$$

It turns out that the seemingly straightforward addition of Y has introduced a flaw into the formulation. We know that the values $x = y = \mathit{True}$ exist in the real world, so $S' \wedge x \wedge y = S \wedge Y \wedge x \wedge y$ should be satisfiable. However, the clauses $\neg r \vee \neg x \vee \neg y$ and $Y = r \vee \neg x \vee \neg y$ of S' imply $\neg x \wedge \neg y$, which is not satisfiable when $x = y = \mathit{True}$. We repair that defect by removing the clause $\neg r \vee \neg x \vee \neg y$ from S', getting a final S' as

$$\begin{aligned} &y \vee \neg x \\ &r \vee s \vee \neg y \\ &z \vee \neg y \\ &r \vee \neg x \vee \neg y \end{aligned} \tag{9.3.8}$$

General Case of Incompleteness

In the general case, *True/False* values are at hand for some variables of a CNF system S, say for $x_1, x_2, \ldots, x_n$. *True/False* values for additional variables of S, say for $r_1, r_2, \ldots, r_p$ are also given. The values for the r_i and x_j are S-acceptable. In the real world, the values of the x_j values exist, but the values for the r_i and x_j values are not possible. Thus, S does not model the real world correctly.

As in (9.2.5), let the values for the r_i and x_j be specified by

$$(9.3.9) \qquad r_i = \begin{cases} \mathit{True} & \text{if } i \in I_+ \\ \mathit{False} & \text{if } i \in I_- \end{cases} \qquad x_j = \begin{cases} \mathit{True} & \text{if } j \in J_+ \\ \mathit{False} & \text{if } j \in J_- \end{cases}$$

We argue as for (9.2.5)–(9.2.8). Thus, the values for the r_i and x_j do not satisfy the CNF clauses X and Y given by

$$(9.3.10) \qquad \begin{aligned} X &= \bigvee_{j \in J_+} \neg x_j \vee \bigvee_{j \in J_-} x_j \\ Y &= \bigvee_{i \in I_+} \neg r_i \vee \bigvee_{i \in I_-} r_i \vee \bigvee_{j \in J_+} \neg x_j \vee \bigvee_{j \in J_-} x_j \end{aligned}$$

The values also do not satisfy the following implication, which is equivalent to Y.

$$(9.3.11) \qquad \Big(\bigwedge_{j \in J_+} x_j \wedge \bigwedge_{j \in J_-} \neg x_j \Big) \rightarrow \Big(\bigvee_{i \in I_+} \neg r_i \vee \bigvee_{i \in I_-} r_i \Big)$$

At the same time, the r_i and x_j values are S-acceptable. Hence, Y is not a theorem of S at any positive level. However, according to the real world, the r_i and x_j are not possible. We summarize the latter situation by saying that Y *is* a theorem of the real world.

There is a simple remedy. We add the CNF clause Y to S, getting the CNF system $S \wedge Y$, for which Y is a theorem at level 100. That naïve approach overlooks an opportunity to detect and eliminate a potential defect of S using X of (9.3.10). We know that the x_j values exist in the real world, and that Y is a theorem in the real world. Thus, $S \wedge \neg X \wedge Y$ should be satisfiable. Suppose this is not the case. We remedy the defect by invoking Algorithm UNSATISFIABILITY (9.2.16). The input CNF system is $S \wedge \neg X \wedge Y$. The set K is computed as in (9.2.9). That is, using C^k to denote clauses of S as before, we have

$$(9.3.12) \qquad K = \{k \mid \text{clause } C^k \text{ has at least one literal of } r_1, r_2, \ldots, r_p\}$$

Note that neither the index of clause Y nor the index of any clause of $\neg X$ is in K. Algorithm UNSATISFIABILITY (9.2.16) requires that deletion of all clauses C^k with index $k \in K$ from the input system, here $S \wedge \neg X \wedge Y$, produces a satisfiable CNF system. The next theorem shows that K of (9.3.12) meets this requirement.

(9.3.13) Theorem. *Let S' be obtained from S by deletion of all clauses C^k with $k \in K$ of* (9.3.12). *Then $S' \wedge \neg X \wedge Y$ is satisfiable.*

Proof. By assumption, the values of the x_j are S-acceptable and, thus, $(S' \wedge \neg X)$-acceptable. By the definition of K, no clause of $S' \wedge \neg X$ involves any variable r_i. Thus, we may assign a *True/False* value to an arbitrarily selected r_i so that Y is satisfied. We conclude that $S' \wedge \neg X \wedge Y$ is satisfiable. □

We still must define the input cost d_k for Algorithm UNSATISFIABILITY (9.2.16). For each $k \in K$, let α_k be the likelihood value associated with clause C^k, and define l_k to be the number of literals of that clause. Select reasonable values for parameters $\gamma \geq 0$ and $\delta \geq 0$; for example, take $\gamma = \delta = 1$. As in (9.2.11), the cost d_k is computed by

$$d_k = \gamma \cdot \alpha_k - \delta \cdot l_k, \quad k \in K \tag{9.3.14}$$

Given these inputs, Algorithm UNSATISFIABILITY (9.2.16) outputs a subset K' of K and a satisfiable CNF system $S' \wedge \neg X \wedge Y$ where S' is derived from S by deletion of clauses C^k with index $k \in K' \subseteq K$. We may also say, equivalently, that the x_j values are $(S' \wedge Y)$-acceptable. By the derivation, the x_j values are S'-acceptable, while the r_i and x_j values are S'-unacceptable. Equivalently, X is S'-acceptable at any positive level, while Y is a theorem of S' at level 100.

Algorithm for Incompleteness

The next algorithm summarizes the steps.

(9.3.15) Algorithm INCOMPLETENESS. *Constructs clause whose addition eliminates incompleteness.*

Input: CNF system S, with $r_1, r_2, \ldots, r_p$ and $x_1, x_2, \ldots, x_n$ among its variables. *True/False* values for the r_i and x_j given by (9.3.9), for some index sets I_+, I_-, J_+, and J_-; equivalently, X and Y of (9.3.10) may be given. The values of the r_i and x_j are S-acceptable; equivalently, Y is not a theorem of S at any positive level. (This implies that X of (9.3.10) is not a theorem of S at any positive level.)
Optional: Rational parameters $\gamma \geq 0$ and $\delta \geq 0$. The parameters are used by Algorithm UNSATISFIABILITY (9.2.16) to compute costs d_k.

Output: A CNF system S' composed of Y and a clause subsystem of S such that the x_j are S'-acceptable, while the r_i and x_j values are S'-unacceptable; equivalently, X is not a theorem of S' at any positive level, while Y is a theorem of S' at level 100.

Requires: Algorithm UNSATISFIABILITY (9.2.16).

Procedure:

1. (Eliminate unsatisfiability of $S \wedge \neg X \wedge Y$ if present.) If $S \wedge \neg X \wedge Y$ is unsatisfiable: If parameters γ and δ are not given, define default values $\gamma = \delta = 1$. Using (9.3.12), (9.3.14), and parameters γ and δ, compute set K and costs d_k. Carry out Algorithm UNSATISFIABILITY (9.2.16) with input CNF system $S \wedge \neg X \wedge Y$ and set K and costs d_k at hand. Let $S' \wedge \neg X \wedge Y$ be the output CNF system of the algorithm.
2. Redefine S' to become $S' \wedge Y$. Output S', and stop.

We emphasize that the added Y has no associated likelihood level. However, there are situations where Y should be assigned a likelihood level. We refer to this case as *uncertain incompleteness*. The next section covers that case, together with uncertain nonmonotonicity.

9.4 Uncertain Nonmonotonicity and Incompleteness

The algorithms for remedying nonmonotonicity and incompleteness in Sections 9.2 and 9.3 assume that S is to be modified so that the current situation is correctly handled. Specifically, if we detect nonmonotonicity, we delete enough clauses from S to get an S' so that the *True/False* values for the r_i and x_j given by (9.2.5) are S'-acceptable; equivalently, Y of (9.2.7) is not a theorem of S' at any positive level. Similarly, if we detect incompleteness, we add a clause Y without likelihood level to S, getting S'. Thus, Y is a theorem of S' at level 100. Both changes are appropriate since, for the present situation, we know with certainty whether or not Y should be a theorem according to the real world.

Likelihood Level for Theorems

The approach of Sections 9.2. and 9.3 may be inappropriate when we want to modify S for future uses, since we may not know with certainty whether or not Y should be a theorem at that time. Indeed, based on prior data, we may estimate that, in the future, Y should or should not be a theorem at some given likelihood level α. We want to modify S

accordingly. The algorithms achieving these goal use r_i and x_j values and the related formulas X and Y given earlier. We list them here for ease of reference.

$$r_i = \begin{cases} \mathit{True} & \text{if } i \in I_+ \\ \mathit{False} & \text{if } i \in I_- \end{cases} \qquad x_j = \begin{cases} \mathit{True} & \text{if } j \in J_+ \\ \mathit{False} & \text{if } j \in J_- \end{cases} \tag{9.4.1}$$

$$X = \bigvee_{j \in J_+} \neg x_j \vee \bigvee_{j \in J_-} x_j$$
$$Y = \bigvee_{i \in I_+} \neg r_i \vee \bigvee_{i \in I_-} r_i \vee \bigvee_{j \in J_+} \neg x_j \vee \bigvee_{j \in J_-} x_j \tag{9.4.2}$$

We note, once more, that Y is equivalent to the implication

$$\Big(\bigwedge_{j \in J_+} x_j \wedge \bigwedge_{j \in J_-} \neg x_j \Big) \rightarrow \Big(\bigvee_{i \in I_+} \neg r_i \vee \bigvee_{i \in I_-} r_i \Big) \tag{9.4.3}$$

The first algorithm deals with *uncertain nonmonotonicity*, where a proof at or above a given level α is possible but not desired. The second algorithm handles *uncertain incompleteness*, where a proof at level α is desired but not possible.

We restate the two cases addressed by the algorithms. For given $1 \leq \alpha \leq 100$, define *True/False* values of variables of S to be *S-acceptable (resp. S-unacceptable) at level* α if the values can (resp. cannot) be extended to a solution that satisfies all clauses C^k of S having likelihood value $\alpha_k \geq \alpha$. In the case of uncertain nonmonotonicity, the x_j values are S-acceptable at level α, while the r_i and x_j values are S-unacceptable at level α but should be S-acceptable at that level. In the case of uncertain incompleteness, the r_i and x_j values are S-acceptable at level α, but should be S-unacceptable at that level; at the same time, the x_j values should remain S-acceptable at level α.

Algorithm for Uncertain Nonmonotonicity

Let S_α be the CNF system defined by the clauses C^k of S having likelihood value $\alpha_k \geq \alpha$. The algorithm for uncertain nonmonotonicity applies Algorithm NONMONOTONICITY (9.2.15) to S_α to derive an S'_α for which Y is not a theorem at any positive level. The CNF system S'_α replaces S_α in S. The resulting S' does not have Y as a theorem at or above level α.

(9.4.4) Algorithm UNCERTAIN NONMONOTONICITY. *Finds clauses whose removal eliminates uncertain nonmonotonicity.*

Input: CNF system S, with $r_1, r_2, \ldots, r_p$ and $x_1, x_2, \ldots, x_n$ among its variables. Likelihood level α, where $1 \leq \alpha \leq 100$. *True/False* values for the r_i and x_j given by (9.4.1), for some index sets I_+, I_-, J_+, and J_-; equivalently, X and Y of (9.4.2) may be given. The x_j values are S-acceptable at level α, while the r_i and x_j values are S-unacceptable at that level; equivalently, X is not a theorem of S at or above level α, while Y is a theorem of S at or above level α.
Optional: Index set K of clauses C^k of S that are to be considered for deletion. The set K must only contain indices k of clauses C^k having likelihood value $\alpha_k \geq \alpha$, and the set must be sufficiently large so that deletion of all clauses C^k with $k \in K$ from S produces an S' for which the values of the r_i and x_j are S'-acceptable at level α. Costs d_k, $k \in K$, expressing the undesirability of deletion. (A large cost indicates that deletion is highly undesirable.)
If K and the d_k are not supplied, the algorithm selects K and uses rational parameters $\gamma \geq 0$ and $\delta \geq 0$ to compute costs d_k. The parameters may be given as part of the input, or the algorithm selects default values.

Output: An index set $K' \subseteq K$ and the CNF system S' derived from S by deletion of the clauses C^k with $k \in K'$. The r_i and x_j values are S'-acceptable at level α; equivalently, Y is not a theorem of S' at or above level α. (Trivially, the x_j values are S'-acceptable at level α, and X is not a theorem of S' at or above level α. In terms of the costs d_k of removing clauses, S' is the best subsystem of S with the desired features.)

Requires: Algorithm NONMONOTONICITY (9.2.15).

Procedure:

1. Let S_α be the CNF system that has all clauses C^k of S with likelihood value $\alpha_k \geq \alpha$. Carry out Algorithm NONMONOTONICITY (9.2.15) with the following input. CNF system S_α; the r_i and x_j values given by (9.4.1), or X and Y of (9.4.2); if supplied, set K and costs d_k, or parameters γ and δ.
2. Using K' of the output of Algorithm NONMONOTONICITY (9.2.15), define S' from S by deleting all clauses C^k with $k \in K'$. Output K' and S', and stop.

Algorithm for Uncertain Incompleteness

In the case of uncertain incompleteness, we not only must introduce into S the clause Y with level α, but also must assure that $S \wedge Y$ is satisfiable. Here is the scheme.

(9.4.5) Algorithm UNCERTAIN INCOMPLETENESS. *Constructs clause whose addition eliminates uncertain incompleteness.*

Input: CNF system S, with $r_1, r_2, \ldots, r_p$ and $x_1, x_2, \ldots, x_n$ among its variables. Likelihood level α, where $1 \leq \alpha \leq 100$. *True/False* values for the r_i and x_j given by (9.4.1), for some index sets I_+, I_-, J_+, and J_-; equivalently, X and Y of (9.4.2) may be given. The r_i and x_j values are S-acceptable at level α; equivalently, Y is not a theorem of S at or above level α. (This implies that X is not a theorem of S at or above level α.) Optional: Rational parameters $\gamma \geq 0$ and $\delta \geq 0$. (The parameters are used by Algorithms INCOMPLETENESS (9.3.15) and UNSATISFIABILITY (9.2.16) to compute costs d_k.)

Output: A CNF system S' composed of Y and a maximal clause subsystem of S, such that the following holds. The x_j are S'-acceptable at level α; the r_i and x_j values are S'-unacceptable at that level, but not at any higher level. Equivalently, X is not a theorem of S' at or above level α, while Y is a theorem at that level. In addition, S' is satisfiable.

Requires: Algorithms INCOMPLETENESS (9.3.15), SOLVE SAT, and UNSATISFIABILITY (9.2.16).

Procedure:

1. Let S_α be the CNF system that has all clauses C^k of S with likelihood $\alpha_k \geq \alpha$. Do Algorithm INCOMPLETENESS (9.3.15) with the following input. CNF System S_α; the r_i and x_j values of (9.4.1), or X and Y of (9.4.2); if supplied, parameters γ and δ. The output of the algorithm consists of CNF system S'. Revise S' by declaring Y of S' to have likelihood level α, and by adding all clauses of S with $\alpha_k < \alpha$. (Thus, the clauses added to S' are precisely the clauses of S that are not in S_α.)
2. Check with Algorithm SOLVE SAT if S' is satisfiable. If this is the case, output S', and stop.
3. (Reduce S' to a satisfiable CNF system by deleting some clauses C^k with $\alpha_k < \alpha$.) Redefine S to become S'. Carry out Algorithm UNSATISFIABILITY (9.2.16). The input consists of CNF system S, set $K = \{k \mid \alpha_k < \alpha\}$, costs d_k of (9.2.11), and, if supplied, parameters γ and δ. The algorithm produces a CNF system S'. Output that system, and stop.

More complicated situations are possible where one wants to change the level at which a theorem can be proved. For example, a theorem may be presently proved at level 70, but should be provable only at level 25. All cases can be handled by judicious use of Algorithms UNCERTAIN NONMONOTONICITY (9.4.4) and UNCERTAIN INCOMPLETENESS (9.4.5). For the cited example, one first eliminates proof at or above level $25-1 = 24$ using Algorithm UNCERTAIN NONMONOTONICITY (9.4.4) and then, if needed, enables proof at level 25 with Algorithm UNCERTAIN

INCOMPLETENESS (9.4.5). We omit a detailed discussion of the possible cases, since the solution approach is straightforward for each situation.

9.5 Further Reading

A good starting point for further reading about nonmonotonicity and related topics is Russell and Norvig (2003). That reference also covers related *truth maintenance systems* for updating knowledge.

9.6 Exercises

The calculations required by the exercises may be accomplished with the software of Exercise (9.6.8).

(9.6.1) Carry out Algorithm NONMONOTONICITY (9.2.15) for the CNF systems S given by

$$r \vee s \vee \neg x \quad \textit{at level } 70$$
$$s \vee \neg y \quad \textit{at level } 25$$
$$\neg x \vee y \quad \textit{at level } 25$$
$$r \vee \neg y \quad \textit{at level } 25$$

The r_i and x_j and their values are as follows. The variable r is the sole r_i variable, with value $r = \textit{False}$, and the variable x is the sole x_j variable, with value $x = \textit{True}$. The algorithm uses the default values for γ and δ.

(9.6.2) Carry out Algorithm UNSATISFIABILITY (9.2.16) where S is given by

$$r \vee \neg s \vee t \quad \textit{at level } 100$$
$$\neg s \vee \neg t \quad \textit{at level } 90$$
$$r \vee s \quad \textit{at level } 75$$
$$\neg r \vee s \quad \textit{at level } 45$$

The variables r and s are the r_i variables, there are no x_j variables, and $Y = \neg r \vee \neg s$. The algorithm uses the default values for γ and δ.

(9.6.3) Define S by the following clauses.

$$p \vee q \vee r \quad \textit{at level } 100$$
$$\neg p \vee \neg q \vee \neg x \quad \textit{at level } 85$$
$$p \vee \neg r \quad \textit{at level } 70$$
$$\neg p \vee x \quad \textit{at level } 60$$

You are given the implication $\neg x \rightarrow \neg r$, which is equivalent to $Y = x \vee \neg r$. Thus, r is the sole r_i variable, and x is the sole x_j variable.

(a) Determine the highest level α at which Y can be proved to be a theorem.

(b) Use Algorithm UNCERTAIN NONMONOTONICITY (9.4.4) to modify S so that Y can no longer be proved at the level found in part (a).

(9.6.4) For the case of S and Y of (9.6.3), use Algorithm UNCERTAIN INCOMPLETENESS (9.4.5) to find S' so that Y can be proved at level 90.

(9.6.5) Develop an algorithm for decreasing the level at which a statement Y can be proved. Use Algorithms UNCERTAIN NONMONOTONICITY (9.4.4) and UNCERTAIN INCOMPLETENESS (9.4.5) as subroutines.

(9.6.6)

(a) Devise a recursive method that is an alternative to Algorithm NONMONOTONICITY (9.2.15). In the recursive step of the method, Algorithm SOLVE MINCLS UNSAT (3.3.14) identifies a set of candidate clauses for deletion. From that set, one selects one clause for deletion, using the cost d_k of (9.2.11) as criterion.

(b) Compare the effectiveness of the method of (a) with that of Algorithm NONMONOTONICITY (9.2.15), using the total cost of the deleted clauses as criterion.

(9.6.7)

(a) Expand Algorithms NONMONOTONICITY (9.2.15) and UNSATISFIABILITY (9.2.16) for the case where some clauses are labeled "always correct." The revised algorithms are not allowed to delete such clauses.

(b) How can the notion of "always correct" clauses be used to avoid cycles of addition/deletion of the same, correct clause in repeated applications of the algorithms of this chapter?

(9.6.8) (Optional programming project) There are two options if the reader wishes to avoid manual solution of some of the exercises. First, the reader may obtain commercially or publicly available software for at least some of the problems. Any search engine on the Internet will point to such software. Second, the reader may opt to write programs that implement the algorithms of (a)–(e) below, and then use that software to solve the exercises. The task is easy if the reader has already created programs for SAT and MINSAT as described in Exercise (2.9.13).

(a) Algorithm NONMONOTONICITY (9.2.15).

(b) Algorithm UNSATISFIABILITY (9.2.16).

(c) Algorithm INCOMPLETENESS (9.3.15).

(d) Algorithm UNCERTAIN NONMONOTONICITY (9.4.4).

(e) Algorithm UNCERTAIN INCOMPLETENESS (9.4.5).

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 10

Question-and-Answer Processes

10.1 Overview

Communication between humans ranges from short, one-sided statements such as "Excuse me" to complex sequences of two-sided conversations and discussions that can go on for hours. Present software technology severely constrains the extent to which computers can take part in such exchanges. Nevertheless, a few types of exchanges can already be processed by computers. In this chapter, we cover a class of such exchanges. In the typical case, a person requests that some conclusions be proved or disproved by a computer system. To decide the cases, the system repeatedly asks questions that are answered by the person. The system then uses the answers to produce the desired results. We call any such exchange a *question-and-answer process*, for short QA *process*.

There is another type of question-and-answer exchange where the person asks a question and the system looks up the answer in a data base. That case is handled by data base query languages and is not addressed here.

We assume that all exchanges are done via keyboard and are displayed on a screen. The process can be expanded to speaking/hearing when suitable software translates voice into text and conversely. That step is part of Natural Language Processing and is not covered here.

As a matter of format, we always assume that each QA process involves a person and a computer system. But it is possible that the role of the person is assumed by a second computer system.

Section 10.2 explains the computational steps of the basic QA process, using a simplified medical diagnosis case as an example. The knowledge about such diagnosis is encoded in a CNF system. The person of the QA process wants to determine which disease a patient suffers from. The computer system asks the person to determine symptom values by some tests, and uses the answers to derive a diagnosis by theorem proving.

Section 10.3 introduces definitions and assumptions for the general QA process. The knowledge is encoded in a CNF system S. The values *True*, *False*, *Absent*, and *Unavailable* are the possible answers that the person may supply. The person determines answers by carrying out tests that have been proposed by the system. Each test entails a cost, so a judicious selection of tests is necessary if the acquisition of information is to be cost-effective.

It turns out that all conclusions of S of interest can be restated as conclusions of a CNF system S' that is derived from S by fixing variables to the *True/False/Absent/Unavailable* values supplied by the person. Section 10.4 describes the derivation of S'.

Define the set of conclusions that are of interest to be the goal set. Section 10.5 deals with the proof of conclusions of the goal set. For each conclusion, we try to achieve one of two results. First, we attempt to establish the conclusion to be a theorem of S'. Second, if a conclusion cannot be proved to be a theorem, we try to show that, no matter what additional information may become available, the conclusion cannot be established to be a theorem of S'. If we succeed in the second case, we say that the conclusion is futile.

As conclusions are decided in the QA process, the goal set must be adapted. The person participating in the QA process may also modify the goal set at any time, thus redirecting the QA process at will. Section 10.6 covers this handling of the goal set.

At the end of the QA process, we compute better ways of proving conclusions by a more efficient use of tests. The results are encoded in so-called low-cost assignments. Section 10.7 describes an algorithm for computing these assignments.

If one cannot prove each conclusion of the goal set or show it to be futile, the computer system must ask for more information. Section 10.8 explains how this can be done by selection of cost-effective tests via low-cost assignments.

Section 10.9 assembles the above steps to an algorithm for the entire QA process.

Section 10.10 sketches the computation of explanations that the person might ask for as part of the QA process.

Sections 10.11 discusses a variation of the QA process involving optimization. Typical examples are systems for regulatory compliance, where one wants to compute best solutions while obeying all regulations.

Section 10.12 describes how logic formulas learned from data with algorithms of Chapter 7 may be used in QA processes.

Section 10.13 suggests references for further reading.

Section 10.14 contains exercises.

10.2 Basic Process

We demonstrate the computational steps of the basic QA process using a simplified case of medical diagnosis.

Example

A doctor wants to determine which of three diseases d_1, d_2, and d_3 a patient suffers from. The symptom variables are x_1, x_2, x_3, and x_4. The symptoms and diseases are related by the following implications.

$$\begin{aligned} &(x_1 \wedge x_2) \rightarrow d_1 \\ &\neg x_3 \rightarrow (\neg d_1 \vee d_2) \\ &(x_1 \wedge x_4) \rightarrow d_3 \end{aligned} \tag{10.2.1}$$

Conversion to CNF results in the following clauses, which define S.

$$\begin{aligned} &\neg x_1 \vee \neg x_2 \vee d_1 \\ &x_3 \vee \neg d_1 \vee d_2 \\ &\neg x_1 \vee \neg x_4 \vee d_3 \end{aligned} \tag{10.2.2}$$

By an initial examination of the patient, the doctor has determined that $x_1 = \mathit{True}$. Given that information, the computer system tries to prove d_1, d_2, and d_3. That is, the system decides which of the implications

$$\begin{aligned} &x_1 \rightarrow d_1 \\ &x_1 \rightarrow d_2 \\ &x_1 \rightarrow d_3 \end{aligned} \tag{10.2.3}$$

are theorems of S. In CNF, the implications are

$$\begin{aligned} Y_1 &= \neg x_1 \vee d_1 \\ Y_2 &= \neg x_1 \vee d_2 \\ Y_3 &= \neg x_1 \vee d_3 \end{aligned} \tag{10.2.4}$$

Thus, the system determines, for $k = 1, 2, 3$, whether $S \wedge \neg Y_k$ is satisfiable. It is easy to check that all $S \wedge \neg Y_k$ are satisfiable, so no Y_k is a theorem of S, and no disease can be determined.

The system decides on a test that can determine the *True/False* value of x_2. The doctor carries out the test and obtains $x_2 = \mathit{True}$. Accordingly, the system redefines the Y_k as

$$\begin{aligned} Y_1 &= \neg x_1 \vee \neg x_2 \vee d_1 \\ Y_2 &= \neg x_1 \vee \neg x_2 \vee d_2 \\ Y_3 &= \neg x_1 \vee \neg x_2 \vee d_3 \end{aligned} \tag{10.2.5}$$

and carries out theorem proving for the new Y_k. It turns out that $S \wedge \neg Y_1$ is unsatisfiable, while both $S \wedge \neg Y_2$ and $S \wedge \neg Y_3$ are satisfiable. Thus, disease d_1 has been proved, while d_2 and d_3 are still open.

Suppose d_1 is a relatively mild disease, while d_2 and d_3 are serious maladies. Thus, the doctor requests that the diagnostic process be continued. This time, the system selects a test that gives the values of x_3 and x_4. The doctor runs the test and obtains $x_3 = x_4 = \mathit{False}$. Since disease d_1 has been settled, the system ignores Y_1 and revises Y_2 and Y_3, getting

$$\begin{aligned} Y_2 &= \neg x_1 \vee \neg x_2 \vee x_3 \vee x_4 \vee d_2 \\ Y_3 &= \neg x_1 \vee \neg x_2 \vee x_3 \vee x_4 \vee d_3 \end{aligned} \tag{10.2.6}$$

This time, $S \wedge \neg Y_2$ is unsatisfiable, while $S \wedge \neg Y_3$ is satisfiable. Thus, disease d_2 is present, while d_3 has not been proved. Since there are no symptoms left to investigate, the QA process terminates with these conclusions.

Features of the Iterative Process

Key elements of each iteration are the acquisition of additional information and subsequent evaluation by theorem proving, where one tries to prove each conclusion of a given goal set. The iterations continue until each conclusion has been proved or shown to be futile.

Declare a *question-and-answer method*, for short QA *method*, to be any algorithm carrying out a version of the iterative process. For example, exhaustive questioning coupled with theorem proving, as done in the above example, constitute a QA method, albeit not an attractive one. Instead, we want QA methods that produce the desired results while holding down the total cost of acquiring answers for questions. We say "holding down total cost" instead of the more desirable "holding total cost to a minimum" for the following reason. Due to uncertainties of the possible answers, the minimum total cost would have to be defined as a minimum expected cost. Computation of that expected cost would require knowledge of probability distributions that often cannot be estimated from available data with reasonable accuracy.

The subsequent sections describe a QA method that holds down total cost. We first introduce relevant definitions.

10.3 Definitions

We assume that the knowledge relevant for the situation at hand is encoded in a CNF system S whose clauses may have likelihood values assigned.

Variables and Values

Among the variables of S are $d_1, d_2, \ldots, d_p$ and $x_1, x_2, \ldots, x_n$. Each variable d_k represents a conclusion and may have the value *True* or *False*. Each variable x_j represents a fact and may take on the value *True*, *False*, *Absent*, or *Unavailable*. The interpretation of *Absent* and *Unavailable* is consistent with that of Section 7.2. Thus, the value *Absent* tells that a *True/False* can be obtained in principle, but cannot be supplied for some reason that is not intrinsic to the case at hand. Example reasons are ignorance of the person who is to supply an answer, and breakdown of test equipment. In contrast, *Unavailable* means that a *True/False* value is unimportant or irrelevant, or cannot be obtained for some reason intrinsic to the case.

We use index sets J_+, J_-, J_a, and J_u to define an assignment of values to the variables x_j as follows.

$$x_j = \begin{cases} \textit{True} & \text{if } j \in J_+ \\ \textit{False} & \text{if } j \in J_- \\ \textit{Absent} & \text{if } j \in J_a \\ \textit{Unavailable} & \text{if } j \in J_u \end{cases} \tag{10.3.1}$$

Let $J_o = \{1, 2, \ldots, n\} - (J_+ \cup J_- \cup J_a \cup J_u)$, which thus is the set of indices j of the x_j without assigned value. Each x_j with $j \in J_o$ is *open*.

In some applications, the x_j have values other than the four cited ones. For example, the value may be a rational number or a member of a set. Section 7.2 of Chapter 7 describes suitable methods for transforming such cases to the four employed here.

Obtaining Values

Unavailable values have a peculiar impact on S and must be obtained at the beginning of a QA process. Section 10.4 covers that step. The values *True*, *False*, and *Absent* are acquired by tests $T_1, T_2, \ldots, T_m$. As a matter of notational convenience, T_i also is the set of indices j of the variables x_j

for which the test determines values. For example, $T_1 = \{1, 2, 4\}$ means that the test T_1 determines the values of the variables x_1, x_2, and x_4. The cost of carrying out test T_i is assumed to be positive and independent of any tests that may have already been done, and is denoted by c_i.

Suppose a test T_i contains index j, yet fails to produce a *True/False* value for x_j. If another test can produce a *True/False* value for x_j, we do not assign any value to x_j, and x_j remains open. Otherwise, we assign *Absent* to x_j. For example, suppose that T_i consists of asking an expert about the *True/False* value of x_j, and that the expert answers "I don't know the value." If another test can produce the *True/False* value of x_j, then x_j remains open. Otherwise, we assign $x_j = Absent$.

If the *True/False* value of a variable x_j found by test T_i can also be obtained by other tests, it is no longer necessary that these tests determine that value. Accordingly, we reduce these tests by removing the index j.

Algorithm to Obtain Values

The following algorithm finds values using a specified test T_i and, based on the obtained values, reduces tests accordingly.

(10.3.2) Algorithm OBTAIN VALUES. *Obtains values for variables and reduces tests accordingly.*

Input: Tests $T_1, T_2, \ldots, T_m$. Test T_i to be performed.

Output: *True/False/Absent* values resulting from application of test T_i. Corresponding to these values, reduced tests $T_1, T_2, \ldots, T_m$.

Requires: No additional algorithm needed.

Procedure:

1. Do test T_i, and output the obtained *True/False* values.
2. (Reduction based on *True/False* values) For each x_j for which T_i determined a *True/False* value, remove the index j from all tests containing that index.
3. (Assignment of *Absent*) For each x_j for which T_i should have determined a value but failed to do so, and for which the value cannot be determined by any other test, output the value *Absent*.
4. Redefine T_i to be the empty set, and stop.

Variations of the above process are possible. For example, a test T_i may not produce a *True/False* value for x_j due to operator error, but a second application of T_i may do so. Also, the fact that a test no longer needs to find a value for x_j may lower the cost of the test. These variations and others can be handled by appropriate changes in Algorithm OBTAIN VALUES (10.3.2). Exercise 10.14.12 asks the reader to work out details for some of these variations.

Conclusions

The variables $d_1, d_2, \ldots, d_p$ represent conclusions. We may also be interested in compound conclusions involving these variables. We reduce such compound cases to single-variable conclusions by introducing additional conclusion variables that represent the compound cases. For example, if we have the compound case $d_1 \vee d_2$, then we define a new conclusion d_l and add CNF clauses to S that represent $d_l \leftrightarrow (d_1 \vee d_2)$. We even consider the conclusion $\neg d_k$ to be a compound case. For that situation, the added clauses are derived from $d_l \leftrightarrow \neg d_k$. We introduce that compound case if the conclusion $\neg d_k$ might ever be proved in the QA process. That approach simplifies the discussion.

With each conclusion, we have an associated likelihood level α_k. If the likelihood level α_k is not explicitly specified, then by default $\alpha_k = 100$. We desire to prove d_k to be a theorem at level $\geq \alpha_k$, or to show that it is impossible to prove that result. For each k, we assume that d_k and the compound case $\neg d_k$ have the same associated likelihood levels. Thus, if d_l represents $\neg d_k$ via $d_l \leftrightarrow \neg d_k$, then $\alpha_l = \alpha_k$.

For each pair (d_k, α_k), let S'_k be obtained from S by fixing d_k to *False* and deleting all clauses with level $< \alpha_k$. We assume that all such S'_k are satisfiable. Indeed, if this is not so for some k, then d_k can be proved at level $\geq \alpha_k$ *a priori* and thus can be eliminated from consideration in the QA process.

Restatement of Proof Process

For convenient exposition of the algorithms and results of this chapter, we rephrase the steps establishing a theorem. We motivate the change by an example. Suppose we know $x_1 = \mathit{True}$. Then the conclusion d_1 is a theorem of a given S for the case $x_1 = \mathit{True}$ if and only if $S \wedge \neg(x_1 \rightarrow d_1)$, which can be rewritten as $S \wedge x_1 \wedge \neg d_1$, is unsatisfiable. Suppose we fix x_1 to *True* in S, delete all literals of x_1 evaluating to *False*, and delete all clauses now satisfied. Let S' result. Then the satisfiability test for $S \wedge x_1 \wedge \neg d_1$ becomes one for $S' \wedge \neg d_1$, and d_1 is a theorem of S for the case $x_1 = \mathit{True}$ if and only if d_1 is a theorem of S'.

In the general case, we have values for x_j variables given by (10.3.1) and want to know whether for this situation d_k is a theorem of S. Since the value *Absent* for x_j means that we have no information about x_j, the fixing of variables in S involves the x_j with *True/False/Unavailable* values. We see in the next section how the fixing of x_j with *Unavailable* value is done. Ignoring that aspect for the moment, let the fixing reduce S to S'. Then d_k is a theorem of S for the given situation if and only if $S' \wedge \neg d_k$ is unsatisfiable.

Due to these relationships, we call the x_j with *True/False/Unavailable* values, that is, with $j \in (J_+ \cup J_- \cup J_u)$, *fixed*, and declare any x_j with *Absent* value or any open x_j, that is, with $j \in (J_a \cup J_o)$, to be *free*.

We restate the general QA process using these definitions. Initially, we determine which of the x_j have value *Unavailable*. In each iteration, we select some test T_i and obtain *True/False/Absent* values for some x_j variables. In S, we fix the x_j for which we have *True/False/Unavailable* values, getting a CNF system S'. For each d_k, we attempt to prove that d_k is a theorem of S' or to show that the proof is not possible. If for each d_k one of the desired results has been obtained, or if all x_j have assigned values, we stop. Otherwise, we begin the next iteration.

In the subsequent sections, we fill in the details of the above process and introduce extensions. First, we describe how *True/False/Unavailable* values for x_j variables reduce S to S'.

10.4 Reduction of CNF System

Let *True/False/Absent/Unavailable* values be given for some of the x_j variables. We use these values to reduce the given S to a CNF system S' that has the same variables as S, with the same classification as fixed, free, and open. However, the clauses are modified to reflect the impact of the given values.

The *Absent* values play no role in the reductions since they just imply ignorance of a *True/False* value. For the *True/False/Unavailable* cases, the next two subsections show details.

Unavailable Value

Chapter 7 introduces the *Unavailable* value in the context of learned logic formulas. In particular, Section 7.2 defines that the value *Unavailable* assigned to any variable of a DNF clause D_1 results in the value *False* for D_1 regardless of the values assigned to the remaining variables of the clause. Accordingly, one would want the negated DNF clause $\neg D_1$, which is a CNF clause, to evaluate to *True* under the same conditions. In agreement with that consideration, we define that assignment of *Unavailable* to any variable of the CNF system S causes all clauses of S containing a literal of x_j to have value *True* and thus to be satisfied. Accordingly, all such clauses can be deleted.

The rule is backed up by an intuitive explanation. Recall that $x_j =$ *Unavailable* may mean that the *True/False* value for x_j is unimportant. On the other hand, any clause of S containing a literal of x_j implicitly says that the *True/False* value of x_j is important. As a matter of consistency,

if x_j is found to be unimportant, then the clause should be declared to be unimportant as well. Accordingly, the clause should be deleted.

The above rule for the effect of *Unavailable* values in CNF clauses and the corresponding rule of Section 7.2 for DNF clauses are consistent if a clause is uniquely classified as being of type CNF or DNF. When this is not the case, the two rules are in conflict. For example, the formula x_1 is both a CNF and DNF clause, and $x_1 = \textit{Unavailable}$ causes the formula to evaluate to *True* (resp. *False*) if it is declared to be a CNF (resp. DNF) clause. We avoid this conflict by always associating with each CNF or DNF formula a classification label declaring it to be in CNF or DNF, and by flipping the label when a formula is negated. The reader may rightly feel that this is a rather cumbersome process. But in each setting of this book, we work either only with CNF formulas or only with DNF formulas, and the classification label is obvious from the context. In the rare exceptional case where both CNF and DNF formulas occur and where variables may have the value *Unavailable*, we assign and maintain explicit classification labels.

True/False Value

For each x_j with *True/False* value, we delete all literals of x_j in S that evaluate to *False* and delete all clauses containing a literal of x_j that evaluates to *True*.

Irrelevance of Order of Reductions

Suppose S is reduced to S' by processing *True/False/Unavailable* values in a given sequence. It is easy to see that the same S' results if any other sequence is used. Thus, the sequence need not be specified.

Example

Let S have the following clauses.

$$\begin{aligned} &\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4 \vee \neg x_5 \vee d_1 \vee d_3 \quad \textit{at level } 80 \\ &x_2 \vee x_4 \vee x_5 \vee \neg d_2 \vee d_3 \\ &\neg x_2 \vee x_3 \vee r \vee d_2 \quad \textit{at level } 60 \end{aligned} \tag{10.4.1}$$

Given are the values $x_1 = \textit{Unavailable}$, $x_2 = \textit{False}$, $x_3 = \textit{True}$, and $x_4 = \textit{Absent}$. The variable x_5 has no value assigned and thus is open.

The value $x_1 = \textit{Unavailable}$ results in deletion of the first clause. The values $x_2 = \textit{False}$ and $x_3 = \textit{True}$ result in deletion of all literals x_2 and

$\neg x_3$, and in removal of the third clause. The combined effect produces S' as

$$x_4 \vee x_5 \vee \neg d_2 \vee d_3 \tag{10.4.2}$$

In S and S', the variables x_1, x_2, and x_3 are fixed, x_4 is free but not open, and x_5 is free and open.

Nonmonotonicity Caused by Unavailable Value

Suppose S is unsatisfiable. If we fix a variable of S to any *True/False* value, then the resulting S' is also unsatisfiable. However, if we fix a variable to *Unavailable*, then the resulting S' may be satisfiable. For example, if each clause of the unsatisfiable S contains a literal of x_j, and if we fix x_j to *Unavailable*, then S' has no clauses and is satisfiable. Thus, the *Unavailable* value causes nonmonotonicity as defined in Chapter 9. We avoid this pitfall by demanding that all *Unavailable* values must be obtained at the beginning of any QA process.

The method for obtaining the initial *Unavailable* values depends on the application. For example, if just a few variables may possibly have the value *Unavailable* and if the applicable cases are readily recognized, one may go through the list of these variables and ask which ones have that value. In more complicated settings, logic formulas learned from historical data may allow prediction of *Unavailable* values, and thus may be used to ask for such values efficiently.

During a QA process, it may happen that an *Unavailable* value surfaces despite careful planning. Since the new *Unavailable* value causes deletion of clauses, all results derived by theorem proving up to that point are potentially invalid. We find out which theorems are affected by restarting the QA process. The initial values are not just the *Unavailable* values, but also all *True/False/Absent* values already obtained.

We are ready to prove conclusions.

10.5 Proof of Conclusions

Suppose we have *True/False/Absent/Unavailable* values for some but not all of the x_j variables. We want to decide whether d_k is a theorem for that situation at level $\geq \alpha_k$. Using the x_j values, we reduce S to S'.

We delete from S' all clauses with level $< \alpha_k$, fix d_k at *True*, and check the resulting S'' for satisfiability. If S'' is unsatisfiable, then $\neg d_k$ is a theorem at level $\geq \alpha_k$, and we say that (d_k, α_k) is *impossible*. Assume that this case is not at hand.

We attempt to prove d_k at level $\geq \alpha_k$. For this, we delete from S' all clauses with level $< \alpha_k$, fix d_k at *False*, and check the resulting S'' for satisfiability.

If S'' is unsatisfiable, we have proved d_k at level $\geq \alpha_k$. For short, we say that (d_k, α_k) is *proved*.

If S'' is satisfiable, we check if d_k can be proved by some *True/False* values for the open x_j. This is an instance of problem Q-ALL SAT (4.2.8). We solve that instance approximately by Heuristic SOLVE Q-ALL SAT (4.6.9). Before proceeding, the reader may want to review that scheme as well as Heuristics SOLVE Q-MINFIX UNSAT (4.6.16) and SOLVE Q-MINFIX SAT (4.6.22) of Section 4.6. In the algorithms to follow, the cited heuristic schemes occur as subroutines. They can be replaced in rather straightforward changes by exact algorithms, provided the substitutes are fast enough for the task at hand.

If Heuristic SOLVE Q-ALL SAT (4.6.9) ascertains that no *True/False* values exist for the open x_j that would prove d_k at level $\geq \alpha_k$, then the proof of d_k at level $\geq \alpha_k$ is *futile*, for short (d_k, α_k) *is futile*.

If (d_k, α_k) is impossible, proved, or futile, we say that (d_k, α_k) is *decided*. In the remaining case, (d_k, α_k) is *undecided*.

If we have *True/False/Absent/Unavailable* values for all and not just some of the x_j, the above process can be simplified. In that case, if d_k cannot be proved, we know that (d_k, α) is futile, and we skip execution of Heuristic SOLVE Q-ALL SAT (4.6.9).

Goal Set and Result Set

We collect in a *goal set* G the pairs (d_k, α_k) we want to prove or show to be futile. The selection of the goal set depends on circumstances. Section 10.6 discusses details. For the moment, we assume that the goal set is given.

When we have determined that a pair (d_k, α_k) is impossible, we remove that pair from G and add the statement "(d_k, α_k) impossible" to a *result set* H. The case where (d_k, α_k) is proved or shown to be futile is handled analogously. That is, the pair is removed from G, and "(d_k, α_k) proved" or "(d_k, α_k) futile" is added to H.

Algorithm to Prove Conclusions

We are ready to summarize the computational steps that prove conclusions. For convenient use, we assume that the input contains both a goal set G of (d_k, α_k) that are still undecided and a result set H of (d_k, α_k) that have already been decided. The algorithm uses H to improve the computational efficiency, as follows. If H contains "(d_k, α_k) impossible" (resp. "(d_k, α_k) proved"), then the clause $\neg d_k$ (resp. d_k) with level α_k can be added to the

CNF system on hand. In both cases, the additional clause induces a fixing of d_k when any goal (d_l, α_l) of G with $\alpha_l \leq \alpha_k$ is to be decided.

(10.5.1) Algorithm PROVE CONCLUSIONS. *Attempts to prove conclusions.*

Input: CNF system S with $d_1, d_2, \ldots, d_p$ and $x_1, x_2, \ldots, x_n$ among its variables. Goal set G and result set H. *True/False/Absent/Unavailable* values for some or all of the x_j.

Output: A reduced set G and an increased set H. (The set G (resp. H) contains all undecided (resp. decided) (d_k, α_k). If there are no open x_j, then G is empty, and all (d_k, α_k) are decided.)

Requires: Algorithm SOLVE SAT and Heuristic SOLVE Q-ALL SAT (4.6.9).

Procedure:

1. Using the given *True/False/Absent/Unavailable* values for the x_j, reduce S to S'. Enlarge S' as follows. For each statement "(d_k, α_k) impossible" (resp. "(d_k, α_k) proved") of H, add the clause $\neg d_k$ (resp. d_k) with level α_k. Initialize a temporary goal set G' to be equal to G.
2. If G' is empty, output the current G and H, and stop. Otherwise, remove a pair (d_k, α_k) from G'.
3. (Case "(d_k, α_k) impossible") Derive a CNF system S'' from S' by deleting all clauses with level $< \alpha_k$ and by enforcing $d_k = \mathit{True}$. Using Algorithm SOLVE SAT, check if S'' is satisfiable. If S'' is unsatisfiable, remove (d_k, α_k) from G, add "(d_k, α_k) impossible" to H, revise S' by adding the clause $\neg d_k$ with level α_k, and go to Step 2.
4. (Case "(d_k, α_k) proved") Derive a CNF system S'' from S' by deleting all clauses with level $< \alpha_k$ and by enforcing $d_k = \mathit{False}$. With Algorithm SOLVE SAT, check if S'' is satisfiable. If S'' is unsatisfiable, remove (d_k, α_k) from G, add "(d_k, α_k) proved" to H, revise S' by adding the clause d_k with level α_k, and go to Step 2.
5. (Case "(d_k, α_k) futile.") If there are no open x_j, go to Step 6. Otherwise, do Heuristic SOLVE Q-ALL SAT (4.6.9) with the satisfiable S'' of Step 4 as input CNF system and with the open x_j as the special input variables $q_1, \ldots, q_l$. The input system R is defined to have no clauses. If the heuristic determines that no *True/False* values for the open x_j exist that render S'' unsatisfiable, go to Step 6. Otherwise, go to Step 2.
6. Remove (d_k, α_k) from G, add "(d_k, α_k) futile" to H, and go to Step 2.

The goal set G used in Algorithm PROVE CONCLUSIONS (10.5.1) depends on the application. The next section covers details.

10.6 Selection of Goal Set

The goal set needed for Algorithm PROVE CONCLUSIONS (10.5.1) may be chosen in several ways. In the simplest setting, G is defined at the beginning of a QA process and includes all (d_k, α_k) pairs of interest. As Algorithm PROVE CONCLUSIONS (10.5.1) is repeatedly invoked with expanding lists of *True/False/Absent/Unavailable* values for the x_j, the set G is reduced. The QA process stops when G has become empty.

In a variation, the person involved in the QA process may repeatedly redefine G to explore conclusions and related likelihood levels. In such a situation, the person becomes a supervisor of the QA process, and the computer system is an assistant that attempts proofs as directed by the goal set.

In Section 10.11, we meet an optimization version of the QA process where, effectively, the goal set is defined by the solution of a MINSAT instance. It is an example for cases where another process, and not a person, selects the goal set.

10.7 Low-cost Assignments

When the QA process has terminated, we compute for each proved or futile (d_k, α_k) a *low-cost assignment* whose values can be obtained at low total cost of tests, yet are sufficient to prove (d_k, α_k) or show it to be futile. The low-cost assignments are used in subsequent QA sessions to select tests, as described in Section 10.8.

The reader may wonder why we do not compute a low-cost assignment for the case where (d_k, α_k) is impossible, since that result shows that $\neg d_k$ can be proved at level $\geq \alpha_k$. According to Section 10.3, S then also contains another conclusion, say d_l, representing the compound case $\neg d_k$ and having associated likelihood $\alpha_l = \alpha_k$. Since (d_k, α_k) is impossible, (d_l, α_l) is proved, and when we compute a low-cost assignment for the latter case, we implicitly also cover the former one.

The steps for finding a low-cost assignment depend on whether (d_k, α_k) is proved or futile. We first deal with the case where (d_k, α_k) is proved.

Proof Case

Suppose (d_k, α_k) is proved. We execute Heuristic SOLVE Q-MINFIX UNSAT (4.6.16) with input created as follows. We fix in the CNF system S each x_j with value *Unavailable* to this value. We also fix d_k to *False*, and finally delete all clauses with level $< \alpha_k$. These reductions produce

a CNF system S' that is the input CNF system for Heuristic SOLVE Q-MINFIX UNSAT (4.6.16). We show that S' is satisfiable, as demanded by the heuristic. In Section 10.3, a CNF system S'_k is derived from S by fixing d_k to *False* and by deleting all clauses with level $< \alpha_k$. It is assumed there that S'_k is satisfiable. We may derive S' from S'_k by fixing each x_j with *Unavailable* value to this value. The latter fixing causes deletion of some clauses from S'_k and thus reduces the satisfiable S'_k to a satisfiable S'.

We continue with the definition of the input for Heuristic SOLVE Q-MINFIX UNSAT (4.6.16). Define the x_j with value *True/False* to be the special input variables $q_1, \ldots, q_l$ for the heuristic. The *True/False* values associated with the x_j define the input assignment for the special variables. Since the *True/False* values prove (d_k, α_k), that assignment renders S' unsatisfiable, as demanded by the heuristic. The input tests T_i, with costs c_i, are the original tests on hand prior to any reductions by Algorithm OBTAIN VALUES (10.3.2).

The output of Heuristic SOLVE Q-MINFIX UNSAT (4.6.16) is a collection of tests T_i that, at low total cost, determine the value of an S'-unacceptable partial assignment of *True/False* values for $q_1, \ldots, q_l$. We assign these *True/False* values to the corresponding x_j and call that partial assignment for the x_j plus the assignment of *Absent* and *Unavailable* values on hand a *low-cost proof assignment*. By the derivation, the assignment can be found by tests at low total cost, and it is sufficient information to prove (d_k, α_k).

When the QA process is executed repeatedly, a number of low-cost proof assignments for (d_k, α_k) may be computed. These assignments are used for the selection of tests, as described in Section 10.8. For that step, we collect the low-cost proof assignments for (d_k, α_k) in a file P_k. We call P_k a *low-cost proof file* for (d_k, α_k). We emphasize that P_k is not a set and thus may contain multiple copies of the same low-cost proof assignment. That way, the relative frequency of cases is correctly represented in P_k.

Example

Let S be given by

$$\begin{aligned} &\neg x_1 \vee \neg x_2 \vee d_1 \\ &x_3 \vee \neg d_1 \vee d_2 \\ &\neg x_1 \vee \neg x_4 \vee d_3 \end{aligned} \tag{10.7.1}$$

We have the values $x_1 = x_2 = \mathit{True}$, $x_3 = \mathit{Unavailable}$, and $x_4 = \mathit{Absent}$. These values allow d_1 to be proved at level $\alpha_1 = 100$. The *True/False* values may be obtained by tests $T_1 = \{1\}$, $T_2 = \{1, 2\}$, $T_3 = \{2, 3\}$, and $T_4 = \{3, 4\}$, with costs $c_1 = 1$, $c_2 = 3$, $c_3 = 5$, and $c_4 = 2$.

The input system S' for Heuristic SOLVE Q-MINFIX UNSAT (4.6.16) is derived from S by fixing x_3 to *Unavailable* and d_1 to *False*. Thus, S' is given by

$$\begin{aligned} &\neg x_1 \vee \neg x_2 \\ &\neg x_1 \vee \neg x_4 \vee d_3 \end{aligned} \tag{10.7.2}$$

The variables x_1 and x_2 are the input variables $q_1, \ldots, q_l$, and $x_1 = x_2 =$ *True* are the values for the input assignment. It turns out that Heuristic SOLVE Q-MINFIX UNSAT (4.6.16) produces a low-cost test collection that consists just of T_2. That test produces the *True/False* values for x_1 and x_2. Thus, $(x_1 = x_2 = \mathit{True}, x_3 = \mathit{Unavailable})$ is the desired low-cost proof assignment for the file P_1.

Futile Case

The case where (d_k, α_k) is futile is handled similarly. Specifically, Heuristic SOLVE Q-ALL SAT (4.6.9) has determined (d_k, α_k) to be futile. Thus, assignment of the *True/False/Absent/Unavailable* values to the x_j and of *False* to d_k, deletion of all clauses with level $< \alpha_k$, and deletion of all literals of the open x_j reduce S to a satisfiable CNF system, say S''. That conclusion establishes (d_k, α_k) to be futile.

We want the same conclusion with *True/False* values for the x_j determined by a low-cost collection of tests. To this end, we execute Heuristic SOLVE Q-MINFIX SAT (4.6.22) with input constructed as follows. We delete from S all literals of open x_j variables, fix each x_j with value *Unavailable* at that value, fix d_k to *False*, and finally delete all clauses with level $< \alpha_k$. The CNF system S' so deduced from S is the input system for the heuristic. The x_j with *True/False* values constitute the special input variables $q_1, \ldots, q_l$. The *True/False* values associated with the x_j define the input assignment for the special variables. That assignment reduces S' to S''. Since Heuristic SOLVE Q-ALL SAT (4.6.9) has shown (d_k, α_k) to be futile, S'' is satisfiable as demanded by Heuristic SOLVE Q-MINFIX SAT (4.6.22). The input tests T_i, with costs c_i, are the original tests on hand prior to any reductions by Algorithm OBTAIN VALUES (10.3.2).

The output of Heuristic SOLVE Q-MINFIX SAT (4.6.22) is a collection of tests T_i that, at low total cost, produce a partial assignment for $q_1, \ldots, q_l$ such that any extension assignment is S'-acceptable. We assign these *True/False* values to the corresponding x_j and call that partial assignment of *True/False* values for the x_j plus the assignment of *Absent* and *Unavailable* values on hand, a *low-cost futile assignment.* By the derivation, that assignment is sufficient information to show that (d_k, α_k) is futile.

Analogously to the case of proved (d_k, α_k), we collect the low-cost futile assignments for (d_k, α_k) obtained in various executions of the QA

process in a file F_k. The file is used in Section 10.8 to select tests. We call F_k a *low-cost futile file* for d_k. Like P_k, F_k is not a set and thus may contain multiple copies of the same low-cost futile assignment.

Example

Define S by the clauses

$$\begin{aligned} &\neg x_1 \vee \neg x_2 \vee d_1 \\ &x_3 \vee \neg d_1 \vee d_2 \\ &\neg x_1 \vee \neg x_4 \vee d_3 \end{aligned} \tag{10.7.3}$$

The values for the x_j are $x_1 = \mathit{False}$, $x_2 = \mathit{Unavailable}$, and $x_3 = \mathit{False}$. The variable x_4 is open. These values cause d_3 to be futile at level $\alpha_3 = 100$. The *True/False* values may be obtained by tests $T_1 = \{1\}$, $T_2 = \{1, 2\}$, $T_3 = \{2, 3\}$, and $T_4 = \{3, 4\}$, with costs $c_1 = 1$, $c_2 = 3$, $c_3 = 5$, and $c_4 = 2$.

The input system S' for Heuristic SOLVE Q-MINFIX SAT (4.6.22) is derived from S by deleting the literals of the open x_4, fixing x_2 at *Unavailable*, and fixing d_3 at *False*. Thus, S' is given by

$$\begin{aligned} &x_3 \vee \neg d_1 \vee d_2 \\ &\neg x_1 \end{aligned} \tag{10.7.4}$$

The variables x_1 and x_3 are the input variables $q_1, \ldots, q_l$, and $x_1 = x_3 = \mathit{False}$ are the values for the input assignment. Heuristic SOLVE Q-MINFIX SAT (4.6.22) outputs a low-cost test collection that consists just of T_1. That test produces the *True/False* value for x_1. Thus, $(x_1 = \mathit{False}, x_2 = \mathit{Unavailable})$ is the desired low-cost futile assignment for the file F_1.

Algorithm for Low-cost Assignment

The following algorithm summarizes the steps.

(10.7.5) Algorithm LOW-COST ASSIGNMENT. *Computes low-cost assignment.*

Input: CNF system S with $d_1, d_2, \ldots, d_p$ and $x_1, x_2, \ldots, x_n$ among its variables. *True/False/Absent/Unavailable* values for some or all of the x_j. A goal (d_k, α_k) that has been proved or shown to be futile. For $1 \leq i \leq m$, test T_i for determining the values of a subset of the x_j at cost c_i. The T_i collectively determine all values.

Output: A low-cost proof assignment or futile assignment for (d_k, α_k), as applicable.

Requires: Heuristics SOLVE Q-MINFIX SAT (4.6.22) and SOLVE Q-MINFIX UNSAT (4.6.16).

Procedure:

1. If (d_k, α_k) is futile, go to Step 2. Otherwise, do Heuristic SOLVE Q-MINFIX UNSAT (4.6.16) with the following input. In S, fix the x_j with given *Unavailable* value to that value, enforce d_k = *False*, and delete all clauses with level $< \alpha_k$. The resulting S' is the input CNF system for the heuristic. Define the x_j with *True/False* values to be the special input variables $q_1, \ldots, q_l$. The *True/False* values of the x_j provide an S'-unacceptable input assignment for $q_1, \ldots, q_l$. Finally, the given T_i and c_i are the input tests and costs for the heuristic. The output collection of tests T_i produces a low-cost partial assignment for $q_1, \ldots, q_l$ and thus for the x_j. Combine that partial assignment with the *Absent* and *Unavailable* values of the x_j to a low-cost proof assignment for (d_k, α_k). Output that assignment, and stop.
2. ((d_k, α_k) is futile.) Do Heuristic SOLVE Q-MINFIX SAT (4.6.22) with input determined as follows. In S, delete all literals of the open x_j, fix the x_j with given *Unavailable* value to that value, enforce d_k = *False*, and delete all clauses with level $< \alpha_k$. The resulting S' is the input CNF system for the heuristic. Define the x_j with *True/False* values to be the special input variables $q_1, \ldots, q_l$. The *True/False* values of the x_j provide an S'-acceptable input assignment for $q_1, \ldots, q_l$. Finally, the given T_i and c_i are the input tests and costs for the heuristic. The output collection of tests T_i produces a low-cost partial assignment for $q_1, \ldots, q_l$ and thus for the x_j. Combine that partial assignment with the *Absent* and *Unavailable* values of the x_j to a low-cost futile assignment for (d_k, α_k). Output that assignment, and stop.

A key part of the QA process is the selection of tests that produce values for the x_j. The next section covers that step.

10.8 Selection of Tests

We want to select tests that produce enough *True/False* values for the x_j so that each pair (d_k, α_k) in the given goal set G can be decided. Subject to that condition, we want the total cost of the tests to be low.

In the general case, we already have *True/False/Absent/Unavailable* values for some but not all of the x_j, and these values do not suffice to decide any conclusion of the current goal set. These values define the *current assignment* w.

We need not select all tests at one time, but are allowed to pick a test, perform it, combine the new values with the current assignment w to a new

current assignment, and check which of the (d_k, α_k) can now be decided. If all (d_k, α_k) can be decided, we stop. Otherwise, we select another test and repeat the above steps.

It is possible that, instead of one test, we may prefer selection of a group of tests. For example, each test of a group may require the same setup, and it is more efficient to do that setup once and carry out all tests of the group. For each such possibility, we introduce an additional test T_l that represents the group. Thus, if I is the set of indices i of the tests T_i in the group, then $T_l = \bigcup_{i \in I} T_i$. The cost c_l of T_l is so computed that it correctly reflects the total cost of doing all tests T_i of the group at the same time. Due to this rule, we only need to consider selection of one test.

The rule for selecting the next test is based on a scheme that has proved to be effective in a related setting. The main idea is to employ the records of the low-cost assignments v_k in the files F_k and P_k at hand to guide the selection of tests. In a slight abuse of set notation, we denote the file created by concatenating the files F_k and P_k by $F_k \cup P_k$. We obtain a *representative file* V_k from $F_k \cup P_k$ by deleting from the latter file all duplicate assignments. Thus, if an assignment occurs f_1 times in F_k and f_2 times in P_k, then the assignment occurs $f_1 + f_2$ times in $F_k \cup P_k$ and just once in V_k.

Relevance of Assignments

Declare an assignment $v_k \in V_k$ to be *irrelevant* if the *True/False* values that v_k has for open x_j cannot help to decide d_k. Clearly, v_k is irrelevant if v_k has no *True/False* values for the open x_j. So suppose that v_k has at least one such *True/False* value. The test for irrelevance of v_k is as follows.

Fix the x_j of S at the values of the current assignment w, fix d_k at *False*, and delete all clauses with level $< \alpha_k$. Finally, assign the *True/False* values that v_k has for open x_j, to those variables. Let S_{v_k} be the resulting CNF system.

If $v_k \in P_k$, then v_k is irrelevant if its values do not lead to a proof of d_k, or equivalently, if S_{v_k} is satisfiable.

If $v_k \in F_k$, then v_k is irrelevant if its values do not show d_k to be futile, or equivalently, if S_{v_k} can be made unsatisfiable by some *True/False* values for the x_j that are still open in S_{v_k}. The decision involves solution of an instance of Q-ALL SAT (4.2.8) where S_{v_k} plays the role of the CNF system S of Q-ALL SAT (4.2.8), and where the still-open x_j of S_{v_k} are the special variables $q_1, \ldots, q_l$. For this instance, the CNF system R of Q-ALL SAT (4.2.8) has no clauses. To this instance of Q-ALL SAT (4.2.8), we apply Steps 1 and 2 of Heuristic SOLVE Q-ALL SAT (4.6.9). Specifically, we declare v_k to be irrelevant if Step 2 of the heuristic determines that deletion of the still-open x_j from S_{v_k} produces an unsatisfiable subsystem.

We focus on the $v_k \in V_k$ that, according to the above process, are not ruled out as irrelevant. Let us call these v_k *relevant.*

Using the relevant v_k, we want to estimate the effectiveness of tests to settle the undecided conclusions. We do this in four steps, using the following definition. If a relevant v_k has a *True/False* value for an open x_j, we say that v_k *covers* x_j. In that case, we also say that x_j is v_k*-covered.*

First, for each relevant v_k, we compute an estimate γ_{v_k} of the likelihood that we will be able to decide d_k once we have obtained *True/False* values for the v_k-covered x_j.

Second, for each open x_j, we combine the results of the first step to a utility value $u_{j,k}$ that estimates the usefulness of the yet to be obtained *True/False* value of x_j for deciding d_k.

Third, for each test T_i, we sum the utility values $u_{j,k}$ with $j \in T_i$ to an effectiveness measure e_i of T_i.

Fourth, we declare the ratio e_i/c_i to be a measure of the cost effectiveness of T_i. Accordingly, we select a test that maximizes that ratio.

Here are the details of the four steps.

Likelihood

Let v_k be relevant. We want an estimate γ_{v_k} of the likelihood that we can decide d_k once we have *True/False* values for the v_k-covered x_j. Let m_{v_k} (resp. n_{v_k}) be the number of x_j for which v_k has a *True/False* value and for which the current assignment w has the same (resp. opposite) *True/False* value. We call m_{v_k} (resp. n_{v_k}) the number of *matches* (resp. *mismatches*) between v_k and w. If there are only matches and no mismatches, it seems likely that the yet to be obtained *True/False* values for the v_k-covered x_j will agree with those of v_k. Since v_k is relevant, it thus seems likely that the yet to be obtained *True/False* values will suffice to decide d_k. A bit optimistically, we declare the likelihood estimate γ_{v_k} for that event to be equal to 100. Now let the number of matches decline and eventually reach 0. It seems reasonable that the estimate γ_{v_k} also declines and eventually reaches 0.

We achieve the desired behavior of γ_{v_k} with the formula

$$\gamma_{v_k} = \begin{cases} 100 \cdot m_{v_k}/(m_{v_k} + n_{v_k}) & \text{if } m_{v_k} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{10.8.1}$$

Example

Suppose we have variables $x_1, x_2, \ldots, x_5$. The current assignment is $w = (x_1 = \mathit{True}, x_2 = \mathit{False}, x_3 = \mathit{True})$. Thus, the variables x_4 and x_5 are

open. Suppose $v_k = (x_1 = \mathit{False}, x_3 = \mathit{True}, x_4 = \mathit{True}, x_5 = \mathit{False})$ is relevant. The value $x_3 = \mathit{True}$ is the only match between v_k and w, so $m_{v_k} = 1$. The value of x_1 is the only mismatch, so $n_{v_k} = 1$. Using these quantities in (10.8.1), $\gamma_{v_k} = 100 \cdot 1/(1+1) = 50$ estimates the likelihood that the yet to be obtained *True/False* values for the v_k-covered x_4 and x_5 will decide d_k.

Utility

For each open x_j, the utility value $u_{j,k}$ estimates the usefulness of the yet to be obtained *True/False* value of x_j for deciding d_k. For each relevant v_k, let p_{v_k} be the number of v_k-covered open x_l. Define f_{v_k} to be the number of copies of v_k occurring in $F_k \cup P_k$. We call f_{v_k} the *frequency* of v_k.

Recall that, for any relevant v_k, γ_{v_k} estimates the likelihood that we can decide d_k with the yet to be obtained *True/False* values for the v_k-covered x_j. It seems reasonable that we use the portion γ_{v_k}/p_{v_k} of γ_{v_k} to measure the contribution any v_k-covered x_j will make toward deciding d_k. For each open x_j, we add up these contributions by considering all relevant v_k and their frequencies f_{v_k}, and get the desired utility value $u_{j,k}$. For a compact formula, we define, for each open x_j, a set $A_{j,k}$ that contains all relevant v_k that cover x_j. Thus,

$$A_{j,k} = \{v_k \in V_k \mid v_k \text{ is relevant and covers } x_j\} \tag{10.8.2}$$

and

$$u_{j,k} = \sum_{v_k \in A_{j,k}} \gamma_{v_k} \cdot f_{v_k}/p_{v_k} \tag{10.8.3}$$

Example

Suppose that x_4 is open, and that there are two relevant assignments v_k^1 and v_k^2 that have *True/False* values for x_4. The frequencies are $f_{v_k^1} = f_{v_k^2} = 1$. Assume $\gamma_{v_k^1} = 50$, $\gamma_{v_k^2} = 33$, $p_{v_k^1} = 2$, and $p_{v_k^2} = 1$. Then $A_{4,k} = \{v_k^1, v_k^2\}$ and $u_{4,k} = \gamma_{v_k^1} \cdot f_{v_k^1}/p_{v_k^1} + \gamma_{v_k^2} \cdot f_{v_k^2}/p_{v_k^2} = 50 \cdot 1/2 + 33 \cdot 1/1 = 58$.

We turn to the third step, where we combine the utility values $u_{j,k}$ to the effectiveness measure e_i.

Effectiveness

For each nonempty test T_i, we define the effectiveness e_i of T_i to be the sum of the utility values $u_{j,k}$ for which x_j is open and determined by T_i, and

for which (d_k, α_k) is not yet decided. For the formula, we use the fact that Algorithm OBTAIN VALUES (10.3.2) reduces tests as values are obtained. Accordingly, each current T_i contains an index j only if x_j is open. We also use the fact that the goal set G contains the undecided pairs (d_k, α_k). Define $K = \{k \mid (d_k, \alpha_k) \in G\}$. Then

$$e_i = \sum_{j \in T_i} \sum_{k \in K} u_{j,k} \tag{10.8.4}$$

Example

Let $T_1 = \{1, 4, 5\}$ and $G = \{(d_1, \alpha_1), (d_2, \alpha_2)\}$. Assume x_4 and x_5 are open, and $u_{4,1} = 58$, $u_{4,2} = 70$, $u_{5,1} = 10$, and $u_{5,2} = 25$. Then $e_1 = u_{4,1} + u_{4,2} + u_{5,1} + u_{5,2} = 58 + 70 + 10 + 25 = 163$.

We come to the fourth step, where we select the next test.

Cost-effective Test

We choose a nonempty test that maximizes the effectiveness per unit cost. Let $I = \{i \mid T_i \text{ is nonempty}\}$. Then the index i^* of the selected test T_{i^*} solves

$$\max_{i \in I} \frac{e_i}{c_i} \tag{10.8.5}$$

In case several indices i achieve the maximum ratio, we select among them one with minimum c_i and break any secondary tie by a random choice.

We carry out Algorithm OBTAIN VALUES (10.3.2) for the selected test T_{i^*} and use the values so obtained to enlarge the current assignment w. With that assignment, we begin another iteration of the QA process.

Example

Suppose we have three nonempty tests T_1, T_2, and T_3 with costs $c_1 = 5$, $c_2 = 7$, and $c_3 = 4$. We also have effectiveness estimates $e_1 = 163$, $e_2 = 274$, and $e_3 = 32$. The ratios e_i/c_i are $e_1/c_1 = 163/5 = 32.6$, $e_2/c_2 = 274/7 = 39.1$, and $e_3/c_3 = 32/4 = 8$. The largest ratio is $e_2/c_2 = 39.1$, so $i^* = 2$, and T_2 is the next test.

Algorithm for Selection of Next Test

The entire selection process is captured by the following algorithm.

(10.8.6) Algorithm SELECT TEST. *Selects test for obtaining values.*

Input: CNF system S with $d_1, d_2, \ldots, d_p$ and $x_1, x_2, \ldots, x_n$ among its variables. *True/False/Absent/Unavailable* values for some but not all of the x_j. Goal set G. For each k such that $(d_k, \alpha_k) \in G$, low-cost assignment files F_k and P_k. For each $(d_k, \alpha_k) \in G$, d_k cannot be decided by Algorithm PROVE CONCLUSIONS (10.5.1) using the values given for the x_j. For $1 \leq i \leq m$, test T_i for determining the values of a subset of the x_j at cost c_i. The T_i collectively determine all values.

Output: A test T_{i^*} that is estimated to be the most cost-effective next test.

Requires: Algorithm SOLVE SAT and Heuristic SOLVE Q-ALL SAT (4.6.9).

Procedure:

1. Using the given *True/False/Absent/Unavailable* values for the x_j, reduce S to S'. Initialize a temporary goal set G' to be equal to G.
2. If G' is empty, go to Step 6. Otherwise, remove a pair (d_k, α_k) from G'. Reduce S' to S'' by fixing $d_k =$ *False* and by deleting all clauses with level $< \alpha_k$. Delete from $F_k \cup P_k$ all duplicate assignments to obtain the representative file V_k.
3. (Check relevance of each $v_k \in V_k$.) Declare each $v_k \in V_k$ which does not cover any open x_j, to be irrelevant. For each remaining $v_k \in V_k$, assign each *True/False* value that v_k has for an open x_j of S'', to that variable, and reduce S'' accordingly. Let S_{v_k} result. Process each S_{v_k} depending on whether v_k is in P_k or F_k.
   If $v_k \in P_k$, check with Algorithm SOLVE SAT if S_{v_k} is satisfiable. If this is the case, v_k is irrelevant.
   If $v_k \in F_k$, do Steps 1 and 2 of Heuristic SOLVE Q-ALL SAT (4.6.9) with S_{v_k} as input CNF system S and with the still-open x_j of S_{v_k} as the special input variables $q_1, \ldots, q_l$. The input system R has no clauses. Declare v_k to be irrelevant if Step 2 of the heuristic determines that deletion of the still-open x_j from S_{v_k} leaves an unsatisfiable subsystem. Define all $v_k \in V_k$ that by the above process are not declared to be irrelevant, to be relevant.
4. (Likelihood γ_{v_k}) For each relevant v_k, do the following. Let m_{v_k} (resp. n_{v_k}) be the number of x_j for which v_k has a *True/False* value and for which the current assignment w has the same (resp. opposite) *True/False* value. Compute $\gamma_{v_k} = 100 \cdot m_{v_k}/(m_{v_k} + n_{v_k})$ if $m_{v_k} > 0$, and $\gamma_{v_k} = 0$ otherwise.
5. (Utility $u_{j,k}$) For each relevant v_k, let p_{v_k} be the number of v_k-covered open x_l. Define f_{v_k} to be the number of copies of v_k in $F_k \cup P_k$. For

each open x_j, let $A_{j,k} = \{v_k \in V_k \mid v_k \text{ is relevant and covers } x_j\}$, and compute $u_{j,k} = \sum_{v_k \in A_{j,k}} \gamma_{v_k} \cdot f_{v_k}/p_{v_k}$. Go to Step 2.

6. (Effectiveness e_i) Let $K = \{k \mid (d_k, \alpha_k) \in G\}$. For each nonempty test T_i, compute $e_i = \sum_{j \in T_i} \sum_{k \in K} u_{j,k}$.
8. (Select most effective test.) Let $I = \{i \mid T_i \text{ is nonempty}\}$. Define i^* to be an index solving $\max_{i \in I} e_i/c_i$. In case several indices achieve the maximum ratio, select among them one with minimum cost, and break any secondary tie by a random choice. Output T_{i^*}, and stop.

With Algorithms PROVE CONCLUSIONS (10.5.1), LOW-COST ASSIGNMENT (10.7.5), and SELECT TEST (10.8.6) at hand, we are ready to define the algorithm for the entire QA process.

10.9 QA Process

We are to decide, for each pair (d_k, α_k) of a given goal set G, whether d_k can be proved or is futile at level $\geq \alpha_k$. For each d_k, we have files F_k and P_k that contain low-cost assignments computed during previous executions of the QA Process.

We first determine which x_j have the value *Unavailable*. Optionally, we may also acquire some *True/False/Absent* values for the x_j. We begin an iterative process. In each iteration, we proceed as follows.

We attempt to prove the pairs $(d_k, \alpha_k) \in G$ or to show them to be futile, using Algorithm PROVE CONCLUSIONS (10.5.1). We remove each (d_k, α_k) decided in this step from G.

If this step leaves G nonempty, we use Algorithm SELECT TEST (10.8.6) to select a next test T_{i^*}. We carry out Algorithm OBTAIN VALUES (10.3.2) for the test T_{i^*}, combine the resulting *True/False/Absent* values with the values already on hand for the x_j, and begin another iteration.

Suppose that G has become empty, so all conclusions have been decided. We use Algorithm LOW-COST ASSIGNMENT (10.7.5) to compute a low-cost assignment for each d_k. If d_k is proved (resp. is futile), we add the low-cost assignment to P_k (resp. F_k).

Algorithm for QA Process

The following algorithm carries out the above steps.

(10.9.1) Algorithm QA PROCESS. *Carries out* QA *process.*

Input: CNF system S with $d_1, d_2, \ldots, d_p$ and $x_1, x_2, \ldots, x_n$ among its variables. Goal set G. The system S remains satisfiable if, for any $(d_k, \alpha_k) \in G$,

$d_k = \mathit{False}$ is enforced. For each k such that $(d_k, \alpha_k) \in G$, low-cost assignment files F_k and P_k. For $1 \leq i \leq m$, test T_i for determining the values of a subset of the x_j at a cost c_i. The T_i collectively determine all values.

Output: For each $(d_k, \alpha_k) \in G$, either "d_k is proved" or "d_k is futile," and a corresponding low-cost assignment that suffices to establish this fact and that is added to F_k or P_k, respectively.

Requires: Algorithms LOW-COST ASSIGNMENT (10.7.5), OBTAIN VALUES (10.3.2), PROVE CONCLUSIONS (10.5.1), and SELECT TEST (10.8.6).

Procedure:

1. (Get initial values) Ask for the applicable *Unavailable* values for the x_j, and if some *True/False/Absent* values for the x_j are available. Use the supplied values to define a current assignment w for the x_j. Remove from the tests T_i all indices j for which x_j now has an assigned value.
2. (Prove conclusions.) Carry out Algorithm PROVE CONCLUSIONS (10.5.1) with the following input: CNF system S, goal set G, and *True/False/Absent/Unavailable* values of assignment w. If the algorithm reduces G to the empty set, go to Step 4.
3. (Select next test.) Carry out Algorithm SELECT TEST (10.8.6) with the following input: CNF system S, goal set G, the values of assignment w, low-cost assignment files F_k and P_k, tests T_i and costs c_i. The output is a cost-effective next test T_{i^*}. Do Algorithm OBTAIN VALUES (10.3.2) with test T_{i^*}, incorporate the resulting values into assignment w, and go to Step 2.
4. (Derive low-cost assignments.) Restore the goal set G and all tests T_i to the original input sets. For each $(d_k, \alpha_k) \in G$, do Algorithm LOW-COST ASSIGNMENT (10.7.5) with the following input: CNF system S, the values of assignment w, the pair (d_k, α_k) and whether it is proved or futile, and tests T_i with costs c_i. The algorithm outputs a low-cost assignment. If d_k is proved (resp. futile), place the low-cost assignment into P_k (resp. F_k). When all $(d_k, \alpha_k) \in G$ have been processed, stop.

We skip discussion of an example since we have already presented example calculations for each step.

Convergence

Suppose Algorithm QA Process (10.9.1) is applied repeatedly in an environment where the initial x_j values as well as the x_j values produced by tests are randomly selected according to a given probability distribution. In that setting, we say that Algorithm QA Process (10.9.1) *converges* if the

repeated executions produce low-cost assignment files F_k and P_k so that the test selections based on these files eventually are done at minimum expected total cost.

Due to the complexity of the QA process and due to the use of heuristics employed in the various steps of Algorithm QA Process (10.9.1), we cannot prove that the algorithm must converge. However, the algorithm has features that aid convergence. Specifically, the test selection formulas (10.8.1)–(10.8.5) are so structured that, for empty or small assignment files F_k and P_k, the test selection is undiscerning and even rambling. That selection process, however flawed it may seem to be, effectively tries out tests when little historical information is available. As the assignment files F_k and P_k grow and contain more assignments that apply to a given situation, the selection process becomes more focused and is likely to make good choices consistently.

If the probability distributions describing the environment change over time, we record with each assignment in F_k and P_k the date of its creation. Before each execution of Algorithm QA PROCESS (10.9.1), we check the dates of all assignments in F_k and P_k and delete those exceeding a specified age. As a matter of computational efficiency, we also delete oldest assignments when convergence has been achieved and when the files F_k and P_k begin to exceed a specified size.

Finally, if we have access to some prior records of decided cases, then we can use the *True/False/Absent/Unavailable* values of these records to compute initial low-cost assignment files F_k and P_k.

The person taking part in the QA process may request an explanation for the decision produced for a pair (d_k, α_k). We sketch how such explanations are computed.

10.10 Explanations

Whenever an explanation is demanded for a decision concerning some (d_k, α_k), the QA process is temporarily halted, and the desired explanation is computed. The explanation depends on whether (d_k, α_k) is impossible, proved, or futile.

Case of Impossible Pair

If (d_k, α_k) is impossible, we know that, for some d_l representing the compound case $\neg d_k$ with $\alpha_l = \alpha_k$, (d_l, α_l) is proved, and an explanation for the latter decision suffices as explanation for the former one. Thus, this case becomes the next one.

Case of Proved Pair

If (d_k, α_k) is proved, we carry out Algorithm EXPLAIN DECISION (5.7.11) with the following input. The clauses of S with level $\geq \alpha_k$ are the input clauses. The x_j are the input variables of layer 0, while d_k is the input decision variable of the top layer. In Step 2 of Algorithm EXPLAIN DECISION (5.7.11), the fixing of the input variables x_j must handle not only *True/False* values but also *Absent/Unavailable* values. The output of Algorithm EXPLAIN DECISION (5.7.11) provides the desired explanation why (d_k, α_k) is proved.

Case of Futile Pair

If (d_k, α_k) is futile, we derive from S a CNF formula S' by deleting all clauses with level $< \alpha_k$, fix the x_j to the *True/False/Absent/Unavailable* values on hand, and fix $d_k = False$. We derive a system S'' from S' by deleting all open x_j, and use Algorithm SOLVE SAT to get a satisfying solution for S''. Such a solution must exist since Heuristic SOLVE Q-ALL SAT (4.6.9) has established (d_k, α_k) to be futile via such a solution. As explanation, we output that solution and say that (d_k, α_k) is futile because the satisfying solution for S'' obviously is valid for S' regardless of the *True/False* values one may obtain for the open x_j.

Had we proved futility of (d_k, α_k) by an exact algorithm instead of by Heuristic SOLVE Q-ALL SAT (4.6.9), we generally would not know how to give a compact explanation. At least, at the present time, it is not known whether an exact algorithm exists that would allow for a compact explanation.

In the next section, we treat a variation of the QA process involving optimization.

10.11 Variation: Optimization

A QA process may be a component of a larger process. In this section, we present an example where the larger process carries out optimization. Applications abound. For example, advisory systems for regulatory compliance involve such optimization processes. There, one must make decisions that satisfy all regulatory constraints while keeping the total cost of the decisions to a minimum.

In the general case, we have a CNF system S whose variables include *action variables* $a_1, a_2, \ldots, a_p$ and *fact variables* $x_1, x_2, \ldots, x_n$. All clauses of S hold with certainty. For the applications we have in mind, *Absent* and

Unavailable values for the x_j make little sense. Thus, we only allow the values *True* and *False* for the x_j.

For each variable of S, we have costs associated with the values *True* and *False*. We refer to these costs as *S-costs*.

In an advisory system for regulatory compliance, the x_j represent facts describing a situation, and the a_l represent possible ways to cope with the situation. The clauses describe the applicable regulations. The S-costs associated with the variables reflect the costs of the various possible decisions.

Define an assignment of *True/False* values to the action variables a_l to be a *plan of action*, for short *plan*. Given *True/False* values for all x_j, we want a satisfying solution of S that respects the values given for x_j and that, subject to that condition, has minimum total cost. The *True/False* values for the a_l of such a solution constitute an *optimal plan*. Since the values for the x_j are given, their S-costs are irrelevant for the choice of the plan. Thus, we may assume that these S-costs are 0.

In an advisory system for regulatory compliance, the optimal plan is the least-cost way to cope with the situation encoded by the values of the x_j.

As stated, the problem is easily solved. We fix the x_j in S to the given values, getting S', and solve the MINSAT instance defined by S' and the S-costs. If S' turns out to be unsatisfiable, we stop with that conclusion. Indeed, that outcome signals a formulation error that can be identified with the techniques of Chapter 5. On the other hand, if we obtain an optimal solution for the MINSAT instance, then we declare the *True/False* values of that solution for the a_l to be the desired optimal plan.

The problem becomes more difficult when the *True/False* values of the x_j can only be obtained by tests T_i with costs c_i. In that case, we want to minimize the total cost of the tests required to compute an optimal plan. We refer to the costs c_i as *T-costs* to emphasize the difference between them and the earlier introduced S-costs.

In a simplistic approach, we select a collection of tests that determine all *True/False* values for the x_j at one time and that, subject to that condition, minimize the total cost of tests. We then carry out the tests, get the *True/False* values for the x_j, and then proceed as described above. Exercise 10.14.9 guides the reader through the construction of an algorithm for this approach.

Potentially, we can do much better by carrying out one test at a time. In that approach, we select a test, carry it out, and combine the resulting *True/False* values with any values already on hand to a current assignment for the x_j. We fix the x_j of S according to that assignment, getting S'. We solve the MINSAT instance given by S' and the S-costs. If S' is unsatisfiable, we have a formulation error and stop. Otherwise, the solution values for the a_l constitute an optimal plan. Suppose that the plan could

also be produced by the above process if values for all x_j and not just the values of the assignment were used to define the MINSAT instance. In that fortuitous situation, the optimal plan constitutes the desired plan of action, and we say that the optimal plan is *feasible.*

We test for feasibility as follows. We fix the x_j in S according to the assignment and fix the a_l according to the optimal plan. Let S'' result. We check if S'' can be made unsatisfiable by any *True/False* values for the open x_j. This is an instance of Q-ALL SAT (4.2.8). We solve it approximately with Heuristic SOLVE Q-ALL SAT (4.6.9). If the heuristic ascertains that no *True/False* values exist for the open x_j that would make S'' unsatisfiable, then the optimal plan is feasible, and we stop. Otherwise, we do not know if the plan is feasible. We then choose another test and repeat the above steps. The process must stop in a finite number of iterations since eventually values are known for all x_j.

The above description ignores details of the test selection. We return to that issue after discussion of an example.

Example: Regulatory Compliance

We have fact variables x_1, x_2, and x_3 and action variables a_1, a_2, and a_3 in an advisory system for regulatory compliance. The regulations can be expressed by the following rules.

$$\begin{aligned} &(x_1 \wedge \neg x_2) \rightarrow (a_1 \vee a_2) \\ &(x_2 \wedge x_3) \rightarrow a_3 \\ &(\neg x_1 \wedge x_3) \rightarrow \neg a_1 \\ &a_1 \vee a_2 \vee a_3 \end{aligned} \tag{10.11.1}$$

The equivalent CNF clauses define S. They are

$$\begin{aligned} &\neg x_1 \vee x_2 \vee a_1 \vee a_2 \\ &\neg x_2 \vee \neg x_3 \vee a_3 \\ &x_1 \vee \neg x_3 \vee \neg a_1 \\ &a_1 \vee a_2 \vee a_3 \end{aligned} \tag{10.11.2}$$

The S-costs of *True* for a_1, a_2, and a_3 are 2, 7, and 5, respectively. All other S-costs are 0.

We want to compute an optimal, feasible plan for a given situation. At the outset, we have no *True/False* values for the x_j. Hence, we define $S' = S$ and solve the MINSAT instance given by S' and the S-costs. There are several optimal solutions, with total cost equal to 2. One of them assigns *False* to all variables except for a_1. That solution defines the optimal plan $a_1^* = \mathit{True}$ and $a_2^* = a_3^* = \mathit{False}$. Is this plan feasible?

To answer the question, we fix the a_l in S' to the values of the optimal plan and reduce accordingly. The resulting S'' is given by

$$\begin{aligned} &\neg x_2 \vee \neg x_3 \\ &x_1 \vee \neg x_3 \end{aligned} \tag{10.11.3}$$

We carry out Heuristic SOLVE Q-ALL SAT (4.6.9) to decide if S'' of (10.11.3) can be made unsatisfiable by some *True/False* values for the open x_j, which are x_1, x_2, and x_3. The heuristic does not conclude the desired answer, which would say that no *True/False* values exist for the open x_j that make S'' unsatisfiable. Thus, we do not know if the optimal plan on hand is feasible, and must get additional values for the x_j.

We skip details of the test selection and assume that the chosen test determines $x_1 = \mathit{False}$ and $x_3 = \mathit{True}$. We repeat the above steps using these values. Thus, we fix $x_1 = \mathit{False}$ and $x_3 = \mathit{True}$ in S, getting S' with clauses

$$\begin{aligned} &\neg x_2 \vee a_3 \\ &\neg a_1 \\ &a_1 \vee a_2 \vee a_3 \end{aligned} \tag{10.11.4}$$

The solution of the MINSAT instance defined by S' of (10.11.4) and the S-costs produces the optimal plan $a_1 = a_2 = \mathit{False}$ and $a_3 = \mathit{True}$. Is this plan feasible?

To settle the question, we fix the a_l in S' of (10.11.4) to the values of the optimal plan, getting S''. The latter CNF system has no clauses and thus cannot become unsatisfiable by any *True/False* value for x_2, which is the only open x_j. We conclude that the optimal plan $a_1 = a_2 = \mathit{False}$ and $a_3 = \mathit{True}$ is feasible and constitutes the desired plan of action. Accordingly, the advisory system recommends that action a_3 be done.

For discussion of the test selection, we first link feasibility of an optimal plan and futility of a certain goal.

Feasibility and Futility

Suppose we have a current assignment for the x_j and a corresponding optimal plan, say with values a_l^* for the a_l. In S, we fix the x_j according to the assignment, getting S'. Then in S', we fix the a_l according to the optimal plan, getting S''. Define index sets L_+ and L_- from the optimal plan by

$$\begin{aligned} L_+ &= \{l \mid a_l^* = \mathit{True}\} \\ L_- &= \{l \mid a_l^* = \mathit{False}\} \end{aligned} \tag{10.11.5}$$

Define a new conclusion variable d by

$$d = \bigvee_{l \in L_+} \neg a_l \vee \bigvee_{l \in L_-} a_l \tag{10.11.6}$$

Then $\neg d$ is given by

$$\neg d = \bigwedge_{l \in L_+} a_l \wedge \bigwedge_{l \in L_-} \neg a_l \tag{10.11.7}$$

By the definition of L_+ and L_-, enforcing $\neg d$ is equivalent to fixing the a_l according to the optimal plan. Accordingly, we may view S'', which is derived from S' by such fixing, to be S' with the added condition $d = \mathit{False}$, where d is given by (10.11.6).

Consider the goal (d, α), with $\alpha = 100$. By the definition of futility of goals, (d, α) is futile for S and for the current assignment for the x_j if and only if S' with $d = \mathit{False}$, which is S'', remains satisfiable no matter which *True/False* values are assigned to the open x_j. We know that the latter condition implies feasibility of the optimal plan. Thus, futility of the goal (d, α) implies feasibility as well. Due to this relationship, we can use the goal (d, α) in the algorithms of Sections 10.7 and 10.8 for the computation of low-cost assignments and test selection, as follows.

Low-cost Assignment

When an optimal plan has been proved to be feasible, we carry out Algorithm LOW-COST ASSIGNMENT (10.7.5) with the following input. The input CNF system is the given S plus the variable d defined by (10.11.6). The input pair (d_k, α_k) is the futile pair (d, α), with $\alpha = 100$. The input tests T_i, with T-costs c_i, are the tests prior to any reductions by Algorithm OBTAIN VALUES (10.3.2). By (10.11.7), the fixing of d in Step 2 of Algorithm LOW-COST ASSIGNMENT (10.7.5) to *False* amounts to fixing the a_l according to the given optimal plan. The output of the algorithm is a low-cost futile assignment.

We collect in a file F the low-cost futile assignments determined during various executions of the optimization process. The file F is utilized for test selection.

Test Selection

We use Algorithm SELECT TEST (10.8.6) to choose the next test. The input CNF system is S plus the variable d defined by (10.11.6). The input values for the x_j are the values on hand. The input goal set G contains

just the pair (d, α), with $\alpha = 100$. That pair has been proved to be futile. The low-cost assignment file associated with (d, α) is F. The input tests T_i, with T-costs c_i, are the tests on hand. The output is a test T_{i^*} that is estimated to be the most cost-effective test.

Sections 10.7 and 10.8 contain examples for the computation of low-cost assignments and for the test selection. Hence, we skip discussion of examples here.

Algorithm for Optimization with QA Process

We are ready to list the algorithm for the entire optimization process.

(10.11.8) Algorithm OPTIMIZE WITH QA PROCESS. *Carries out optimization with a* QA *process.*

Input: CNF system S with $a_1, a_2, \ldots, a_p$ and $x_1, x_2, \ldots, x_n$ among its variables. S-costs for the variables of S. The S-costs for the x_j are all 0. Low-cost assignment file F. For $1 \leq i \leq m$, test T_i for determining the values of a subset of the x_j at T-cost c_i. The T_i collectively determine all values.

Output: Either: "Formulation error. The current assignment makes S unsatisfiable." Or: An optimal plan that is feasible, and a corresponding low-cost assignment that is added to F.

Requires: Algorithms LOW-COST ASSIGNMENT (10.7.5), OBTAIN VALUES (10.3.2), SELECT TEST (10.8.6), and SOLVE MINSAT, and Heuristic SOLVE Q-ALL SAT (4.6.9).

Procedure:

1. (Get initial values.) Ask if some *True/False* values for the x_j are available. Use the supplied values to define a current assignment w for the x_j. Remove from the tests T_i all indices j for which x_j now has an assigned value.
2. (Get optimal plan.) Fix the x_j of S according to the assignment w, getting S'. With Algorithm SOLVE MINSAT, solve the MINSAT instance defined by S' and the S-costs. If S' is unsatisfiable, output that a formulation error exists since the assignment w makes S unsatisfiable, and stop. Otherwise, define an optimal plan from the optimal values a_l^* for the a_l. In S', fix the a_l according to the optimal plan. Let S'' result. Define $L_+ = \{l \mid a_l^* = \mathit{True}\}$ and $L_- = \{l \mid a_l^* = \mathit{False}\}$.
3. (Check for feasibility.) Do Heuristic SOLVE Q-ALL SAT (4.6.9) with S'' as input CNF system and with the open x_j as the special variables. The input system R is defined to have no clauses. If the heuristic determines that no *True/False* values for the open x_j exist that render S'' unsatisfiable, output the current plan as optimal and feasible, and go to Step 5.

4. (Select next test.) Carry out Algorithm SELECT TEST (10.8.6) with the following input: CNF system S plus $d = (\bigvee_{l \in L_+} \neg a_l) \vee (\bigvee_{l \in L_-} a_l)$, the values of assignment w, goal set G having just one goal (d, α) with $\alpha = 100$, low-cost assignment file F, and tests T_i and T-costs c_i. The output is a cost-effective next test T_{i^*}. Do Algorithm OBTAIN VALUES (10.3.2) with test T_{i^*}, incorporate the resulting values into assignment w, and go to Step 2.
5. (Derive low-cost assignment.) Restore all tests T_i to the original input sets. Do Algorithm LOW-COST ASSIGNMENT (10.7.5) with the following input: CNF system S plus $d = (\bigvee_{l \in L_+} \neg a_l) \vee (\bigvee_{l \in L_-} a_l)$, the values of assignment w, the pair (d, α) having $\alpha = 100$ and shown to be futile, and tests T_i with T-costs c_i. The algorithm outputs a low-cost futile assignment. Add that assignment to F, and stop.

QA processes may be used to evaluate DNF formulas that have been learned from data as described in Chapters 7 and 8. The next section provides details.

10.12 Evaluation of Learned Formulas

The evaluation of learned DNF formulas may involve sequential tests where one successively obtains additional information about the situation at hand. Subproblem 4 in Section 7.5 of Chapter 7 as well as the corresponding case in Section 8.4 of Chapter 8 involve such sequential tests. In this section, we view the selection of the sequential tests as a particular QA process.

Regardless of the specific case, the QA process requires that the learned DNF formulas are inserted into a CNF system. Such insertion may be useful in other settings. Thus, we treat a more general situation. At first glance, insertion of DNF formulas into a CNF system may appear trivial. But it requires care, as we shall see next.

Insertion into CNF System

Let D be a learned DNF formula with variables $x_1, x_2, \ldots, x_n$. Suppose that, for any values assigned to $x_1, x_2, \ldots, x_n$, the formula evaluates to *True* if and only if a certain conclusion d holds. Assume we want to add one of the implications $d \rightarrow D$ and $d \leftarrow D$ or possibly both of them to a CNF system S. That step is easy if none of the variables $x_1, x_2, \ldots, x_n$ can ever take on the value *Unavailable*. Indeed, we convert the given implications to equivalent CNF formulas, as described in Section 2.2 of Chapter 2, and adjoin the latter formulas to S. However, if fixing of some x_j at the value *Unavailable* may occur, then such an insertion may introduce an error. Here is an example.

Example: Erroneous Use of DNF Formula

Suppose we want to use the learned DNF formula $D = x_1$ to express the condition $d \rightarrow D$. Naïvely, we represent that fact by $d \rightarrow x_1$ or, equivalently, by $\neg d \vee x_1$, and then add the latter CNF clause to S. Consider the case where $x_1 = \mathit{Unavailable}$. According to Section 10.4, the CNF clause $\neg d \vee x_1$ evaluates to *True* and thus does not constrain d. On the other hand, $x_1 = \mathit{Unavailable}$ causes the DNF formula $D = x_1$ to become $D' = \mathit{False}$. Thus, the condition $d \rightarrow D$ becomes $d \rightarrow D' = d \rightarrow \mathit{False} = \neg d$. Accordingly, the clause $\neg d \vee x_1$ produced by the naïve approach is in error.

Insertion Rule

A simple rule avoids potential errors of direct insertion. We need a few definitions. Let D be a DNF formula $D = D_1 \vee D_2 \vee \cdots \vee D_k$, where the D_l are the DNF clauses. Assume that $x_1, x_2, \ldots, x_n$ are the variables of D. Let R be a propositional formula which uses a variable D and which does not use any one of the variables x_j. Define S to be a CNF formula that does not have D as a variable, but that may have any number of variables x_j.

We want a CNF formula that is equivalent to $S \wedge R \wedge [D \leftrightarrow (D_1 \vee D_2 \vee \cdots \vee D_k)]$ while some of the x_j are fixed at the value *Unavailable*. The rule for constructing the desired CNF formula is as follows.

(10.12.1) Rule. *First, apply the Unavailable values to reduce the* DNF *formula D to, say, D'. Specifically, each clause D_l of D containing a literal of x_j with $j \in J$, evaluates to False. If by this process all D_l evaluate to False, then $D' = \mathit{False}$. Otherwise, D' is obtained from D by deleting all D_l having value False. Second, replace in R each occurrence of the variable D by the reduced formula D', getting R', and then convert R' to an equivalent* CNF *formula R''. Third, reduce S to S' by deleting all* CNF *clauses containing a literal of any x_j with $j \in J$. Then $S' \wedge R''$ is the desired* CNF *formula.*

Proof of Validity. By the derivation, D', R', and S' correctly represent D, R, and S for the given *Unavailable* values. Since R'' is equivalent to R', $S' \wedge R''$ correctly represents $S \wedge R \wedge [D \leftrightarrow (D_1 \vee D_2 \vee \cdots \vee D_k)]$ for these values. □

We present an example.

Example: Medical Diagnosis

Suppose we have learned the DNF formula

$$D = (x_1 \wedge x_2) \vee (x_3 \wedge \neg x_4) \tag{10.12.2}$$

The formula indicates whether a patient has a certain disease d, and it has value *True* (resp. *False*) when the symptoms $x_1, x_2, \ldots, x_n$ take on *True/False/Unavailable* values of a patient with (resp. without) the disease.

Suppose we want to add the condition $R = d \to D$ to a CNF system S. Let x_1 be the only variable with value *Unavailable*. That value of x_1 produces the value *False* for the DNF clause $x_1 \wedge x_2$ of D and does not affect the clause $x_3 \wedge \neg x_4$. Thus, $x_1 = \textit{Unavailable}$ reduces D to

$$D' = x_3 \wedge \neg x_4 \tag{10.12.3}$$

The condition $R = d \to D$ becomes $R' = d \to D' = d \to (x_3 \wedge \neg x_4)$, and a CNF formula R'' equivalent to R' is given by the CNF clauses

$$\begin{aligned} &\neg d \vee x_3 \\ &\neg d \vee \neg x_4 \end{aligned} \tag{10.12.4}$$

Suppose $x_1 = \textit{Unavailable}$ reduces S to S'. Then $S' \wedge R''$ correctly represents $S \wedge R$ for the assumed case $x_1 = \textit{Unavailable}$.

Special Case of Direct Insertion

There are situations where Rule (10.12.1) is not needed, and where a direct use of a DNF formula in S is correct. An important case is as follows.

(10.12.5) Theorem. *Let D be a* DNF *formula $D = D_1 \vee D_2 \vee \cdots \vee D_k$, where the D_l are the* DNF *clauses. Then the condition $D \to d$ is correctly enforced by the addition of the* CNF *formula $\bigwedge_{l=1}^{k} (\neg D_l \vee d)$ to S even if some of the variables of D may take on the value Unavailable.*

The proof of the theorem is not difficult. One establishes that direct insertion as described and indirect insertion via Rule (10.12.1) produce the same result for any possible assignment of *Unavailable* to the variables of D. Part (b) of Exercise (10.14.11) asks the reader to fill in the details.

Section 7.5 of Chapter 7 defines, but does not solve, a problem called Subproblem 4. We use Theorem (10.12.5) and Algorithm QA PROCESS (10.9.1) to handle that problem here. Before proceeding, the reader may want to review Chapter 7 up to the introduction of Subproblem 4.

Subproblem 4 of Section 7.5

The input of the subproblem consists of the following: optimized record sets A^* and B^*, which have been derived from training sets A and B; optimized test sets K^* and L^*; and optimized DNF formulas D^{*min}, D^{*max}, E^{*min}, and E^{*max}.

The training sets A and B have been taken from populations $\mathcal{A}$ and $\mathcal{B}$, respectively. The optimized test set K^* is defined by $K^* = \{K_r^* \mid r \in A\}$, where each K_r^* is a cost-effective set of tests for classifying record $r \in A$ via the formula D^{*min} or D^{*max}. Similarly, the optimized test set L^* is $L^* = \{L_s^* \mid s \in B\}$, where each L_s^* is a cost-effective set of tests for classifying record $s \in B$ via the formula E^{*min} or E^{*max}. It is convenient to define

$$\tilde{K}^* = \bigcup_{r \in A} K_r^* \qquad \tilde{L}^* = \bigcup_{s \in B} L_s^* \tag{10.12.6}$$

Thus, $\tilde{K}^*$ (resp. $\tilde{L}^*$) is the set of tests that are considered cost-effective for classification of the records of the training sets A and B by D^{*min} or D^{*max} (resp. E^{*min} or E^{*max}).

Subproblem 4 demands that we use the above sets and formulas to classify a record r of $\mathcal{A} \cup \mathcal{B}$ where the record entries are obtained by sequentially selected tests. Below, we solve that problem by a QA process.

QA Process with Learned Formulas

We begin with the special case where we just use the DNF formula D^{*min} to classify record r and where we ignore the remaining formulas D^{*max}, E^{*min}, and E^{*max}. The record entries of r are represented by variables x_1, $x_2, \ldots, x_n$ whose *True/False/Unavailable* values are determined by tests.

Define a variable d that has the value *True* if record r is predicted to be in population $\mathcal{A}$ (resp. $\mathcal{B}$). Since D^{*min} is predicted to have the value *True* (resp.*False*) for the records of $\mathcal{A}$ (resp. $\mathcal{B}$), the following implication is valid.

$$D^{*min} \rightarrow d \tag{10.12.7}$$

Suppose $D^{*min} = D_1 \vee D_2 \vee \cdots \vee D_k$, where the D_l are the DNF clauses. The variables of D^{*min} are taken from the set $\{x_1, x_2, \ldots, x_n\}$. Define S to be the CNF system with variables d and $x_1, x_2, \ldots, x_n$, and with CNF clauses equivalent to (10.12.7). Using Theorem (10.12.5), the CNF clauses are

$$\neg D_l \vee d, \quad l = 1, 2, \ldots, k \tag{10.12.8}$$

Example

Let $D^{*min} = (x_1 \wedge x_2) \vee \neg x_3$. Thus, D^{*min} consists of the DNF clauses $D_1 = x_1 \wedge x_2$ and $D_2 = \neg x_3$. Using (10.12.8), the input CNF system S for the QA process has the clauses

$$\begin{aligned} \neg D_1 \vee d &= \neg x_1 \vee \neg x_2 \vee d \\ \neg D_2 \vee d &= x_3 \vee d \end{aligned} \tag{10.12.9}$$

By the definition of S, getting enough values for the entries x_j of record r so that D^{*min} evaluates to *True* (resp. *False*) is equivalent to getting enough values for the variables x_j of S so that one can conclude that d is (resp. cannot possibly become) a theorem. Thus, if d can be proved to be a theorem, record r is predicted to be in population $\mathcal{A}$. If such a proof is shown to be futile, record r is predicted to be in population $\mathcal{B}$. We acquire enough values for $x_1, x_2, \ldots, x_n$ with Algorithm QA PROCESS (10.9.1) to decide which case applies. The input for the algorithm consists of the following: CNF system S defined above; a goal set G containing just the element (d, α), with $\alpha = 100$; low-cost assignment files P and F defined below; and tests T_i with costs c_i, also defined below.

The low-cost assignment file P contains the optimized records of A^*, since they allow cost-effective proof of membership of records in $\mathcal{A}$.

We do not have corresponding records for the file F, which is to contain low-cost assignment records for the case where proof of d is futile. Indeed, the records of B^*, which may seem to be candidates, are not appropriate, since they were not used to establish D^{*min}. Instead, we define a set J to consist of the indices j, $1 \leq j \leq n$, for which no literal of x_j occurs in D^{*min}. Let $\overline{B}$ be derived from the training set B by replacing, in each record of B, the values of the x_j with $j \in J$ by *Unavailable*. By this definition, D^{*min} evaluates to *False* for all records of $\overline{B}$.

The tests T_i, $j = 1, 2, \ldots, m$, are the tests specified in $\tilde{K}^*$ of (10.12.6). The cost c_i of test T_i is the given cost for that test.

With the above input, Algorithm QA PROCESS (10.9.1) determines *True/False/Unavailable* values for record r in a cost-effective manner so that either d is proved to be a theorem of S, or such a proof is shown to be futile. In the first (resp. second) case, record r is predicted to be in population $\mathcal{A}$ (resp. $\mathcal{B}$). Step 4 of Algorithm QA PROCESS (10.9.1) determines low-cost assignments for P and F. Since we do not know whether the classification of r is correct, we skip that step.

We could expand the above discussion so that the QA process does not evaluate just D^{*min}, but also evaluates D^{*max}, E^{*min}, and E^{*max} of Subproblem 4 of Section 7.5. Below, we handle that case as well as others by a more general procedure. In particular, the evaluation of $10k$ optimized formulas by sequential tests described in Section 8.4 of Chapter 8 is also covered.

General Collection of Formulas

We assume the following, general situation. For $g = 1, 2, \ldots, N$, we have DNF formula $H^g = H_1^g \vee H_2^g \vee \cdots \vee H_{k(g)}^g$, where the H_l^g are the DNF clauses. The variables of H^g are taken from $\{x_1, x_2, \ldots, x_n\}$. We also have an optimized record set HP^g and a second, not optimized, record set HF^g. The formula H^g evaluates to *True* (resp. *False*) for the records of HP^g (resp. HF^g). Finally, we have a set K^g of tests and their costs. The tests are a cost-effective way to get record values for evaluation by H^g.

As an example, we define H^g, HP^g, HF^g, and K^g for the case where D^{*min}, D^{*max}, E^{*min}, and E^{*max} are to be evaluated. Thus, $N = 4$. The DNF formulas are $H^1 = D^{*min}$, $H^2 = D^{*max}$, $H^3 = E^{*min}$, and $H^4 = E^{*max}$. The optimized sets on which the H^g have the value *True* are $HP^1 = HP^2 = A^*$ and $HP^3 = HP^4 = B^*$. The non-optimized sets on which the H^g have the value *False* are $HF^1 = HF^2 = B$ and $HF^3 = HF^4 = A$. Finally, the test sets are $K^1 = K^2 = \tilde{K}^*$ and $K^3 = K^4 = \tilde{L}^*$.

Algorithm for QA Process with Learned Formulas

The algorithm presented next is a cost-effective way to evaluate a record r simultaneously by all H^g.

(10.12.10) Algorithm QA WITH LEARNED FORMULAS. *Evaluates record by learned formulas, using a* QA *process.*

Input: Record r, with entries denoted by variables $x_1, x_2, \ldots, x_m$. Each x_j has assigned value *Absent*, which means that the *True/False/Unavailable* value is not known. For $g = 1, 2, \ldots, N$, the following formulas and sets are given. DNF formula $H^g = H_1^g \vee H_2^g \vee \cdots \vee H_{k(g)}^g$, where the H_l^g are the DNF clauses. The variables of each H^g are taken from $\{x_1, x_2, \ldots, x_n\}$. Optimized record set HP^g and a second, not optimized, record set HF^g. The formula H^g evaluates to *True* (resp. *False*) for the records of HP^g (resp. HF^g). Set K^g of tests and their costs. The tests are a cost-effective way to get record values for evaluation of H^g.

Output: Enough entries for record r so that each formula H^g evaluates to *True* or *False*. The output includes these *True/False* values.

Requires: Algorithm QA PROCESS (10.9.1).

Procedure:

1. (Define CNF system S.) Let S be the following CNF system. The variables are $d_1, d_2, \ldots, d_N$ and $x_1, x_2, \ldots, x_n$. The clauses are, for $1 \leq g \leq N$ and $1 \leq l \leq k(g)$, $\neg H_l^g \vee d_g$.
2. (Define goal set G.) Let G be the goal set whose elements are, for $1 \leq g \leq N$, (d_g, α_g), with $\alpha_g = 100$.

3. (Define low-cost assignment files P and F.) For $1 \leq g \leq N$, let P_g be the file containing the records of HP^g. Let J_g be the set of indices j such that H^g has no literal of x_j. Define the records of the file F_g from the records of HF^g by replacing in each such record all entries x_j with $j \in J_g$ by *Unavailable*.
4. (Define tests T_i and costs c_i.) Let $T_1, T_2, \ldots, T_m$ be the elements of $\bigcup_{g=1}^{N} K^g$, and denote the associated costs by $c_1, c_2, \ldots, c_m$.
5. (Carry out QA process.) Carry out Algorithm QA PROCESS (10.9.1) with the input at hand, except that Step 4 of the algorithm is skipped. For $1 \leq g \leq N$, if the algorithm outputs "d_g is proved" (resp. "d_g is futile"), output that the DNF formula H^g has the value *True* (resp. *False*). Output these values, plus the values for the entries of r determined by the algorithm, and stop.

The proof of validity of Algorithm QA WITH LEARNED FORMULAS (10.12.10) is almost identical to that given earlier for the special case where just one formula, D^{*min}, is evaluated. Exercise (10.14.13) calls for details of the proof.

Exercise (10.14.14) asks the reader to define the input for Algorithm QA WITH LEARNED FORMULAS (10.12.10) when several DNF formulas produced by Algorithm OPTIMIZED DNF FORMULAS (7.5.11) or Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2) are to be simultaneously evaluated. The solution of the exercise thus covers the case of formulas D^{*min}, D^{*max}, E^{*min}, and E^{*max} of Subproblem 4 in Section 7.5 of Chapter 7 as well as the corresponding case of $10k$ formulas in Section 8.4 of Chapter 8.

Looking Ahead

Sections 11.4 and 11.8 of Chapter 11 discuss example applications for QA processes.

10.13 Further Reading

For a basic discussion of expert systems, see recent introductory artificial intelligence texts such as Nilsson (1998), Luger (2002), Negnevitsky (2002), and Russell and Norvig (2003).

The reader may also want to delve into books on expert systems such as Leondes (1993), Giarratano and Riley (1994), Poole, Mackworth, and Goebel (1998), and Hopgood (2001).

The approach to QA processes described in this chapter is based on Straach (1998) and Truemper (1999).

10.14 Exercises

The calculations required by some of the exercises may be accomplished with the software of Exercise (10.14.16).

(10.14.1) (Alternate representation of *Unavailable*) Consider the following modification of a CNF system S which has $x_1, x_2, \ldots, x_n$ among its variables.

(a) For each x_j, introduce an additional variable u_j.

(b) In each clause of S, replace each literal x_j (resp. $\neg x_j$) by $x_j \vee u_j$ (resp. $\neg x_j \vee u_j$).

(c) Encode a *True/False/Absent/Unavailable* value for x_j as follows.
True/False: Fix x_j at the *True/False* value and define $u_j = False$.
Absent: Declare x_j to be free and $u_j = False$.
Unavailable: Declare x_j to be free and $u_j = True$.

Show that this representation is equivalent to the one introduced in Section 10.4, where *Unavailable* triggers deletion of clauses.

(10.14.2) Let S be given by the clauses

$$\neg x_1 \vee \neg x_2 \vee d_1$$
$$x_3 \vee \neg d_1 \vee d_2$$
$$\neg x_1 \vee \neg x_4 \vee d_3$$

Suppose $T_1 = \{1, 2\}$, $T_2 = \{2, 3\}$, and $T_3 = \{1, 3\}$. Carry out Algorithm OBTAIN VALUES (10.3.2) with this input and with the requirement that test T_1 is to be performed. Assume that T_1 produces $x_1 = x_2 = True$.

(10.14.3) Derive S' from the CNF system S with the clauses

$$\neg x_1 \vee x_2 \vee d_1 \vee \neg d_1$$
$$x_1 \vee \neg x_3 \vee x_4 \vee d_3$$
$$x_2 \vee x_4 \vee \neg x_5 \vee d_2$$

by fixing $x_1 = Unavailable$ and $x_2 = False$.

(10.14.4)

(a) In Section 10.4, the reduction of S to S' due to fixing of variables assumes that the values are known with certainty. Modify the reduction process for the case where x_j values are known only at some likelihood levels. (*Hint*: A value for x_j at level α should apply only to clauses with likelihood level less than or equal to α.)

(b) Let $x_1 = Unavailable$, $x_2 = True$, $x_3 = False$ at level 60, $x_4 = Unavailable$ at level 70, and $x_5 = Absent$. Use these values to derive S' from the CNF system S defined by the clauses

$$\neg x_2 \vee x_3 \vee x_4 \vee \neg x_5 \vee d_1 \vee d_2 \quad \textit{at level } 90$$
$$x_2 \vee x_3 \quad \textit{at level } 80$$
$$x_1 \vee \neg x_3 \vee \neg d_3 \quad \textit{at level } 40$$
$$\neg x_2 \vee x_3 \vee d_2 \vee d_3 \quad \textit{at level } 20$$

(10.14.5) Carry out Algorithm PROVE CONCLUSIONS (10.5.1) with the following input. The CNF system has the clauses

$$\neg x_1 \vee x_2 \vee d_1 \vee d_2 \quad \textit{at level } 90$$
$$\neg x_3 \vee \neg d_1 \quad \textit{at level } 85$$
$$x_3 \vee \neg x_4 \vee d_4 \quad \textit{at level } 60$$

The goal set is $G = \{(d_1, 80), (\neg d_1, 80), (d_2, 70)\}$. The result set H is empty. The known x_j values are $x_1 = \textit{True}$, $x_2 = \textit{False}$, and $x_3 = \textit{True}$.

(10.14.6) Provide details of the steps of Algorithm LOW-COST ASSIGNMENT (10.7.5) for the two example cases of Section 10.7, where S is given by (10.7.1) and (10.7.3).

(10.14.7) Carry out Algorithm SELECT TEST (10.8.6) with the following input. The CNF system is given by

$$\neg x_1 \vee \neg x_2 \vee d_1$$
$$x_3 \vee \neg d_1 \vee d_2$$
$$\neg x_1 \vee \neg x_4 \vee d_3$$

There is just one known x_j value, which is $x_1 = \textit{True}$. The goal set G is $G = \{(d_1, 100), (d_2, 100), (d_3, 100)\}$. We have the following low-cost assignment files F_k and P_k, where each pair of parentheses delimits one assignment. For example, $(x_1 = \textit{False})$ and $(x_1 = x_2 = \textit{True})$ are two assignments. Note that multiple copies occur for some assignments, as is allowed in the files F_k and P_k.

F_1: $(x_1 = \textit{False})$; $(x_2 = \textit{False})$; $(x_2 = \textit{False})$.
P_1: $(x_1 = x_2 = \textit{True})$.
F_2: $(x_1 = \textit{False})$; $(x_1 = \textit{False})$; $(x_2 = \textit{False})$; $(x_2 = \textit{False})$.
P_2: $(x_1 = x_2 = \textit{True}, x_3 = \textit{False})$.
F_3: $(x_1 = \textit{False})$; $(x_1 = \textit{False})$; $(x_4 = \textit{False})$.
P_3: $(x_1 = x_4 = \textit{True})$.

The tests are $T_1 = \{1, 2\}$, $T_2 = \{2\}$, and $T_3 = \{2, 3, 4\}$, with costs $c_1 = 5$, $c_2 = 3$, and $c_3 = 12$.

(10.14.8) Carry out Algorithm QA PROCESS (10.9.1) using the data of Exercise (10.4.7), except that initially no x_j value is known. Assume that

the underlying values of the x_j produced by tests are $x_1 = x_2 = x_3 = \mathit{True}$ and $x_4 = \mathit{False}$.

(10.14.9) Define an algorithm for the optimization case where initially tests are done to get values for all x_j. (*Hint*: For the test selection, consider Steps (a)–(c) below.

(a) For each x_j, let I_j be the set of indices i for which $j \in T_i$, and define the clause $\bigvee_{i \in I_j} \mathit{choose}(T_i)$.

(b) For each $\mathit{choose}(T_i)$, define the cost of *True* (resp. *False*) to be c_i (resp. 0).

(c) Solve the MINSAT instance given by the clauses and costs of Steps (a) and (b).)

(10.14.10) Carry out Algorithm OPTIMIZE WITH QA PROCESS (10.11.8) for the following input. The CNF system is given by

$$\neg x_1 \vee \neg x_2 \vee d_1$$
$$x_3 \vee \neg d_1 \vee d_2$$
$$\neg x_1 \vee \neg x_4 \vee d_3$$

The S-costs of *True* for d_1, d_2, d_3, and d_4 are 7, -5, 2, -10, respectively. All other S-costs are 0. The low-cost assignment file F is empty. The tests are $T_1 = \{1, 2\}$, $T_2 = \{2\}$, and $T_3 = \{2, 3, 4\}$, with T-costs $c_1 = 5$, $c_2 = 3$, and $c_3 = 12$. Assume that the underlying values of the x_j produced by the tests are $x_1 = \mathit{True}$, $x_2 = \mathit{False}$, $x_3 = \mathit{True}$, and $x_4 = \mathit{False}$.

(10.14.11)

(a) Let D be the learned DNF formula $(x_1 \wedge x_3) \vee (\neg x_1 \wedge x_5 \neg \wedge x_6) \vee (x_4 \wedge \neg x_5)$. For the case $x_1 = \mathit{Unavailable}$, find CNF clauses that correctly represent the condition $D \leftrightarrow d$.

(b) Let D be the DNF formula $D = D_1 \vee D_2 \vee \cdots \vee D_k$, where the D_l are the DNF clauses. The variables of D are $x_1, x_2, \ldots, x_n$. Prove that the condition $D \rightarrow d$ is correctly encoded by the CNF clauses $\neg D_l \vee d$, $l = 1, 2, \ldots, k$ even if some of the variables x_j take on the value *Unavailable*. (*Hint*: Assume that some of the x_j have the value *Unavailable*. Eliminate CNF clauses $\neg D_l \vee d$ accordingly. Show that the same CNF clauses result when Rule (10.12.1) is used instead.)

(10.14.12) Modify the algorithms of this chapter so that the following cases are correctly handled.

(a) A test T_i is done, but due to operator error the value for some x_j with $j \in T$ is not determined. A second application of T_i, at a given lower cost, is guaranteed to produce a value for x_j.

(b) Test T_i has produced a value for x_j. Since the value of x_j is known now, another test T_k with $j \in T_k$ no longer needs to produce the value of x_j and thus has lower cost.

(*Hint*: Check Algorithms OBTAIN VALUES (10.3.2), LOW-COST ASSIGNMENT (10.7.5), and SELECT TEST (10.8.6).)

(10.14.13) Validate Algorithm QA WITH LEARNED FORMULAS (10.12.10). (*Hint*: Review the arguments made in Section 10.12 for the special case where D^{*min} is to be evaluated.)

(10.14.14) Define the input for Algorithm QA WITH LEARNED FORMULAS (10.12.10) when some subset of formulas produced by one of the algorithms of (a) or (b) below is to be evaluated by sequential tests.
(a) Algorithm OPTIMIZED DNF FORMULAS (7.5.11).
(b) Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2).

(10.14.15) Carry out Algorithm QA WITH LEARNED FORMULAS (10.12.10), where the optimized DNF formulas and related sets are computed by Exercises (7.10.6) and (7.10.8). The tests and related costs are given in Exercise (7.10.6). The values of the entries of record r, which may only be obtained by tests, are $x_1 = \mathit{False}$, $x_2 = x_3 = \mathit{True}$, $x_4 = \mathit{False}$, and $x_5 = \mathit{Unavailable}$.

(10.14.16) (Optional programming project) There are two options if the reader wishes to avoid manual solution of some of the exercises. First, the reader may obtain commercially or publicly available software for at least some of the problems. Any search engine on the Internet will point to such software. Second, the reader may opt to write programs that implement the algorithms of (a)–(g) below, and then use that software to solve the exercises. The task is easy if the reader has already created programs for SAT and MINSAT as described in Exercise (2.9.13), and for the Heuristics SOLVE Q-ALL SAT (4.6.9), SOLVE Q-MINFIX UNSAT (4.6.16), and SOLVE Q-MINFIX SAT (4.6.22).
(a) Algorithm OBTAIN VALUES (10.3.2).
(b) Algorithm PROVE CONCLUSIONS (10.5.1).
(c) Algorithm LOW-COST ASSIGNMENT (10.7.5).
(d) Algorithm SELECT TEST (10.8.6).
(e) Algorithm QA PROCESS (10.9.1).
(f) Algorithm OPTIMIZE WITH QA PROCESS (10.11.8).
(g) Algorithm QA WITH LEARNED FORMULAS (10.12.10).

Part V

Applications

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Chapter 11

Applications

11.1 Overview

We take a tour of some applications. For each case, we cover particularly important aspects in detail and summarize the remainder. At the end of each section, we include relevant references as well as a list of the preceding chapters containing relevant theoretical results and tools. The latter list is intended for the reader who has begun reading the book in this chapter and who wants to handle a specific application. In that case, the reader need only study the listed chapters.

Section 11.2 shows how correctness of design of certain equipment can be achieved by theorem proving.

Section 11.3 describes a component of a music composition assistant.

Section 11.4 covers a cost-effective management system of hazardous materials.

Section 11.5 describes an intelligent control system for traffic lights.

Section 11.6 shows that creditworthiness can be assessed using logic formulas learned from data.

Section 11.7 treats the problem of deciding the sense of words in texts.

Section 11.8 defines a process that virtually automatically creates certain medical diagnostic systems.

11.2 Correctness of Design

The control of automated manufacturing equipment essentially consists of checks of equipment status and commands for actions. The status checks may trigger, interrupt, or terminate an action. Some of the checks are difficult to design and costly to implement, and often turn out to be incomplete. For example, when a robot arm delivers a subassembly, it may be difficult to check whether that movement is on a collision course with the simultaneous move of a second arm.

The designer of the equipment handles such conflicts by visualizing or modeling the production process and by inserting checks as conflicting actions arise. Despite diligent effort, the initial design typically is not quite correct, and flaws surface during testing of the equipment. By suitable adjustments, these errors are corrected, and a seemingly correct machine is delivered and installed. We say "seemingly," since even small changes in the action sequence may cause an unanticipated collision, and costly repairs.

We can eliminate these difficulties by designing checks that are correct regardless of the selected sequence of actions. The construction of such checks is difficult for humans, but can be effectively handled by an intelligent system. In this section, we cover the logic computations carried out by such a system. We begin with an example.

Machine Crash

Suppose the movements of two robot arms are potentially in conflict. We assume that each arm moves from one point to a second point and back along a fixed trajectory. Suppose the trajectories of the two robot arms intersect. Thus, simultaneous movement of the arms may produce a collision.

There are several ways to avoid that event. The easiest and most reliable way consists of a check that prevents simultaneous movements of the two arms. Suppose two sensors tell the control program when the first arm is in one of its two end positions. Let s_1 and s_2 be logic variables representing the sensors. Assume that $s_i = \mathit{True}$ (resp. $s_i = \mathit{False}$) if the arm is (resp. is not) at the end position equipped with sensor s_i. Since the arm can be at most at one end point, at most one s_i can have the value *True*. The clause

$$\neg s_1 \vee \neg s_2 \tag{11.2.1}$$

expresses this fact.

Analogously, let t_1 and t_2 be logic variables representing two sensors of the trajectory end points for the second arm. We thus have the clause

$$\neg t_1 \vee \neg t_2 \tag{11.2.2}$$

Let m_1 be a logic variable that has the value *True* if the control program issues the command that the first arm should move. To simplify the discussion, we assume that the command always results in the desired move. Thus, once the command has been issued, the arm is moving and is not at one of the two endpoints. Accordingly, both sensors s_1 and s_2 must have the value *False*, and the clause

$$m_1 \rightarrow (\neg s_1 \wedge \neg s_2) \tag{11.2.3}$$

is satisfied. Let m_2 represent the movement of the second arm. Since t_1 and t_2 are the sensors for the endpoints of that arm, we have analogously to (11.2.3) the clause

$$m_2 \rightarrow (\neg t_1 \wedge \neg t_2) \tag{11.2.4}$$

Together, the clauses (11.2.1)–(11.2.4) are an abstract *model* of the two arm movements of the machine.

To prevent collisions, we allow a movement of the first arm only if the second arm is at one of the end points. That condition is expressed by

$$m_1 \rightarrow (t_1 \vee t_2) \tag{11.2.5}$$

For the second arm, we have

$$m_2 \rightarrow (s_1 \vee s_2) \tag{11.2.6}$$

We call the clauses (11.2.5) and (11.2.6) the *check conditions* of the machine.

When converted to CNF, the model clauses (11.2.1)–(11.2.4) and the check conditions (11.2.5) and (11.2.6) define the following CNF system S.

$$\begin{aligned}
&\neg s_1 \vee \neg s_2\\
&\neg t_1 \vee \neg t_2\\
&\neg m_1 \vee \neg s_1\\
&\neg m_1 \vee \neg s_2\\
&\neg m_2 \vee \neg t_1\\
&\neg m_2 \vee \neg t_2\\
&\neg m_1 \vee t_1 \vee t_2\\
&\neg m_2 \vee s_1 \vee s_2
\end{aligned} \tag{11.2.7}$$

Suppose the control program of the machine enforces that movements of the two arms must obey the check conditions. Thus, if any check condition evaluates to *False*, the control program stops the machine.

Can anything go wrong? If so, it would be simultaneous movements of the two arms. We describe that *crash scenario* by the statement

$$m_1 \wedge m_2 \tag{11.2.8}$$

The crash scenario can never happen if and only if $T = \neg(m_1 \wedge m_2)$ is a theorem of S. To decide the question, we check if $S \wedge \neg T$ is satisfiable. If this is the case, then T is not a theorem, and a crash may occur. Otherwise, a crash is not possible. Since enforcing $\neg T = m_1 \wedge m_2$ is equivalent to fixing $m_1 = m_2 = \mathit{True}$, we check satisfiability of $S \wedge \neg T$ by enforcing these values of the crash scenario in S of (11.2.7). One readily confirms that the resulting CNF system has the clauses

$$\begin{aligned} &\neg s_1 \vee \neg s_2 \\ &\neg t_1 \vee \neg t_2 \\ &\neg s_1 \\ &\neg s_2 \\ &\neg t_1 \\ &\neg t_2 \\ &t_1 \vee t_2 \\ &s_1 \vee s_2 \end{aligned} \tag{11.2.9}$$

and is unsatisfiable. Thus, a crash is not possible.

Suppose we do not include the condition $m_1 \rightarrow (t_1 \vee t_2)$ of (11.2.5) among the check conditions. Thus, S of (11.2.7) becomes S_1 with clauses

$$\begin{aligned} &\neg s_1 \vee \neg s_2 \\ &\neg t_1 \vee \neg t_2 \\ &\neg m_1 \vee \neg s_1 \\ &\neg m_1 \vee \neg s_2 \\ &\neg m_2 \vee \neg t_1 \\ &\neg m_2 \vee \neg t_2 \\ &\neg m_2 \vee s_1 \vee s_2 \end{aligned} \tag{11.2.10}$$

When we enforce the values $m_1 = m_2 = \mathit{True}$ of the crash scenario in S_1, the reduced CNF system is unsatisfiable, and a crash is not possible. Thus, the check condition (11.2.5) could be omitted. However, we shall see later that dropping that condition turns out to be a bad decision if sensors fail.

Now suppose that, due to an oversight, both check conditions (11.2.5) and (11.2.6) are omitted. Then S of (11.2.7) becomes S_2 with clauses

$$\begin{array}{l} \neg s_1 \vee \neg s_2 \\ \neg t_1 \vee \neg t_2 \\ \neg m_1 \vee \neg s_1 \\ \neg m_1 \vee \neg s_2 \\ \neg m_2 \vee \neg t_1 \\ \neg m_2 \vee \neg t_2 \end{array} \tag{11.2.11}$$

When we enforce $m_1 = m_2 = \mathit{True}$, we see that the reduced system has the satisfying solution $s_1 = s_2 = t_1 = t_2 = \mathit{False}$, and the crash given by T can happen. Indeed, the satisfying solution describes a configuration of the machine under which the crash may occur.

It is not difficult to verify that the conditions $\neg s_1 \vee \neg s_2$ and $\neg t_1 \vee \neg t_2$ of (11.2.7) are not needed. Such redundancy of clauses is typical when one formulates the model of the machine. This aspect is of little concern here.

General Case

We have an automated machine that executes a number of steps involving various robotic arms, tools, grippers, fixtures, and so on. We create a CNF model M of the machine that represents the various operations.

We define a CNF system N of check conditions that, according to an initial analysis, should rule out any crash.

We specify crash scenarios $C_1, C_2, \ldots, C_m$ that, we believe, cover all possible crashes.

Define $S = M \wedge N$. For $i = 1, 2, \ldots, m$, we test if $T_i = \neg C_i$ is a theorem of S. For the test, we enforce in S the values of the crash scenario C_i and check the resulting S' for satisfiability. If S' is unsatisfiable, then crash C_i is not possible. Otherwise, the satisfying of S' together with the values defined by C_i describe a possible crash situation. We analyze that case, add or change check conditions to prevent that event, and repeat the above tests for $C_1, C_2, \ldots, C_m$. The process stops when all crashes have become impossible.

When we carry out the above process, the construction of M typically is not difficult, but the initial check conditions may not be easy to define. However, any errors or omissions become apparent during the recursive use of the satisfiability tests, and that process stops only when all crashes have been eliminated.

But how can we be sure that all possible crash scenarios are represented in the list $C_1, C_2, \ldots, C_m$? There are two aids. First, the designer has considerable insight into the operation of the machine and is likely to

provide a quite complete list of possible crashes. Second, computer-aided design (CAD) systems can search for geometric conflicts and thus may be used to identify possible crashes.

Increasing Machine Reliability

When a sensor fails, effectively the machine has changed, and crashes originally ruled out may become possible. One often can avoid such a disaster by expanding the check conditions used in the control program, or by introducing redundant sensors. We discuss the two approaches.

Additional Check Conditions

Recall that the clauses of the CNF system M are a model of the machine. When the *True/False* values for movements and sensor values do not satisfy the clauses of M, we know that some inconsistency is at hand; thus, we should stop the machine. Due to these considerations, each clause of M for which the *True/False* values of the variables are always known to the control program, can be made part of N.

For example, for the case of M and N defined by (11.2.1)–(11.2.6), we could add the clauses

$$\begin{aligned} &\neg s_1 \vee \neg s_2 \\ &\neg t_1 \vee \neg t_2 \end{aligned} \tag{11.2.12}$$

of M to N.

If we are certain that commands issued by the control program always produce arm movements, then we could also add the clauses

$$\begin{aligned} &\neg m_1 \vee \neg s_1 \\ &\neg m_1 \vee \neg s_2 \\ &\neg m_2 \vee \neg t_1 \\ &\neg m_2 \vee \neg t_2 \end{aligned} \tag{11.2.13}$$

Redundant Sensors

One may add sensors to obtain additional information about the status of the machine or to back up existing sensors. An example of the first case is a sensor that verifies that a command of arm movement does result in that movement. An example of the second case is addition of a duplicate sensor that performs exactly like an existing sensor. We look at the second case in detail.

When we back up a sensor s by a duplicate, redundant sensor s', then we add to N the clauses

$$\begin{array}{l} s \vee \neg s' \\ \neg s \vee s' \end{array} \tag{11.2.14}$$

to enforce that s and s' must have the same *True/False* value.

Where should redundant sensors be added? We answer that question in the remainder of this section. First, we look at possible sensor failures.

Sensor Failures

A sensor can fail in three ways. It may always supply the value *True*, or may always supply *False*, or may erratically supply *True/False* values. In the first (resp. second, third) case, we say that the sensor is *stuck closed* (resp. *stuck open*, *intermittently failing*). The third case of intermittent failure is rather rare and typically may be viewed as a sequence of temporarily stuck-open or stuck-closed cases. Due to this fact, it suffices that we focus on the stuck-open and stuck-closed cases. In both of them, the machine changes, while the control program remains the same. Accordingly, a correct representation of the failure of a sensor, say s, requires that the model M of the machine is changed, say to $\overline{M}$, while the check conditions N remain the same.

Revised Model of Machine

The derivation of $\overline{M}$ from M depends on the type of failure of the sensor s and in principle can be complicated. For example, other sensors may be affected due to a mechanical or electrical link. However, a good design excludes such possibilities, and in the vast majority of situations, $\overline{M}$ is derived from M according to the following rule.

(11.2.15) Rule. *If sensor s fails by becoming stuck open (resp. stuck closed), then the modified model $\overline{M}$ of the machine is derived from M by deleting all clauses containing a literal of s and by adding the clause $\neg s$ (resp. s). If both types of failures are possible, the same deletion of clauses is carried out, but the clause addition is skipped.*

Example: Sensor Failure

We discuss an example case. Let M be the CNF system defined by (11.2.1)–(11.2.4). Thus, the clauses of M are

$$
\begin{aligned}
&\neg s_1 \vee \neg s_2 \\
&\neg t_1 \vee \neg t_2 \\
&\neg m_1 \vee \neg s_1 \\
&\neg m_1 \vee \neg s_2 \\
&\neg m_2 \vee \neg t_1 \\
&\neg m_2 \vee \neg t_2
\end{aligned}
\tag{11.2.16}
$$

Suppose the sensor s_1 is stuck closed. According to Rule (11.2.15), we drop the clauses containing a literal of s_1, that is, $\neg s_1 \vee \neg s_2$ and $\neg m_1 \vee \neg s_1$, and add the clause s_1. Thus, $\overline{M}$ is given by

$$
\begin{aligned}
&\neg t_1 \vee \neg t_2 \\
&\neg m_1 \vee \neg s_2 \\
&\neg m_2 \vee \neg t_1 \\
&\neg m_2 \vee \neg t_2 \\
&s_1
\end{aligned}
\tag{11.2.17}
$$

Let the check conditions N be defined by (11.2.5) and (11.2.6). Thus, the clauses of N are

$$
\begin{aligned}
&\neg m_1 \vee t_1 \vee t_2 \\
&\neg m_2 \vee s_1 \vee s_2
\end{aligned}
\tag{11.2.18}
$$

Define $\overline{S} = \overline{M} \wedge N$. Consider the crash scenario $C = m_1 \wedge m_2$ of (11.2.8). It is easily checked that $\overline{S} \wedge C$ is unsatisfiable, so the stuck-closed s_1 cannot result in that crash.

We saw earlier that the crash scenario C cannot happen for the correctly working machine even when the check condition $\neg m_1 \vee t_1 \vee t_2$ is dropped. We now show that this is not the case when s_1 is stuck closed. Indeed, let N_1 represent the remaining check condition $\neg m_2 \vee s_1 \vee s_2$. Then for $\overline{S}_1 = \overline{M} \wedge N_1$, it is readily seen that $\overline{S}_1$ is satisfiable, which proves that crash scenario C is possible. This example demonstrates that it is a poor strategy to eliminate check conditions without considering the impact of sensor failures.

Critical Sensors

Define a sensor s to be *critical* if its failure may cause a crash. For each critical sensor s, one may want to add to the machine a duplicate, redundant sensor s'.

Multiple Failures

The situation becomes more complicated when several sensors can fail simultaneously. We address here just one issue, where we determine the minimum number of sensors that must fail if a given crash scenario is to occur. We answer that question by solving a certain MINSAT instance, as follows.

For each sensor s that may possibly fail, we define a variable $\mathit{fail}(s)$ and modify the clauses of M according to the following rule.

(11.2.19) Rule. *For each sensor s that may fail, add to each* CNF *clause of M containing a literal of s the literal $\mathit{fail}(s)$. If sensor s can only be stuck open (resp. stuck closed), also add the* CNF *clause $\neg\mathit{fail}(s) \vee \neg s$ (resp. $\neg\mathit{fail}(s) \vee s$), which represents the implication $\mathit{fail}(s) \rightarrow \neg s$ (resp. $\mathit{fail}(s) \rightarrow s$). If both types of failure are possible for sensor s, no such clause addition is done.*

Define $\overline{M}$ to be the CNF system derived from M by Rule (11.2.19). Let $\overline{S} = \overline{M} \wedge N$. Define the cost of *True* for each variable $\mathit{fail}(s)$ to be equal to 1. Declare all other *True/False* costs of the variables of $\overline{S}$ to be 0. Given a crash scenario C, we solve the MINSAT instance defined by the CNF system $\overline{S} \wedge C$ and the costs.

If the MINSAT instance is unsatisfiable, then failure of all sensors cannot result in a crash. Otherwise, the variables $\mathit{fail}(s)$ with optimal value *True* define a minimum-cardinality collection D of sensors whose simultaneous failure can produce the crash scenario C. Thus, at least $|D|$ sensors must have failed if crash C occurs. We say that the sensors of D are *jointly critical*.

For example, let M be specified by the clauses

$$\begin{aligned} &\neg s_1 \vee \neg s_2 \\ &\neg t_1 \vee \neg t_2 \\ &\neg m_1 \vee \neg s_1 \\ &\neg m_1 \vee \neg s_2 \\ &\neg m_2 \vee \neg t_1 \\ &\neg m_2 \vee \neg t_2 \end{aligned} \tag{11.2.20}$$

Suppose all four sensors s_1, s_2, t_1, and t_2 can become stuck closed and cannot become stuck open. We introduce variables $\mathit{fail}(s_1)$, $\mathit{fail}(s_2)$, $\mathit{fail}(t_1)$, and $\mathit{fail}(t_2)$ and apply Rule (11.2.19) to M. The resulting $\overline{M}$ has the clauses

$$
\begin{aligned}
&\neg s_1 \vee \neg s_2 \vee \mathit{fail}(s_1) \vee \mathit{fail}(s_2)\\
&\neg t_1 \vee \neg t_2 \vee \mathit{fail}(t_1) \vee \mathit{fail}(t_2)\\
&\neg m_1 \vee \neg s_1 \vee \mathit{fail}(s_1)\\
&\neg m_1 \vee \neg s_2 \vee \mathit{fail}(s_2)\\
&\neg m_2 \vee \neg t_1 \vee \mathit{fail}(t_1)\\
&\neg m_2 \vee \neg t_2 \vee \mathit{fail}(t_2)\\
&\neg \mathit{fail}(s_1) \vee s_1\\
&\neg \mathit{fail}(s_2) \vee s_2\\
&\neg \mathit{fail}(t_1) \vee t_1\\
&\neg \mathit{fail}(t_2) \vee t_2
\end{aligned}
\tag{11.2.21}
$$

Suppose N has the clauses

$$
\begin{aligned}
&\neg m_1 \vee t_1 \vee t_2\\
&\neg m_2 \vee s_1 \vee s_2
\end{aligned}
\tag{11.2.22}
$$

Let $\overline{S} = \overline{M} \wedge N$. Define the cost of *True* for $\mathit{fail}(s_1)$, $\mathit{fail}(s_2)$, $\mathit{fail}(t_1)$, and $\mathit{fail}(t_2)$ to be 1, and let all other costs of the variables of $\overline{S}$ be 0.

Let C be the crash scenario $C = m_1 \wedge m_2$. We solve the MIN-SAT instance given $\overline{S} \wedge C$ and the costs. A bit of checking confirms that there is an optimal solution with $\mathit{fail}(s_1) = \mathit{fail}(t_1) = \mathit{True}$ and $\mathit{fail}(s_2) = \mathit{fail}(t_2) = \mathit{False}$. Thus, $D = \{s_1, t_1\}$ is a set of jointly critical sensors, and at least $|D| = 2$ sensors must have failed if the crash occurs.

Relevant Chapters

Chapters 2, 3, and 5 contain all material needed for this application.

References

The material of this section is based on previously unpublished work for a manufacturer of automated machining and assembly equipment.

11.3 Music Composition Assistant

Music is an art form that is often described in terms of rhythm, melody, and harmony. Useful for the composer are intelligent systems that suggest

options and variations regarding, for example, musical progressions and melodic or rhythmic themes. Such a system allows the composer to try out many ideas in short order. When suitably enhanced, intelligent systems may be employed in areas of music other than composition; for example, for training and analysis, in historical and comparative studies, and, with the use of biometrics and biofeedback, in studies of psycho-physiological effects of music.

In music composition, any sequence of notes in principle can be claimed to be a piece worthy of performance. The degree of acceptance of that claim by others determines whether the claim is valid or not. This fact implies that any restraints on composition are in principle self-imposed and thus need not be, in fact generally are not, logically consistent according to some system of rules. Accordingly, if a system is to be helpful for the composer, it must allow for vagueness in the rules and imprecise conclusions.

Chord Progressions

We sketch one component of such a system. It deals with progressions of *chords*, each of which consists of three or more notes played at the same time. Using texts on harmony, for example, Piston, DeVoto, and Jannery (1987), one may group progressions according to level of perceived pleasantness or unusual tonality. A reasonable choice seem to be six levels, where level 1 has the most commonly used progressions, level 2 has very frequently occurring cases, and so on, down to level 6, which contains rare cases.

We express the fact that a chord x is frequently followed by chord y, by the clause

$$chord(x, current) \rightarrow chord(y, next) \quad frequently \tag{11.3.1}$$

with the obvious interpretation of the cited variables. If chord y follows chord x sometimes, the likelihood term *frequently* is replaced by *sometimes*. Indeed, we use the likelihood terms of (6.1.1) in Chapter 6, which are *always*, *very frequently*, *frequently*, *half the time*, *sometimes*, and *rarely* to handle the six levels. The level *always* is interpreted here as *virtually always*.

For the development of rules concerning chords, it is convenient that one characterizes each chord by the starting pitch, the number of notes it contains, and specification as *major*, *minor*, *diminished*, or *augmented*. In addition, variations such as *inversions* and *extensions* must be considered.

We focus on the starting pitch. It is denoted by a Roman numeral that, for a major key, ranges from I to VII where I specifies the first note of the chosen scale, II the second note, and so on, up to VII for the seventh note. For a minor key, one uses lower-case Roman numerals instead. Since

we want rules that cover both cases, we use Arabic numerals instead. Thus, for $1 \leq p \leq 7$ and $t \in \{current, previous, next\}$, we define $chord(p, t)$ to be *True* if the chord in the relative position t of a sequence of chords has starting pitch equal to the pth note of the chosen scale.

Rules for Starting Pitch

We show a small portion of a set of rules of a composition assistant created by J. W. Davidson. The system helps with the composition of songs. The rules shown next cover the case where the previous chord has as starting pitch the third note of the chosen scale.

$$
\begin{array}{ll}
chord(3, previous) \rightarrow chord(1, current) & sometimes \\
chord(3, previous) \rightarrow chord(2, current) & frequently \\
chord(3, previous) \rightarrow chord(4, current) & very\ frequently \\
chord(3, previous) \rightarrow chord(5, current) & half\ the\ time \\
chord(3, previous) \rightarrow chord(6, current) & always \\
chord(3, previous) \rightarrow chord(7, current) & rarely
\end{array}
\tag{11.3.2}
$$

The rules do not link $chord(3, previous)$ and $chord(3, current)$, since it is assumed that the option of repeating a chord is always available and thus need not be covered in the rules.

Next, we include a few rules showing how a next chord with the fifth note as starting pitch may be preceded. Here, too, the case of a chord with same starting pitch is assumed always possible and thus is not explicitly shown.

$$
\begin{array}{ll}
chord(5, next) \rightarrow chord(1, current) & frequently \\
chord(5, next) \rightarrow chord(2, current) & very\ frequently \\
chord(5, next) \rightarrow chord(3, current) & rarely \\
chord(5, next) \rightarrow chord(4, current) & always \\
chord(5, next) \rightarrow chord(6, current) & half\ the\ time \\
chord(5, next) \rightarrow chord(7, current) & sometimes
\end{array}
\tag{11.3.3}
$$

Superficial Inconsistency of Rules

One might think that the rules implying the current chord from the next one should be nothing but inverse relationships of the rules implying the current chord from the previous one. According to that reasoning, the implications $chord(3, previous) \rightarrow chord(5, current)$ of (11.3.2) and $chord(5, next) \rightarrow chord(3, current)$ of (11.3.3) should have the same likelihood value. However, the former implication has the value *half the time*, while the latter has

the value *rarely*. This is an instance of the fact mentioned earlier, that the rules are not created by some mathematically consistent process, but by an evaluation process that need not be based on mathematical considerations. In the specific example, the likelihood choices are based on an opinion how likely sequences of three chords, labeled *previous*, *current*, and *next*, should be. The person who created the rule set felt that the selected likelihood values are more apt to produce the desired results for such sequences.

We discuss the use of the rules by an example.

Example: Selection of Progression

Suppose in a progression of three chords with $t = previous$, $t = current$, and $t = next$, we already have the starting pitch for $t = previous$, say $chord(3, previous) = True$. Suppose we also have the starting pitch for the next chord, say $chord(5, next) = True$. We want to find the starting pitch for the current chord. In agreement with the theorem-proving approach of Chapter 6, we iterate through the levels *always*, *very frequently*, ..., *rarely*, delete all clauses with level below the specified one, and check which cases of $chord(p, current)$ can be proved to be theorems. The highest level at which a chord can be proved is then the likelihood level at which the chord is suggested.

We have chosen the example case so that the clauses of (11.3.2) and (11.3.3) suffice to compute the results, which are shown in Table (11.3.4). The cases of $chord(3, current)$ and $chord(5, current)$ are skipped, since they are repetitions of the previous or next chord.

(11.3.4)

Chord	Proved at Level
$chord(1, current)$	*frequently*
$chord(2, current)$	*very frequently*
$chord(4, current)$	*very frequently*
$chord(6, current)$	*always*
$chord(7, current)$	*sometimes*

Selection of Current Chord

Composition Process

The composer selects a key signature, defines the form of the song, and then chooses chord patterns to fill out each of the sections defined by the form. At that stage, the rules for the selection of starting pitches of chords are useful. Once a rough composition is at hand, the composer investigates details and variations involving a multitude of substitutes and extensions. We cannot cover details here and mention only that the same approach used

for the starting pitch of chords can be employed. Thus, substitutions and extensions can be expressed by rules with likelihood levels, and theorem proving selects appropriate cases.

Relevant Chapters

Chapters 2, 3, 5, and 6 suffice to handle this application.

References

The chord progression component described in this section is part of a music composition and analysis system under development by J. W. Davidson.

11.4 Management of Hazardous Materials

Management of industrial chemical exposure must comply with numerous federal, state, and industry regulations. Indeed, the regulations are sometimes so complex that even an expert has difficulty making fully-compliant decisions. On the other hand, one may encode the regulations in a logic formulation and then use techniques for question-and-answer processes of Chapter 10 to construct an easy-to-use expert system that searches for best, compliant decisions.

We sketch such a system. It deals with the management of asbestos in an electric utility in Texas and is called OCHEM (Optimal Chemical Hazardous Exposure Management). Numerous federal, state, and industry regulations control the handling and treatment of asbestos. Nevertheless, just 154 variables and 233 clauses are needed to encode all regulations using the decision pyramid approach of Chapter 5. The 154 variables are made up of 72 input variables, 23 intermediate variables, and 59 conclusion variables.

The input variables characterize the situation at hand, the conclusion variables tell the decisions, and the intermediate variables are introduced as part of the decision pyramid approach as described in Chapter 5.

The clauses select lab testing options, determine the category of asbestos, decide the job type and the class of material, establish respirator requirements, select among wet and dry options, determine level of supervision needed, cover control methods and containment rules, and consider inspection requirements. We include three example clauses.

To express that the treatment of an asbestos case must be removal, repair, maintenance, or custodial, we have the clause

$$\begin{aligned} & fact(asbestos) \rightarrow \\ & [fact(removal) \vee fact(repair) \vee \\ & fact(maintenance) \vee fact(custodial)] \end{aligned} \tag{11.4.1}$$

For the choice of control methods, we have the following rule. If it is an asbestos case and if there is no negative exposure assessment, then one must employ one of the following control methods: local exhaust, enclosure, isolation, or ventilation. The rule is encoded by

$$\begin{aligned} & [fact(asbestos) \wedge fact(no\_negative\_exposure\_assessment)] \rightarrow \\ & [fact(local\_exhaust) \vee fact(enclosure) \vee \\ & fact(isolation) \vee fact(ventilation)] \end{aligned} \tag{11.4.2}$$

A rule of the federal Occupational Safety and Health Administration (OSHA) specifies that, if the activity involves removal of surfacing asbestos-containing material, then it is a case of Class I or III. The rule is represented by

$$\begin{aligned} & [fact(removal) \wedge \\ & fact(surfacing\_asbestos\_containing\_material)] \rightarrow \\ & [fact(class\_I) \vee fact(class\_III)] \end{aligned} \tag{11.4.3}$$

Let S be the CNF system defined by the 154 variables and 233 clauses. Each decision variable has an associated cost for *True*, which is incurred if the actions associated with the decision are carried out.

In principle, OCHEM could proceed as follows. The system asks for the *True/False* values of the 72 input variables, fixes the input variables in S to these values, and solves the MINSAT instance defined by the modified S and the given costs. The optimal *True/False* values for the decision variables tell which actions are to be taken.

The above approach suffers from the fatal flaw that each session would require an extensive effort by the user to supply the *True/False* values for the 72 input variables. That effort can be drastically reduced by the algorithms of Chapter 10 for question-and-answer processes. Section 10.11 of that chapter covers precisely the case at hand. Specifically, the input (resp. conclusion) variables of OCHEM are denoted in Section 10.11 by x_1, $x_2, \ldots, x_n$ (resp. $a_1, a_2, \ldots, a_p$).

After each session, OCHEM computes low-cost assignments as described in Section 10.11 of Chapter 10. These assignments are collected in files that guide the selection of questions about the values of input variables

in subsequent sessions. Upon request, OCHEM also supplies explanations using the approach of Section 5.7 of Chapter 5.

In an experiment, it turned that after 75 sessions OCHEM had accumulated enough low-cost assignments so that the questions for input variable values matched the question selection by an expert of that domain. That is, the system asked as few questions, typically 8 or 9, as the expert. The drastic reduction from 72 possible questions to 8 or 9 actually asked questions demonstrates the effectiveness with which OCHEM learned questioning via low-cost assignments.

Relevant Chapters

Chapters 2–5 and 10 contain the relevant material for this application.

References

Rules and regulations of hazardous materials and hazardous waste management are covered in Woodside (1999).

OCHEM is described in Straach (1998) and Straach and Truemper (1999).

11.5 Traffic Control

Traditional techniques for the control of traffic lights rely on hourly flow rates of cars and mathematical optimization models and methods. In contrast, recently introduced techniques use information about queues and other momentary traffic patterns at each intersection in logic programming models. In this section, we describe a system of the latter type. It is called *TraVers*.

Operation

Each intersection has a control unit that is locally connected to the control units of neighboring intersections by a low-volume data link. There is no hierarchy of units, and there is no central control unit. At each intersection, the control unit is aware of car counts and, possibly, speeds in different sections of the roads approaching the signal. Typically, cameras obtain these data. The control unit uses that information, the time expired in a given signal phase, and the status of neighboring intersections to decide whether the current phase should be terminated. If the control unit decides that termination is appropriate, it also selects which signal phase is to start next.

Decision Problem

The decision problem is formulated as a MINSAT instance. The formulation relies on a decision pyramid composed of status logic variables, intermediate variables, preliminary decision variables, and final decision variables. We discuss these variables and related clauses next. For general results about decision pyramids, see Section 5.6 of Chapter 5.

To simplify the discussion, we assume that all intersections have the usual configuration where two roads cross. The two portions of each road which lead up to the intersection are *segments* of the road. We also assume that each segment has a separate left-turn lane. We emphasize that these assumptions are made just to simplify the discussion and that TraVers readily handles more complicated cases.

Status Variables

Layer 0 of the decision pyramid, which is the lowest layer, contains *status variables* that describe the status of the intersection and its neighbors. Examples are variables representing the number of cars in the segments of the roads, the time expired in the current signal phase, and the extent to which the intersection currently is synchronized with the neighboring intersections.

Qualitative terms are employed in the conversion of rational data to logic *True/False* values for the status variables. For example, qualitative terms for car counts, with associated numerical values in parameters, are *none* (0), *few* (1–3), *several* (4–6), *many* (7–11), and *very_many* (12–20). Thus, if the queue in segment s of road r has four cars, then the term *several* applies. Correspondingly, the status variable $queue(r, s, several)$ has the value *True*, while the variables $queue(r, s, none)$, $queue(r, s, few)$, $queue(r, s, many)$, and $queue(r, s, very\_many)$ have the value *False*.

A similar approach is used to characterize time intervals. For example, the following terms, with corresponding time interval in seconds, are employed: *now* (0), *next* (0–4), *shortly* (0–7), and *soon* (0–14).

Intermediate Variables

Various clauses define the intermediate variables, which constitute layer 1 of the decision pyramid, from the status variables. The intermediate variables represent the congestion on the roads at, near, or far from the intersection. For example, one clause effectively says that, if the queue in segment s of road r has at most a few cars, and if the number of cars in the adjacent arrival portion of the segment has at most a few cars, then the congestion in that segment is at most low.

Preliminary Decision Variables

Layer 2 of the decision pyramid contains the preliminary decision variables. They encode decisions of phase termination based on values such as queue length, other counts in road segments, congestion, and time expired during the current phase. For example, one clause represents the following consideration. Suppose that we are in the left-turn phase of segment s of road r, that at least 80% of the maximum time for that phase has expired, that the congestion in the left-turn lane of segment s of road r is at most medium, and finally that the congestion in the crossing road is high. Then we should terminate the left-turn phase in segment s due to congestion considerations.

Final Decision Variables

The topmost layer 3 of the decision pyramid has the final decision variables. They encode the choice of next phase, which depends on the situation at hand as well as rules externally imposed on the system. For example, if may be required that left turns must be done at the beginning of all green phases for a road segment.

Costs of Variables and MINSAT Solution

The decision of phase termination and the selection of the next phase can be handled by theorem proving. Since a number of such questions must be answered, the accelerated theorem-proving approach of Section 5.8 of Chapter 5 is advantageous. In that method, a cost of 1 is assigned to *True* for each final decision variable. All other costs of *True/False* values are 0. Then the MINSAT instance defined by the clauses and costs is solved, and the final decision variables with optimal value *True* indicate which actions are to be taken.

The accelerated theorem-proving method of Section 5.8 of Chapter 5 demands additional theorem proving to check whether the actions implied by the optimal *True* values of decision variables are really to be done. In this application, we forgo these additional tests, since potential errors have few if any negative consequences.

Development Subsystem of TraVers

The clauses of the MINSAT formulation are determined by the traffic engineer with an interactive subsystem of TraVers that allows human insight and reasoning to be inserted directly into the logic formulation, without intervening use of mathematical parameters, functions, or optimization methods.

The core of the subsystem is a simulation program that, at each simulated intersection, carries out the very same logic calculations that eventually will be done by the control units. A graphic display shows the roads, intersections, traffic lights, and cars in appropriate scale. The traffic engineer observes the traffic flow and checks if the decisions at a given intersection agree with his/her intuitive insight. If a decision seems inappropriate, the traffic engineer stops the simulation and uses a logic module to determine in the logic formulation the cause for the objectionable behavior. The module essentially carries out a portion of the validation method of Section 5.7 of Chapter 5 and advises the traffic engineer to add, delete, or modify clauses for the intersection. Changes are easily made since the names of the variables are self-explanatory and since qualitative terms are employed for counts and time intervals. Once the change has been made, the traffic engineer restarts the simulation program and continues the evaluation process.

The development process is completed when the performance has become appropriate for the entire grid. At that time, the clauses developed for the intersections are transferred to the corresponding control units and thus reflect precisely the thinking and reasoning of the traffic engineer.

Additional Features

The control units carry out additional functions with appropriate logic modules. For example, one module continuously checks if the cameras are performing correctly. If data supplied by a camera are diagnosed to be faulty, the module demands that these data are to be ignored. The flaw may be a camera breakdown, in which case the control unit alerts the system administrator. It may also be a temporary difficulty, such as a rising sun shining directly into the camera lens. In the latter case, the control unit continues to analyze the camera data and begins using them once they have become appropriate again. Similar diagnostic modules check for accidents or events requiring emergency action.

Integration into Existing Systems

In contrast to the TraVers system, current traffic control is usually carried out by a central control unit. The timing of the traffic lights is based on hourly traffic flows, which are used to determine a fixed time for one complete cycle of all phases at a given intersection. Within that fixed period, the phases are allowed to vary, with the final phase of a complete cycle always selected so that the fixed time is used up.

The TraVers system can be modified so that it can become part of such centrally controlled systems. After completion of a cycle of phases, each

intersection sends the central control unit a proposed cycle time. From the cycle times proposed by the intersections, the central unit derives a suitable compromise, which is returned to each intersection. That cycle time is used at each intersection to guide the decisions for the next cycle.

Performance

In computer experiments, the TraVers system converged rapidly to steady-state behavior and from then on closely tracked changes in traffic flows. To-date, an experimental installation has been made at one intersection in Europe. There, it performed significantly better than two competitors.

Relevant Chapters

Chapters 2, 3, and 5 contain material relevant for this section.

References

The TraVers system is the result of research on traffic control by several institutes and universities in Italy and the United States. A summary is given in Felici, Rinaldi, Sforza, and Truemper (2001).

11.6 Credit Rating

The evaluation of creditworthiness is a difficult problem for banks and other loan-granting institutions. In this section, we summarize current evaluation methods and describe an approach based on the learning logic techniques of Chapters 7 and 8.

We begin with some definitions.

Scorecards

An *application scorecard* evaluates the creditworthiness of loan applicants. It predicts the probability that a loan, once granted, will end up in default. A *behavioral scorecard* forecasts future performance of a loan portfolio. It identifies delinquent accounts having a high probability of default and meriting particular attention. Thus, the two types of score cards predict the probability of default before or after a loan has been granted.

Input Data for Scorecards

For application scorecards, example input data are yearly income, length of stay at current residence, home ownership versus rental, length of current employment, total number of credit inquiries during the last six months, and aggregate balance past due during the past twelve months. For behavioral scorecards, the input data are similar, but do not include items which become available only during the loan application process and which may not longer be correct.

Prediction Methods

The predictions of scorecards are usually based on an analysis of large amounts of historical data by *statistical techniques* and, increasingly, *neural nets*. Recent approaches also use *support vector machines*.

Regardless of the method, a key problem is the selection of relevant input variables or *attributes*. The choice is typically made in a stepwise selection process that uses various techniques plus human insight. Another difficulty for current approaches is missing entries. Each missing value is typically approximated by some estimate using reasonable rules. That process cannot account correctly for cases where the missing entry, by itself, is a telling sign and should be explicitly considered.

Learning Logic Approach

We have used the learning logic techniques of Chapters 7 and 8 for the prediction task. The reasons for that choice include the following. First, the selection of attributes is an integral part of the learning logic process. Second, the missing entries are readily handled by *Absent* and *Unavailable* values as appropriate. Third, the entire learning process is automatic once the data have been collected.

We sketch the learning process. We first divide the available data into two sets of records. The first set contains the instances where the loan was repaid, while the second set has the default cases. The two sets typically contain rational and set data besides logic data. We must convert the rational and set data to logic data. The transformation task may seem straightforward, but actually is quite difficult. The main problem is the definition of markers that determine how the rational data are converted to logic data. We have found that a selection of markers based on human insight can be quite inappropriate, in the sense that logic formulas learned from the resulting records may be very large and may make quite inaccurate predictions. These facts and similar, negative results for other applications motivated the search for a transformation scheme that does not suffer from

these difficulties. According to experiments, the *cutpoint method* is one such transformation scheme, and we use it for the case at hand. References for that method and all other schemes mentioned so far are included in Section 7.9 of Chapter 7.

Let A and B be the two sets of training records produced by the cutpoint method from the original data. Suppose that A contains the creditworthy or non-default cases, while B has the default cases. We view A and B to be randomly selected subsets of populations $\mathcal{A}$ and $\mathcal{B}$, respectively.

Computation of Formulas

We apply Algorithm FORTY MIN/MAX DNF FORMULAS (8.3.1) to the training sets A and B and get 40 DNF formulas separating the two sets. If the vote total produced by these formulas is greater than or equal to a yet to be computed threshold, we classify the record into population $\mathcal{A}$. Otherwise, we declare the record to be in population $\mathcal{B}$.

Probability Distributions and Optimal Threshold

We assume that prior probabilities are available for membership in the two populations $\mathcal{A}$ and $\mathcal{B}$. We also assume that we have the costs of misclassifying a record for each of the two populations. We use that information and the learned logic formulas in Algorithms VOTE DISTRIBUTIONS (8.5.7) and CLASSIFICATION CONTROL (8.6.5) and get the following results: the conditional probability distributions that the vote total is below a given threshold, given that the record is in $\mathcal{A}$ or $\mathcal{B}$; for all possible thresholds, the unconditional probability of misclassification; and an optimal threshold z^* that minimizes expected total misclassification cost.

We use these results for credit scorecards as follows.

Loan Evaluation

Given a record r that is to be evaluated, we apply the 40 DNF formulas, get a vote total z, and predict default if $z \leq z^*$. Besides this binary decision, we can use the above conditional probabilities and the overall misclassification probabilities for further analysis of the case.

Test Case

We describe results obtained for training and testing data of a behavioral scorecard. The data are proprietary and concern delinquent mortgages.

The data consist of 253 training and 251 testing records, where each set is about evenly split into two classes of non-default and default cases. Each record has 15 attributes, of which 13 are rational entries and 2 are logic entries.

An initial, manual transformation of the rational data, attempted by the company supplying the data, results in records with 48 logic entries for which Algorithm FORTY MIN/MAX DNF FORMULAS (8.3.1) computes large and unusable logic formulas. Thus, this is an instance where human insight fails to achieve a proper transformation to logic data. In contrast, the cutpoint method produces records with 23 logic entries, from which Algorithm FORTY MIN/MAX DNF FORMULAS (8.3.1) deduces reasonably sized logic formulas. When applied to the transformed testing records, 95% of the records are correctly classified, using a threshold of 0. An unexpected result is that some erroneously classified testing records have vote totals far different from 0. It would be enlightening to investigate this unusual behavior, but we have no data or access to do so. One conjecture is that at least some of the unusual cases of misclassification involve data collection errors. A second conjecture is that the erroneously classified records with vote totals far different from 0 signal that one should look for additional, possibly unorthodox, attributes.

Cost of Data Collection

Up to this point, we have ignored costs connected with the collection of attribute data for a given case. If such costs are of concern, we can minimize them or at least keep them low by use of an expert system that diligently asks for data until the decision can be made. Section 11.8 shows how such an expert system can be constructed by a process that requires almost no manual effort.

Relevant Chapters

Chapters 2, 7, 8 contain the material needed for this application.

References

A comprehensive text on credit scoring techniques is Thomas, Edelman. and Crook (2002).

For references regarding data mining, machine learning, and, in particular, the cutpoint method, see Section 7.9 of Chapter 7.

11.7 Deciding Word Sense

Work of the 20th century, in particular, of L. Wittgenstein, shows that *word sense* is an elusive concept that defies axiomatic definition. Wittgenstein proposes a pragmatic viewpoint where the various senses of a word are largely defined by the usage of the word. In that approach, one obtains an understanding of the senses of a word by examining a large number of example sentences and paragraphs.

Indefiniteness of Word Sense

A corollary of Wittgenstein's proposal is that one cannot discuss or investigate word sense the same way one can examine a plant specimen under a microscope. However, one may conjecture that a computational process exists where the sense of a word, sentence, or collection of sentences is established and revised as a stream of words in a given environment is processed. Whenever that stream of words is temporarily interrupted or finally stopped, one can talk about a *sense* of individual words, sentences, paragraphs, or even of the entire stream of words processed so far. We say "a sense" since typically several senses are valid interpretations.

For the moment, let us focus on the possible senses of an entire stream of words. From the possible senses that are valid for that stream of words, the human brain typically picks just one sense and declares it to be *the* sense of the stream of words. If subsequent sentences invalidate that interpretation, the brain changes the selected sense with amazing speed and equally astounding lack of remorse about that faulty performance. Indeed, one might conjecture that the fundamental feature jokes have in common is an unexpected, huge change of sense toward the end of the story. Declaring that change to be funny is then one way with which the brain copes with that embarrassing failure.

The following sentence demonstrates the above claims. The brain processes the sentence "Henry says,'I am not married to Anne because I love her'." Does the sentence implicitly mean that he is married to her? The brain picks "yes" or "no" as answer and, when requested to supply reasons for that choice, supplies convincing arguments. Yet, both conclusions are equally valid. For example, "yes" is correct if Henry does not love Anne and married her for money, and "no" is correct if he loves her and believes that marriage destroys love.

One may point to a word in a stream of sentences and ask the brain to identify a sense of that word as part of the overall interpretation. Here, too, the term "a sense" is appropriate since generally several different senses can be assigned, even though the human brain almost always selects just one sense and declares it to be *the sense* of the word. The corrective mechanism discussed above applies here, too. Thus, the sense may have

to be changed as additional sentences are processed. In extreme cases, the sense is changed a number of times.

If an intelligent system is to emulate the performance of the brain for deciding word sense, it must not only assign a reasonable sense to each word in a given stream of words, but also must be able to revise the choice if, subsequently, additional words or sentences become available. In this section, we describe how learned logic formulas may be used to accomplish, somewhat inaccurately, the first task. Toward the end of the section, we sketch how learned logic formulas may also be used to handle a portion of the second task.

We begin with a definition.

Word Sense Disambiguation

The *Word Sense Disambiguation* (WSD) problem demands the following: Given a stream of words occurring in a particular environment, one must determine the sense of each word which the majority of a given group of persons would assign.

A number of methods have been proposed for the WSD problem. To-date, the most effective ones assume that text is available where, for each occurrence of the word whose sense is to be decided, the sense has been assigned. We say that each occurrence of the word has been *sense-tagged.* The methods use such tagged text to learn how sense should be assigned. For references of these methods, see the end of this section.

Two difficulties are connected with any method using sense-tagged text. First, the tags generally are not unique; this fact is implied by the preceding discussion. Second, since tagging is done by hand, the amount of tagged text is necessarily small relative to the total amount of text available. We ignore the first difficulty for the moment and focus on the second one. To obtain a significant volume of tagged text, projects have been launched which draw many persons, via the Internet, in the tagging effort. But even if one ignores the problem of checking the accuracy and consistency of such efforts, such projects can only produce comparatively small amounts of tagged texts for the cases where a given sense of a word occurs rarely. Indeed, in such a case, even reading large amounts of text can unearth just a few instances of that sense. This negative conclusion does not apply, if, somehow, we can identify instances with rarely occurring sense. We describe an approach for the WSD problem that implicitly has this feature.

Define W to be the word whose sense is to be decided. Let W have q senses. We are to select the appropriate sense p, $1 \leq p \leq q$, of W in a given stream of words. To simplify the discussion, we assume that each stream of words where the sense of W is to be decided, is just one sentence. This restriction implies that in some cases we do not have even a hint of the

sense of W. For example, the sentence "It ate everything" allows for the word "eat" to have any one of the senses *take into mouth*, *consume*, *erode*, and *devastate*.

The method relies on learning of logic formulas as described in Chapters 7 and 8. The learning process is iterative and begins with some example sentences containing the word W. Specifically, we assume to have, for each sense p, a few sentences containing the word W with the sense p. We call these sentences the *training sentences*. One iteration of the process involves five steps.

Step 1: Create Training Records

We derive from each training sentence a record where each entry is a certain set of words, as follows.

With standard Natural Language Processing (NLP) techniques, we parse the sentence and thus make visible the syntactic connections between the word W and other words u of the sentence. In a preliminary investigation, we have deduced which of these connections are important. We skip a detailed discussion of that step and mention only that the preliminary investigation uses a simplistic learning of logic formulas to decide the sense of W. These logic formulas, imperfect as they may be, indicate which words u of a sentence are likely to play a role in the determination of the sense of W.

For each of the selected words u, we record the part-of-speech of u and three sets. The sets depend on whether we know the sense of u. Suppose we know that sense. Then the three sets of words are as follows: the set containing as single element the *category* of u, which is selected from a list of about forty choices; the *synset* of u, which is the set of synonyms of u; and a set of *hypernyms* of u, which are generalizing concepts of u. We obtain these sets from WordNet, which is an elaborate dictionary.

The above sets are modified when we do not know the sense of u. In that case, we replace the category set, the synset, and the hypernym set by the union of the respective sets which are generated by the various senses of u.

The record resulting from the given sentence contains, for each selected word u, the following items: the word u itself, its part-of-speech, and the three sets as described above.

Step 2: Transform Set Data to Logic Data

Using the techniques of Section 7.2 of Chapter 7, we transform the set data of the records to logic data.

Step 3: Define Training Sets

For each sense p, we collect the logic records produced from the training sentences where word w has sense p, in a set A^p. We view each A^p to be a subset of a population $\mathcal{A}^p$. If we have q senses for word W, we thus have $A_1, A_2, \ldots, A_q$. For each sense p, let $B^p = \bigcup_{l \neq p} A^p$.

Step 4: Compute Logic Formulas

For each sense p, we compute with Algorithm FORTY MIN/MAX DNF FORMULAS (8.3.1) 40 logic formulas that separate A^p from B^p, and collect these formulas in a set $\mathcal{H}^p$.

Step 5: Test Validity of Formulas

We apply the logic formulas to logic records which are obtained from sentences with word W as described as above, except that the sense of W is not known. Suppose that, for a given sentence, the vote total v_p produced via the 40 formulas of $\mathcal{H}^p$ for the various senses p are such that one v_p, say v_{p^*}, is close to 40, while all others are close to -40. Now a vote total close to 40 (resp. -40) indicates that membership in A^p (resp. B^p) is highly likely. Thus, the fact that v_{p^*} is close to 40 and that all other v_p are close to -40 implies that the record likely belongs to population $\mathcal{A}^{p^*}$. Accordingly, we define the vote totals to be *consistent* and declare the word W in that sentence to have the sense p^*. If we have consistency for all sentences with W that are available to us for testing, we declare that the learned formulas suffice to identify the sense of W. Otherwise, we pick a small k and select the k sentences with the most inconsistent vote totals using the following method. For a given sentence, consider the vote totals v_p sorted in descending order and with v_{p^*} in first position. Let $v_{p^{**}}$ be the vote total in second position, and define $d = v_{p^*} - v_{p^{**}}$. Since the v_p values may range from -40 to 40, we have $0 \leq d \leq 80$. Indeed, a d value close to 0 (resp. 80) signals a large degree of inconsistency (resp. consistency). Thus, we select the sentences associated with the k smallest d values and assign in each sentence the correct sense to the word W. Finally, we add the newly tagged sentences to the current collection of training sentences, and begin another iteration.

In tests done to-date, the method converged rapidly to logic formulas that rather reliably determined word sense. The number of training

sentences needed for computation of the formulas was reasonable and required only a modest manual tagging effort.

Joint Evaluation of Words

Once we have learned logic formulas to decide the sense for all words, we can apply them simultaneously to determine the sense of each word in a given sentence. The process is as follows.

For each word W of the sentence, we define a record and translate it to a logic record as described above. For each sense p of W, we apply the relevant 40 formulas and get a vote total p. Finally, we use the vote totals to compute the measure d of consistency.

We select the word W of the sentence with the most consistent vote totals, as measured by d, and fix the sense of the word W accordingly. Then the entire evaluation process repeats, except that we use the fact that the sense of W is known when we derive the logic records from the original records. Thus, the category set, synset, and hypernym set connected with W become smaller. Due to this change, the transformation rules of Section 7.2 of Chapter 7 produce more precise logic records. Here, "more precise" means that, when we apply the learned logic formulas to these logic records, we tend to obtain more consistent vote totals.

The evaluation scheme stops when the sense of each word has been fixed.

The above method can also be employed to recompute the sense of words in a sentence if the sense of one word is changed. This is the case, for example, if subsequent sentences invalidate a sense choice for a word and replace it by another sense. In such a case, the above method may produce quite different senses for the other words of the sentence. Since computing effort for the evaluation of formulas is very small—typically around 10^5 formulas can be evaluated in 1 sec on a computer of modest speed—the re-evaluation process is almost instantaneous and thus matches the astounding capability of the brain to recompute senses in a fraction of a second.

Beyond Word Sense Disambiguation

The above description skips over numerous issues. For example, it does not cover the case where more than one sentence is to be used to evaluate the sense of a word W. It also does not treat the determination of sense of a sentence or of an entire stream of words. For most of these problems, presently there are at best some partial solutions. We conjecture that logic formulations and algorithms will play an important role in the eventual solution of many of these problems.

Relevant Chapters

Chapters 2, 7, and 8 contain material relevant for this section.

References

Over several decades, Wittgenstein developed his views on meaning. Wittgenstein (1953) was the final result, published soon after his death in 1951. Helpful for an understanding of that work are his earlier, also posthumously published, works, in particular Wittgenstein (1958). The biography by Monk (1991) provides a good introduction to Wittgenstein and his work.

A detailed review and references of WSD approaches and methods is given by Ide and Véronis (1998). The reference includes material on dictionaries such as WordNet and on basic NLP techniques such as parsing.

The material of this section is based on so-far unpublished work with S. Granberry and R. Mihalcea.

11.8 Differential Medical Diagnosis

Teaching, practice, and research in medicine can be supported by intelligent systems in many ways. Since there are so many opportunities, we need methods for constructing entire classes of such systems in a cost-effective manner. Ideally, one designs a process which takes as input existing medical data of a given problem and which outputs an intelligent system for that problem. In this section, we describe one such process.

The class of problems is *differential diagnosis*, where a physician must tell which of several closely related diseases is present in a patient. For example, the erythemato-squamous diseases are a problem instance. They are closely related skin diseases which not only share the clinical features of erythema and scaling but which generally are quite similar. The diseases are psoriasis, seboreic dermatitis, lichen planus, pityriasis rosea, chronic dermatitis, and pityriasis rubra pilaris; see the reference cited at the end of this section for details.

We describe an automated process which creates an expert system for each problem instance of differential diagnosis. In contrast to the discussion so far in this chapter, we are forced to invoke a number of concepts and results of preceding chapters, in particular, of Chapters 7, 8, and 10. Thus, if the reader has not yet studied that material, it would be a good idea to scan those chapters before proceeding.

Suppose the instance of differential diagnosis involves a collection of q related diseases. Assume that, for each disease p, we have a set of patient records. It is likely that the records contain rational or set data besides

logic data. With the techniques of Section 7.2 of Chapter 7, we transform all such entries to logic data. For each disease p, we collect the transformed records in a training set A^p. For each p, we define a second training set $B^p = \bigcup_{l \neq p} A^p$. Conceptually, the transformation that produces the sets A^p can be applied to each record of the population of patients with disease p. The resulting records constitute a population $\mathcal{A}^p$ that contains the records of training set A^p. We represent the entries of any record of population $\mathcal{A}^p$ by variables $x_1, x_2, \ldots, x_n$ and refer to these entries as *symptoms*.

We are also given tests $T_1, T_2, \ldots, T_m$ that determine symptom values or, more precisely, values that by the transformations determined above become values for the symptoms $x_1, x_2, \ldots, x_n$. With each test T_i, we have a cost c_i for carrying out the test.

Finally, we have, for each p, a cost $c_{err|\mathcal{A}^p}$ of misclassifying a record of population $\mathcal{A}^p$.

The above data suffice for us to construct an expert system for the case of differential diagnosis at hand. The construction process involves three steps. First, we compute logic formulas and decision criteria for the classification of records. Second, we define the diagnostic decision step. Third, we establish a procedure for getting record entries. We cover the three steps in detail.

Computation of Logic Formulas

For each disease p, we use Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2) to compute 40 optimized DNF formulas for separating the set A^p from the set B^p. The input for the algorithm consists of the sets A^p and B^p, the tests T_i with cost c_i, and parameters $\lambda_A^p \geq 1.0$ and $\lambda_B^p \geq 1.0$. Usually, $\lambda_A^p = \lambda_B^p = 1.2$ is a good choice. The output consists of some minimum separation costs that are not of interest here, optimized record sets A^{*p} and B^{*p}, optimized test sets $K^{*p} = \{K_r^{*p} \mid r \in A^p\}$ and $L^{*p} = \{K_s^{*p} \mid s \in B^p\}$, and, for $i = 1, 2, \ldots, 10$, optimized DNF formulas $D_i^{*min,p}$, $D_i^{*max,p}$, $E_i^{*min,p}$, and $E_i^{*max,p}$. For each p, the 40 formulas make up a set $\mathcal{H}^p$. Let $\tilde{K}^{*p} = \bigcup_{r \in A^p} K_r^{*p}$ and $\tilde{L}^{*p} = \bigcup_{s \in B^p} L_s^{*p}$.

Decision Strategy

We carry out Heuristic MULTICLASSIFICATION CONTROL (8.7.38) with training sets A^p, formula sets $\mathcal{H}^p$, and misclassification costs $c_{err|\mathcal{A}^p}$ as input. The output is a good if not optimal decision strategy $d^1, d^2, \ldots, d^q$ for minimization of the expected multipopulation classification cost.

We recall the following process of Section 8.7 of Chapter 8 for the classification of records with decision strategies: Given a record r, one obtains the vote total z_r^P produced by the applicable DNF formulas, finds

an index p^* that maximizes $e_r^p = z_r^p + d^p$, and declares the record r to be in population $\mathcal{A}^{p^*}$. For the case at hand, we implement the process as follows.

Diagnosis

Suppose we have initial symptom values for a patient plus, possibly, additional symptom values obtained so far by tests. Let r be the patient record containing these values. Thus, each entry x_j of r is equal to *True*, *False*, *Unavailable*, or *Absent*, where the first three cases represent initial values or values obtained by tests, and where the fourth value, *Absent*, indicates that so far no value has been obtained. Below, we refer to the case of *Absent* as a *missing symptom* value.

For each disease p, we apply the logic formulas of $\mathcal{H}^p$ to record r, compute the vote total z_r^p of these formulas, and define $e_r^p = z_r^p + d^p$. Let p^* be an index for which e_r^p is maximum. In case of a tie, we use a reasonable tie-breaking rule that must be independent of the vote totals z_r^p. For example, if all costs $c_{err|\mathcal{A}^p}$ of misclassification are distinct, we may break a tie by choosing the p^* with the largest such cost.

If we have *True/False/Unavailable* values for all symptoms x_j, we declare that the disease p^* is present. Otherwise, we decide if additional values for currently missing symptoms could lead to a different diagnosis, assuming that none of the additional values are equal to *Unavailable*. The check involves solution of $q - 1$ MINSAT instances that are indexed by the diseases $p \neq p^*$. The solution of MINSAT instance p is to provide the largest possible difference between the vote total obtainable for p^* and the vote total obtainable for p under all possible *True/False* choices for the missing symptoms values. We achieve that goal by defining the CNF system and the costs of the MINSAT instance p as follows.

We abbreviate the notation for the family of formulas in $\mathcal{H}^p$ so that the 20 formulas of type $D_i^{*min,p}$, $D_i^{*max,p}$ (resp. $E_i^{*min,p}$, and $E_i^{*max,p}$) are denoted by $D_1^p, D_2^p, \ldots, D_{20}^p$ (resp. $E_1^p, E_2^p, \ldots, E_{20}^p$). We fix in these formulas all variables for which record r has *True/False/Unavailable* values, to these values, and reduce the formulas accordingly to, say, $D_1'^p, D_2'^p, \ldots, D_{20}'^p$ (resp. $E_1'^p, E_2'^p, \ldots, E_{20}'^p$).

For $i = 1, 2, \ldots, 20$, we convert the formulas

$$\begin{aligned} u_i^p &\rightarrow D_i'^p \\ E_i'^p &\rightarrow v_i^p \\ D_i'^{p^*} &\rightarrow u_i^{p^*} \\ v_i^{p^*} &\rightarrow E_i'^{p^*} \end{aligned} \tag{11.8.1}$$

to equivalent CNF clauses. The clauses associated with a given $p \neq p^*$ define the CNF system of the MINSAT instance p. We denote that CNF

system by S^p. For $i = 1, 2, \ldots, 20$, we declare the cost of *True* (resp. *False*) for $u_i^{p^*}$ and v_i^p to be 1 (resp. -1). Furthermore, the cost of *True* (resp. *False*) of $v_i^{p^*}$ and u_i^p is -1 (resp. 1). All other costs of the variables of S^p are 0. Due to these costs, for any optimal solution of the MINSAT instance p defined by S^p and the stated costs, the following holds. Let z^{p^*} (resp. z^p) be the total cost contribution by the optimal *True/False* values for the $u_i^{p^*}$ and $v_i^{p^*}$ (resp. u_i^p and v_i^p) variables. Then $z^{p^*} - z^p$ is minimum. Define $e^{p^*} = z^{p^*} + d^{p^*}$ and $e^p = z^p + d^p$. If $e^p + d^p < e^{p^*} + d^{p^*}$ or $e^p + d^p = e^{p^*} + d^{p^*}$ with the tie-breaker preferring p^* to p, then additional *True/False* symptom values for record r cannot possibly cause us to replace the tentative diagnosis p^* by a diagnosis of p. In that situation, we permanently discard p as potential diagnosis. If that reduction rule eliminates all $p \neq p^*$, then we declare p^* to be the final diagnosis, and stop. Otherwise, we acquire additional symptom values as described in the next subsection.

Up to this point, we have assumed that none of the missing symptom values can be equal to *Unavailable*. Suppose that, to the contrary, the value *Unavailable* is possible for some missing symptoms. To accommodate that possibility, we modify the DNF formulas $D_i'^p$ and $E_i'^p$ of (11.8.1). That is, for each missing symptom x_j that may take on the value *Unavailable*, we introduce a new variable y_j and replace each literal x_j (resp. $\neg x_j$) in the DNF formulas by $x_j \wedge y_j$ (resp. $\neg x_j \wedge y_j$). In the expanded formulas, the case of *True* (resp. *False*) for the missing symptom corresponds to $x_j =$ *True* and $y_j =$ *True* (resp. $x_j =$ *False* and $y_j =$ *True*). The case of *Unavailable* corresponds to an arbitrary *True/False* value for x_j and $y_j =$ *False*.

Additional Symptom Values

From now on, p refers to any disease that so far has not been eliminated. We use Algorithm QA PROCESS (10.9.1) to select a cost-effective test that produces additional symptom values. For a compact description of the input of the algorithm, we introduce some definitions.

Let H^g, $g = 1, 2, \ldots, N$ denote the DNF formulas in the sets $\mathcal{H}^p$. That is, each formula in each $\mathcal{H}^p$ defines one H^g. If such H^g is the formula $D_i^{*min,p}$ or $D_i^{*max,p}$ (resp. $E_i^{*min,p}$ or $E_i^{*max,p}$) of $\mathcal{H}^p$, then let HP^g be the optimized record set A_i^{*p} (resp. B_i^{*p}), define HF^g to be the non-optimized record set B_i^p (resp. A_i^p), and let K^g be the test set $\tilde{K}^{*p}$ (resp. $\tilde{L}^{*p}$). We associate with the tests of K^g the given costs.

We are ready to define the input for Algorithm QA PROCESS (10.9.1).

The input CNF system S of Algorithm QA PROCESS (10.9.1) has the variables $x_1, x_2, \ldots, x_n$ and additional variables $h_1, h_2, \ldots, h_N$. The clauses are, for $g = 1, 2, \ldots, N$,

$$\neg H_l^g \vee h_g \tag{11.8.2}$$

The input goal set G has as elements (h_g, α_g), $g = 1, 2, \ldots, N$, with $\alpha_g = 100$. The input low-cost assignment file P^g is equal to HP^g. Let J_g be the set of indices j such that H^g has no literal of x_j. Define the records of the low-cost assignment file F_g from the records of HF^g by replacing in each such record all entries x_j with $j \in J_g$ by *Unavailable*.

The input tests are those of $\bigcup_g K^g$, with associated input costs.

The initial values called for in Step 1 of Algorithm QA PROCESS (10.9.1) are the *True/False/Unavailable* values already obtained for record r. As soon as Step 3 of the algorithm has selected the next T_{i^*} and has incorporated the values produced by that test into record r, we stop the algorithm.

With the expanded values for record r, we start another iteration where we determine a tentative diagnosis p^*, evaluate if it is the final diagnosis, and so on. The iterative process stops when p^* is determined to be the final diagnosis.

In the interest of clarity, the above definitions of the CNF systems S^p and S via (11.8.1) and (11.8.2) are somewhat simplified. Indeed, these CNF systems must be enlarged by certain clauses that arise from the transformation of the rational and set data of the patient records. We explain the format and role of the additional clauses by an example.

Suppose the jth entry of the patient records is rational and has been transformed to logic data using four markers $p_1 < p_2 < p_3 < p_4$. Let x_1, x_2, x_3, and x_4 be the corresponding four logic variables. Thus, each rational jth entry is represented by *True/False* values according to the formula (7.2.16), which produces exactly one of the following cases for a given rational number. Either $x_1 = x_2 = x_3 = x_4 = \mathit{False}$; or, for some $1 \leq l \leq 3$, $x_1 = \cdots = x_l = \mathit{True}$ and $x_{l+1} = \cdots = x_4 = \mathit{False}$; or $x_1 = x_2 = x_3 = x_4 = \mathit{True}$. Clearly, we want the satisfying solutions of S to admit just these cases. It is readily verified that we achieve the desired effect by adding, for $2 \leq l \leq 4$, the clause $x_l \rightarrow x_{l-1}$.

The example motivates how additional clauses should be defined for the general case of transformation of rational data to logic data via $k \geq 1$ markers. Specifically, if $k = 1$, then we do not add any clause. If $k \geq 2$, and if $x_1, x_2, \ldots, x_k$ are the variables corresponding to the k markers, then we add, for $2 \leq l \leq k$, the clause $x_l \rightarrow x_{l-1}$.

We also add clauses when set data of the patient records have been transformed to logic data as discussed in Section 7.2 of Chapter 7. Given the above discussion, the derivation of the clauses is not difficult, and we skip details.

Algorithm for Construction of Diagnostic System

We summarize the above process. Since it applies to any situation where one must differentiate between various similar defects, we use terms such

as *defect* and *case* instead of *disease* and *patient*.

(11.8.3) Algorithm CONSTRUCT DIFFERENTIAL DIAGNOSTIC SYSTEM. *Constructs expert system for differential defect diagnosis from given data.*

Input: Records with data for each defect. Tests and related costs for getting record entries. Costs of diagnostic errors.

Output: An expert system for differential defect diagnosis.

Requires: Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2), Heuristic MULTICLASSIFICATION CONTROL (8.7.38), and Algorithm QA PROCESS (10.9.1).

Procedure:

1. (Transform data.) If rational or set entries are part of the records, transform them to logic data using the techniques of Section 7.2 of Chapter 7. For each defect p, define A^p to be the set of transformed records, and let $B^p = \bigcup_{l \neq p} A^p$.
2. (Determine formulas.) For each defect p, use Algorithm FORTY OPTIMIZED DNF FORMULAS (8.3.2) to obtain optimized DNF formulas separating A^p from B^p.
3. (Compute decision strategy.) For each defect p, apply Heuristic MULTICLASSIFICATION CONTROL (8.7.38) to obtain a good if not optimal decision strategy $d^1, d^2, \ldots, d^q$.
4. (Construct expert system.) Define the following procedure to be the expert system.
   (a) Ask for readily available, initial data of the case at hand, and transform that information to logic data. Let r denote the resulting record, which has *True/False/Absent/Unavailable* entries.
   (b) Decide upon a tentative diagnosis p^* using the logic formulas and the decision strategy computed in the above Steps 2 and 3.
   (c) Compute if other defects p can possibly displace p^* by additional information, by solving MINSAT instances whose CNF systems and costs are defined via (11.8.1) and the discussion following (11.8.1). Eliminate all defects p which cannot possibly displace p^*.
   (d) If p^* is the only possible defect left, output p^* as final diagnosis, and stop.
   (e) Carry out Algorithm QA PROCESS (10.9.1) to get additional values. The input for the algorithm is defined by the discussion surrounding (11.8.2). Go to Step (b).

Relevant Chapters

Chapters 2, 7, 8, and 10 contain the material needed for this application.

References

An introductory text for differential diagnosis in many areas of medicine is Sam and Beynon (2003). A growing number of books cover specialized cases; for details, the reader should search the Internet using the key words "differential medical diagnosis."

Classification of the erythemato-squamous diseases is covered in detail in Güvenir, Demiröz, and İlter (1998). That reference uses a voting approach that is quite different from that used here.

The material of this section is based on previously unpublished work.

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993), *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

Bartnikowski, S., Granberry, M., Mugan, J., and Truemper, K. (2004), Transformation of Rational and Set Data to Logic Data, in *Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques* (E. Triantaphyllou and G. Felici, eds.), Kluwer Academic Publishers, New York, 2004.

Charniak, E., and McDermott, D. (1985), *Introduction to Artificial Intelligence*, Addison-Wesley, Reading, Massachusetts, 1985.

Chvátal, V. (1983), *Linear Programming*, Freeman, San Francisco, 1983.

Cook, S. A. (1971), The complexity of theorem-proving procedures, in: *Proceedings of Third Annual ACM Symposium on Theory of Computing*, Shaker Heights, Ohio, 1971. ACM, New York, 1971, pp. 151–158.

Cristianini, N., and Shawe-Taylor, J. (2000), *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*, Cambridge University Press, Cambridge, UK, 2000.

Eiter, T., and Gottlob, G. (1995), The Complexity of Logic-based Abduction, *Journal of the Association for Computing Machinery* 42 (1995) 3–42.

Felici, G., Rinaldi, G., Sforza, A., and Truemper, K. (2001), Traffic Control: A Logic Programming Approach and a Real Application, *Ricerca Operativa* 30 (2001) 39–60.

Felici, G., Sun, F., and Truemper, K. (2004), Learning Logic Formulas and Related Error Distributions, in *Data Mining and Knowledge Discovery*

Approaches Based on Rule Induction Techniques (E. Triantaphyllou and G. Felici, eds.), Kluwer Academic Publishers, New York, 2004.

Franco, J., Kautz, H., Kleine Büning, H., van Maaren, H., Selman, B., and Speckenmeyer, E. (2004), *Theory and Applications of Satisfiability Testing*, forthcoming volume of *Annals of Mathematics and Artificial Intelligence*, Kluwer Academic Publishers, Amsterdam, 2004.

Garey, M. R., and Johnson, D. S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

Gent, I., van Maaren, H., and Walsh, T. (2000), *SAT2000*, Kluwer Academic Publishers, Amsterdam, 2000.

Giarratano, J., and Riley, G. (1994), *Expert Systems: Principles and Programming* (2nd edition), PWS Publishing Company, Boston, 1994.

Güvenir, H. A., Demiröz, G., and İlter, N. (1998), Learning differential diagnosis of erythemato-squamous diseases using voting feature interval, *Artificial Intelligence in Medicine* 13 (1998) 147–165.

Hand, D., Mannila, H., and Smyth, P. (2001), *Principles of Data Mining*, MIT Press, Cambridge, Massachusetts, 2001.

Hopgood, A. A. (2001), *Intelligent Systems for Engineers and Scientists*, CRC Press, Boca Raton, Florida, 2001.

Ide, N., and Véronis, J. (1998), Word Sense Disambiguation: The State of the Art, *Computational Linguistics* 24 (1998) 1–40.

Kleine Büning, H., and Lettmann, T. (1999), *Propositional Logic: Deduction and Algorithms*, Cambridge University Press, Cambridge, UK, 1999.

Kneale, W., and Kneale, M. (1984), *The Development of Logic*, Clarendon Press, Oxford, 1984.

Kosko, B. (1992), *Neural Networks and Fuzzy Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

Leondes, C. T. (2001), *Expert Systems*, Volume 1–6, Academic Press, New York, 2001.

Luger, G. F. (2002), *Artificial Intelligence* (4th edition), Addison-Wesley and Pearson Education Limited, Harlow, England, 2002.

Mendelson, S., and Smola, A. (2003), *Advanced Lectures on Machine Learning*, vol. 2600 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 2003.

Mitchell, T. M. (1997), *Machine Learning*, McGraw-Hill, Boston, 1997.

Monk, R. (1991), *Ludwig Wittgenstein: The Duty of Genius*, Penguin Books, New York, 1991.

Negnevitsky, M. (2002), *Artificial Intelligence*, Addison-Wesley and Pearson Education Limited, Harlow, England, 2002.

Newman, J. R. (1956), *The World of Mathematics*, Vols. 3 and 4, Simon and Schuster, New York, 1956.

Nguyen, H. T., and Walker, E. A. (1997), *A First Course in Fuzzy Logic*, CRC Press, Boca Raton, Florida, 1997.

Nilsson, N. J. (1998), *Artificial Intelligence: A New Perspective*, Morgan Kaufmann, San Francisco, 1998.

Papadimitriou, C. H. (1994), *Computational Complexity*, Addison-Wesley, Reading, Massachusetts, 1994.

Piston, W., DeVoto, M., and Jannery, A. (1987), *Harmony* (5th edition), W. W. Norton & Company, New York, 1987.

Poole, D., Mackworth, A., and Goebel, R. (1998), *Computational Intelligence*, Oxford University Press, New York, 1998.

Russell, S., and Norvig, P. (2003), *Artificial Intelligence: A Modern Approach* (2nd edition), Prentice-Hall, Englewood Cliffs, New Jersey, 2003.

Sam, A. H., and Beynon, H. (2003), *Rapid Differential Diagnosis* (2nd edition), Blackwell Publishing, Oxford, 2003.

Schrijver, A. (1986), *Theory of Linear and Integer Programming*, Wiley, Chichester, 1986 and 1998.

Straach, J. (1998), *Effective Optimization in Expert Systems*, Ph.D. thesis, University of Texas at Dallas, 1998.

Straach, J., and Truemper, K. (1999), Learning to ask relevant questions, *Artificial Intelligence* 111 (1999) 301–327.

Sun, F.-S. (1998), *Error Prediction in Data Mining*, Ph.D. thesis, University of Texas at Dallas, 1998.

Thomas, L. C., Edelman, D. B.. and Crook, J. N. (2002), *Credit Scoring and Its Applications*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 2002.

Triantaphyllou, E., and Felici, G. (2004), *Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques*, Kluwer Academic Publishers, New York, 2004.

Truemper, K. (1998), *Effective Logic Computation*, Wiley, New York, 1998.

Wittgenstein, L. (1953), *Philosophical Investigations* (E. Anscombe and G. H. von Wright, eds.), Blackwell, Oxford, 1953.

Wittgenstein, L. (1958), *The Blue and Brown Books* (R. Rhees, ed.), Blackwell, Oxford, 1958.

Woodside, G. (1999), *Hazardous Materials and Hazardous Waste Management* (2nd edition), Wiley, New York, 1999.

Wos, L., Overbeek, R., Lusk, E., and Boyle, J. (1992), *Automated Reasoning* (2nd edition), McGraw-Hill, New York, 1992.

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Author Index

Design of Logic-based Intelligent Systems. Klaus Truemper
 ISBN: 0-471-48403-2

Subject Index

M