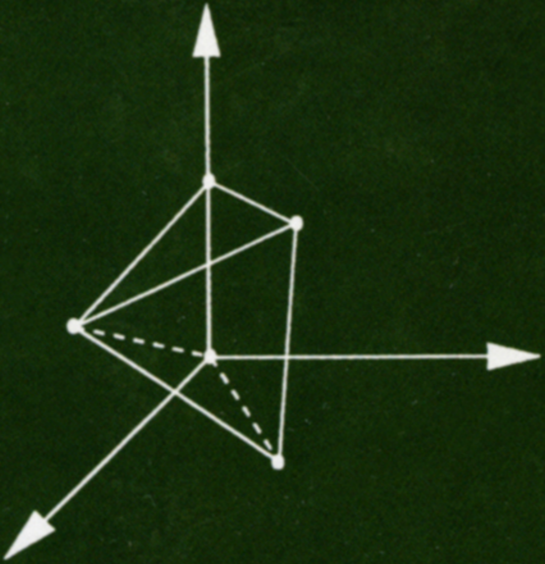
Wiley-Interscience Series in Discrete Mathematics and Optimization

OPTIMIZATION METHODS FOR LOGICAL INFERENCE



VIJAY CHANDRU

JOHN N. HOOKER

Optimization Methods For Logical Inference

Optimization Methods For Logical Inference

Vijay Chandru

John Hooker

A Wiley-Interscience Publication

JOHN WILEY & SONS, INC.

New York • Chichester • Weinheim • Brisbane • Singapore • Toronto

Library of Congress Cataloging in Publication Data:

Chandru, Vijay, 1953–
Optimization methods for logical inference / Vijay Chandru, John Hooker.
p. cm. — (Wiley-Interscience series in discrete mathematics and optimization)
"A Wiley-Interscience publication."
Includes bibliographical references.
ISBN 0-471-57035-4 (cloth : alk. paper)
1. Combinatorial optimization. 2. Logic, Symbolic and mathematical. I. Hooker, John, 1944– . II. Title. III. Series.
QA402.5.C447 1999
519.3—dc21 98-44622
CIP

10 9 8 7 6 5 4 3 2

To our parents

Contents

Preface

This book owes much to others. To a large degree it rests upon intellectual foundations laid by Robert E. Jeroslow and H. Paul Williams, both of whom explored the deep connections between logic and mathematical programming.

As authors, we are indebted to a number of people who read parts of the manuscript or just helped us to think about logic and optimization. They include Kim Allan Andersen, Raphael Araque, Egon Balas, Peter Barth, Alexander Bockmayr, Vivek Borkar, Endre Boros, Collette Coullard, Gérard Cornuéjols, Giorgio Gallo, Harvey Greenberg, Ignacio Grossmann, Peter Hammer, Jean-Louis Lassez, Sanjoy Mitter, Arun Pujari, Ron Rardin, Gerald Thompson, Klaus Truemper, M. Vidyasagar, V. Vinay, and John Wilson.

We have also benefited from collaboration with our present and former students: Srinivas Bollapragada, Geon Cho, Milind Dawande, Chawki Fedjki, Farid Harche, V. S. Jayachandran, Hak-Jin Kim, Miguel Montañez, Srinath Naidu, N. R. Natraj, Maria Auxillio Osorio, Greger Ottosson, Arnab Paul, Gabriella Rago, Ramesh Raman, Suman Roy, Anjul Shrivastava, and Hong Yan.

We acknowledge Neal Glassman and Abraham Waksman of the U.S. Air Force Office of Scientific Research, and particularly Donald Wagner of the U.S. Office of Naval Research, who foresaw the potential of combining logic and optimization and provided continuing support in the form of research grants.

Finally, we thank our editors at Wiley for their patience and support during the lengthy gestation of this book.

VIJAY CHANDRU
JOHN HOOKER

Bangalore, India
Pittsburgh, Pennsylvania, USA
January, 1999

Optimization Methods For Logical Inference

Chapter 1

Introduction

Solving logical inference problems with optimization methods may seem a bit like eating sauerkraut with chopsticks, because the two come from vastly different worlds. Logical inference comes from a "left brain" world of formal languages and symbolic manipulation. Optimization comes from a "right brain" world of spatial models and numerical calculation. Logical inference is associated with artificial intelligence and computer science, and optimization with operations research and engineering.

But on second thought it should not be so surprising that optimization methods are useful in logic. It is the mathematical structure of a problem that determines whether an optimization model can help solve it, not the context in which the problem occurs. Tasks as different as routing soft drink trucks and manufacturing circuit boards, for instance, can pose the same optimization problem (in this case, the traveling salesman problem). We should therefore expect that logical inference might pose some familiar optimization problems.

The thesis of this book is that many deductive inference problems do in fact have the sort of mathematical structure that optimization methods, or methods suggested by optimization, can exploit. These problems arise in logics that have important applications in artificial intelligence, computer science, decision support, and manufacturing:

propositional logic
first-order predicate logic
probabilistic and related logics
logics that combine evidence (e.g., Dempster-Shafer theory)
rule systems with confidence factors
constraint logic programming systems

The last several years have already seen orders-of-magnitude improvements in inference algorithms for propositional and probabilistic logic, due in part to the techniques described in this book.

A mathematical analysis of inference problems also reveals some interesting parallels between logic and mathematics. These can lead to better algorithms, both numeric and symbolic. For instance, the inference problem in propositional logic can be solved as an integer programming problem. Furthermore, a well-known inference procedure in logic (unit resolution) generates cutting planes that help solve the integer programming problem. The result is a kind of symbiosis: logic helps integer programming to solve logic problems. Similar insights can allow one to identify special cases of inference problems that are easy to solve. This is important to do because the general inference problem in most logics of interest can be very hard computationally. For example, inference is easy for the "Horn" propositions used in most expert systems because they are associated with polyhedra that have a least element property. Since the same property is shared by other polyhedra, this leads to the extension of Horn propositions to a much larger class for which inference is equally easy.

Most of the pioneering work in the logic/optimization interface was done within the last three decades, notably by H. P. Williams [290, 291, 296], who showed some connections between inference in propositional logic and the projection of polyhedra, and R. Jeroslow [23, 165], who used integer programming to solve inference problems and introduced a number of other seminal ideas. In addition, T. Hailperin found in the work of George Boole the elements of a linear programming model for probabilistic logic [27, 28, 124, 126]. Most of the research in the area is even more recent, having taken place within the last dozen years (that is, after [141]).

It is therefore possible for a single book to describe a large fraction of what has been done, and this we undertake to do. We also present a number of new results of our own.

Although the book is addressed to a technical audience, we are aware that our readers represent a large variety of backgrounds: operations research, computer science, logic, artificial intelligence, and engineering. We therefore presuppose as little as possible. The entire book should be readily intelligible to an otherwise qualified reader with absolutely no background in logic. It is more difficult to make the same promise with respect to optimization, on which we draw more deeply. Some exposure to linear or integer programming would no doubt be helpful. But we try to explain everything we do and provide an appendix to cover basic concepts that we do not explain elsewhere.

The book is organized by the types of logic to which optimization methods have been applied. *Propositional logic*, the most basic sort of logic, has been the most thoroughly investigated from the point of view of optimization. It is allotted two long chapters. Chapter 2 deals with specialized methods for particular classes of propositional inference problems, and Chapter 3 with general methods. *Probabilistic logics* assign propositions probabilities rather than regarding them simply as true or false. The quantitative nature of probability naturally invites numerical methods, as witnessed by Boole's early contribution. Chapter 4 covers these as well as some applications to Dempster-Shafer theory (an approach to combining evidence) and rule systems with confidence factors.

First-order predicate logic, which contains such expressions as "for all" and "there exists," is a powerful system and consequently much harder to completely solve inference in. Chapter 5 contains a description of partial instantiation, a new methodology for inference in predicate logic, that was motivated by paradigms in large-scale optimization. Compactness and other structural properties of infinite-dimensional mathematical programming are used to analyze inference in first-order logic in this penultimate chapter. Chapter 6 completes our description of how optimization methods can be used to address logical inference by presenting brief applications of mathematical programming in many valued logics and nonclassical logics such as nonmonotonic logics, modal logics, and constraint logic programming. We have limited our discussion in this book to the use of optimization methods in deductive inference.

The rest of this introductory chapter puts the research described here into context. The application of quantitative methods to logic is part of a larger movement toward the merger of the two worlds of logic and mathematics. Moreover, logic has recently become a basic modeling tool alongside mathematics, and the two styles of modeling are beginning to combine. Thus the need for logical inference methods, particularly those that involve quantitative methods, is growing. The need becomes particularly urgent as logic models become larger, because the difficulty of inference increases very rapidly with the size of the model.

1.1 Logic and Mathematics: The Twain Shall Meet

For more than two millennia, methods of formal reasoning have followed two largely separate lines of development. On the one hand is formal logic, which stretches from the Jains of ancient India, to Aristotle's systematic investigations, through the medieval logicians, to such mathematical logicians as Gottlob Frege and Kurt Gödel, and finally to the large community of researchers now applying logic to problems in computer science and artificial intelligence. On the other hand is mathematical computation, which began even earlier in Egypt and Mesopotamia and developed into today's highly advanced methods.

Although largely parallel, these two paths have crossed on occasion, sometimes with spectacular results. This book aims to participate in what we believe is the latest crossing. But a proper appreciation of this phenomenon requires some more background.

The seventeenth century philosopher Gottfried Wilhelm von Leibniz brought about the first important synthesis of logical deduction and mathematical computation. His contribution was to point out that they are fundamentally the same. To follow the steps of a numeric algorithm is to perform a series of deductions. Conversely, a series of deductions in formal logic can be viewed as computation and can be automated just as long division can. Leibniz went so far as to envision a calculus of reasoning (*calculus ratiocanator*) in which all truths can in principle be obtained from self-evident premises by calculation.

Leibniz did not have the technical wherewithal even to begin to realize his vision, but George Boole did. He showed how to compute inferences in propositional logic with his famous algebra, and he did the same for probabilistic logic using an idea that anticipates linear programming. He began a series of developments, too long to recount here, that culminated in the field of artificial intelligence, which aims in part to carry out Leibniz's project of automating the reasoning process. More relevant for our purposes, however, is what was *not* accomplished by this and subsequent encounters of logic and mathematics, until very recently. Although Boole saw that mathematical and logical calculation are of a piece, he did not bring them together into a single calculus. On the contrary, a major point of Boolean algebra was to show that computation can be purely nonnumeric, or "symbolic" as we now sometimes say. Boole used arithmetical symbols to denote logical operations, but this was only a notational convenience [185].

Other encounters grew out of Leibniz's work but likewise fell short of true fusion. Leibniz realized that logical computation would be possible only within a formal language (*characteristica universalis*). This idea eventually led to the development of such formal languages as predicate logic and set theory, within which Bertrand Russell, Alfred North Whitehead, and their successors attempted to formalize mathematics in a "logicist" manner. (It is no accident that Russell wrote his dissertation on Leibniz.) Alfred Tarski and others devised decision procedures for logics that contain arithmetic. In fact, the whole formalist paradigm of doing mathematics, which in its starkest form (due to D. Hilbert) regards mathematics as the study of how uninterpreted symbols may be manipulated in a formal language, is largely inspired by the Leibnizian legacy. But through all these developments, it is hard to find an occasion on which someone actually introduced numerical methods into the logical deduction process, or logical methods into numerical computation.

Perhaps the first marriage of logical and mathematical computation (with the interesting exception of Boole's probabilistic logic) occurred with the arrival of digital electronic computers, which use logic gates to implement arithmetic operations in a circuit. But it was a marriage without consummation, because the logical operations remain at the micro level and leave the higher-level numeric algorithms unaffected.

The relationship was brought a bit closer when work in artificial intelligence revived interest in decision procedures for logics, including logics of arithmetic. Some algorithms were devised that solve numeric problems by replacing key numeric procedures with symbolic ones. There is, for instance, a partially symbolic algorithm for linear programming (the "SUP-INF method" [261]). These algorithms had limited impact on methods for numeric problems, however, because symbolic procedures are often unable to exploit the special structure of these problems to the extent that purely numeric algorithms do. The SUP-INF method, for instance, is extremely inefficient (it has doubly exponential complexity).

The latest mathematical/logical encounter is bidirectional. This book focuses on the application of numeric methods to logical deduction. The reverse influence is also underway. Logic programming and its successors are now used to solve problems that have long been attacked by the numerical methods of operations research. The successors include constraint logic programming and constraint satisfaction techniques [207, 286], which not only apply discrete and logical methods but integrate linear programming as well, adding another layer of interaction.

The work described in this book can therefore be seen as an episode in the cross-fertilization of logic and mathematics initiated by Leibniz. The word "optimization" appears in the title because most of the mathematical ideas applied to logical inference in the last few years have related to optimization methods, which turn out to be particularly suited to computing inferences. It is perhaps pure coincidence that the concept of optimization played a key role in Leibniz's philosophy.

Not all methods discussed herein are optimization methods, and not all are numeric. Yet all are inspired by such methods in one way or another, even if only in the sense that they result from a similar style of thinking. In fact, the main benefit of focusing on optimization may be simply that it encourages a research community rooted in the mathematical tradition to try its hand at inference problems.

1.2 Inference Methods for Logic Models

As inference problems grow in size and importance, it is more important than ever to solve them efficiently. Some of these problems come from specific domains that pose problems of a logical nature. Electronic circuit design and testing, for instance, have presented hard logic problems for some decades, and these problems get harder as circuits get more complex. Research in automated theorem proving for mathematics yielded a number of new inference methods. But there is a more general phenomenon that has made inference problems a standard feature of the technological landscape. This is the rise of *logic models* and, more recently, logico-mathematical models.

Just as logical and mathematical computation are essentially the same, a logic model is essentially the same kind of structure as a mathematical model. Either kind of model describes a problem in a formal language that allows one to deduce facts about its solution. In a mathematical model, the formalism is a mathematical theory, and the deduction takes the form of algebraic manipulation or numeric calculation.

The practice of logic modeling grew partly out of attempts to create artificial intelligence. An intelligent computer should not only be able to solve well-structured problems traditionally attacked with mathematical models, but it should be able to solve such "messier" problems as scheduling operations in a factory, designing a building layout, guiding a robot over unfamiliar terrain, diagnosing an illness, or interpreting natural language. The first two problems and perhaps the third may be formulable in mathematical terms but tend to be very hard to solve with mathematical methods. The remaining problems do not even admit a mathematical formulation.

One response to this dilemma in the artificial intelligence community has been to describe problems in a general-purpose formalism that permits deductions but presupposes no mathematical structure. Thus we see the development of such "knowledge-based systems" as expert systems. An expert system for diagnosing failures in a piece of machinery, for instance, might contain a few hundred "rules" that look something like, "If indicator light A is red and noise B is audible, then circuit C is defective." The user adds his observations to the set of rules, whereupon an "inference engine" deduces what may be wrong with the machine.

Another response has been the development of logic programs, normally written in PROLOG after a fashion initially advocated by Kowalski [188]. PROLOG accepts statements written in a restricted form of predicate logic and computes some of their implications. Inference is incomplete because of the lack of practical methods to solve the problem completely.

More recently, logic programming has been largely supplanted by *constraint programming* [207], which offers a number of specialized predicates that are useful for real-world problems, along with specialized algorithms to deal with them. They include both discrete predicates, such as "all-different," and arithmetical predicates.

Expert systems and other decision support systems are often designed to reason under uncertainty or incomplete information, the former most often captured by probabilities. The classical framework for reasoning with probabilities is Bayesian inference, which is the basis for decision trees. More recent variations include influence diagrams, which are based on Bayesian networks rather than trees.

Several logics are designed to account for uncertainty and incomplete information. Boole's original probabilistic logic does both, as do Dempster-Shafer theory and other logics of belief. These form the basis for "belief nets," of which Bayesian networks are only one example. Their nodes represent propositions and their arcs dependencies among propositions that permit inferences.

Data bases are important components of many decision support systems and can likewise be regarded as logic models in which predicate logic plays an important role. "Default logic" and "nonmonotonic logic" were developed partly to deal with logical problems they posed. Default logic allows one to make generalizations that are not strictly supported by data, and nonmonotonic logic allows one to retract an inference when additional information becomes available. Both types of reasoning (they are typically combined) now play a role in modeling systems other than data bases. Modal logics, including temporal logic, are also used to model knowledge acquisition as well as to verify computer programs, for example.

In addition to all this is a vast literature on fuzzy logic systems, which were originally intended to account for vagueness but now seem to serve other purposes as well.

1.3 Logic Modeling Meets Mathematical Modeling

Not only are logic models proliferating, but a newer trend is underway: their merger with more traditional mathematical models. Inference procedures that combine numeric and nonnumeric elements may be particularly appropriate for models that do the same.

One impetus for the merger lies in the fact that, in many situations, neither type of modeling alone is true to reality. At one point in history, an elegant mathematical reality seemed to underlie the world, and mathematical models sufficed. But the "modern" age of Newtonian simplicity has given way to a "postmodern" age of bewildering complexity that calls for a mixture of modeling styles. We want to understand and manage such complex systems as the economy and the atmosphere and large manufacturing plants, but there are no elegant mathematical models from which we can deduce predictions, or prescriptions for action, with only second-order error. To get even a first approximation we must work with a system description that may be nearly as complex as the system itself. One attraction of chaos theory was that it seemed to bring complex, baffling phenomena once again under the purview of simple mathematical models. But if such writers as Freeman Dyson [90] are right, it may only be a momentary respite from an overwhelming trend toward messiness.

We should therefore expect a postmodern model to be a mosaic of mathematical and nonmathematical elements. Some aspects of a problem may display mathematical structure, and a model would be remiss to neglect this—not only for the sake of verisimilitude, but to make deduction (i.e., computation) easier. A factory subsystem that admits a linear programming submodel, for instance, should be so modeled, if only to get a correct solution quickly for that part of the system. But other aspects of the problem will likely not submit easily to mathematical modeling and may call for logic-based modeling. It has become commonplace to mix the two in constraint programming, and a similar mixture seems on the horizon in mathematical programming.

It is hard to predict where logico-mathematical modeling will lead. Already, new formalisms are evolving that are classifiable as neither mathematical nor logical but have some structural similarities with both. In the meantime, the research described in this book can play a role in its development. One obvious contribution derives from its use of mathematical programming models to compute the implications of a logic model. But more generally, a persistent theme of the research is the discovery of structural and algorithmic commonality between logic and mathematics. One can expect this sort of discovery to hasten the fusion of logical models and methods with those of mathematics.

1.4 The Difficulty of Inference

Logical inference can be a very hard combinatorial problem. It is perhaps curious that one of the fundamental tasks of information science—deducing the implications of what we already know—could be so hard. It is hard in the sense that the amount of computation required to check whether a knowledge base implies a given proposition grows very rapidly with the size of the knowledge base. Thus as logic models grow larger, we face a need for continuing breakthroughs in the speed of inference algorithms, or else new ways to structure models so as to make inference easier.

The difficulty of inference in several logics can be stated quite precisely. In propositional logic, the most basic we consider, no known algorithm always solves the inference problem in better than exponential time (i.e., the solution time increases exponentially with the size of the knowledge base in the worst case). Technically, the problem is "NP-complete" [66, 108]). Robinson's well-known resolution algorithm [244], when specialized to propositional logic, requires exponential time in the worst case when applied to propositional logic [128]. Computational experience [139] indicates that the running time rapidly explodes in the typical case as well. We prove in Chapter 3 that a tree search algorithm (the Davis-Putnam-Loveland algorithm) is also exponential in the worst case. Franco and Paul [97] showed that it requires exponential time with probability approaching one when large random problems are chosen from a reasonable distribution.

The situation is even worse in first-order predicate logic. In this case the problem is not only hard but insoluble in general. There are procedures, such as the resolution procedure, that verify that any given implication in fact follows from the premises. But Alonzo Church proved in 1936 (and Alan Turing shortly thereafter) that there is no procedure that can always verify in finite time that a given nonimplication does not follow [57, 279]. To have a finite decision procedure one must restrict oneself to a fragment of predicate logic, and even then inference is very hard. One such fragment consists of formulas in Schönfinkel-Bernays form, in which all existential quantifiers ("there exists") precede all universal quantifiers ("for all"). Checking the satisfiability of Horn formulas in Schönfinkel-Bernays form, which comprise a very limited fragment, requires exponential time in the worst case [235]. Doing the same for an arbitrary formula in Schönfinkel-Bernays form requires "nondeterministic exponential" time and is therefore even harder [195].

Because propositional logic is a subset of probabilistic logic, inference in the latter is likewise hard and is easily shown to be an NP-complete problem [113].

The proper reaction to these observations is not to despair of solving large inference problems. Clever algorithms can overcome their difficulty in many cases, and analysis can identify subsets of logics for which inference is easier.

This page intentionally left blank

Chapter 2

Propositional Logic: Special Cases

We begin with propositional logic, the simplest sort of logic. Propositional logic, sometimes called sentential logic, may be viewed as a grammar for exploring the construction of complex sentences (propositions) from "atomic statements" using the logical connectives such as "and," "or," and "not." The fundamental problem of inference in logic is to ascertain if the truth of one formula (proposition) implies the truth of another. Even for elementary propositional logic, the inference problem is not entirely well-solved.

The problem of testing the satisfiability of a propositional formula is the core problem of inference. This ***satisfiability problem*** is simply stated as the problem of finding a set of truth assignments for the atomic propositions that renders the formula true. An obvious algorithm for this problem is to enumerate all possible truth assignments and evaluate the formula until it is satisfied or proved unsatisfiable. This would entail an unacceptable amount of work for all but small and uninteresting formulas. Our focus in this chapter will be on special formulas for which satisfiability can be tested extremely rapidly by algorithms whose run times are, typically, within a constant factor of the time it takes a computer to read the formula.

Quadratic and Horn formulas represent two classical examples of structured propositions that admit highly efficient (linear-time) inference algorithms. We shall explore these and related structures in this chapter using a quantitative or optimization-based perspective. This approach has two advantages. The first is that it leads to an understanding of the mathematical structure of these formulas which the purely syntactic approach of symbolic computation misses altogether. We will also demonstrate that this perspective has been very useful in identifying rich new classes of formulas which, like the Quadratic and Horn classes, admit fast inference algorithms. These developments, many of which are quite recent, have the potential for substantively increasing the expressive power of highly tractable fragments of propositional logic.

The main optimization-based perspective of satisfiability is developed on an (0-1) integer programming formulation. This formulation is a manifestation of a well-known reduction of satisfiability to testing the solubility of a system of linear inequalities defined on variables that are restricted to binary values of 0 or 1. Since solvability and optimization in integer programming are closely related we can view satisfiability as an optimization problem. A central idea in integer programming is to relax the integrality of the variables by treating the 0-1 variables as continuous variables taking values in the range [0,1] to obtain the *linear programming relaxation*. Linear programming problems are far easier to handle, both mathematically and computationally, than integer programming problems. The linear programming relaxations of satisfiability problems are particularly easy to solve because unit resolution can be adapted to design a complete solution method for these special linear programs.

The linear programming relaxation of a Horn formula turns out to have a remarkable property. The linear program has an integral least element that corresponds to the unique minimal model of the Horn formula. We will see that this least element property can be generalized in a very natural way to realize a class of Extended Horn Formulas [44]. The least element property of Horn formulas also permits a rich theory of proof signatures for these formulas based on the duality theorem of linear programming.

We will also study a class of "Q-Horn" formulas [34] that simultaneously generalizes Quadratic and Horn formulas. The class is characterized by a special property of a related linear programming problem. Another paradigm from optimization that we will find useful in studying special propositional formulas is that of forbidden minor characterizations of special structures. This paradigm will help in the study of Q-Horn formulas as well as some recursive structures built on Horn formulas that still yield tractable formulas that are called Generalized Horn Formulas [5, 106, 302].

We will start with a quick review of some of the basic concepts of propositional logic. This will lead us to the (0-1) integer programming formulation of satisfiability and on to the study of special structures in propositional logic.

2.1 Basic Concepts of Propositional Logic

In propositional logic we consider formulas (sentences, propositions) that are built up from atomic propositions ($x_1, x_2, \ldots, x_n$) that are unanalyzed. In a specific application, the meaning of these atomic propositions will be known from the context. The traditional (symbolic) approach to propositional logic is based on a clear separation of the syntactical and semantical functions. The syntactics deals with the laws that govern the construction of logical formulas from the atomic propositions and with the structure of proofs. Semantics, on the other hand, is concerned with the interpretation and meaning associated with the syntactical objects. Propositional calculus is based on purely syntactic and mechanical transformations of formulas leading to inference. The "principle of resolution" is the most important transformation rule for this purpose. We will maintain this traditional approach to propositional logic in this section. However, it should be noted that the quantitative or optimization perspective that will be introduced in the next section is not beholden to this doctrine of separating syntactics and semantics. In fact, much of its power derives from the integration of the two functions.

We will first review the syntactics of forming propositional logic formulas. The assignment of truth values (true/false) to atomic propositions and the evaluation of truth/falsehood of formulas is the essence of the semantics of this logic. The central problem of inference in propositional logic is the satisfiability problem. This is the problem of determining whether a given formula is true (is satisfied) for some assignment of truth values to the atomic propositions. Two formulas are (semantically) equivalent if their satisfying truth assignments are equivalent under some mapping. We will discuss rewriting "well-formed formulas" in equivalent canonical representations called "normal forms." A particular normal form, the "conjunctive normal form" (CNF), is the canonical representation of propositions that we use in the ensuing discussion of propositional logic. The artificial intelligence (AI) literature often describes the calculus of propositional logic in the syntax of "if-then" rules, which are sometimes called "inference rules." We shall see that this is just an alternate schema for CNF formulas.

2.1.1 Formulas

Propositional logic formulas are built up from atomic propositions using various logical connectives. The *primary connectives* are $\wedge, \vee, \neg$ which are understood to represent the semantics of *and, or, not,* respectively. An inductive definition of well-formed formulas (wffs) using these connectives is given by:

(a) Atomic propositions are wffs.

(b) If S is a wff so is $\neg S$.

(c) If S_1 and S_2 are wffs, so are $S_1 \vee S_2$ and $S_1 \wedge S_2$.

It is customary to use parentheses "(" and ")" to distinguish the start and end of the field of a connective operation. Consider the formula

$$((\neg x_1 \vee (x_1 \wedge x_3)) \wedge ((\neg(x_2 \wedge \neg x_1)) \vee x_3)) \tag{2.1}$$

We notice that since $\neg$ is a unary operator, the field on which it operates follows immediately. The subformula $\neg x_1$ has $\neg$ applied to the atomic proposition x_1 and so no parentheses are required. Whereas the subformula $(\neg(x_2 \wedge \neg x_1))$ has $\neg$ applied to the nonatomic proposition $x_2 \wedge \neg x_1$ and parentheses are required to delineate the field. The binary operators $\vee$ and $\wedge$ have their field to the left and right of them and the parentheses delineate them. For example, the first open parenthesis "(" and the last close parenthesis ")" delineate the field of the $\wedge$ connective that appears in the middle of the formula. A formula is a wff if and only if there is no conflict in the definition of the fields of the connectives. Thus a string of atomic propositions and primitive connectives, punctuated with parentheses, can be recognized as a well-formed formula by a simple linear-time algorithm. We scan the string from left to right while checking to ensure that the parentheses are nested and that each field is associated with a single connective. Incidentally, in order to avoid the use of the awkward abbreviation "wffs," we will henceforth just call them propositions or formulas and assume they are well formed unless otherwise noted.

The calculus of propositional logic can be developed using only the three primary connectives $\{\neg, \vee, \wedge\}$. However, it is often convenient to permit the use of certain additional connectives. Three such connectives that we will find occasion to use are $\rightarrow$, $\bigvee$, and $\bigwedge$. They are, respectively, the connective "implies," "p-ary disjunction," and the "p-ary conjunction." They are essentially abbreviations that have equivalent formulas using only the primary connectives. The equivalences are detailed below.

$$(S_1 \rightarrow S_2) \quad \text{is equivalent to} \quad (\neg S_1 \vee S_2)$$

$$(\bigvee_{i=1}^{p} S_i) \quad \text{is equivalent to} \quad (\cdots(S_1 \vee S_2) \vee S_3)\cdots S_p)$$

$$(\bigwedge_{i=1}^{p} S_i) \quad \text{is equivalent to} \quad (\cdots(S_1 \wedge S_2) \wedge S_3)\cdots S_p)$$

The truth or falsehood of a formula is a semantic interpretation that depends on the values of the atomic propositions and the structure of the formula. The elements of the set {T,F} are called truth values with "T" denoting "true" and "F" denoting "false." A *truth assignment* is simply the assignment of values T or F to all the atomic propositions. To evaluate a formula we interpret the connectives $\neg$, $\vee$, and $\wedge$ with the appropriate meaning of "not," "or," and "and." As an illustration, consider the formula (2.1). Let us start with an assignment of true (T) for all three atomic propositions x_1, x_2, and x_3. At the next level, of subformulas, we have $\neg x_1$ evaluates to false (F), $(x_1 \wedge x_3)$ evaluates to T, $(x_2 \wedge \neg x_1)$ evaluates to F, and x_3 is T. The third level has $(\neg x_1 \vee (x_1 \wedge x_3))$ evaluating to T since it is the "or" of two subformulas one of which is false and the other true. Eventually $((\neg(x_2 \wedge \neg x_1)) \vee x_3))$ also evaluates to T. The entire formula is the "and" of two propositions both of which are true, leading to the conclusion that the formula evaluates to T. This process is simply the inductive application of the rules:

(a) S is T if and only if $\neg S$ is F.

(b) $(S_1 \vee S_2)$ is F if and only if both S_1 and S_2 are F.

(c) $(S_1 \wedge S_2)$ is T if and only if both S_1 and S_2 are T.

We now introduce a variety of inference questions related to the truth or falsehood of propositions. The ***satisfiability problem*** is the problem of determining whether a given formula is "satisfied" (evaluates to T) for some assignment of truth values to the atomic propositions. We showed that the formula (2.1) is indeed satisfiable, since it is satisfied by the truth assignment $(x_1, x_2, x_3) = (\text{T,T,T})$. A satisfying truth assignment is called a *model*. A formula with no model is called unsatisfiable. A formula for which every truth assignment is a model is called a *tautology*. The formula $(x_1 \vee \neg x_1)$ is a tautology. A formula S_1 is said to imply another formula S_2, defined on the same set of atomic propositions as S_1, if every model of the former is also a model of the latter. Two formulas are said to be equivalent if they share the same set of models. We now have three basic inference problems in propositional logic.

- Is a given formula satisfiable?
- Is a given formula a tautology?
- Does one formula imply another?

Notice that all three problems can be solved by enumeration. We simply list all possible truth assignments (there are 2^n of them on n atomic propositions). By evaluating the formula on each of them we could certainly answer all three questions. In many textbooks on logic, this technique would be called the method of "truth tables." The method is, of course, not a serious contender since the size 2^n is unmanageable. However, the fact that one method solved all three problems is indicative of a possible relationship between these problems. Indeed there is a simple equivalence between all three problems.

Proposition 1 *The following statements are equivalent:*

1. *S implies $\tilde{S}$.*
2. *$(\tilde{S} \vee \neg S)$ is a tautology.*
3. *$(S \wedge \neg\tilde{S})$ is unsatisfiable.*

The proposition substantiates the earlier claim that satisfiability is the central problem of inference in propositional logic. In the next section we will study syntactic transformation rules that modify a formula without affecting its models. We will use these rules to obtain special representations called normal forms of formulas. These normal forms are particularly suitable for devising solution strategies for the satisfiability problem and for developing the optimization-based perspective of inference.

2.1.2 Normal Forms

The normal forms of formulas are special representations that make evident the duality of the binary connectives: "conjunction" $\wedge$ and "disjunction" $\vee$. To begin with, we need to get rid of the unary "negation" operator "$\neg$." This is done by expanding the set of atomic propositions $\{x_1, x_2, \ldots, x_n\}$ into the *set of literals* $\{x_1, x_2, \ldots, x_n; \neg x_1, \neg x_2, \ldots, \neg x_n\}$. Given any formula, our task will be to absorb the negation connective "$\neg$" into the literals. This is achieved by repeated application of the familiar De Morgan's laws,

$$\neg(S_1 \vee S_2) \text{ is equivalent to } (\neg S_1 \wedge \neg S_2) \tag{2.2}$$

$$\neg(S_1 \wedge S_2) \text{ is equivalent to } (\neg S_1 \vee \neg S_2) \tag{2.3}$$

along with the involutory property of negation,

$$\neg\neg S = S \tag{2.4}$$

A moment's reflection will convince the reader that given any subformula operated on by $\neg$, there is a linear-time algorithm to get rid of $\neg$ as an operator. We apply (2.3, 2.4) recursively until $\neg$ operates only on atomic propositions. Thus we only have formulas with literals connected by disjunctions $\vee$ and conjunctions $\wedge$. A subformula that is a pure disjunction of literals is called a *clause*. A formula is said to be in *conjunctive normal form* or CNF if it is the pure conjunction of a set of clauses. Such a formula would have the appearance

$$\begin{aligned} S &= \bigwedge_{i=1}^{m} C_i \\ C_i &= \bigvee_{j \in J_i} \pm x_j \text{ for } i = 1, 2, \ldots, m \end{aligned}$$

The notation $+x_j$ denotes the "positive literal" or atomic proposition x_j, while $-x_j$ denotes the "negative literal" $\neg x_j$. A formula is said to be in *disjunctive normal form* or DNF if it is the pure disjunction of *terms*, each of which is a pure conjunction of literals. Such formulas have the appearance

$$\begin{aligned} S &= \bigvee_{i=1}^{m} T_i \\ T_i &= \bigwedge_{j \in J_i} \pm x_j \text{ for } i = 1, 2, \ldots, m \end{aligned}$$

The two normal forms CNF and DNF are dual representations with symmetric properties. Although there are some applications of propositional logic for which the DNF may be the more accepted normal form, we will stay with CNF as the standard form for propositions in this book. The traditional technique of transforming a given formula of literals connected by $\vee$ and $\wedge$ connectives into an equivalent CNF formula is by repeated application of the "distributive law":

$$S_1 \vee (S_2 \wedge S_3) \text{ is equivalent to } (S_1 \vee S_2) \wedge (S_1 \vee S_3) \tag{2.5}$$

We now have a complete procedure for transforming a given formula into CNF.

Procedure 1: *Reduction to CNF*

Step 1. Use the transformation rules of De Morgan's laws (2.3) and involution of negation (2.4) to absorb all $\neg$ into the literals.

Step 2. Use the distributive law (2.5) to move the conjunctions out of the subformulas until each subformula is a clause of pure disjunctions.

As an illustration of Procedure 1, let us consider the formula (2.1). The step by step transformations on this sample are depicted below.

$$\begin{array}{rcl} (\neg x_1 \vee (x_1 \wedge x_3)) & \wedge & ((\neg(x_2 \wedge \neg x_1)) \vee x_3) \\ (\neg x_1 \vee (x_1 \wedge x_3)) & \wedge & ((\neg x_2 \vee x_1) \vee x_3) \\ (\neg x_1 \vee x_1) \wedge (\neg x_1 \vee x_3) & \wedge & (x_1 \vee \neg x_2 \vee x_3) \\ (\neg x_1 \vee x_3) & \wedge & (x_1 \vee \neg x_2 \vee x_3) \end{array} \tag{2.6}$$

In the last step we removed the clause $(x_1 \wedge \neg x_1)$ since this is a tautology whose truth is invariant. In general, however, this simple procedure for reducing formulas to CNF runs into trouble. The main difficulty is that there may be an explosive growth in the length of the formula. The length of a formula is measured by the total number of literals in the description of the formula. Consider the action of Procedure 1 on the family of DNF formulas

$$(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2n-1} \wedge x_{2n}) \tag{2.7}$$

It is not difficult to see that the CNF formula produced, by Procedure 1, is made up of the 2^n clauses

$$(x_{p(1)}, x_{p(2)}, \ldots, x_{p(n)}) \text{ where } x_{p(j)} \in \{x_{2j-1}, x_{2j}\} \ (j = 1, 2, \ldots, n) \tag{2.8}$$

These arguments prove that, in the worst case, Procedure 1 is an exponential-time algorithm.

Theorem 1 ([23]) *In the worst case, Procedure 1 may generate a CNF formula whose length is exponentially related to the length of the input formula.*

One may the tempted to argue that the exponential growth in the formula is caused by "redundant" clauses. Let us pursue this suspicion and convince ourselves that it is unfounded. A clause, in a CNF formula, is called redundant if it is implied by the remaining clauses. A single clause C *absorbs* clause D if every literal in C appears in D. Further, a clause D is logically implied by a single clause C if and only if C absorbs D or if D is a tautology. In a CNF formula therefore a clause will be redundant if it is absorbed by another. What makes inference, in propositional logic, so complex is the fact that redundancy can be deeply embedded. Fortunately, there is a simple principle that, on repeated application, reveals the embedded structure. This is the *principle of resolution*.

In any CNF formula, suppose there are two clauses C and D with exactly one atomic proposition x_j appearing negated in C and posited in D. It is then possible to resolve C and D on x_j. We call C and D the ***parents*** of the ***resolvent clause***, which is formed by the disjunction of literals in C and D, excluding x_j and $\neg x_j$. The resolvent is an implication of the parent clauses.

As an illustration of this principle consider the last line of formula (2.6).

$$\begin{array}{rcl} C & = & (\neg x_1 \lor x_3) \\ D & = & (x_1 \lor \neg x_2 \lor x_3) \\ \hline & & (\neg x_2 \lor x_3) \end{array}$$

The resolvent of C and D is the clause below the line. The resolution principle is a manifestation of case analysis, since if C and D have to be satisfied and (case 1) x_1 is true, then x_3 must be true; or if (case 2) x_2 is false then $(\neg x_2 \lor x_3)$ must be true. Hence every model for C and D is also a model for the resolvent. A ***prime implication*** of a CNF formula S, is a clause that is an implication of S that is not absorbed by any clause that is an implication of S, except by itself. In the above example, the prime implications are

$$\{(\neg x_1 \lor x_3), (\neg x_2 \lor x_3)\}$$

A simple procedure for deriving all prime implications of S is to repeatedly apply the resolution principle while deleting all absorbed clauses. If we are left with an empty clause, the formula S has no model. Otherwise, we will be left with all the prime implications. The correctness of this procedure will be established in Section 3.1.1, as will its computational complexity. This elegant principle of resolution therefore provides a complete inference mechanism for propositional logic based on purely syntactic techniques. However, akin to "truth tables", the principle of resolution is not a computationally viable inference method for all but small examples, in the general case. In the section, and indeed for much of this chapter, we will work with a weakened form of resolution, called "unit resolution", in which one of the parent clauses is always taken to be a clause with exactly one literal. This weaker principle is complete only on special classes of propositions that we will define.

Returning to the class of formulas (2.8) that proved Theorem 1, we see that no further resolution is possible and none of the clauses absorb each other. Therefore they are prime implications. This debunks the notion that "succinct" CNF representations of any formula can be guaranteed by the "distributive law" approach of Procedure 1. In order to obtain such "succinct" representations, we will have to resort to a "rewriting" technique that introduces additional atomic propositions corresponding to subformulas of the given formula. The two formulas will be equivalent in that every model for the original formula can be extended to a model for the rewritten one (with appropriate truth values for the new atomic propositions). Conversely, every model of the rewritten formula, when restricted to the original atomic propositions (projected), corresponds to a model of the original formula. Extended formulations have been the subject of study in integer programming for at least two decades now. The central theme of the monograph by Jeroslow [168] is on this interplay between reformulations in logic and their counterparts in integer programming.

That these rewriting techniques lead to CNF representations that are polynomially bounded in length was first noted by Tseitin [278] in a remarkable paper written in 1968. This rewriting technique was improved in later papers by several authors [23, 66, 297]. We will use the most recently published, and most efficient technique due to Wilson [297]. We will first illustrate the technique on an example and then state the general idea. Consider a small instance of the formulas that were the ruin of Procedure 1.

$$(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6) \vee (x_7 \wedge x_8) \tag{2.9}$$

Let us introduce new propositions $x_{12}, x_{34}, x_{56}, x_{78}$ that represent the four parenthetic conjunct subformulas. For each $x_{2j-1,2j}$ $(j = 1, 2, 3, 4)$ we write the clauses

$$(\neg x_{2j-1,2j} \vee x_{2j-1}) \text{ and } (\neg x_{2j-1,2j} \vee x_{2j}) \tag{2.10}$$

We also need a clause to knit the four subformulas together.

$$(x_{12} \vee x_{34} \vee x_{56} \vee x_{78}) \tag{2.11}$$

Putting these clauses together we have rewritten the formula (2.9) in CNF using 12 atomic propositions and 9 constraints. Procedure 1 would have produced a CNF using only the original 8 atomic propositions but 16 clauses. The numbers are more dramatic for the general case (2.7). Procedure 1 obtained a CNF formula with $2n$ atomic propositions and 2^n clauses. The obvious extension of the rewriting technique introduced above would result in a CNF formula with $3n$ atomic propositions and $2n + 1$ clauses. Thus rewriting has brought the size down from exponential to linear for this class of examples. This result can be generalized for all formulas in propositional logic with a general rewriting technique described below.

Procedure 2: ***Rewriting to CNF***

Input. A well-formed formula F with subformulas $\{F_i\}$.