2000

Complete Randomized Cutting Plane Algorithms for Propositional Satisfiability

Stephen Lee Hansen
Nova Southeastern University, stephenhansenphd@gmail.com

Complete Randomized Cutting Plane Algorithms for Propositional Satisfiability

by

Stephen Lee Hansen

A Dissertation Report
submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

School of Computer and Information Sciences
Nova Southeastern University

2000

We hereby certify that this dissertation report, submitted by Stephen Lee Hansen, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the disseratation requirements for the degree of Doctor of Philosophy.

Lee J. Leitner, Ph.D. Chairperson of Dissertation Committee	Date
S. Rollins Guild, Ph.D. Dissertation Committee Member	Date
Sumitra Mukherjee, Ph.D. Dissertation Committee Member	Date

Approved:

Edward Lieblein, Ph.D. Dean, School of Computer and Information Sciences	Date

School of Computer and Information Sciences
Nova Southeastern University

2000

Certification Statement

I hereby certify that this dissertation constitutes my own product and that the words or ideas of others, where used, are properly credited according to accepted standards for professional publications.

Signed:

Stephen Lee Hansen

Date

An Abstract of a Dissertation Submitted to Nova Southeastern University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Complete Randomized Cutting Plane Algorithms for Propositional Satisfiability

by
Stephen Lee Hansen

August 7, 2000

The propositional satisfiability problem (SAT) is a fundamental problem in computer science and combinatorial optimization. A considerable number of prior researchers have investigated SAT, and much is already known concerning limitations of known algorithms for SAT. In particular, some necessary conditions are known, such that any algorithm not meeting those conditions cannot be efficient. This paper reports a research to develop and test a new algorithm that meets the currently known necessary conditions.

In chapter three, we give a new characterization of the convex integer hull of SAT, and two new algorithms for finding strong cutting planes. We also show the importance of choosing which vertex to cut, and present heuristics to find a vertex that allows a strong cutting plane. In chapter four, we describe an experiment to implement a SAT solving algorithm using the new algorithms and heuristics, and to examine their effectiveness on a set of problems. In chapter five, we describe the implementation of the algorithms, and present computational results. For an input SAT problem, the output of the implemented program provides either a witness to the satisfiability or a complete cutting plane proof of unsatisfiability.

The description, implementation, and testing of these algorithms yeilds both empirical data to characterize the performance of the new algorithms, and additional insight to further advance the theory. We conclude from the computational study that cutting plane algorithms are efficient for the solution of a large class of SAT problems.

Acknowledgements

I could not have pursued this research without the loving and enduring support of my domestic partner, Dennis Adams. Dennis both gave spiritual and emotional support, and took care of nearly all of the domestic chores for more than five years.

I would like to thank Glenn Weber of Christopher Newport University. Glenn introduced me to the SAT problem more than twenty years ago, showed me some of the deep connections between mathematical logic and discrete optimization, and inspired my early interest in the field.

I would like to thank my dissertation advisor, Lee Leitner, for his encouragement and support during the research, and for helping me to maintain a sense of balance. I would also like to thank my committee members, Rollie Guild and Sumitra Mukherjee for their careful reading and suggestions. Rollie suggested the approach of considering necessary conditions during a pre-class conversation in 1996.

Rollie Guild was unable to sign the final copy of this dissertation report due to a sudden medical emergency. S. Rollins Guild, Ph.D., passed from this earth on Friday, July 29, 2000. We shall remember him fondly.

In Memorium Rollie Guild.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The search problem in artificial intelligence and combinatorial optimization is to search a combinatorial structure in a feasible time to find a solution that is optimal or satisfactory in some sense. The combinatorial structure arises from problem representations that involve multiple independent variables, each having multiple possible values. The complexity class NP captures the essential difficulty of these combinatorial search problems. A decision problem is in NP if there exists a nondeterministic Turing machine that decides every instance of that problem in a polynomially bounded number of steps. Garey and Johnson (1979) give a catalog of decision and optimization problems that are in NP. Many of the problems in NP have high economic value, or are theoretically important.

The propositional satisfiability problem (SAT) is complete for complexity class NP (Cook, 1971). Cook showed how to reduce an arbitrary nondeterministic Turing machine (NDTM) to a proposition in conjunctive normal form (CNF), so that the CNF term evaluates to true on exactly the inputs that are accepted by the NDTM. Hence, every decision problem in NP can be reduced to a SAT problem.

The SAT problem is considered the simplest of the NP-complete problems. The

SAT problem captures all of the difficulty of any NP-hard problem, and is also very simply stated:

> Given a Boolean proposition in conjunctive normal form, decide whether or not there exists a substitution of values for variables that satisfies the proposition.

No polynomially bounded deterministic algorithm for the SAT problem has been found. Numerous researchers have spent many years working on this problem. A great variety of algorithms have been constructed for satisfiability, but none of them is efficient. The obvious algorithm for SAT is exhaustive search. For a proposition having N variables, exhaustive search requires $O(2^N)$ steps and is obviously inefficient. Much of satisfiability research has concentrated on finding algorithms that are less inefficient than exhaustive search.

A decision problem is in P if there exists a deterministic Turing machine (DTM) that decides every instance of that problem in a polynomially bounded number of steps. The distinction between deterministic and nondeterministic is important. We can construct deterministic computers, but we cannot physically construct nondeterministic computers. A deterministic machine may simulate a nondeterministic machine, but the known methods require an exponential increase in the number of steps.

The question of whether $P = NP$ is open. If SAT can be solved in deterministic polynomial time, then that algorithm would demonstrate $P = NP$. If $P \neq NP$, we cannot expect to find any efficient algorithm for satisfiability. It is widely believed that $P \neq NP$.

This dissertation describes and evaluates some new algorithms that may be useful for satisfiability. We do not provide an efficient algorithm for SAT, or a definitive

solution to the $P = NP$ question. This dissertation only describes and evaluates some alternative algorithms that have not previously been described or evaluated. The new algorithm is less inefficient than existing algorithms for a class of SAT problems. More important, the algorithm demonstrates a class of complete SAT algorithms that can utilize efficient randomized and incomplete search algorithms without sacrificing completeness.

1.1 Relevance And Significance

The satisfiability problem is a fundamental problem in mathematical logic, inference, automated reasoning, machine learning, and discrete optimization. NP problems occur in every branch of Computer Science, and in many applications. Cook (1971) showed that efficient algorithm for the SAT problem would give efficient algorithms for all of the problems in NP.

Garey and Johnson (1979) gives a catalog of NP optimization and decision problems. Hundreds of authors have found other problems that are important for various reasons, and are in NP. Crescenzi and Kann (1995) maintain a current catalog of NP problems and approximation results. The problems include a wide variety of economic problems such as resource allocation, production scheduling, and logistic distribution problems. For many problems in NP, any efficient algorithm would have significant economic and theoretical consequences. Many NP problems are economically important enough that even a partial solution, which solves some useful subset of the practical examples, would be valuable.

One problem of particular interest is found in software engineering and programming languages. The construction of correct programs is a vital issue for software engineering, particularly for safety-critical and mission-critical applications. The problem

of proving that a given program meets a given specification can be viewed as a search problem. Assuming that the program is written in some computable programming language, that it takes only some fixed amount of input, and that it terminates, the problem is to decide whether or not there exists any input for which the program gives incorrect output. It is easy to see that this problem is NP. We may nondeterministically try each possible input to the program, and compare the output to the specified correct output for that input. That this problem is equivalent to a SAT problem follows directly from the result of Cook (1971).

1.2 Barriers and Issues

It is widely believed that $P = NP$, and that no efficient algorithm for SAT can exist. The literature contains a large number of approaches to the SAT problem, none of which is efficient. Various researchers have described and evaluated a number of algorithms for SAT, and have sought to minimize the constants in the complexity of those algorithms. Those algorithms can be grouped into several families of similar algorithms.

The fact that the problem has been studied by a large number of researchers, yet not solved, indicates that the problem is extremely difficult. The large number of approaches that have been tried indicate that it may be difficult to identify a previously untried approach. In chapter 2 we review some of the approaches to SAT that have been tried.

It is clear that straightforward search algorithms cannot solve SAT efficiently. Tree search algorithms use reasoning to avoid searching some some sub-trees, but must still search an exponentially sized portion of the complete tree. Algebraic rewriting cannot solve SAT efficiently, because there can be no compact canonical form. Approaches

for special subclasses of SAT are efficient, but are only applicable to a vanishingly small fraction of all problems. A wide variety of heuristic or randomized search methods have been tried, but cannot provide complete proof of unsatisfiability. Linear relaxations allow the application of integer programming techniques, but the common branch-and-bound techniques for integer programming are essentially just tree-search algorithms.

The problem of finding concise proofs of unsatisfiability is not effectively addressed by any of the known approaches. The complete search algorithms produce proofs of unsatisfiability, but in each case the resulting proof size is exponential in the number of variables. The randomized algorithms that are known for SAT are not complete, so they do not address the problem of finding proofs of unsatisfiability. It is not obvious what other approaches to SAT might be more successful.

Cutting plane algorithms for SAT have not been fully explored, and seem to offer the possibility of allowing concise proofs of unsatisfiability. Cutting plane proofs have the advantage that it is not necessary to divide the problem into multiple subproblems at each step. This advantage may be critical in allowing small proofs of unsatisfiability and low complexity of algorithms to find those proofs. However, existing methods of generating cutting plane proofs tend to require large numbers of cutting plane steps. There are also some theoretical results that some particular cutting plane proof systems do not allow small proofs of some propositions.

1.3 Plan and Approach

In this dissertation, a family of complete algorithms for SAT is constructed by considering necessary conditions. A number of conditions are known to be necessary in the sense that any algorithm not meeting the condition must be inefficient. By

avoiding these known causes of inefficiency, we construct an algorithm that has better average-case performance than previous algorithms for some classes of hard SAT problems.

A complete algorithm for satisfiability requires either finding a witness of satisfiability or finding a proof that no witness exists. Any proof that no witness exists is necessarily based on some system for proving theorems of propositional logic. Different formalizations of propositional logic include different sets of axioms and inference rules, but are equivalent in the sense that they admit proofs of the same theorems. However, the lengths of the proofs is often very different (Cook & Reckhow, 1974, 1979). Clearly, the number of steps required by a satisfiability algorithm to show that some problem is unsatisfiable cannot be less than the number of lines required in the proof.

It is well known that various proof systems based on search and resolution do not allow short proofs of unsatisfiability (Håstad, 1987; Chvátal & Szemerédi, 1988; Urquhart & Fu, 1996). Satisfiability algorithms based on those proof systems cannot be efficient, because the algorithms require a number of steps at least as great as the length of proof. It is a necessary condition that any efficient algorithm must be based on a proof system for which short proofs can exist, or at least for which it is not known that short proofs cannot exist.

A variety of hard satisfiability problems are known, for which particular algorithms are known to require super-polynomial time. The hard part of the satisfiability problem is finding proofs of unsatisfiability, or refutation, for unsatisfiable problems (Franco, Dunn, & Wheeler, 1992). A variety of hard unsatisfiable problems have been proposed. For some of these hard problems, particular algorithms have been found that can solve the particular hard problem efficiently. It is a necessary condition that any efficient algorithm must be able to solve all of these known hard problems

efficiently. For the problems for which algorithms are known, some guidance may be available by examining the structure of the known algorithms.

1.3.1 Cutting Plane Proofs

Every satisfiability problem can be represented as an integer programming problem. Each clause $\bigvee_i x_i \vee \bigvee_j \overline{x_j}$ can be represented as a linear constraint $\sum_i x_i + \sum_j (1 - x_j) \geq 1$. The use of linear inequalities with methods developed for linear integer programming may be viewed as an extended logic. Each method of deriving new valid inequalities may be viewed as an inference rule for the proof system. A sequence of matrix inequalities, where the first is a linear representation of the SAT problem and the rest are derived by valid inference rules, may be considered as a proof. A proof of unsatisfiability based on this type of proof system is called a *cutting-plane proof*. If the last line of a cutting-plane proof is a false inequality, such as $0 \geq 1$, then the premises of that proof are unsatisfiable.

Cutting-plane proofs also possess something like a Church-Rosser property. A set of inequalities in N variables describes a polytope in N dimensions. If we choose a partial ordering of polytopes based on a measure such as the volume of the polytope, the addition of each cutting plane reduces the polytope. The order in which several cutting planes are added does not matter. Any order of additions gives the same reduced polytope. A method of generating cutting planes is called *complete* if it can generate every cutting plane necessary to find the integer convex hull of the problem. If at least one complete family of cutting planes is used, then for any unsatisfiable SAT problem, any cutting plane proof can be extended to an empty polytope.

Cutting plane proof systems are known to be as efficient as general Frege proof systems, and more efficient than constant-depth Frege proof systems (Clote, 1995). In

addition, short cutting plane proofs are known to exist for various problems that are hard for other proof systems (Barth, 1994). Hence, a cutting plane proof system may exist that meets both of our identified necessary conditions. In addition cutting plane proofs can be represented using nice regular data structures, and many algorithms are known for finding cutting planes.

1.3.2 Cutting Plane Algorithms

Cutting plane algorithms for integer programming are similar in many respects to cutting plane proof systems. The problem of finding good cutting planes for a cutting plane algorithm is closely related to the problem of constructing short proofs. Good cutting planes are those that give maximum reduction of some measure of the polytope. Such large reductions often lead to short proofs. Conversely, short proofs require that at least some of the steps correspond to large reductions. A proof of only one step uses a single cutting plane that reduces the measure to zero.

Unfortunately, it is difficult to find good cutting planes for satisfiability problems. The number of individual cutting planes that may be inferred in one step is too large to search completely. The number of basic solutions of a linear program (LP) is combinatorial in the number of variables, and there may exist a large number of distinct cutting planes for each basic solution. Even if we limit the linear combinations of the inequalities to using only a small number of distinct coefficients, the number of possible cutting planes is exponential in the number of inequalities.

Traditional randomized algorithms for SAT have been oriented only to searching for a model to verify satisfiability, and have had some considerable success. Complete algorithms have been viewed as being necessarily deterministic, based on the assumption that they must necessarily search an entire refutation tree. In contrast,

we adopt the view that even complete algorithms for SAT can be viewed as randomized algorithms, in the sense that at each step a number of next steps are possible and one must be chosen. *The order in which a search is conducted, or the choices that are made during a construction, may be interpreted as a random choices.*

A cutting plane algorithm is not just one algorithm. Every cutting plane algorithm requires some method of choosing which vertex of the polytope to cut. Every cutting plane algorithm requires some method of generating cutting planes for a selected vertex, and of choosing which cutting planes to use when several are available.

The LP formulation allows the choice of an objective function. The objective function is usually chosen using some simple heuristic, such as the sum of the linear constraints. However, for satisfiability there is no particular reason to choose any particular objective function. It may be useful to modify the objective function during the computation. *It may be useful to derive a cutting plane from some basic solution of the linear program other than the optimal solution.*

For a given basic solution of the LP relaxation, we want to find the strongest possible cutting plane. The restriction of variables to the domain $\{0, 1\}$ may allow stronger cutting plane techniques than are available for general integer programming. Methods of *lifting* to find stronger cuts have been developed that are applicable when all coefficients of a valid inequality are in $\{0, 1\}$ (Balas & Jeroslow, 1972; Hooker, 1992; Barth, 1995). *It may be possible to develop similar lifting methods to find stronger cuts for the general case, in which the coefficients are not restricted.* Some work along these lines has been reported. For a survey see Balas, Ceria, Cornuéjols, and Natraj (1996). This dissertation presents a new method of lifting, in which non-integral coefficients of a cutting plane are modified by addition of integer quantities.

When several cutting planes are available, the choice of which ones to add to the linear program may be critical. Methods to choose from multiple valid cutting

planes have been suggested, but are often dismissed as too expensive or impractical (Nemhauser & Wolsey, 1988, pp. 374), (Jünger, Reinelt, & Thienel, 1995, pp. 133). A measure of cutting plane strength is clearly needed, but is not sufficient. *An efficient method is needed to evaluate the strength of a cutting plane, independent of any particular objective function.*

When the choice of cutting plane is viewed in combination with the choice of basic solution from which to derive a cutting plane, it seems clear that these two choices may be made together. Hence, it may be useful to search some subset of the possible basic solutions, and some subset of the possible cutting planes for each of those basic solutions, as part of the search for a strong cutting plane. Because the number of possible basic solutions and cutting planes is impossibly large, it is clear that any practical algorithm may examine only a small subset of the possible cutting planes. Hence, *a randomized algorithm may be useful in the search for strong cutting planes.*

1.4 Statement of Hypothesis

In this dissertation, we develop the theory and practice of 0-1 integer programming, with the purpose of extending or specializing that theory for the satisfiability problems. The research was specifically oriented to finding short cutting-plane refutations of hard unsatisfiable SAT problems. The research goal was to find support for or refutation of the following hypothesis:

> *Short cutting-plane proofs of unsatisfiability exist for many hard SAT problems. Algorithms and heuristics for finding strong cutting planes can be used to construct such short refutation proofs.*

The approach involved research toward two theoretical goals, and a series of experiments.

The first goal was to develop methods for finding strong cutting planes, so as to minimize the number of cutting plane steps required to either generate a proof of unsatisfiability or to find a witness to satisfiability. Intuitively, strong cutting planes are those that yield large reductions of some measure of the polytope, and should allow more concise refutation proofs. Algorithms are developed in chapter 3 to update the objective function of the LP relaxation, to measure the strength of a cutting plane, and to search a neighborhood of basic solutions to find a strong cutting plane.

The second goal was to understand the performance characteristics of the resulting algorithms. The key measure is the number of proof steps required in refutation proofs of unsatisfiable SAT problems. Some bound on the number of cutting plane steps in a proof may be possible, but no subexponential bound is presented in this work. Instead, the expected number of cutting planes is predicted, depending on the average strength of the cutting planes. This result indicates the importance of algorithms for finding strong cutting planes.

To quantify the performance of the new randomized algorithms, an experiment is described in chapter 4, and the results of that experiment are reported in chapter 5. The experiment used a computer program in which the various algorithms and heuristics were applied to test problems. The program incorporates algorithms and heuristics that seem promising on a theoretical basis, to see if they work in practice. The experiment used several varieties of hard unsatisfiable test problems to verify that short refutation proofs can indeed be found.

1.5 Summary

The SAT problem is a problem of fundamental importance in computer science, artificial intelligence, and discrete optimization. A large number of algorithms have been developed for SAT, none of which is efficient. A class of algorithms based on linear programming relaxations, using cutting planes, has not been fully explored. Linear inequalities can be viewed as an extension of propositional logic, and cutting planes can be viewed as lemmas for that extended logic. No superpolynomial lower bounds are currently known for SAT algorithms using extended logics, so there remains a possibility that such an extended logic may allow short proofs of unsatisfiability.

In this dissertation, we consider some possible algorithms for SAT that are based on linear programming relaxations, using cutting planes and randomized algorithms. The theoretical goals included developing algorithms for finding strong cutting planes, and minimizing the number of cutting planes required to solve a SAT problem. The research hypothesize asserts that randomized methods of searching for good cutting planes are useful for constructing short cutting plane proofs of unsatisfiability.

Chapter 2

Review of the Literature

This chapter surveys some of the approaches to SAT that have been tried, and summarizes those results. Lack of success in finding an efficient algorithm for SAT has not been due to lack of trying. A large number of researchers have proposed and evaluated a wide variety of algorithms. This chapter is organized into five major sections. Section 2.3 gives a broad review of algorithms and heuristics that have been tried. Section 2.4 reviews methods of linear and integer programming that can be applied to SAT. Section 2.8 reviews the relationship between SAT algorithms and propositional proof systems. These proof systems have been used to show exponential lower-bounds on the complexity of some SAT algorithms. Section 2.9 reviews various definitions of cutting-plane proof systems in detail, and identifies the common features of cutting-plane proof systems. Sections 2.10 and 2.11 review known complexity results on cutting plane proof systems.

Section 2.3 reviews algorithms that have been proposed for SAT. A very great variety of algorithms have been proposed. Complete algorithms are those that can definitively solve a SAT problem. Classes of complete algorithms include tree-search, resolution, and cutting plane algorithms. Incomplete algorithms are those that can

solve some SAT problems quickly, but cannot solve all problems. Classes of incomplete algorithms include those based on rewriting systems and randomized or heuristic search. Some of the ideas in incomplete algorithms may be useful for further research into complete algorithms.

Section 2.4 introduces the approach of reducing a SAT problem to an integer problem and solving it using integer programming methods. Integer programming is a variation of linear programming, in which the optimal solution domain is also constrained to integers. Section 2.5 gives a short tutorial introduction to linear-programming. Section 2.6 gives a short introduction to cutting plane methods integer programming. Cutting plane algorithms for solving integer programs are based on generating additional linear constraints such that non-integral solutions to the linear program are excluded by the added constraints.

In section 2.8, we consider propositional proof systems that provide the mathematical basis for complete SAT algorithms. Each complete algorithm for SAT is based on some proof system for propositional theorems. To decide that a term is unsatisfiable a SAT algorithm must, at least implicitly, construct a proof of unsatisfiability. Any proof that no satisfying solution exists is necessarily based on some system for proving theorems of propositional logic. The shortest proof of a proposition provides a lower-bound on complexity of algorithms based on that propositional proof system. Lower bounds on proof length are known for a variety of propositional proof systems. These lower-bounds proofs show why most SAT algorithms cannot be efficient.

Section 2.9 reviews cutting plane proof systems that have been proposed for SAT. In this section, definitions of cutting-plane proof system will be given in detail. Cutting-plane proof systems are based on cutting-plane algorithms for integer programming, but there is a crucial difference. The cutting plane proof systems that have been proposed do not introduce slack variables, so they keep the inequalities

as inequalities rather than converting them to higher-dimensional equalities. As a consequence of this choice of definition, multiplication is permitted only by positive values to preserve the sense of the inequalities.

Two lower-bounds results (using Craig interpolation formula) that are specifically for cutting-plane systems need to be explained in some detail. These lower bounds use a theorem by Rasborov, that certain functions require exponential size monotone circuits. We will see that these lower bounds depend critically on the definitions of cutting plane proof systems. The several definitions each introduce a critical restriction, which enables the lower bounds proof to work.

The lower bounds results stand in contrast to several positive results on cutting plane algorithms. Some problems that require exponential tree-search and resolution proofs can be proved by polynomial length cutting plane proofs. In particular, problems such as pigeonhole formula that involve *counting* are hard for resolution and easy for cutting planes. This shows that cutting-plane proof systems are stronger than tree-search and resolution.

Other results on cutting-plane proof systems include a bi-simulation result showing that cutting-plane proofs are polynomially-equivalent to Frege proofs, and a paper asserting that the methods used to establish "lower bounds" for other proof systems cannot work for Frege systems. These results seem to contradict the lower bounds results for cutting plane methods. As is common when mathematical results appear to conflict, there is a difference in the definitions.

2.1 Propositional Logic

Numerous volumes with chapters on propositional logic are available, and there are numerous conventional variations of the notation. In this chapter, we will not provide

a full introduction to the theory of Boolean algebra. We will instead provide only a few basic definitions. Early formulations of propositional logic include those by Boole (1854/1958) and (Whitehead & Russell, 1912/1950). Boole demonstrated that it is possible to define a mathematical notation for logic. The volumes by Whitehead and Russell are widely referenced as the first rigorous approach to mathematical logic.

In modern form, various systems of notation are used for propositional logic. For complete information on the development of propositional logic and the various notations that are often used, the reader may consult any of the numerous chapters and monographs on propositional logic and finite set theory (Church, 1956; Suppes, 1960/1972; Stoll, 1961, 1963/1979; Curry, 1963/1977; Quine, 1951; Bernays, 1968/1991; Ershov & Palyutin, 1979/1984; Mendelson, 1987). Deskins (1978) gives a very terse introduction which also includes charts of equivalent symbols. Epstein (1995) gives a thorough introduction to various formalizations of propositional logic, and a fairly rigorous argument that all of the formalizations are equivalent.

A propositional calculus is a term algebra. The terms are constructed recursively in the usual manner. A set of symbols are reserved for use as *operator* symbols to denote propositional functions. Operator symbols are required for a subset of propositional functions that can be used to define all propositional functions. Such a set of symbols is called a *basis* for the propositional algebra. Several basis for propositional algebra are known. The functions *and*, *or*, *not*, *true*, and *false* provide one such basis. The zero-arity functions $true$ and $false$ are also called *constants*.

Table 2.1 lists the correspondence between logic concepts, function names, and the operator symbols that we will use to denote the respective functions. We will use the operator symbols and the function names interchangeably throughout the remainder, as the distinction is usually clear from the context. Following tradition, we will often write the negation as a line above the negated term, as $\overline{x}$, rather than as a prefix.

Logic Concept	Function Name	Operator Symbol	Example
Conjunction	and	$\wedge$	$x_1 \wedge x_2$
Disjunction	or	$\vee$	$x_1 \vee x_2$
Negation	not	$\neg$	$\neg x_1$
False	false	0	0
True	true	1	1

Table 2.1: Logic concepts, function names, and operator symbols.

The set of *propositional variable* symbols is any set of other symbols, excluding the operator symbols. Terms are constructed recursively as follows: A variable symbol is a term. A constant symbol is a term. For every term T, $\overline{T}$ is a term. For every pair of terms T_1 and T_2, $T_1 \wedge T_2$ and $T_1 \vee T_2$ are terms.

Any set of function symbols isomorphic to the set {*and, or, not, false, true*} could be used as the set of operator symbols. Many alternative set of symbols may be found in the literature. It is also customary to use N-arity forms of the two binary operators, which are each associative, commutative, and idempotent. The N-arity terms may alternatively be defined as *abbreviations* for equivalent binary terms. All of these variations of the notation are equivalent, so we may use any convenient notation without loss of generality.

The term $not(T)$ is called the *negation* of T. The term $and(T_1, \ldots, T_n)$ is called the *conjunction* of $T_1, \ldots, T_n$. The term $or(T_1, \ldots, T_n)$ is called the *disjunction* of $T_1, \ldots, T_n$. An *atom* is either a variable or a constant. A *literal* is either an atom or the negation of an atom. A *clause* is either a single literal or a disjunction of literals. A term that is either a single clause or a conjunction of clauses is said to be

in *conjunctive normal form*. Many authors use an equivalent *clausal form*, in which the operator symbols are omitted and the lists of operands are considered as sets.

The *variables* of a term, $V(T)$, is the set of variable symbols that appear in the term. A *ground term* is a term that has no variables. An *interpretation* of a term T is a substitution mapping from the variables of T to constants. Every interpretation of a term is a ground term. The *value* of a ground term is determined by the semantic functions of the operator symbols applied to the values of the operands. The semantic functions are often specified as *truth tables*. An interpretation is called *satisfying* if the value of the term is true under the interpretation.

2.2 The Propositional Satisfiability Problem

The propositional satisfiability problem (SAT) is to decide whether or not there exists a substitution such that a given proposition is true. The converse problem is the tautology problem, which is to decide whether or not there exists any substitution such that a given proposition is false. A proposition P is satisfiable, if and only if the negation $\neg P$ is not a tautology. Satisfiability is a fundamental problem in mathematical logic.

The origin of the propositional satisfiability problem traces to the very beginnings of mathematical logic. Boole (1854/1958) formulated the first algebra of propositional logic, and explored various methods of proving the truth of particular propositions. Boole argues that every proposition can be written in an equational form $V = 0$, where V is a sum of products (ch. X). However, the satisfiability problem is not specifically stated, as Boole was primarily concerned with showing that the algebra can be used to write proofs.

Russell (1902/1938) presents and defends a thesis that mathematics and logic are

identical. This work includes a comprehensive and careful development of mathematical logic, and presents Russell's famous paradox in class theory. In the calculus of classes (pp. 21), Russell identifies that a class can be determined by a proposition, and that the question of whether or not such a class is empty is an important question. This appears to be the earliest explicit statement of the satisfiability problem.

During the next 30 years, the development of mathematical logic concentrates on finding the resolution of Russell's paradox, and attempting to find a completely satisfactory foundation for all of mathematics. Whitehead and Russell (1912/1950) made a significant advance in this direction, giving a formal logical foundation to much of mathematics. Gödel (1931/1992) eventually showed that in every system of mathematics that admits arithmetic, some statements are true but not provable. At about the same time, Church (1936b, 1936a) and Turing (1936–1937) came to the same conclusion by very different routes. Turing's work considered the question of computation, and showed that some numbers are not computable. The computation of a number, in this sense, corresponds to the proof of a statement.

In the meantime, Herbrand (1930/1967) gave the first definite procedure to decide if a proposition is a tautology. This major result for theorem proving is stated as a little remark. Remark 3 states a procedure for deciding the truth of a proposition where each variable takes only a fixed number of alternative values. The truth of such a proposition can be determined by searching the finite set of interpretations. After giving a procedure to decide whether or not a proposition is satisfiable, the mathematical logicians considered the problem solved.

During the 1940's and 1950's, digital computers were developed. As a consequence of this development, some mathematicians were drawn to the problems associated with programming. Algorithms were developed for various problems, and the problem of algorithm efficiency arose. Von Neumann (1948/1986) identified the problem, "The

thing which matters is not only whether it can reach a certain result in a finite number of steps at all but also how many such steps are needed." It was natural to also ask if there might exist efficient algorithms for some problems for which only inefficient algorithms were then known.

Edmonds (1965) proposed polynomial-time computability as a criteria for the separation of good, efficient algorithms from bad, inefficient algorithms. This proposal was immediately taken as a theoretical definition of an efficient algorithm. However, to be practical and useful, an algorithm does not necessarily have to be polynomially bounded. Edmonds observed that an exponentially bounded algorithm with a low-enough exponent could be more useful than a polynomially bounded algorithm with a high-degree polynomial.

Cook (1971) showed that the subgraph, graph-isomorphism, prime-number, tautology, and 3-tautology problems are all solvable by a nondeterministic Turing machine in polynomially bounded time. Further, Cook showed that any decision problem that can be solved by a polynomial time-bounded nondeterministic Turing machine can be reduced to the tautology problem. Cook also conjectures that the tautology/satisfiability problem is not in $L*$ (now known as P), and that it is worth spending considerable effort to prove it. Karp (1972) showed that a great variety of hard problems are equivalent to satisfiability, and introduced the terms P and NP.

Garey and Johnson (1979) gave a catalog of NP problems, with citations to reduction results, that included more than 300 problems. Many of those problems are economic or managerial problems, such as integer programming, combinatorial optimization, and machinery scheduling. Others arise from the design of computing machinery and software, including the design of networks, the design of storage and retrieval systems, checking program equivalence and correctness, and program optimization. Every one of these important problems can be reduced to a polynomial

number of satisfiability problems.

Hence, the satisfiability problem is important for two reasons. First, from a theoretical point of view, a major open question is whether or not there exists any polynomially bounded algorithm for satisfiability. Second, a large number of efficient algorithms for significant economic and theoretical problems could be derived from any efficient algorithm for satisfiability. While it would be theoretically interesting to show that there can be no polynomially bounded algorithm for SAT, it would be much more profitable to find an efficient algorithm.

2.3 Algorithms for SAT

This section reviews some of the approaches to SAT that have been tried. The various algorithms that have been proposed for SAT fall roughly into several classes. Tree search algorithms split the problem into multiple subproblems. Algebraic rewriting attempts to solve the problem by manipulating formulas according to some Markov algorithm. Special subclasses of SAT have been investigated by a number of researchers. A wide variety of heuristic or randomized algorithms have been proposed. Finally, linear relaxations transform the problem into a linear programming problem.

2.3.1 Search algorithms

Various search algorithms can be categorized as recursive tree-search, resolution, and connections methods. A large number of variations and refinements on these basic algorithms have been proposed. Surveys of search algorithms for propositional theorem proving include those by Gu, Purdom, Franco, and Wah (1997), Gabbay (1992), Bibel and Eder (1993) and Eisinger and Ohlbach (1993).

Resolution Search Algorithms

Resolution algorithms use an inference rule that also effectively searches all possible assignments, but is implemented differently (Davis & Putnam, 1960; Robinson, 1965). Resolution algorithms use the inference rule $(x \vee y) \wedge (\overline{x} \vee z) \vdash (y \vee z)$ to construct a new clause without a variable x. If all copies of a selected variable are eliminated in all possible ways, the resulting term is equivalent to the original term and has one less variable. For a CNF term $f(x_1, \ldots, x_n)$, if resolution is used to eliminate x_n, then the resulting term is just the conjunctive normal form of $f(x_1, \ldots, x_{n-1}, 0) \vee f(x_1, \ldots, x_{n-1}, 1)$.

Most implementations of resolution do not eliminate all copies of a variable at once. Instead, they search a tree of resolution operations to find an empty clause. Starting from any clause, all possible resolution inferences involving that clause as an antecedent are tried. If an empty clause can be inferred by resolution, then the original term is unsatisfiable. The size of the search tree is of course exponential in the number of clauses. Galil showed that every resolution procedure has exponential complexity, because in many cases the shortest refutation requires an exponential number of resolution steps (Galil, 1974, 1975b, 1975a).

Model Search Algorithms

The basic approach of model search algorithms is to try all possible assignments of constant values for each variable in the term. The possibility of searching all possible assignments for algebras in finite-domain was identified first by Herbrand (1930/1967). Herbrand showed that it is possible to enumerate all interpretations of a term by enumerating the cross-product of the interpretations of the variables that appear in the term. In propositional logic, there are exactly two possible values

for each variable, corresponding to "true" and "false". For Boolean expressions of n variables, there are 2^n such assignments. Enumeration is obviously not a practical approach for problems with large numbers of variables.

The first computer program to search all possible assignments is due to Davis, Logemann, and Loveland (1962). The algorithm uses *unit clause* and *pure literal* rules as heuristics for choosing a variable. The algorithm assigns the value *true* to any literal that appears by itself as a clause. When no such variable is available it chooses one from the shortest non-unit clause. The chosen variable is added to the current formula as a new unit clause, and the algorithm recurses. If the guess leads to an unsatisfiable subproblem, the guess is reversed and the other subproblem is tried. If both subproblems are unsatisfiable, then the problem is reported unsatisfiable. The complexity of this algorithm is exponential in the number of variables. Some simple algebraic reductions based on the pure-literal and subsumption rules allow some branches of the tree to be avoided, but that does not significantly reduce the complexity of the search.

Recently, much of SAT research has had the goal of reducing the constants or the exponent in the exponential complexity of tree-search algorithms (Kullmann, 1998; Freeman, 1995; Wang, 1997; Zhang, 1997). These approaches have had some success. Kullmann's algorithm currently has the lowest exponent among the tree-search algorithms, with complexity approximately $O(2^{.5893N})$ for 3-SAT. However, the number of steps is still exponential in the number of variables. The size of problems that can be solved by these methods is still severely restricted.

Connection Search Algorithms

Connection methods also search all possible assignments, but are again implemented differently. Connection calculi is presented in a rectangular grid of literals in which

the columns represent the clauses of the term. A proof in a connection calculi is an arrangement of arcs between the literals such that the connected graph spans the entire grid, and each arc connects two complementary literals. The connections between complementary literals represent applications of the resolution rule. Bibel and Eder (1993) gives a good presentation of several variations of connection calculi.

The advantage of the connection methods is the ease of automating the search, as compared to the methods using traditional formula to represent the term. The difficulty with the connection methods is the large number of possible proofs that must be searched. The number of possible arcs in a matrix of M clauses with N variables is at only MN, so the size of a finished proof is small. However, the number of possible sets of arcs that must be searched to find a proofs is 2^{MN}. Hence, connection methods are exponentially less efficient than even resolution methods.

Summary

None of the search algorithms is efficient. In each case, the difficulty arises from either the size of the search tree or the size of the intermediate terms. Håstad (1987) showed that functions that can be expressed by polynomially bounded circuits at each depth $N > 1$ require super-polynomial circuits at depth $N - 1$, even if some variables are randomly restricted to constants. Håstad's result directly implies that satisfiability algorithms based on resolution and/or model search cannot be efficient. We will review the lower-bounds results in more detail in section 2.8.

2.3.2 Special Cases

Some efficient algorithms have been found for various specific subclasses of SAT (Dantsin, 1997; Truemper, 1998). The review in Truemper (1998, Ch. 5) is both

recent and very thorough. The subclasses for which polynomial algorithms are known include 2-CNF, Horn, Balanced, and some others. These classes of problems include some common problems. For example, Horn satisfiability problems include all (pure) prolog programs. However, only a vanishingly small proportion of satisfiability problems possess such special structures (Franco, 1997).

2-CNF Formula

The class of 2-CNF propositions includes CNF formula in which each clause contains at most two literals. The satisfiability of 2-CNF formula can be solved in polynomial time, using an algorithm given by Evan, Itai, and Shamir (1976).

Evans algorithm is very simple, and consists essentially of applying resolution. The satisfiability of every 2-CNF proposition can be solved in polynomial time even by resolution. There are fewer than $4N^2$ distinct clauses of at most two literals in N variables. A resolvent of two such clauses gives another such clause, so that set of clauses is closed under resolution. Hence, at most $4N^2$ resolution steps are required to generate all possible clauses, and then to inspect the resulting set of clauses to see if it contains an empty clause.

Horn Formula

The class of Horn propositions includes CNF formula in which each clause contains at most one positive literal. A second class of propositions, which can be converted to Horn by reversing the signs of all variables, includes CNF formula in which each clause contains at most one negated literal. Satisfiability of Horn formula can be solved in polynomial time using an algorithm given by Dowling and Gallier (1984). That algorithm, and variants of it, provide the basis for the logic programming language Prolog.

The class of hidden-Horn propositions include CNF formula which can be converted to Horn by reversing the signs of some variables. An algorithm due to Lewis (1978) recognizes hidden Horn formula. Several algorithms are known to determine the subset of variables to reverse (Truemper, 1998).

Balanced Formula

Propositional terms can be represented by matrices. Each row represents a clause, and each column represents a variable. If a positive (resp. negative) literal appears in a clause, the corresponding matrix entry is 1 (resp. 0). A matrix is a cycle matrix if it is connected, and has exactly two nonzero values in each row and in each column. A cycle matrix is balanced if the sum of its entries is divisible by 4. A general balanced matrices is a matrix in which every cycle submatrix is balanced. Every linear solution of balanced matrices is integral (Chvátal, 1983; Schrijver, 1986). Hence, if a proposition is represented by a balanced matrix, then linear programming methods can be used to solve the satisfiability problem efficiently.

Single-Solution Formula

Dantsin (1997) defines the all-0 (resp. all-1) class of SAT problems includes problems that are satisfied by setting all variables to value 0 (resp. 1). For these classes of problems, the algorithm consists of simply trying the value to see if it satisfies the problem. Of course, there are 2^N such classes of special problems, corresponding to 2^N possible assignments. While testing membership in any polynomial number of these classes can be done in polynomial time, testing membership in all such classes clearly requires exponential time.

Size of Special Subclasses

Franco (1997) found that only a vanishingly small proportion of satisfiability problems possess such special structures. For large problems, the probability that a randomly selected problem has special structure goes to zero in the limit. Franco formalizes this result in a theorem:

Theorem 1 (Franco 1977). *For any fixed $r > 4/(k(k-1))$, the probability that a random formula is SLUR, q-Horn, extended Horn, CC-Balanced, or renamable Horn tends to* 0 *as* $n \to \infty$.

The parameter k is the number of literals per clause. The parameter r is the ratio M/N, where formulas have M clauses in N variables. For 3-SAT, we have $4/(k(k-1)) = 2/3$. Hence, for $M > 2N/3$, the theorem applies. SAT problems with $M \leq N$ are trivially easy, and can be solved by a greedy method. The range of problems with $M > 2N/3$ includes all of the interesting 3-SAT problems.

Decomposition Method

Truemper (1998) gives methods that detect sub-problems with special structure, and decompose a larger problem to take advantage of the easily solved subproblems. The decomposition technique allows some problems that do not have special structures to be solved efficiently, if they have subproblems with special structure.

Even for problems that do possess easy subproblems, the problem of finding good decompositions is a search problem. The problem of finding the best decomposition is NP-hard. Truempers method uses several integer programming problems to find decompositions. Hence, an NP problem is used as a subproblem in a method to solve an NP problem. The method does not actually require the best decomposition, and so it can use approximate solutions to these subproblems.

Franco's (1997) results on the frequency of problems with special structure apply to subproblems as well as to whole problems. Hence, only a vanishingly small proportion of subproblems have special structure. Truemper asserts that many important real problems have special subproblems and can be solved by his method. In any case, Truemper's decomposition method is not a general algorithm, and cannot solve all SAT problems efficiently.

Summary

Polynomial algorithms are known for some limited subclasses of satisfiability problems that have special structure. These classes of problems include some common problems. Some problems can be decomposed into subproblems that can be solved efficiently.

Such special cases account for only a vanishingly small proportion of satisfiability problems. For random SAT problems, the probability that the problem can be solved by the known efficient methods tends to zero as the size of the problem increases. In the limit, the probability that any of these methods applies is essentially zero.

2.3.3 Algebraic Rewriting

Various algebraic approaches have also been attempted, generally grouped under the name *rewriting systems*. In rewriting systems, rules are applied to reduce an algebraic term to a unique minimal form relative to some term ordering. Le Chenadec (1986) gives a concise catalog of rewriting results. Klop (1992) and Dershowitz and Jouannaud (1994) each give detailed treatments, with references to more recent results.

The earliest formal presentations of rewriting rules as processes of computation are due to (Church, 1941). Rewriting is central to the λ-calculus and functional program-

ming. Church's lambda calculus is based on rewriting rules known as α-conversion and β-conversion. Church and Rosser (1936) shows that λ-calculus is consistent by showing the consistency of α and β conversion. The Church-Rosser theorem shows that β-reduction allows exactly one normal form for every λ-expression. This Church-Rosser property can be used to decide if two λ-expressions are β-equivalent.

Barendregt (1984) provides a full exposition of the theory of λ-calculus and the Church-Rosser theorems. Hankin (1993) provides an accessible introduction. Salomaa (1973) presents definitions of rewriting systems, and a definition of Turing Machine in terms of rewriting systems. This simulation of an arbitrary Turing Machine by a rewriting system shows that rewriting systems can simulate Turing machines.

The first explicit procedures for computing in term algebras using generators as rewriting rules is due to Evans (1951). In rewriting, a few simple "rules" are applied to a single copy of the problem, the goal being to maintain the value of the expression, while reducing it to some canonical form. Dershowitz and Jouannaud (1994) is a recent introduction to term rewriting. The notion of rewriting has been widely applied to numerous algebraic systems other than λ-calculus, and many results have been published (Jouannaud, 1987/1987; Lescanne, 1987; Dershowitz, 1989; Book, 1991, 1993; Hsiang, 1995; Ganzinger, 1996; Comon & Jouannaud, 1993; Heering, Mienke, Möller, & Nipkow, 1993).

Confluence

A confluent term rewriting system is a rewrite system that solves the word problem in an algebra by reducing terms that are semantically equivalent to terms that are syntactically identical. The minimal form of a term is called a canonical form. Two terms that are equal for all Herbrand interpretations are said to be equivalent, or to be in the same equivalence class. Within each equivalence class, we must have

some criteria to select which equivalent term is the canonical form of the equivalence class. This ordering must have certain properties to be useful. In particular, every term must be greater than any of its subterms, and the ordering must be stable under substitution for variables. Orderings that satisfy these properties are called simplification orderings (Dershowitz, 1987/1987).

The Church-Rosser theorem (Church & Rosser, 1936) asserts that there exists a unique minimal normal form for certain systems of relations. When a rewrite system is confluent, the rules may be applied in any order. For a given starting term, there is exactly one irreducible normal form.

Definition 1 (Confluence). *A rewriting system R said to be confluent iff For all terms M, M_1, M_2 such that $M \rightarrow_R^* M_1$ and $M \rightarrow_R^* M_2$, there exists a term P such that $M_1 \rightarrow_R^* P$ and $M_2 \rightarrow_R^* P$.*

The confluence property is a global property, and is difficult to check mechanically because each rewriting may require multiple steps. There are too many combinations to check. The property of local confluence is related to confluence by restricting the divergent rewriting to a single application of a single rule.

Definition 2 (Local Confluence). *A rewriting system R said to be confluent iff For all terms M, M_1, M_2 such that $M \rightarrow_R M_1$ and $M \rightarrow_R M_2$, there exists a term P such that $M_1 \rightarrow_R^* P$ and $M_2 \rightarrow_R^* P$.*

Newman's lemma (Newman, 1942) allows us to check local confluence rather than global confluence. Local confluence is much less difficult to check mechanically, because there are fewer combinations to be checked.

Lemma 1 (Newman's Lemma). *Let R be a noetherian rewriting system, then: R is confluent iff R is locally confluent.*

Completion

Knuth and Bendix (1970) observed that any nonconfluence must reduce to some elementary divergence between rules. Such a divergence results when two rules may each rewrite the same term. The two rules need not rewrite the same occurrence of the term. It is enough that they rewrite overlapping occurrences. These divergent points are called critical pairs.

The Knuth-Bendix theorem gives an algorithm to test the local confluence of a list of rules via the critical pairs. For any pair of rules, we can detect the critical pairs. To show the local confluence of the list of rules, it suffices to show the confluence of the critical pair members. By Newman's lemma, local confluence implies global confluence. So the Knuth-Bendix theorem and Newman's lemma give an algorithm to test the global confluence of a list of rules.

When a list of rules is not locally confluent, there exists at least one non-confluent critical pair between two rules. The essential part of the Knuth-Bendix algorithm is that the divergence can be eliminated by adding a new rule to the rewriting system. Each critical pair may potentially generate a new rule. If a critical pair is resolved by application of the existing rewriting rules to both sides, then no new rule is necessary. If a critical pair is not resolved, a new rule is necessary.

Each new rule is created by orienting the reduced critical pair. The rule is oriented according to a reduction ordering to assure that the rewriting system will remain Noetherian. If the rule cannot be oriented, then the Noetherian induction would fail to show termination. The completion algorithm may halt when an equation cannot be oriented, loop indefinitely, or stop in success when all critical pairs are resolved.

Negative Results for Rewriting

Convergent rewrite systems exist that reduce Boolean terms to canonical forms, but the resulting terms have super-polynomial term size (Hsiang & Huang, 1997). Hsiang's canonical form is based on the "and" and the "exclusive-or" operators. Unfortunately, term rewriting cannot reduce terms of propositional algebra to any compact canonical form, because the needed term ordering cannot exist (Quine, 1952; Freese, Jezek, & Nation, 1993).

Hsiang (1983) found a convergent rewriting system for Boolean rings. Hsiang's canonical form is based on the "and" and the "exclusive-or" operators. The rules are arranged so that they generates an exhaustive list of every possible satisfying assignment, in the form of an exclusive-or of (potentially very many) conjunctions of literals. The calculation of Hsiang's canonical form is not an efficient algorithm, because the size of the term increases exponentially with the number of variables. Le Chenadec (1986) reports that (Fages, 1983) extended Hsiang's result to Boolean algebras by adding rules to eliminate the other operators. The result of Fages' system is also in exclusive-or form, so it is also not efficient.

Unfortunately, term rewriting cannot reduce Boolean terms to any compact canonical form, because the needed term ordering cannot exist (Quine, 1952; Freese et al., 1993). Quine showed that there is no complete set of reductions for free Boolean algebras to conjunctive normal form, because there is no unique minimal set of prime implicants. This follows from a demonstration that one Boolean function may be represented in CNF by multiple different irreducible terms, which are identical except for a renaming of the variables. Because the terms are identical up to a renaming of variables, Quine asserts that there is no reasonable criteria to select one over the other as the canonical form of that function.

Quine's result implies that if the terms are kept in conjunctive normal form, then the word problem for propositional logic cannot be solved by any finite rewriting system. However, this does not eliminate the possibility that some other normal form might be found. Freese et al. (1993) showed that there is no other normal form that can work, and so no rewriting system can solve the propositional word problem. Freese formalizes this result with a proof that there is no finite, convergent term rewriting system for the equational theory of lattices. This does not conflict with Hsiang's result, because the theory of lattices does not include the exclusive-or operator. This result shows that rewriting systems by themselves cannot efficiently solve the SAT problem.

2.3.4 Randomized Algorithms

Various heuristic search algorithms for propositional satisfiability have been proposed and tested. Much of artificial intelligence research has focused on developing efficient heuristics for searching combinatorial structures. A variety of search heuristics are surveyed by Aarts and Lenstra (1997) and Hochbaum (1995). The goal of search heuristics is to avoid the exhaustive search, so that the computation may be finished in feasible time. Examples of heuristic search algorithms include hill climbing, simulated annealing and genetic algorithms.

Randomized algorithms are algorithms that receive as input both the problem to be solved and a source of random bits that are used to make decisions (Motwani & Raghavan, 1995). Hill climbing with random starting points, neural networks with random starting states, simulated annealing with random state changes, and genetic algorithms with random initial genes are all examples of randomized algorithms. The theory of randomized search is based in probability theory, using tail inequalities and

Chernoff bounds to assert high probability of finding a solution if one exists (Motwani & Raghavan, 1995).

Randomized search algorithms for SAT are incomplete, because the random search does not cover the entire Herbrand universe of the problem. Each incomplete search algorithm has some probability of finding a satisfying solution, or witness, if the function is satisfiable. However, that probability is less than one. Each incomplete search algorithm has some nonzero probability of failing to find a witness when one exists. Hence, none of these algorithms can prove that a SAT problem is unsatisfiable.

Other randomized algorithms have the goal of finding approximate solutions for optimization problems. These are useful for some decision problems, if the decision problem can be posed as an optimization problem. However, for SAT, the problem of finding an approximate solution is as hard as that of finding exact solutions (Arora, Lund, Motwani, Sudan, & Szegedy, 1998; Håstad, 1997).

Randomized Algorithms for SAT

A number of randomized algorithms have been specifically designed for SAT. Each of the randomized algorithms is incomplete. Most are concerned with direct search for satisfying solutions, and are not capable of finding a proof of unsatisfiability. For problems which possess many solutions, the direct search for a witness is successful with high probability. Heuristics to guide that search can increase the probability somewhat, or reduce the expected length of the search. One algorithm, by Løkkentangen and Glover (1997), describes a method of using a heuristic algorithm to control parameters of another randomized method.

A large number of random search algorithms have been proposed. In each of these algorithms, multiple random starting points are used, and some random decisions occur at various points in the search. Franco and Ho (1986) considered an algorithm

that simply tries random truth assignments until it finds a satisfying truth assignment. The GSAT algorithm is a hill-climbing algorithm that proceeds by randomly flipping any variable that results in greatest decrease in the number of unsatisfied clauses (Selman, Levesque, & Mitchell, 1992). The WSAT algorithm is a variant on the GSAT algorithm that with probability p picks some random variable in some unsatisfied clause, and with probability $1 - p$ follows the GSAT rule (Selman, Kautz, & Cohen, 1996). The SASAT algorithm is a similar hill-climbing algorithm, using a simulated-annealing schedule for several control parameters (Spears, 1996). The GRASP algorithm is a greedy randomized hill climbing algorithm, which adjusts some parameters after each local search (Resende & Feo, 1996). The RandomUC algorithm uses a local search in which the variables are ordered, and values are flipped when a unit-clause is found under a partial substitution of the variables in a prefix of that ordering (Zane, 1998). Zane also adds a preprocessing step that uses resolution to expand the set of clauses, thereby increasing the probability that unit clauses occur during the search.

All of these search algorithms quickly find satisfying solutions for SAT problems that have large numbers of satisfying assignments. All of these algorithms require exponential time for problems that have only one satisfying assignments. None of the these algorithms is complete. No randomized search algorithm can determine that a problem is unsatisfiable. For each of the randomized search algorithms, there remains a possibility that the algorithm will fail to find a satisfying assignment that does exist.

Løkkentangen and Glover (1997) describe a method of controlling randomized algorithms, based on surrogate constraints. Surrogate constraints are extended clauses or sums of clauses. A surrogate constraint is a linear combination of simple constraints, similar to the valid inequalities of linear programming. Løkkentangen and

Glover use surrogate constraints to guide heuristic search. Initially, a vector of weights w is chosen. The linear combination $w^T Ax \geq w^T b$ of the problem constraints $Ax \geq b$ controls the probabilities that are used in a randomized algorithm. The randomized algorithm uses the probabilities to choose the variable to be assigned at each node of the search tree, or to choose starting points for hill-climbing. After each iteration, the index set of the violated constraints is used to change the weights w, giving new probabilities.

Summary

Various randomized algorithms for propositional satisfiability have been proposed and tested for SAT. We have listed a number of these randomized algorithms. None of these is complete as a search algorithm. Løkkentangen and Glover (1997) observed that randomized algorithms can be used to guide or control other algorithms.

2.3.5 Cutting Plane Algorithms

The satisfiability problem can be represented as a general integer programming problem. Each clause $\bigvee_i x_i \vee \bigvee_j \overline{x_j}$ can be represented as a linear constraint $\sum_i x_i + \sum_j (1 - x_j) \geq 1$. Using this representation, methods developed for integer programming can be applied to satisfiability. Linear relaxation is a method for integer programming that involves relaxing the domain constraint on the discrete variables. Popular texts on linear and integer programming include those by Wolsey (1998), Nemhauser and Wolsey (1988), Schrijver (1986), Chvátal (1983), Dantzig (1963), and Dorfman, Samuelson, and Solow (1958/1987).

Two general classes of integer programming algorithms are called branch-and-bound and cutting-plane. Branch-and-bound algorithms are tree-search algorithms,

and so have exponential complexity due to the branching of the search tree. Cutting plane algorithms are based on the derivation of valid inequalities that cut off non-integral extrema of the feasible linear polytope (Chvátal, 1983; Gomory, 1963). Hybrid algorithms are possible. Some branch-and-bound algorithms use weak cutting plane algorithms to fathom nodes and prune the search tree (Joy, Mitchell, & Borchers, 1997; Jünger et al., 1995).

After forming the linear relaxation of the SAT problem and choosing an objective function, the linear relaxation can be treated as a linear program. Linear programs can be solved by various algorithms. The most popular algorithm is the simplex algorithm, based on pivot operations (Dantzig, 1963). In the pivoting algorithms for linear programming, row operations similar to Gaussian elimination are used to find the optimal basis.

Cutting plane algorithms for integer programming iteratively solve the linear relaxation to optimality, add a valid inequality as a new constraint, and repeat until either an integral solution is found or no feasible linear solution exists. Valid inequalities are generated so as to eliminate non-integral solutions of the linear program, while retaining integral solutions. Gomory (1963) gave the first cutting-plane algorithm and showed finite termination of that algorithm.

2.4 Reduction from SAT to Integer Program

Linear programming was developed in the 1950's to solve problems of operations research (Dantzig, 1963). The initial problems involved military logistics and network design, but the methods were quickly found to be applicable to a wide variety of economic problems. Popular texts on linear programming include those by Chvátal (1983), and Dantzig (1963).

In this section, we show the reduction from SAT to Integer Programming. In sections 2.5 and 2.6, we will discuss methods that have been proposed to solve integer programming problems.

2.4.1 The Integer Programming Problem

A linear programming problem consists of a linear objective function to be minimized (resp. maximized) and a set of linear inequalities that constrain the feasible solutions.

$$\begin{aligned} &\text{maximize: } C^T X \\ &\text{such that: } AX \leq B \\ &X \geq 0 \end{aligned} \tag{2.1}$$

where C is an N-vector of constants, A is an $M \times N$ matrix of constants, B is an M-vector of constants, and X is an N vector of variables. The linear $C^T X$ is the objective function. The vector inequality $AX \leq B$ expresses a set of linear constraints on the permissible values of X. $X \geq 0$ expresses that all variables must be non-negative. It is common to omit the explicit statement of the non-negativity constraints.

It is customary to assume that the coefficients in a linear or integer programming problem are integers. In the case where rational numbers are given, the entire array can be multiplied by the least common multiple of the divisors. In the case where the constants are irrational, rational approximations are used. This assumption allows the arithmetic required by various algorithms to be performed easily.

The integer programming problem expresses the additional constraint that the

solution values must be integers.

$$\begin{aligned} \text{maximize: } & C^T X \\ \text{such that: } & AX \leq B \\ & X \geq 0 \\ & X \text{ integer} \end{aligned} \tag{2.2}$$

2.4.2 Writing a SAT as an Integer Program

The satisfiability problem can be represented as a general integer programming problem (Karp, 1972; Garey & Johnson, 1979). Each Boolean truth value can be encoded as an integer, with 0 representing *false* and 1 representing *true*. Using this encoding of truth values, each conjunction of the satisfiability problem can be represented as a linear inequality. Suppose we have a conjunction:

$$\bigvee_i x_i \quad \vee \quad \bigvee_j \overline{x_j} \tag{2.3}$$

We can replace the conjunction by a linear inequality:

$$\sum_i x_i + \sum_j (1 - x_j) \geq 1 \tag{2.4}$$

It is clear that the set of $x_i \in \{0, 1\}$ that satisfy the inequality (2.4) encodes the set of truth values that satisfy the clause (2.3). Multiplying by -1 and moving the constants to the right-hand side allows us to rewrite the $\geq$ inequality as a $\leq$ inequality, suitable for use as a constraint of an integer program.

$$\sum_i -x_i + \sum_j x_j \leq |j| - 1 \tag{2.5}$$

Repeating the transformation for each clause in the SAT problem gives a set of inequalities for use as constraints of the integer program.

To finish constructing the integer program, we need additional constraints to enforce that the variables may take only the values that represent truth values.

$$\begin{aligned} x_i &\leq 1 \\ x_i &\geq 0 \\ X_i &\text{ integer} \end{aligned} \tag{2.6}$$

Finally, we must choose an objective function, and state all of the inequalities. The satisfiability problem does not require any particular objective function, so we may freely choose the coefficient vector C. Suppose we choose to maximize the objective function $C^T X$. Finally, the SAT problem may be written as an integer program:

$$\begin{aligned} &\text{maximize: } C^T X \\ &\text{such that: } \sum_i -x_i + \sum_j x_j \leq |j| - 1, \\ &\qquad\qquad \text{for each clause } \bigvee_i x_i \quad \vee \quad \bigvee_j \overline{x_j} \\ &x_i \leq 1, \text{ for each variable } x_i \\ &-x_i \leq 0, \text{ for each variable } x_i \\ &X_i \text{ integer} \end{aligned} \tag{2.7}$$

2.4.3 Writing a SAT as a Hitting-Set IP

There are various other methods of representing a SAT as an integer programming problem. One method that is often used is to first reduce the SAT problem to a hitting set problem, then represent the hitting-set problem as an integer program (Karp, 1972; Garey & Johnson, 1979).

The hitting-set problem is given a finite set S, a set C of subsets of S, and an integer $K \leq |C|$. The hitting-set problem is feasible if and only if there is a subset $S' \subseteq S$ with $|S'| \leq k$ such that S' contains at least one element from each subset in C.

The key to the reduction from SAT to hitting-set is to define a new variable to represent the negation of each original variable. Let $z_i = \overline{x_i}$ for each i. Using the convention of 0 representing *false* and 1 representing *true*, we have $z_i = 1 - x_i$. To write the domain constraints, we require that for each i, at least one of x_i or z_i must be true.

$$\begin{aligned} x_i + z_i &\geq 1 \\ x_i, z_i &\in \{0, 1\} \end{aligned} \tag{2.8}$$

In the hitting set interpretation, these constraints directly encode the existence of a set in C containing the elements x_i and z_i, for each i.

We then write a constraint for each clause in the SAT problem. Suppose the SAT problem has a conjunction:

$$\bigvee_i x_i \quad \vee \quad \bigvee_j \overline{x_j} \tag{2.9}$$

We replace the conjunction by a linear inequality:

$$\sum_i x_i + \sum_j z_j \geq 1 \tag{2.10}$$

It is clear that the set of $x_i, z_i \in \{0, 1\}$ that satisfy the inequality (2.10) encodes the set of truth values that satisfy the clause (2.9). For each clause, the constraint (2.10) encodes the existence of a set in C containing the elements x_i and z_j. Repeating the transformation for each clause in the SAT problem gives a set of inequalities for use as constraints of the integer program.

The domain constraints to enforce that the variables may take only the values that represent truth values are not directly represented in the hitting set formulation. It would be possible for some i to have both $x_i = 1$ and $z_i = 1$. The hitting-set formulation uses the limit value k to limit the number of x_i and z_i that may be true. The IP formulation of the hitting-set problem uses the objective function to find a

feasible solution with the minimum sum of all of the variables. The complete integer program is then:

$$\begin{aligned} \text{minimize: } & \sum_i x_i + \sum_i z_i \\ \text{such that: } & \sum_i x_i + \sum_j z_j \geq 1 \text{ , for each clause } \bigvee_i x_i \quad \vee \quad \bigvee_j \overline{x_j} \\ & x_i + z_i \geq 1 \text{ , for each variable } x_i \end{aligned} \tag{2.11}$$

The hitting set problem is solved affirmatively if there exists a feasible solution to the integer program with objective value less than or equal to k. The SAT problem in N variables is satisfiable if and only if there is a feasible solution to the integer program (2.11) with objective value equal to N. For each variable x_i, at least one of x_i or z_i must be 1 in order to satisfy the constraint $x_i + z_i \geq 1$. If for each i, only one of x_i or z_i is nonzero, then the objective function has value N. If both x_i and z_i must be 1 for some i to hit all of the clauses, then the objective value will be greater than N.

The formulation of SAT as a set-cover problem, and the translation of the set-cover problem as an integer program, is almost identical to the formulation as a hitting set problem. The difference is that the set cover problem does not explicitly limit the number of variables that may be true. The SAT problem in N variables is satisfiable if and only if there is a feasible solution to the integer program (2.11) with objective value equal to N.

2.5 Algorithms for Linear Programming

In this section, we give a very terse explanation of an algorithm for solving linear programs, similar to the simplex algorithm due to Dantzig (1963). Other variations of the simplex algorithm exist (Chvátal, 1983). Also, other algorithms called interior-point

methods exist for linear programming (Schrijver, 1986). For the present purpose, it is enough to outline a simple algorithm. This material has been available in textbook form for many years. Popular texts on linear programming include Schrijver (1986), Chvátal (1983), Dantzig (1963), and Dorfman et al. (1958/1987).

Suppose we are given a linear programming problem:

$$\begin{aligned} &\text{maximize: } C^T X \\ &\text{such that: } AX \leq B \\ &X \geq 0 \end{aligned} \tag{2.12}$$

The linear function $C^T X$ is referred to as the *objective function*. The matrix inequality $AX \leq B$ represents a set of linear inequalities, $\sum_j a_{ij} X_j \leq B_i$, for each j. These linear inequalities are referred to as the *constraints*.

In the simplex approach to linear programming, the system of inequalities is first converted to an equivalent system of linear equalities, and Gaussian elimination is used to solve the system of linear equalities. Because the system of linear equalities is under-constrained, it has multiple solutions. Dantzig's major insight was that one of these basic solutions is an optimal solution to the linear program. While other solutions may exist, every optimal solution to the linear program is a basic solution to the system of equalities.

2.5.1 Equality Conversion

The first step in solving a linear programming problem is to state it as a system of linear equalities. This is accomplished in two steps: the objective function is stated as an equality by defining a new variable equal to the objective value; and the constraint inequalities are converted to equalities by adding a new slack variable S_i to each inequality.

We first consider the objective function, $C^T X$. To state the objective function as an equality, we define a new variable Z as the value of the objective function. Then we have:

$$Z - C^T X = 0 \tag{2.13}$$

We then consider the inequality constraints. Suppose we have an inequality:

$$A_i X \leq B_i \tag{2.14}$$

where A_i is the i-th row of A, and B_i is the i-th element of B. To convert the inequality (2.14) to an equality, we add a new *slack* variable S_i. The slack variable is a new variable that does not otherwise appear in the problem. The slack variable is also constrained to be non-negative. The inequality can then be rewritten as:

$$A_i X + S_i = B_i \tag{2.15}$$

The addition of a slack variable lifts the inequality to a higher-dimensional vector space. There is a distinct slack variable for each inequality, so each inequality is converted to a corresponding equality. For $S_i \geq 0$, the set of X that satisfy the inequality (2.14) is identical to the set of X that satisfy the equality (2.15).

$$\{X : A_i X \leq B_i, X \geq 0\} = \{X : A_i X + S_i = B_i, X \geq 0, S_i \geq 0\} \tag{2.16}$$

After converting the objective function and each inequality constraint to equalities, we have the linear program (2.12) converted to the system of linear equalities:

$$\begin{aligned} & Z - C^T X = 0 \\ & A_i X + S_i = B_i, \text{ for } i = 1, \ldots, M \\ & X \geq 0 \end{aligned} \tag{2.17}$$

Hence, the linear programming problem is converted into finding the solution of the equality constraints (2.17) that gives the greatest possible value for the objective variable Z. This can be expressed:

$$\begin{aligned} &\text{maximize: } Z \\ &\text{such that: } Z - C^T X = 0 \\ &\qquad A_i X + S_i = B_i, \text{ for } i = 1, \ldots, M \\ &\qquad X \geq 0 \\ &\qquad S_i \geq 0 \end{aligned} \tag{2.18}$$

The constraints $Z - C^T X = 0$ and $A_i X + S_i = B_i$ can be combined into a single vector equation by renaming of the variables. Let $X_{i+N} \equiv S_i$, for $1 \leq i \leq M$. Then the linear programming problem can be written as:

$$\begin{aligned} &\text{maximize: } Z \\ &\text{such that: } Z - C^T X = 0 \\ &\qquad AX = B \\ &\qquad X \geq 0 \end{aligned} \tag{2.19}$$

2.5.2 Basic Solutions

Assume that we have written the linear program as a set of linear equalities, such as 2.19. Various solutions to the matrix equality $AX = B$ are possible. A trivial solution is given by:

$$X_i = \begin{cases} B_{i-N}, & \text{if } (N+1) \leq i \leq (M+N) \\ 0, & \text{otherwise} \end{cases} \tag{2.20}$$

In terms of the original variable names, this solution sets each slack variable S_i equal to the corresponding B_i, and all X_i to zero. This gives an initial solution with objective value $Z = C^T X = 0$. However, this trivial solution is not the only possible solution.

Suppose we select a set of M variables $\hat{X} \subset X$. Let $\hat{A}$ be the columns of A that are the coefficients of those variables. Then, if $\hat{A}$ is nonsingular, a *basic solution* of $AX = B$ is given by setting $\hat{X} = \hat{A}^{-1}B$, and all other variables to zero. The *basic variables* are the variables that appear in $\hat{X}$. All other variables are called *non-basic*.

Further, the matrix A has rank M, because there exists at least one non-singular $M \times M$ sub-matrix. Indeed, the $M \times M$ identity matrix is a submatrix of A. This identity matrix is the submatrix that is the coefficients of the slack variables in the trivial solution.

The number of distinct basic solutions is determined by the number of distinct non-singular $M \times M$ submatrices of A. There are at most $\binom{M+N}{M}$ such non-singular sub-matrices.

2.5.3 Pivot Operations

Starting from the initial basic solution, subsequent steps of the simplex algorithm seek to improve the solution. Iterations change the variables in the basis one at a time, until an optimal solution is achieved or it is determined that no feasible solution exists. Each iteration consists of selecting one non-basic variable to be added to the basis, one basic variable to be removed from the basis, and some arithmetic to effect the change of basis.

To change the basis, it is not necessary to compute the inverse of the new basis submatrix from scratch. Instead, row operations similar to Gaussian elimination are applied. Suppose that we want to introduce variable X_j to the basis, and remove variable $\hat{X}_i$. We apply row operations similar to Gaussian elimination to generate a unit vector in column j, with the 1 in row i. The element $T_{i,j}$ is called the the *pivot element*. The operation of changing the basis is called a *pivot*.

In the pivot operation, row i is divided by $A_{i,j}$, to generate a 1 in position i, j. Then multiples or row i are added to each of the other rows to generate 0 in each other position of column j. The same operations are performed on the vector B. The objective function $C^T X = Z$ is treated as one of the rows, to generate a 0 in the j-th element of C.

These operations are justified by the rules of arithmetic, which allow us to derive new equations from existing equations by addition or by multiplication. The following inference rules encode these derivations:

Addition

$$\frac{\sum v_i x_i = P \qquad \sum w_i x_i = Q}{\sum (v_i + w_i) x_i = (P + Q)} \tag{2.21}$$

Multiplication For $c \neq 0$:

$$\frac{\sum v_i x_i = P}{\sum (c \times v_i) x_i = (c \times P)} \tag{2.22}$$

The addition rule (2.5.3) says that a new equation may be derived by adding two existing equations. The multiplication rule (2.5.3) says that a new equation may be derived by multiplying every constant in an existing equation by the same constant c. Note that negative constants c are allowed.

2.5.4 Integer Only Methods

Suppose that all of the coefficients and constants in an initial linear program are rational numbers. Those rational numbers can be represented with a common denominator, say d. Then, after a pivot operations, all of the coefficients and constants in the resulting linear program are also rational numbers. Further, if the coefficient of the pivot element has value p, the resulting numbers can all be represented with a common denominator pd.

At each pivot, the common denominator is multiplied by the pivot value. To see this, consider a 2×2 submatrix of coefficients:

$$\begin{matrix} A_{i,j} & A_{i,l} \\ A_{k,j} & A_{k,l} \end{matrix} \tag{2.23}$$

We will pivot on the element $A_{i,j}$. The k'th row is representative of rows other than the one containing the pivot element. The l'th column is representative of columns other that the one containing the pivot element. After the pivot, the submatrix becomes:

$$\begin{matrix} 1 & \frac{A_{i,l}}{A_{i,j}} \\ 0 & \frac{A_{k,l}A_{i,j} - A_{k,j}A_{i,l}}{A_{i,j}} \end{matrix} \tag{2.24}$$

Note that each element of the resulting matrix can be written with denominator $A_{i,j}$:

$$\begin{matrix} \frac{A_{i,j}}{A_{i,j}} & \frac{A_{i,l}}{A_{i,j}} \\ \frac{0}{A_{i,j}} & \frac{A_{k,l}A_{i,j} - A_{k,j}A_{i,l}}{A_{i,j}} \end{matrix} \tag{2.25}$$

The common denominator is closely related to the determinant of a submatrix of the coefficients. Gomory (1963) gives a concise algebraic proof, and Schrijver (1986) gives a good account of the historical development. Suppose the initial problem has constraints $Ax = B$, and after some pivot operations the problem has constraints $\bar{A}x = \bar{B}$. A sequence of pivots is just a sequence of steps of Gaussian elimination. Then there is a square matrix D such that $DA = \bar{A}$ and $DB = \bar{B}$, and the common denominator is just the determinant of D.

Clearly, it would be possible to multiply each equation by the common denominator, so that the linear program is represented as an equivalent system of linear equations with integer coefficients and constants. The common denominator may be kept separately, so that the rational numbers may be constructed whenever re-

quired. The advantage of this method is that linear programming algorithms may be implemented using only integer arithmetic.

2.5.5 Tableau Representation

Implementations of the simplex algorithms use a convenient data structure, called a tableau. A tableau is just a $M + 1 \times M + N + 1$ matrix, with some auxiliary information. A simplex tableau stores all of the information that is needed in the simplex algorithm, in a data structure that is more compact and more convenient for programming.

Consider the linear program:

$$\begin{aligned} &\text{maximize: } Z \\ &\text{such that: } \quad Z - C_1X_1 - \ldots - C_NX_N = 0 \\ &\qquad\qquad S_1 + A_{1,1}X_1 + \ldots + A_{1,N}X_N = B_1 \\ &\qquad\qquad S_2 + A_{2,1}X_1 + \ldots + A_{2,N}X_N = B_2 \\ &\qquad\qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ &\qquad\qquad S_M + A_{M,1}X_1 + \ldots + A_{M,N}X_N = B_M \end{aligned} \tag{2.26}$$

The tableau representation simply dispenses with the repetitive syntax and variable names, and uses a single matrix to keep track of all of the coefficients. The rows and columns are indexed starting from zero. Row 0 of the tableau represents the objective function. Rows 1 through M represent the individual constraints. Column 0 represents the B vector, while columns 1 through $M + N$ represent the individual variables. The objective variable Z is not represented in the tableau. The linear program (2.26) is represented in a tableau as follows:

$$\left[\begin{array}{c|cccccccc} 0 & -C_1 & -C_2 & \dots & -C_N & 0 & 0 & \dots & 0 \\ \hline B_1 & A_{1,1} & A_{1,2} & \dots & A_{1,N} & 1 & 0 & \dots & 0 \\ B_2 & A_{2,1} & A_{2,2} & \dots & A_{1,N} & 0 & 1 & \dots & 0 \\ \dots & \multicolumn{8}{c}{\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots} \\ B_M & A_{M,1} & A_{M,2} & \dots & A_{M,N} & 0 & 0 & \dots & 1 \end{array}\right] \tag{2.27}$$

The row operations of Gaussian elimination can be applied directly to the tableau matrix. This tableau format is very convenient for computation. However, it is not the smallest representation for the information. For very large problems, another optimization is available to further reduce the size of the tableau.

Note that the tableau contains the $M \times M$ identity matrix as a submatrix. In the simplex algorithm, the tableau always contains the $M \times M$ identity matrix as a submatrix. In cases where M is large, the identity matrix requires a large amount of storage. It is not necessary to actually store that constant submatrix. It is only necessary to provide storage for the columns that are coefficients of non-basic variables.

To keep track of which columns are actually stored in the reduced tableau, it is also necessary to keep two vectors of variable names. One keeps the indexes of the basic variables, the other keeps the indexes of the non-basic variables. These vectors of variable indexes (or names) are the auxiliary information. The resulting data structure consists of one matrix of coefficients and two vectors of variable names. The pivot algorithm using the reduced tableau is complicated by the need to exchange variable names between the basis and non-basis vectors, and by the arithmetic of generating in the pivot column the coefficients of the new nonbasic variable. However, it is clear that an actual implementation could use the reduced tableau.

2.5.6 The Primal Simplex Algorithm

We are now ready to state the primal simplex algorithm. We suppose that the problem is given as a system of equalities and has been placed into a tableau T as presented in formula (2.27) of subsection 2.5.5. For the primal algorithm, we further suppose that the initial solution is feasible. That is, we have $T_{i,0} \geq 0$ for $1 \leq i \leq M$.

The essential idea of the simplex algorithm is very simple. If any C_j is positive for a non-basic variable X_j, then the objective function can be increased by increasing the value of X_j. The objective function is stored in row 0 of the tableau, $T_{0,j} = -C_j$, so the algorithm looks for negative values in row 0 of the tableau. If any are found, the algorithm considers the pivot steps that would bring X_j into the basis, and chooses one that preserves the feasibility of the solution. Figure 2.1 shows the primal simplex algorithm.

The choice in step 1 of a variable to enter the basis is important. If a correct choice is always made, then the simplex algorithm terminates after at most M iterations. However, it is not easy to make such correct choices. Dantzig's original simplex algorithm included a deterministic procedure to choose the entering variable. That procedure gives good average performance (Smale, 1983). However, in the worst-case it requires an exponential number of steps. Klee and Minty gave an example in 1972 for which Dantzig's algorithm requires an exponential number of steps (Schrijver, 1986, pp. 139). Recently, randomized simplex algorithms have been developed which are both simple and polynomial (Kalai, 1992; Gärtner & Ziegler, 1994). A randomized algorithm just chooses randomly in step 3 from among all $\bar{j}$ such that $T_{0,j} < 0$.

Polynomially bounded algorithms for linear programming do exist. Khachiyan showed in 1979 an ellipsoid method that runs in polynomial time (Schrijver, 1986, ch. 13). However, in practice such methods are rarely used because they are so

Figure 2.1: Primal Simplex Algorithm

1. (Test for optimality.) If $T_{0,j} \geq 0$, for $1 \leq j \leq M + N$, then stop. The current solution is optimal.

2. (Choose a variable to enter the basis.) Choose an index $\bar{j}$ such that $1 \leq \bar{j} \leq M + N$ and $T_{0,j} < 0$.

3. (Test for unbounded solution.) If $T_{i,\bar{j}} < 0$ for $1 \leq i \leq M$, then fail. An unbounded solution exists.

4. (Choose a basis variable to exit the basis.) Choose an index $\bar{i}$ with $T_{\bar{i},0}/T_{\bar{i},\bar{j}} = \min(T_{i,0}/T_{i,\bar{j}})$, for $T_{i,\bar{j}} \geq 0$.

5. (Pivot to the new basis.) Divide each element of row $\bar{i}$ by $T_{\bar{i},\bar{j}}$. For each other row $k \neq \bar{i}$, subtract $T_{k,j}/T_{\bar{i},\bar{j}}$ times row $\bar{i}$ from row k. This generates a 1 in tableau position $T_{\bar{i},\bar{j}}$, and a 0 in tableau position $T_{\bar{i},\bar{j}}$, for each $k \neq \bar{i}$.

6. Go to step 1.

complex. In practice, Dantzig's original algorithm gives good performance with high probability, and randomized algorithms give good performance with probability 1.

2.5.7 Duality

Given any linear program, there is another linear program that is closely related to the first. In the context of this relation, the first linear program is referred to as the *primal*, while the second is referred as the*dual*. This section follows the presentation of Chvátal (1983, ch. 5). Suppose the primal linear programming problem is:

$$\begin{aligned} &\text{maximize: } C^T X \\ &\text{such that: } AX \leq B \end{aligned} \tag{2.28}$$

The dual linear program is defined using the same set of coefficient vectors and a new set of variables. Using variables Y_i the dual problem is:

$$\begin{aligned} &\text{minimize: } B^T Y \\ &\text{such that: } A^T Y \geq C \end{aligned} \tag{2.29}$$

We can write the dual linear program in standard form by multiplying the objective by -1 to express the dual program as a maximization problem, and multiplying the constraints by -1 to express them using $\leq$ relations. Then the dual linear program can be written as:

$$\begin{aligned} &\text{maximize: } -B^T Y \\ &\text{such that: } -A^T Y \leq -C \end{aligned} \tag{2.30}$$

It is easy to see that the dual of the dual is identical to the primal linear program. To see this, we write the dual of the linear program 2.30:

$$\begin{aligned} &\text{minimize: } -C^T X \\ &\text{such that: } -AX \geq -B \end{aligned} \tag{2.31}$$

Then we rewrite the dual of the dual in standard form, as a maximization problem with $\leq$ constraints, we obtain exactly 2.28.

There are several very interesting symmetries between the primal and the dual problem. Recall that a basic solution of the primal problem, or *primal solution*, is said to be optimal if $C_j \leq 0, for 1 \leq j \leq N$. A primal solution is said to be feasible if $B_i \geq 0, for 1 \leq i \leq M$. If all $C_j \leq 0$ and all $B_i \geq 0$, then the primal solution is both optimal and feasible.

By symmetry, a basic solution of the dual problem, or *dual solution*, is said to be optimal if $B_i \geq 0, for 1 \leq i \leq M$. A dual solution is said to be feasible if $C_j \leq 0, for 1 \leq j \leq N$. If all $C_j \leq 0$ and all $B_i \geq 0$, then the dual solution is both optimal and feasible.

A variety of theorems are available describing the relation between the primal and dual problems. Every feasible solution of the primal LP (2.28) provides a lower bound on the optimal (minimal) feasible value of the dual (2.29), and visa-versa. The proof of these bounds is particularly nice:

$$C^T X = \sum_{j=1}^{n} C_j X_j \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} A_{i,j} Y_i \right) X_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} A_{i,j} X_j \right) Y_i \leq \sum_{i=1}^{m} b_i y_i = B^T Y \tag{2.32}$$

A stronger duality theorem is due to Gale, Kuhn, and Tucker (Chvátal, 1983, pp. 57).

Theorem 2. *If the primal LP has an optimal feasible solution X, then the dual has an optimal and feasible solution Y such that $C^T X = B^T Y$.*

The proof of the strong form of the duality consists of constructing an optimal and feasible solution to either problem, by using slack variables, and showing that the dual of that solution is a feasible and optimal solution to the other problem.

By observing the symmetries, it is easy to verify that optimal solutions of the primal problem correspond exactly with feasible solutions to the dual problem, and optimal solutions of the dual problem correspond with feasible solutions of the primal

problem. If no feasible solution exists for one problem, then no optimal solution exists for the other, which must then be either unbounded or infeasible. It is possible that no feasible solution exists for either problem.

2.5.8 The Dual Simplex Algorithm

The dual simplex algorithm can be applied when the initial solution is optimal, but not feasible. The dual simplex algorithm solves the dual problem to optimality. However, it is not necessary to perform the matrix transposition. By swapping the indexes in the primal algorithm, a dual algorithm can be derived that solves the dual optimization problem using the primal tableau. Figure 2.2 shows the primal simplex algorithm.

2.5.9 The Primal-Dual Algorithm

A solution is optimal if $B_j \geq 0, for 1 \leq j \leq M + N$. A solution is feasible if $C_i \leq 0, for 1 \leq i \leq M$. The initial solution $\hat{X} = \hat{A}^{-1}B$ may suboptimal, infeasible, or both. In the initial tableau, some of the variable values $T^0_{i,0}$ may be negative, violating the non-negativity constraint of the primal simplex algorithm. Further, some of the $T^0_{i,0}$ may be negative, violating the non-negativity constraint of the dual simple algorithm.

The primal-dual algorithm is a variant algorithm that uses both primal and dual pivots. The primal-dual algorithm does not require that the $T^0_{i,0}$ (resp. $T^0_{i,0}$) tableau entries must be non-negative. Instead, it uses dual pivots to obtain non-negativity in the $T^0_{i,0}$ without regard to the $T^0_{i,0}$, then primal pivots to obtain the solution.

The pivoting procedure proceeds in two phases. In the first phase, dual pivots are applied as in the dual simplex algorithm until a feasible solution is found, or it

Figure 2.2: Dual Simplex Algorithm

1. (Test for Feasibility) If $T_{i,0} \geq 0$, for $1 \leq i \leq M$, then stop. The current solution is feasible.

2. (Choose a variable to exit the basis) Choose an index $\bar{i}$, where $1 \leq \bar{i} \leq N$, and $T_{\bar{i},0} < 0$.

3. (Test for No Feasible Solution) If $T_{\bar{i},j} \geq 0$ for $1 \leq j \leq N$, then fail. An unbounded solution exists.

4. (Choose a variable to enter the basis) Choose an index $\bar{j}$ with $T_{0,\bar{j}}/T_{\bar{i},\bar{j}} = \min(T_{0,j}/T_{\bar{i},j})$, for $T_{\bar{i},j} \geq 0$.

5. (Pivot to the new basis.) Divide each element of row $\bar{i}$ by $T_{\bar{i},\bar{j}}$. For each other row $k \neq \bar{i}$, subtract $T_{k,j}/T_{\bar{i},\bar{j}}$ times row $\bar{i}$ from row k. This generates a 1 in tableau position $T_{\bar{i},\bar{j}}$, and a 0 in tableau position $T_{\bar{i},\bar{j}}$, for each $k \neq \bar{i}$.

6. Go to step 1.

is determined that no feasible solution exists. In the second phase, primal pivots are applied as in the primal simplex algorithm to find the optimal feasible solution, or to determine that an unbounded solution exists.

2.6 Algorithms for Integer Programming

Various algorithms for integer programming have been proposed. Search algorithms are commonly used for integer programming, but suffer from the same combinatorial explosion as search algorithms for satisfiability. Indeed, the integer programming problem is polynomially reducible to SAT (Garey & Johnson, 1979, pp. 245). The other major approach to integer programming uses a class of algorithms called cutting plane algorithms.

Cutting plane algorithms for integer programming avoid the search. In cutting plane algorithms, the integer program is relaxed to a linear program, which is solved as a subproblem. Additional valid inequalities are added to the linear program to eliminate non-integral solutions of the linear program, while retaining integral solutions. Each addition of a new valid inequality increases the size of the linear program, which must then be solved as a new subproblem. Typically, many iterations are required.

Given a integer programming problem with constraints $Ax \geq b, x \in I^n$, an inequality $c^T x \geq d$ is a *valid inequality* if and only if $c^T \bar{x} \geq d$ for all $\bar{x} \in \{x | Ax \geq b, x \in I\}$. That is, every integral solutions of the constraints is also a solution of the valid inequality. If the integer program is unsatisfiable, then no integral solutions exist, and every inequality is a valid inequality.

A valid inequality is called a *cutting plane* with respect to a point y if in addition we have $c^T y < d$. Geometrically, a cutting plane *separates* the point from all of the integral solutions. The point y is a non-integral solution of the original constraints,

and the cutting plane provides a new constraint that excludes the non-integral point.

A new valid inequality may be derived by multiplying an existing valid inequality by a positive constant, or by addition of existing valid inequalities, just as in linear algebra. However, the key to the cutting plane algorithms is to eliminate non-integral vectors from the solution set of the linear program. This is done by introducing cutting planes. Two principle methods for deriving new cutting planes are provided by Gomory and Chvátal (Gomory, 1963; Chvátal, 1983).

2.6.1 Gomory Cutting Planes

Gomory's derivation of cutting planes uses modular arithmetic. For any set of equations, the equations hold also in the algebraic module defined by those equations for any modulus. That is, each equation implies that a set of modular equivalences hold, and the modulus may be freely chosen. If we have an equation:

$$\sum_i a_i x_i = b \tag{2.33}$$

where the a_i and b are integers, then for every integer modulus d, all integral values of x that satisfy the inequality also satisfy:

$$\sum_i a_i x_i \equiv b \pmod{d} \tag{2.34}$$

From this, Gomory derives the linear inequality:

$$\sum_i -(a_i \pmod{d}) x_i \leq -(b \pmod{d}) \tag{2.35}$$

Gomory's actual derivation is quite intricate. However, the actual concept is relatively simple, and so simplified presentations appear in various textbooks (Schrijver, 1986; Nemhauser & Wolsey, 1988; Wolsey, 1998). In particular, we may choose modulus

(mod 1). This has the effect of keeping only the fractional part of each coefficient. The simplified derivation starts from an equality:

$$\sum_i a_i x_i = b \tag{2.36}$$

By some simple analysis, we may derive:

$$\sum_i \lfloor a_i \rfloor x_i \leq \lfloor b \rfloor \tag{2.37}$$

Then subtracting 2.36 from 2.37, we obtain:

$$\sum_i (\lfloor a_i \rfloor - a_i) x_i \leq \lfloor b \rfloor - b \tag{2.38}$$

If we define a function to extract the fractional part of a coefficient: $f(x) = x - \lfloor x \rfloor$, Gomory's cutting plane may be written as:

$$\sum_i -f(a_i) x_i \leq -f(b) \tag{2.39}$$

The presentation using (mod 1) is equivalent to the presentation using arbitrary modulus, because the original equation may be multiplied by any desired positive rational number before taking the cut. That is, for each $\lambda > 0$, we may multiply the equality by λ,

$$\sum_i \lambda a_i x_i = \lambda b \tag{2.40}$$

and then use that inequality to derive the cut

$$\sum_i -f(\lambda a_i) x_i \leq -f(\lambda b) \tag{2.41}$$

For an arbitrary modulus d, the equivalent cut is derived by setting $\lambda = 1/d$. Then we have $f(\lambda a_i)/\lambda = f(a_i/d) \times d = a_i \pmod d$, and similarly $f(\lambda b)/\lambda = f(b/d) \times d = b \pmod d$. So equation 2.41 is equivalent to 2.34, if we choose $\lambda = 1/d$.

Gomory showed that the addition of these inequalities to a linear program cuts off some non-integer solutions of the linear program. In particular, each new inequality 2.39 is an unsatisfied constraint, because the constant $-f(\lambda b)$ on the right-hand side is negative. Hence, a dual-simplex algorithm may be used to find again a feasible solution. Since each added cut properly reduces the polytope without eliminating any integral solutions, we eventually obtain the integral hull of the polytope, and the optimal solution of that integer hull is an integer optimal solution of the original problem.

2.6.2 Chvátal Cutting Planes

Chvátal gave a simplified version of cutting plane derivation, and proved that the simplified version is complete. Given an inequality:

$$\sum_i a_i x_i \geq b \tag{2.42}$$

with the domain restriction $x_i \in I$, Chvátal derives a cutting plane by first rounding the coefficients:

$$\sum_i \lceil a_i \rceil x_i \geq b \tag{2.43}$$

Rounding the coefficients up to the next integer can only increase the left hand side, so every solution of (2.43) is also a solution of (2.42). Now, because the variables are all constrained to integer values, the left-hand side of (2.43) must be integral. Hence, we may also round the right-hand side:

$$\sum_i \lceil a_i \rceil x_i \geq \lceil b \rceil \tag{2.44}$$

Chvátal showed that every integer solution of (2.44) is also a solution of (2.42), and that there exists a valid derivation for every valid inequality. It is interesting that the

usual derivation of Gomory's cutting planes actually uses Chvátal 's cutting planes as an intermediate step.

The completeness theorem states that every valid inequality can be derived using only Chvátal -style cutting planes, multiplication of inequalities by positive numbers, and addition of inequalities. It is significant that the completeness theorem only asserts that a derivation exists for every valid inequality, but does not say anything about the length of the derivation. This completeness proof gave rise to the notion of cutting-plane proofs.

2.6.3 Basic Cutting Plane Algorithm

Gomory (1958) provided the original cutting plane algorithm. Gomory (1963) gives a full development with a proper proof of finite termination. Gomory's algorithm finds integral solutions to linear programs by adding cutting planes to the linear program one at a time. Figure 2.3 gives Gomory's cutting plane algorithm.

Subsequent algorithms using cutting planes have largely followed Gomory's original algorithm. The principle differences between cutting plane algorithms are in the methods used to select the row in step 5, and the methods used to actually construct the cutting plane from that row in step 6.

During the solving of a linear program, all of the values in the tableau can be expressed using a common denominator, which is the product of all of the pivot elements. (Each pivot step divides by the pivot element, and hence multiplies the common denominator by the pivot element.) The usual sequence of steps in cutting plane algorithms is to keep the linear program fully solved at all times, and to add cutting planes and use dual pivots to eliminate the fractions only when the fully solved linear program contains fractional values.

Figure 2.3: Gomory's Dual Cutting Plane Algorithm

1. Construct the tableau T of the linear programming problem.

2. Solve the linear programming problem using the dual simplex algorithm.

3. If the solution $B = (T_{0,1}, \ldots, T_{0,M})$ is integral, stop. (The optimal integral solution has been found.)

4. If no feasible solution exists, stop. (No integral solution exists.)

5. Select a row to use to generate the cut. Let i be the index that row. (One possible rule is to use the index of the first non-integral component of B.)

6. Generate a new cutting plane from the i'th equation:

$$\sum_{j=1}^{n} -f(T_{i,j})x_j + s = -f(T_{i,0})$$

where $f(y)$ is the fractional part of y, and s is a new slack variable.

7. Adjoin this new cutting plane to the tableau as a new row. Increment M, the count of rows in the tableau. Then go to step 2.

Gomory notes that the common denominator D is multiplied by the pivot element at each pivot step. With $D = 1$, the matrix is all integer, but with $D \neq 1$, the matrix cannot be all integer.

In Gomory's second finiteness proof, Gomory suggests a variation of the method. In this variation, the use of primal pivots, dual pivots, and the addition of cutting planes may be interspersed, rather than occurring in strict sequence (Gomory, 1963, pp. 290). In the proof, a method is suggested in which cutting planes and dual pivots are applied whenever the common denominator exceeds some threshold.

Whenever a cutting plane is added, and a pivot is performed using one of the fractions in the cutting plane as the pivot element, the value of D is reduced. After some finite number of cutting planes and dual pivots, the denominator is again reduce to the threshold. Some logic concerning the sequence of denominator values suffices to show that the algorithm finds an integer solutions, which is assumed to exist. Gomory credits E. M. L. Beale for a suggestion that fractional values might be eliminated immediately whenever they appear. That algorithm adds cutting planes whenever the denominator D is not 1.

The weakness of this proof is that it assumes the existence of an integer solution. Because of this weakness, the algorithm that is implicit in this proof has been largely ignored by the integer programming community.

Empirical Experience with Cutting Plane Algorithms

Early cutting-plane algorithms for integer programming seem to require large numbers of iterations, which in turn require solving a large number of large linear-programming subproblems. This empirical inefficiency was observed very early, but very few results were published to either demonstrate or explain the difficulty.

As a result, cutting-plane algorithms were quickly dismissed as too inefficient, and

most subsequent research on integer programming has concentrated on branch-and-bound approaches.

A few authors attempted to explain the difficulty. Jeroslow and Kortanek (1971) shows that pathological examples of integer programs exist such that Gomory's fractional cutting plane algorithm converges very slowly, if it is implemented using finite-precision arithmetic. For any given n, there exists an integer program with only two variables for which Gomory's algorithm requires at least n iterations. The problem occurs when a linear solution has a term with coefficient $-1/t$ for large t, so the cutting plane has a coefficient $(t-1)/t$, and the pivot on that coefficient makes a new term $-1/(t-1)$. If $-1/(t-1)$ gets rounded $-1/t$ by the finite-precision arithmetic, then $O(t)$ iterations are needed to eliminate the fractions.

The results of Jeroslow and Kortanek apply to a large class of fractional cutting-plane algorithms. However, the cutting planes considered by Jeroslow and Kortanek do not include all possible cutting planes. In this article, they consider only Gomory cuts with $\lambda = 1$. To be fair, they report that they completed a proof for a larger class of algorithms using more general cutting planes, but were convinced by the editor to present only the simple case in the published article. Even with these restrictions, Jeroslow and Kortanek demonstrated that the use of finite precision (floating-point) arithmetic could be a significant source of inefficiency in fractional cutting plane algorithms.

In subsequent literature, some attempts have been made to reduce the inefficiency, even though the causes were not fully understood.

2.6.4 The Method of Decreasing Congruences

Gondran (1973) presents a method of decreasing congruences, which is similar to the algorithm that is implicit in Gomory's second finiteness proof. The algorithm is also reported in english by Minoux (1983/1986). The basic idea is rather elementary. Suppose an initial integer program has integer coefficients. The linear relaxation of that integer program is solved to obtain a result that is optimal and feasible, but has non-integer values.

The basic idea is elementary. Suppose an initial integer program has integer coefficients, and we solve linear relaxation to obtain a result that is optimal and feasible but has non-integer values. Using Gomory's (1963) integer-only method, the numbers in that solution tableau have a common denominator which is the product of pivot elements. The method of decreasing congruences eliminates the fractions by decreasing the common denominator until it reaches one. The decrease is obtained by adding Gomory cutting planes, and performing pivots using pivot elements with value $n/d < 1$. The method is as follows:

1. If the solution is in integers, stop.
2. Generate a Gomory cutting plane, and add it to the tableau.
3. Perform exactly one dual-simplex pivot, using a pivot element $p/d < 1$ (i.e. with numerator $n < d$).
4. goto step 1.

The algorithm as presented by Minoux also incorporates a heuristic to look for an integral solution, before step 2. If an integral solution is found, then the algorithm stops. That heuristic may avoids some final iterations, but does not seem essential to the correctness of the algorithm.

In the method of decreasing congruences, the fractions are eliminated because the sequence of common denominators is strictly decreasing. At each iteration, the common denominator is multiplied by (n/d). Now, $n/d \leq (d-1)/d < 1$, so the algorithms strictly reduces the common denominator at each step. Gondran provided empirical results that the average number of iterations required to obtain an integer solution is $log_2(d)$, where d is the common denominator, if the best available cut is used at each step.

However, the method can result in some constraints becoming infeasible. If some constraints are infeasible after all fractions are eliminated, then additional iterations of the dual simplex method are required. These additional pivots may introduce fractions, which then must be eliminated by another application of the decreasing congruences method. Finite termination is assured because a finite number of cutting planes is sufficient, just as in other cutting plane methods. Because the number of cutting planes can be large, decreasing congruences is not an efficient algorithm for general integer programming.

2.6.5 Finding Strong Cutting Planes

There has been relatively little research on cutting plane algorithms for general integer programming, other than applications to specific problems. The widely held belief that cutting plane algorithms are inherently inefficient seems to have discouraged most research toward finding better cutting plane algorithms.

Blair (1976) gives two methods for deriving new valid inequalities from systems of linear inequalities. Given a system of linear inequalities $Ax \geq b$, a new inequality $\sum t_i x_i \geq R$ can be *linearly deduced* iff there are $\Theta_1, ..., \Theta_m \geq 0$ and $\gamma_1, ..., \gamma_n \geq 0$ such that $\sum_{j=1}^{m} a_{ji}\Theta_j - \gamma_i \leq t_i$ and $\sum_1^m b_j\Theta_j - \sum_1^n \gamma_j \geq R$. The linear deduction

rule is equivalent to finding a weighted sum of the inequalities $\sum a_{ij}x_j \geq b_i$ and the inequalities $-x_j \geq -1$. Blair's *nameless rule* allows a new valid inequality to be deduced from two similar inequalities: If $t_1x_1 + t_1x_2 + \ldots + sx_k + \ldots + t_nx_n \geq P$ and $t_1x_1 + t_1x_2 + \ldots + rx_k + \ldots + t_nx_n \geq T$ then $t_1x_1 + t_1x_2 + \ldots + (r + P - T)x_k + \ldots + t_nx_n \geq P$.

Vizvári (1989) gives two methods to find strong Gomory cuts, and several excellent definitions. The cut $\sum_j \hat{a}_j \pmod d x_j \geq \hat{b} \pmod d$ is a *strengthening* of the Gomory cut $\sum_j a_j \pmod d x_j \geq b \pmod d$ if $\hat{b} \pmod d \geq b \pmod d$ and $\hat{a}_j \pmod d \leq a_j \pmod d$, and we have at least one strict inequality.

The *rank* of a Gomory cut $\sum_j a_j \pmod d x_j \geq b \pmod d$ is the smallest nonnegative integer t such that the equation $\sum_j a_j x_j = d + td$ has at least one nonnegative integer solution. For $t > 0$, the equality gives a stronger cut than the original. Vizvári (1989) gives a dynamic programming algorithm to solve for the t giving the strongest cut, for each modulus $\in \{1 \ldots d\}$, in one pass for all moduli. The running time of the algorithm is linear in d, the denominator. Because the denominator can be exponential in the number of pivots/steps used in the cutting-plane algorithm, this algorithm is not efficient. However, if the algorithm is interrupted early, a partial result can be useful.

In a second method, the problem of choosing a multiplier is considered. The cut $\sum_j \hat{a}_j \pmod d x_j \geq \hat{b}(\mod d)$ is a λ-*strengthening* of the Gomory cut $\sum_j a_j \pmod d x_j \geq b \pmod d$ if $\lambda\hat{b} \pmod d \geq b(\mod d)$ and $\lambda\hat{a}_j \pmod d \leq a_j \pmod d$, and we have at least one strict inequality. The theorem shows that if there is a λ-strengthening, then there is a λ-strengthening such that $\lambda = \lceil p\frac{d}{b} \rceil - 1$, for some index $p \in \{1 \ldots s\}$, where s is the rank of the Gomory cut. This theorem allows the search for multipliers to be constrained to a finite set of multipliers.

Ceria, Cornuéjols, and Dawande (1995) consider the problem of finding a linear

combination of those cutting planes to give the strongest possible cut. The approach uses rational coefficients with a fixed denominator. The procedure first maximized the fraction of the constant on the right-hand side, using a nice observation about the gcd of the coefficients. Then it minimizes the fractions of the coefficients on the left-hand side of the cut, one at a time. This minimization uses a result of J.B. Rosser on the solution of Diophantine equations. Each ordering of the left-hand variables gives a different cut.

Ceria et al. (1995) implemented their algorithm only for the case with up to three inequalities, requiring only three weights. This is because the method requires solving a combinatorial subproblem. Even for this case, the implemented algorithm does not always give the strongest cut when slack variables are involved. In a final section, they suggest that eliminating the slack variables before the optimization might give better results.

2.6.6 Summary

Several approaches to finding strong Gomory cutting planes each require solving a combinatorial subproblem, or inspecting a combinatorial number of cases. The second method of Vizvári (1989) appears to be the least impractical. If the rank of the cut can be restricted to some small value, then it may be possible to search for the value of λ that gives a strong cut. By any method, the problem of finding strong Gomory cutting planes is a search problem, and the number of possibilities is large enough so that exhaustive search is not feasible.

2.7 The Convex Hull of SAT

A critical problem in a cutting-plane algorithm is to find tight cuts, so as to minimize the number of cuts that are needed. Various authors have considered the problem of finding facet inequalities. Facet inequalities are inequalities that describe the convex hull of the integer solution to a problem. In order to find facet cuts, it is essential to have some information about the facets of the integer hull.

2.7.1 Canonical Inequalities

Balas and Jeroslow (1972) made a major step toward describing the convex hull of the integer solutions for the SAT problem. Balas and Jeroslow do not specifically mention the SAT problem. Instead, they consider the convex hull of subsets of the vertices of the n-dimensional unit hypercube. Of course, each satisfying solution of a SAT problem is exactly a vertex of an n-dimensional unit hypercube, so the convex hull of the SAT problem is exactly the convex hull of a subset of such vertices. The theory requires some rather technical definitions:

The n-dimensional *unit hypercube* is the set

$$K = \{x \in R^n | 0 \leq x_j \leq 1, j \in N\} where N = \{1, \ldots, n\} \tag{2.45}$$

A *vertex* of K is a point $x \in K$ such that exactly N of the inequalities (2.45) are tight. Let V be the set of vertices. Two vertices $x, y \in V$ are *adjacent* if they differ in exactly one component.

Let $x, y \in V$ be adjacent vertices, then $[x, y] = \{z \in R^n | z = \lambda x + (1 - \lambda)y, 0 \leq \lambda \leq 1\}$ is an *edge* of K, and $(x, y) = \{z \in R^n | z = \lambda x + (1 - \lambda)y, 0 < \lambda < 1\}$ is an *open edge* of K. An edge is sometimes termed a *closed edge*, to differentiate from an open edge. Two distinct edges of K are *adjacent* iff they have a common endpoint.

A *k-dimensional face* F^k of K is a set of points $x \in K$ such that exactly $n-k$ of the $2n$ inequalities (1) are tight. A k-dimensional face F^k can be written as a conjunction of k literals. Using DeMorgans law, a point x is on a face if the the disjunction of the negations of those literals is false. Let F be a k-dimensional face. let $N(F)^+ = \{j \in N | \forall x \in F \cap V, x_j = 1\}$, $N(F)^- = \{j \in N | \forall x \in F \cap V, x_j = 0\}$, and $N(F)^0 = N - N(F)^+ - N(F)^-$. If $|N(F)^+| + |N(F)^-| = k$, then a k-dimensional face can be written as:

$$\sum_{i \in N(F)^+} (1 - x_i) + \sum_{i \in N(F)^-} x_i = 0$$

The *dimension* or *order* of a k-dimensional face is just $n-k$. Note that the number of variables not used in the equation, $|N(F)^0|$, is equal to the dimension of the face.

The *distance* $d(x, y)$ between two vertices $x, y \in V$ is the number of indices such $k \in N$ such that $x_k \neq y_k$. The distance $d(x, F)$ between a vertex x and a face F of K is $d(x, F) = min_{y \in F \cap V} d(x, y)$.

A canonical hyperplane is defined by Balas and Jeroslow as the set of vertices that are all at some fixed distance d from a given face. Let F be a k-dimensional face. A *canonical hyperplane* associated with a face F and denoted $H(F)_d$ is defined by:

$$\sum_{i \in N(F)^+} (1 - x_i) + \sum_{i \in N(F)^-} x_i = d$$

for a constant d. The *order* of a canonical hyperplane $H(F)_d$ is the number of zero coefficients in the equation, $|N(F)^0|$. The half-space bounded by a canonical hyperplane,

$$\sum_{i \in N(F)^+} (1 - x_i) + \sum_{i \in N(F)^-} x_i \leq d$$

is denoted $H(F)_d^+$.

Now, it gets interesting. It turns out that canonical hyperplanes have a property in common with integer hulls.

Proposition 1 (Balas and Jeroslow (1972)). *Let $V' \subset V$ be a subset of vertices, and let $C(V')$ be the integer hull of V'. Then $C(V')$ contains a point of an open edge (x, y) iff it contains the whole edge $[x, y]$.*

Proposition 2 (Balas and Jeroslow (1972)). *A canonical hyperplane H contains a point of an open edge (x, y) of K if an only if it contains the whole edge $[x, y]$.*

The convex hull of a subset of vertices is exactly the integer hull of SAT. This gives an indication that canonical hyperplanes may have something in common with faces of the integer hull of SAT. The result suggests is that canonical hyperplane might be the faces of the integer hull of SAT, but Balas and Jeroslow do not obtain that result. They do show one simple case in which the integer hull of a set of vertices is coincident with a canonical hyperplane. For a set of vertices that lay on a canonical hyperplane, the integer hull of those vertices is exactly that canonical hyperplane.

Theorem 3 (Balas and Jeroslow (1972)). *Let $C(V \cap H)$ be the convex hull of all vertices of K lying on a canonical hyperplane H. Then $C(V \cap H) = H \cap K$.*

In an extension, Balas and Jeroslow give an inference rule that allows a k-dimensional half space to be derived from a conjunction of $n - k$ k+1-dimensional half-spaces.

Proposition 3 (Balas and Jeroslow (1972)). *Let $H(F^k)_d$, $d \neq 0$, be a canonical hyperplane and let $F_i^{k+1}, i = 1, \ldots, n-k$ be all the faces of order $k+1$ containing F^k. Then*

$$x \in V \cap \left\{ \bigcap_{i=1}^{n-k} H(F_i^{k+1})_d^+ \right\} \Longrightarrow x \in H(F^k)_{d+1}^+ \tag{2.46}$$

The notation of equation (2.46) is rather dense. The $n-k$ half-spaces $H(F_i^{k+1})_d^+$ are just the inequalities that result from omitting one of the literals from the inequality $H(F^k)_d^+$. Their proof of the proposition makes this clear.

Proof. The left-hand side of the implication is equivalent to the statement that x satisfies the relations:

$$\sum_{j\in N(F)^+-i} x_j - \sum_{j\in N(F)^-} x_j \leq |N(F)^+| - 1 - d \text{ , for } i \in N(F)^+$$

$$\sum_{j\in N(F)^+} x_j - \sum_{j\in N(F)^--i} x_j \leq |N(F)^+| - d \text{ , for } i \in N(F)^-$$

where the number of inequalities is $|N(F)^+| + |N(F)^-| = n-k$. Adding the inequalities yields

$$(n-k-1)\left[\sum_{j\in N(F)^+} x_j - \sum_{j\in N(F)^-} x_j\right] \leq (n-k-1)\left[|N(F)^+| - d\right] - d$$

or

$$\sum_{j\in N(F)^+} x_j - \sum_{j\in N(F)^-} x_j \leq |N(F)^+| - d - \frac{d}{n-k-1}$$

which implies, in view of the condition $x_j \in \{0,1\}$, $j \in N$,

$$\sum_{j\in N(F)^+} x_j - \sum_{j\in N(F)^-} x_j \leq |N(F)^+| - d - 1$$

□

The proof just adds the $n-k$ inequalities together, divides by the common coefficient of the literals, then takes a Chvátal cut of the result to obtain the lower-order inequality.

Balas and Jeroslow also give an inference rule that is exactly equivalent to *resolution*, in which a new canonical half-space can be inferred from two the canonical half-spaces defined by two adjacent canonical hyperplanes. Finally, they give an algorithm

to derive from an arbitrary inequality the set of strongest canonical inequalities that define the same vertices. Finally, Balas and Jeroslow show that the decision problem of an arbitrary integer program can be reduced to a set cover problem.

2.7.2 Lifting Procedures for Set Covering

The set-cover problem is closely related to the SAT problem. By introducing a new variables $z_i = \overline{x_i} = (1 - x_i)$ for each 0-1 variable x_i in SAT, and adding constraints $x_i + z_i \geq 1$, a given SAT problem can be stated as a set-cover problem. The SAT problem in n variables is satisfiable if there is a cover of only n nonzero variables. Nobili and Sassano (1989) use the natural bipartite graph of a set-cover problem $B = (V, U, E)$ with node set $V \cup U$ and edge set E, where there is an edge between $v_i \in V$ and $u_j \in U$ if and only if variable v_i appears in clause u_j.

Definition 3. *Let $C \subseteq V, D \subseteq V$ be two arbitrary node sets. The induced subgraph obtain from $V \setminus (C \cup D), U \setminus \mathcal{N}(D))$ by removing from $U \setminus \mathcal{N}(D)$ the dominated nodes is denoted as B_D^C and is said to be a* minor *of B. B_D^C is said to be obtained by* contracting *the nodes in C and* deleting *the nodes in D.*

The function $\mathcal{N}(D)$ is the set of neighbors of nodes in D. This definition corresponds with the usual notion of contracting and deleting columns in 0-1 matrices. The sets that contain variables in D and the variables in C are removed from the set-cover problem. The resulting set-cover problem is smaller, in the sense that it has fewer variables and/or fewer sets that must be covered. In a matrix representation, the columns correspond to the variables, and the rows correspond to the sets or constraints. A 1 entry indicates that the variable of that column is present in the set of that row. The contracting operation removes a column from the matrix, while the deleting operation removes the rows that contain 1 in a column, and also removes

that column.

Nobili and Sassano found several results that allow us to consider these minors as natural sub-problems of SAT. Basically, if an inequality is a facet of a minor sub-problem, then it can be extended to find a facet of the larger problem. The lifting lemma gives that for each facet of a minor sub-problem, a facet of the larger problem exists. The sequential lifting lemma gives that there is a sequence of minor sub-problems, and that the variables may be added one at a time rather than all at once.

Lemma 2 (Lifting Lemma). *Let $H = (V_H, U_H, E_H)$ be a minor of B and let $Q(H)$ be the convex hull of H. Suppose that the inequality $\sum_{v \in V_H} a_v x_v \geq a_0$ defines a non-trivial facet of $Q(H)$ and that $Q(H)$ is full dimensional. Then there exist non-negative numbers $b_v (v \in V \setminus V_H)$ and b_0 such that the inequality $\sum_{v \in V_H} a_v x_v + \sum_{v \in V \setminus V_H} b_v x_v \geq a_0 + b_0$ defines a facet of $Q(B)$.*

Lemma 3 (Sequential Lifting). *Let $H = B_D^C = (V_H, U_H, E_H)$ be a minor of B and assume that the inequality $\sum_{v \in V_H} a_v x_v \geq a_0$ defines a nontrivial facet of $Q(H)$. The the following statements hold:*

1. *For each $w \in C$ the inequality*

$$\sum_{v \in V_H} a_v x_v + (a_0 - \beta^a(B_{D \cup \{w\}}^{C \setminus \{w\}})) x_w \geq a_0$$

   defines a facet of the polytope $Q(B_D^{C \setminus \{w\}})$.

2. *For each $w \in D$ the inequality*

$$\sum_{v \in V_H} a_v x_v + (\beta^a(B_{D \cup \{w\}}^{C \setminus \{w\}}) - a_0) x_w \geq \beta^a(B_{D \cup \{w\}}^{C \setminus \{w\}})$$

   defines a facet of the polytope $Q(B_{D \setminus \{w\}}^C)$.

The function β^a is defined as being the minimum weight of a cover of a bipartite graph. The sequential lifting lemma says that, given a facet of a proper minor of a set-cover problem, we may compute a facet of another minor with one additional variable. The computation is not necessarily straightforward, because the needed values of the function β^a are in general not easy to compute.

2.7.3 Diagonal Sum Inequalities

Hooker (1992) gives a method of deriving a valid inequality called a diagonal sum, and an algorithm using that rule to solve 0-1 integer programming problems. A *diagonal sum cut* is an inference rule based on a rank-one cut, in which the inequality:

$$a_1x_1 + a_2x_2 + ... + a_nx_n \geq b \tag{2.47}$$

may be inferred from the set of inequalities:

$$\begin{aligned} (a_1 - 1)x_1 + a_2x_2 + ... + a_nx_n &\geq b - 1 \\ a_1x_1 + (a_2 - 1)x_2 + ... + a_nx_n &\geq b - 1 \\ &\dots \\ a_1x_1 + a_2x_2 + ... + (a_n - 1)x_n &\geq b - 1 \end{aligned} \tag{2.48}$$

The name *diagonal sum* derives from the fact that the coefficients on the diagonal are each reduced by one in the antecedents. The proof shows that there is a Chvátal cut of a linear combination of the antecedents that gives the consequent. The structure of Hooker's proof of the diagonal sum rule is identical to the structure of Balas and Jeroslow's proposition 7 (Balas & Jeroslow, 1972). It consists of adding the antecedent inequalities together, then dividing by the common coefficient, and taking a Chvátal cut to obtain the consequent. The algorithm consists of repeated application of resolution and diagonal sum cuts. A theorem is given that repeated application of resolution and diagonal sum cuts yield a complete set of prime inequalities.

Hooker also considers specifically the case of set-covering inequalities, in which the coefficients a_i are restricted to $\{0, 1\}$ and the right-hand side b is 1. Set covering problems are closely related to SAT problems. An extended set-covering inequality allows positive integers on the right-hand side. The application of diagonal sum inference procedure to set-covering inequalities uses a difficult combinatorial algorithm to search for diagonal-sum cuts. Hookers method requires combinations of inferences to derive the antecedents of the diagonal-sum rule from a given set of extended set-covering inequalities.

A set of extended clauses *dominates* an extended clause if the extension of the set is a subset of the extension of the extended clause. Hooker (1992) gives several easy lemmas to decide domination. Barth (1994) gives a very nice theorem that an extended clause $L \geq d$ dominates $L' \geq d'$ iff $|L \setminus L'| \leq d - d'$. This gives an easy decision predicate for the dominance relation between constraints. If $L \geq d$ dominates $L' \geq d'$, then $L \geq d$ implies $L' \geq d'$, and so the second constraint is redundant and can be removed.

Barth (1993) considers the problem of transforming a linear 0-1 constraint into an equivalent set of extended clauses, which he defines as an inequality having the form $L_1 + \ldots + L_n \geq d$, where $0 \leq d \leq n + 1$, and the L_i are literals. Extended clauses constrain the *number* of propositions that must be true to make the clause true, and are essentially identical to the canonical inequalities defined by Balas and Jeroslow. Starting from a linear 0-1 constraint, $c_1L_1 + \ldots + c_nL_n \geq d$, the procedure removes the terms with small coefficients by adding multiples of $-x_i \leq -1$ as long as the right hand side remains positive. The result has the fewest literals, at least one of which must be true. A Chvátal cut then discards the remaining coefficients.

Barth (1995) gives another algorithm to search for diagonal sum cuts in an integer program, which is much nicer than Hookers original algorithm. The algorithm essen-

tially uses a recursive procedure to find a diagonal sum. To infer $\sum_{i=1...N} L_i \geq d$, the algorithm first finds or infers each of the N required antecedent inequalities.

Later, Barth (1996) shows how to use diagonal sum cuts to show in polynomial time that the pigeonhole problem is unsatisfiable. The algorithm for the pigeonhole uses the diagonal sum cuts in a specific order, so that only a polynomial number of the cuts must actually be computed. The use of diagonal sum cuts to solve the pigeonhole problem efficiently shows that diagonal sum cuts are quite strong.

2.7.4 Summary

Based on Balas and Jeroslow's results, we might suspect that the facets of the integer hull of SAT are canonical hyperplanes. However, they do not achieve that result. They do achieve several other results that anticipate the results of Hooker and Barth. Those results were later used by Barth to show that a cutting plane algorithm can efficiently solve pigeonhole problems.

Hooker and then Barth build on the results of Balas and Jeroslow without citing Balas and Jeroslow (1972). Nevertheless, Hooker's diagonal-sum rule gives a strong cutting plane, and Barth showed that diagonal-sums can be used to solve certain hard SAT problem efficiently.

2.8 Proof Systems and Lower Bounds

To decide that a term is unsatisfiable, a SAT algorithm must implicitly construct a proof of unsatisfiability. Any proof that no witness exists is necessarily based on some system for proving theorems of propositional logic. Different formalizations of propositional logic include different sets of axioms and inference rules, but are equivalent in the sense that they admit proofs of the same theorems. However, the

length or size of the proofs is often very different. Lower bounds for the complexity of a complete SAT algorithm have been derived by considering the algorithm as a proof system.

The length or size of the proof in a proof system is not less than the complexity of the corresponding algorithm. For a given proof system, if there exists a problem for which the shortest possible proof has length $f(n)$, where X is the size of the input, then the worst-case complexity of the algorithm is at least $O(f(n))$. The size of a proof is just the number of symbols in the proof, while the length is the number of lines in the proof. If the size of one line is limited by some polynomial in the number of variables, then the two measures of proofs can be used interchangeably.

The research in this area takes two general approaches. Some research has been directed at finding unsatisfiable problems which require exponentially long proofs in a specific proof system. Other research has considered polynomial simulation of one proof system by another. A proof system A is said to polynomially simulate a proof system B, if, for every for every proof in system B of length L, a proof can be can be constructed in system A with length $P(L)$, for some polynomial P. Both sorts of results serve to show that one proof system is more or less powerful than another.

2.8.1 Relative Strength of Proof Systems

Cook and Reckhow (1974) defines a *super* proof system as one for which there exists a polynomial $P(n)$ such that every tautology has a proof of length not more than $P(n)$. Cook then considers some proof systems that might be used to approach the SAT problem, and gives a number of polynomial transforms between them. Analytic tableaux, Davis Putnam tree search, and a restricted from of resolution called regular resolution are shown to be "not super". Gentzen's system without cut, unrestricted

resolution, and several others are not classified by Cook as super or not super. Resolution with extension, Gentzen's system with cut, natural deduction, and all Frege systems are shown to be polynomially equivalent in proof length.

2.8.2 Hard Problems

Cook and Reckhow (1979) considers the length of proofs in natural deduction systems. For this, Cook uses the pigeonhole problem as an example of a proposition having a proof with a polynomial number of lines in an unextended Frege system, but in which the lines are exponentially long. Cook does not claim that this proof is the shortest possible Frege proof of the pigeonhole formula, so this is not a lower bound. Cook does show in this paper that all unextended Frege systems are equivalent, up to a polynomial factor.

Based on Cook's (1979) result, pigeonhole problems have become popular for analysis of satisfiability algorithms. For a given problem, a specialized proof system that allows a short proof can be defined, and a specialized algorithm can construct that short proof. Several specialized algorithms are able to construct short proofs for pigeonhole problems in various proof systems (Buss, 1987; Bibel, 1990; Urquhart & Fu, 1996).

2.8.3 Extended Proof Systems

Buss (1987) gives a proof that (unextended) Frege systems do admit a polynomial-length proof of pigeonhole principle tautologies. The Frege proof of PHP is constructed by coding arithmetic operations as propositions, and then giving a simple counting argument. The constructions of arithmetic operations are proved by polynomial Frege proofs, and the counting argument is polynomial size, so the composition

is polynomial size. An exponential gap between resolution proof systems and Frege proof systems is given by a theorem that propositional pigeonhole tautologies have polynomial size Frege proofs, but require exponential size resolution proofs. Hence, Frege systems are *strictly* more powerful than resolution systems.

Bibel (1990) constructs polynomial size proofs or pigeonhole problems using an extended connection method. Connection methods use matrices to display propositional formula, and arcs to show connections, which are essentially resolutions between complementary literals in two clauses. To enable these proofs, Bibel extends the connection method by two new rules. Using these new rules, proofs for a pigeonhole problem can be given in polynomial number of steps. This shows that the extended connection method is strictly stronger than resolution. The key to the short proof is that smaller subproblems are isolated, and then each smaller subproblem is solved using a lemma rather than constructing individual proofs.

2.8.4 Bounded Depth Frege Systems

The depth of a proof in a Frege system is the maximum depth of the formula that appear as lines of the proof. The depth of a formula is the tree-height of the formula. A bounded-depth proof systems is a proof system in which the depth of formula is bounded by a constant. Several important lower bounds have been found for bounded-depth proof systems. This is an important class of proof systems, because shallow formulas are considered easier for most people to work with in natural deduction and Frege proof systems.

Håstad (1987) gives several complexity theorems using Boolean terms that compute parity functions. Håstad showed that certain functions that can be expressed by polynomially bounded circuits at each depth $N > 1$ require super-polynomial cir-

cuits at depth $N - 1$. This directly implies that satisfiability algorithms based on resolution cannot be efficient. Similar results are provided by (Beame et al., 1992; Beame & Pitassi, 1996), but with a simpler and more direct proof. This reduces the list of proof systems that not known to be not super to those that Cook and Reckhow (1974) identified as polynomially equivalent with Frege systems.

Urquhart and Fu (1996) show that proofs of pigeonhole tautologies require exponential number of steps in bounded-depth Frege systems, but polynomial number of steps in unbounded-depth Frege systems. For any depth d, any finite Frege system that can have polynomial proofs of PHP must have formulas of depth $> d$ in proof steps. For any bounded depth, the proof lengths go exponential.

2.9 Cutting-Plane Proof Systems

Cutting plane proof systems use linear inequalities as the lines of the proofs. In each cutting plane proof system, rules of inference allow new linear inequalities to be derived from previous linear inequalities. For SAT, the initial inequalities are derived as for an integer program. Chvátal (1984), Goerdt (1991), and Clote (1995) have each defined a version of cutting plane proof system. The several proof systems differ in the details of their rules of inference, but are polynomially equivalent. Every theorem proved of any of the proof systems can also be proved in each of the others, and the lengths of the proofs are polynomially related.

2.9.1 Chvátal Cutting Plane Proofs

The earliest definition of "cutting plane proof", as distinct from cutting plane methods for integer programming, seems to be due to Chvátal (1984). As reported by Schrijver (1986), the definition is as follows:

> Let $Ax \leq B$ be a system of linear inequalities, and let $cx \leq \delta$ be an inequality. We say that a sequence of linear inequalities $c_1x \leq \delta_1, c_2x \leq \delta_2, \ldots, c_mx \leq \delta_m$ is a *cutting plane proof of* $cx \leq \delta$ *(from* $Ax \leq B$*)*, if each of the vectors $c_1, \ldots, c_m$ is integral, if $c_m = c$, $\delta_m = \delta$, and if for each $i = 1, \ldots, m$: $c_i \leq \delta_i'$ is a nonnegative linear combination of the inequalities $Ax \leq B, c_1x \leq \delta_1, \ldots, c_{i-1}x \leq \delta_{i-1}$ for some δ_i' with $\lfloor \delta_i' \rfloor \leq \delta_i$.

According to this definition, a new inequality may be derived from any nonnegative linear combination of inequalities appearing earlier in the proof, such that the coefficients c are integral, by rounding the constant on the right-hand side of the inequality. The restriction to nonnegative coefficients is necessary, because a negative coefficient would reverse the sense of an inequality and allow derivation of an invalid inequality.

The restriction to integer coefficients seems innocuous, because an inequality with rational coefficients can be multiplied by the least common multiple of the denominators. However, any such multiplication reduces the difference $\delta_i - \lfloor \delta_i' \rfloor$ that can be introduced by the rounding operation. Hence, this method of deriving new valid inequalities appears to give weaker inequalities than the cutting planes of Gomory or Chvátal .

Schrijver gives a completeness theorem, which asserts:

1. If $Ax \leq B$ has at least one integral solution, and $cx \leq \delta$ is a valid inequality, then there is a cutting plane proof of $cx \leq \delta$ from $Ax \leq B$;

2. If $Ax \leq B$ has no integral solutions, then there is a cutting plane proof of $0x \leq -1$ from $Ax \leq B$.

This completeness theorem shows that the cutting plane proof system can serve as the proof system for a complete algorithm for SAT. If the proposition is satisfiable,

then cutting plane proofs can derive every linear inequality needed to find the integral polytope. Further, if the proposition is not satisfiable, then a cutting plane proof of $0x \leq -1$ exists.

2.9.2 Goerdt Cutting Plane Proofs

Goerdt (1991) gives a simplified cutting plane proof system named CP that admits three rules of inference:

Addition

$$\frac{\sum v_i x_i \geq P \qquad \sum w_i x_i \geq Q}{\sum (v_i + w_i) x_i \geq (P + Q)} \tag{2.49}$$

Multiplication For $c \in Z^+$:

$$\frac{\sum v_i x_i \geq P}{\sum (c \cdot v_i) x_i \geq (c \cdot P)} \tag{2.50}$$

Division For $c \in Z^+ \setminus 0$, $\quad v_i = c \cdot w_i$ for all i,

$w_i \in Z$ for all i, $\qquad$ not all $v_i = 0$:

$$\frac{\sum v_i \cdot x_i \geq P}{\sum w_i \cdot x_i \geq \lceil \frac{1}{c} \cdot P \rceil} \tag{2.51}$$

(In Goerdt's article, the division rules contains a typographical error. The fraction as printed on page 176 of that article is $\frac{\sum v_i \cdot x_i \geq P}{\sum w_i \cdot v_i \geq Q}$. The version given above is corrected.)

Goerdt's CP system is clearly equivalent to the definition of cutting plane proof given by Chvátal and Schrijver. The use of $\geq$ rather than $\leq$ for the inequalities has the effect of reversing the signs of all of the numbers, and requires rounding up instead of rounding down. The coefficients are required to be integers, and only the constant can be rounded. The addition and multiplication rules allow construction, by multiple steps, of any nonnegative linear combination of the preceding inequalities.

The division rule provides the rounding of the constants. Subtraction of inequalities and multiplication by negative constants are not allowed, to preserve the validity of the derived inequalities.

2.9.3 Clote Cutting Plane Proofs

Clote (1995) gives yet another equivalent definition, and also an extension named $CP+$. Clote first defines a language of expressions as follows: If $a \in \mathbf{Z}$ and $i \in \mathbf{N}$, then $a \in \mathsf{E}$ and $(a \cdot x_I) \in \mathsf{E}$. If $E, F \in \mathsf{E}$, then $(a \cdot E) \in \mathsf{E}$ and $(E + F) \in \mathsf{E}$. Using these expressions, Clote's system of cutting plane proofs is defined by five inference rules:

Transitivity:

$$\frac{E \geq F \qquad F \geq G}{E \geq G} \tag{2.52}$$

Simplification Simplification of arithmetic expressions

Addition

$$\frac{E \geq F \qquad G \geq H}{E + G \geq F + H} \tag{2.53}$$

Multiplication

$$\text{for } c \in N, \frac{E \geq F}{c \times E \geq c \times F} \tag{2.54}$$

Division for $c \in N, c > 0, E, F, E' \in \mathsf{E}, E = c \cdot E'$,

$$\frac{E \geq F}{E' \geq \lceil F/c \rceil} \tag{2.55}$$

Note that the multiplication rule allows only positive integers as multipliers. Also, the division rule requires that the coefficients (in E') must be integers. With these restrictions, it is not difficult to see that this proof system is equivalent to the proof system of Goerdt. The difference of presentation is intended to make it easier to define Clote's system CP^+.

CP^+ is then defined as a modification of CP, which removes the restriction in the division rule that the division must yield an integral value. The division rule is then restated as:

Division for $c \in N, c > 0, E, F \in \mathsf{E}$,

$$\frac{E \geq F}{\lceil E/c \rceil \geq \lceil F/c \rceil} \tag{2.56}$$

Clearly it is intended that the $\lceil\rceil$ operators, when applied to a sum of the form $\lceil c_1 x_1 + \ldots + c_n x_n \rceil$, should be distributed over the elements as $\lceil c_1 \rceil x_1 + \ldots + \lceil c_n \rceil x_n$.

Note that proof system $CP+$ retains the requirement that the multiplication rule allows only positive integers as multipliers.

2.9.4 Summary

Cutting planes proofs differ from linear program, in that only nonnegative linear combinations of inequalities are allowed. Several versions of cutting plane proofs have been defined. The traditional version due to Chvátal allows nonnegative linear combinations of inequalities, with integral coefficients (Chvátal, 1984; Schrijver, 1986). An equivalent system due to Goerdt (1991) allows only binary addition of inequalities, and multiplication by positive integers. A definition due to Clote (1995) allows the division rule to be applied even where the coefficients do not all divide the divisor evenly. Clote's proof system is CP^+ is identical to that of Chvátal (1984),

while Goerdt's system is slightly more restrictive. In all three definitions, only nonnegative multipliers are allowed. Clote showed that all three definitions are equivalent in the sense that they can polynomially simulate each other.

In each case, the restriction to nonnegative multipliers preserves the validity of the inequalities. In each case, the proof is constructed entirely using only the variables that appear in the original set of inequalities.

2.10 Lower Bounds for Cutting Plane Systems

Several lower-bound and simulation results are available for cutting plane and Frege proof systems. Positive results indicate that cutting plane proof systems are as powerful as general Frege proof systems (Clote, 1995). Negative results include lower bounds for restricted forms of cutting plane proof systems (Bonet, Esteban, Galesi, & Johannsen, 1998), and a general lower bound for an unrestricted cutting plane proof system based on Chvátal cuts (Pudlák, 1997). One misleading result shows an exponential lower bound for cutting plane proofs, but only for an input problem of exponential size (Cook, Coullard, & Turán, 1987). Cook (1990) shows that even for such problems, cutting plane proofs may be constructed in polynomial work space.

2.10.1 Bounded Cutting Plane Systems

The first lower bound result on the length of cutting plane proofs of unsatisfiability is due to Cook et al. (1987). The particular cutting plane proof system is uses only Chvátal cuts and inequalities with bounded integral coefficients. The lower bound is based on an unsatisfiable system having 2^N inequalities in N variables, where each inequality cuts off exactly one integral value. For this problem, the cutting plane proof of unsatisfiability requires length at least $2^N/N - 1$ cuts. The lower bound is

based on the number of variables and not on the size of the input. Of course, for an input problem of size $O(2^N)$, the number of steps required just to read the problem is exponential in N. This result does not apply to the complexity of SAT, because the complexity of SAT must be stated in terms of the size of the input. We should also note that the proof depends critically on allowing only two inequalities to be added together at each step.

2.10.2 Lower Bounds by Interpolation

Krajíček (1997) gives version of the propositional Craig interpolation theorem for cut-free sequent calculus, shows the existence of small monotone interpolations for certain non-monotone propositions, and uses previous results on the size of monotone circuits to show lower bounds for several proof systems including a restricted form of cutting plane proofs.

The interpolation theorem states that every implication $A(p,q) \longrightarrow B(p,r)$ with $p = (p_1, ..., p_n)$ occurring in both A and B, $q = (q_1, ..., q_s)$ occurring only in A, and $r = (r_1, ..., r_t)$ occurring only in B, has an interpolant $C(p)$ such that $A(p,q) \longrightarrow C(p)$ and $C(p) \longrightarrow B(p,r)$.

The idea of monotonicity appears in each of several representations of propositional logic. A Boolean formula is monotone if it contains no NOT operators. A CNF formula is monotone if it contains no negated literals. A logic circuit is monotone if it contains no NOT gates. Translated into pseudo-Boolean inequalities, a formula is monotone if the set of inequalities contains no negative coefficients.

Krajíček constructs a non-monotone unsatisfiable formula $A(p,q) \longrightarrow B(p,r)$ in which the common variables $p = (p_1, ..., p_n)$ appear only positively, and for which there is a monotone interpolant. Because the interpolant is monotone, any proof of that

interpolant can be simulated by a monotone circuit. Using a result by Rasborov, that monotone circuits must have exponential size for certain formula, Krajíček concludes that any CP proof must of the interpolant must have exponential size.

The particular version of cutting plane proof system used by Krajíček is the one defined by Cook et al. (1987), allowing only bounded coefficients and Chvátal cuts. The key to Krajíček lower bound proof for CP proof system is that the formula $A(p, q)$ has no negative coefficients, and the Chvátal cuts cannot introduce negative coefficients. Because all formulas have no negative coefficients, they can be simulated by monotone circuits. If negative coefficients were present, the simulation would fail.

Pudlák (1997) extends the results of Krajíček (1997) to provide lower bounds for CP proofs with unbounded coefficients. Pudlák uses the same monotone interpolation theorem, and a very similar construction. The lower bound is based on an exponential lower bound on the size of a monotone real circuit computing the clique function, and an implication that has the clique function as an interpolant. The result is proved by defining monotone circuits over the domain of real numbers R^n , rather than B^n. After extending Razborov's result to this new class of monotone circuits, the lower bound on CP proofs with real coefficients follows immediately.

2.10.3 Graph Structure of Cutting Plane Proofs

Bonet et al. (1998) give a separation theorem that separates tree-like proofs from dag-like proofs for cutting plane proofs. The difference is, of course, that in a tree-like proof each line of the proof can be used only once in the subsequent steps of the proof. There is a problem for which every tree-like cutting-plane refutation requires exponential size, but a dag-like refutation has polynomial size. Hence, cutting-plane proof systems that allow dag-like proofs are more efficient than those that allow only

tree-like proofs. This suggests that a proof system should not discard an inequality after it has been used to generate another inequality.

2.10.4 Summary

Any SAT algorithm must implicitly construct proofs of unsatisfiability. Various formalizations of propositional logic include different sets of axioms and inference rules, but are equivalent in the sense that they admit proofs of the same theorems. The length or size of a proof, as a function of the size of the theorem, provides a lower bound for the complexity of any algorithm to construct that proof. Various proof systems based on search or resolution have been shown to require exponentially long proofs for certain theorems.

Lower bounds are known for some restricted forms of cutting plane proof systems. Several restricted cutting plane proof systems have been studied.

Cutting-plane proof systems allow only positive integral multipliers, to maintain the orientation of the inequalities. Those cutting plane proof systems are subject to a lower bound derived using monotone interpolations (Krajíček, 1997; Pudlák, 1997). A critical point in each lower bound proof is that any derivation using only positive multipliers cannot introduce negative coefficients into a problem that contains only non-negative coefficients. The absence of negative coefficients allows the proofs to be simulated by monotone circuits. The simulation by monotone circuits is critical to the proof of the lower bound in each case.

The lower bound using monotone circuits affects only cutting plane proof systems that are not capable of introducing negative coefficients. The usual formulations of cutting plane proof systems allow only positive multipliers, to maintain the inequalities. This appears to be an innocuous restriction. No lower bounds are known for

cutting plane proof systems that can introduce negative coefficients. However, no such cutting plane proof systems have been proposed.

2.11 Positive Results for Cutting Planes

Several positive theoretical results have been reported for cutting plane algorithms. Peter Barth used Chvátal cutting planes in an algorithm that can solve pigeonhole problems efficiently (Barth, 1993, 1994, 1996). Clote (1995) showed that cutting plane proofs can polynomially simulate general Frege proofs with unbounded term depth. Hence, cutting plane proofs are as powerful as general Frege proofs. Bockmayr and Eisenbrand (1997) showed that the Chvátal rank of the polytope of an integer program representing an unsatisfiable SAT is surprisingly small. For a problem with n variables, only n iterations of cutting plane algorithm are required, if every possible cut is generated at each iteration.

The literature on computational experience using Gomory cutting planes is very scant. The reputation for inefficiency does not seem to be backed by very many papers reporting actual computational experience. A survey of those empirical results appears in the introduction by Balas et al. (1996), including encouraging results for the set-cover problem, which is very closely related to satisfiability. Balas et al. (1996) also reports a recent set of experiments using Gomory cutting planes to prune a search tree for mixed-integer programming problems.

2.11.1 Fast Solution of Pigeonhole Problems

Peter Barth used extended clauses of the form $L_1 + \ldots + L_n \geq d$, where $0 \leq d \leq n+1$, to demonstrate an algorithm that can solve pigeonhole problems efficiently (Barth, 1993, 1994, 1996). Each extended clause may be written also as a linear inequality of

the form $\sum_{i \in I} x_i + \sum_{j \in J}(1 - x_j) \geq d$, where $0 \leq d \leq n+1$. Hence, methods developed for linear integer programming are applicable. A set of extended clauses is said to *dominate* an extended clause if the extension of the set is a subset of the extension of the extended clause. Barth gives an easy decision predicate for the dominance relation between extended clauses. $L \geq d$ dominates $L' \geq d'$ iff $|L \setminus L'| \leq d - d'$. If $L \geq d$ dominates $L' \geq d'$, then $L \geq d$ implies $L' \geq d'$, and so the second constraint is redundant and can be removed.

To solve the pigeonhole problem, Barth used a family of cutting planes called diagonal sums to convert a pigeonhole problem into a linear programming problems for which no feasible solution exists. The algorithm uses sets of inequalities which appear in the problem and have a certain combinatorial property. After adding the inequalities in such a set, a Chvátal cut of the sum gives the diagonal sum cut. The result of a diagonal sum cut is a new inequality that is satisfied when all-but-one of the literals is true.

In the solution of the pigeonhole problem, each new extended clause is added to the problem only if it dominates at least one old extended clause, which is removed. Hence, the size of the problem does not increase with the addition of new extended clauses. Barth's algorithm generates all available diagonal-sum cuts before the resulting problem is solved as a linear program. The number of diagonal-sum constraints that must be generated to solve a pigeonhole problem is polynomial in the dimension of the problem.

Barth's results build on previous work by Cornuéjols and Sassano (1989), who identified necessary and sufficient conditions for an extended clause to be a facet-defining inequality for the set cover problem. A set covering problem $Ax \geq 1$ defines a bipartite graph (V, U, E) in a natural way: there is an edge between $v_i \in V$ and $u_j \in U$ if the variable v_i appears in the clause u_j. A cutset induced by a set of

variables $S \subseteq U$ is the set of clauses that contain at least one variable in S and one variable in $\bar{S} = V \setminus S$. Cornuéjols and Sassano showed that an inequality $\sum_{j \in S} x_j \geq b$ is a facet-defining equality if for every $j \notin S$, the cutset induced by $S \cup \{x_j\}$ has a certain counting property.

2.11.2 Simulation Results

Goerdt (1991) showed that Frege proof systems can polynomially simulate cutting plane proof systems, showing that cutting plane proofs are not more powerful than Frege proofs. Clote (1995) provides the reverse direction, that cutting plane proofs can polynomially simulate general Frege proofs with unbounded term depth, using a slightly extended cutting plane proof system, CP^+.

The main result of Clote (1995) is that CP^+ polynomially-simulates Frege proofs Frege proof systems. This proof uses a result of Cook and Reckhow (1979) that any Frege system simulates any other Frege system with at most a polynomial increase in proof size. Hence, it is only necessary to show that CP^+ polynomially simulates one particular Frege proof system. For this, Clote uses a particular Frege proof system $\mathcal{F}$. $\mathcal{F}$ has propositional variables x_i, terms defined using the logical connectives $\not\wedge\vee$. The symbol $\neg$ denotes the *not* function, and the symbol $\vee$ denotes the *or* function. Of course, the and function can be simulate because $x_i \wedge x_j = \not(x_i \vee x_j)$, and so the two function symbols provide a basis for propositional logic. $\mathcal{F}$ has four rules of inference:

Contraction

$$\frac{A \vee A}{A} \tag{2.57}$$

Expansion

$$\frac{A}{B \vee A} \tag{2.58}$$

Associativity

$$\frac{A \vee (B \vee C)}{(A \vee B) \vee C} \tag{2.59}$$

Cut

$$\frac{A \vee B \qquad \neg A \vee C}{B \vee C} \tag{2.60}$$

Clote then gives the definition of a Frege proof as follows:

> A Frege proof is a sequence $A_1, \ldots, A_n$ such that for each $1 \leq i \leq n$ either A_i is a substitution instance of an axiom, or there exist $j, k < i$ such that A_k is deduced from A_j, A_k by the application of a rule of inference.

(It appears that Clote's definition contains a typographical error. Clearly, A_i, not A_k, should be deduced from A_j, A_k by the application of a rule of inference) Assuming this correction, Clote's proof simply shows that the operation of each inference rule can be simulated by a sequence of valid inequalities, such that the last valid inequality is the consequent of the inference rule.

Clote gives two distinct proofs that CP^+ can polynomially simulate Frege proof systems: one showing that cut elimination can be simulated; the other by giving a direct translation of any Frege proof into CP^+. Clote also shows that cutting plane proofs are strictly more powerful than resolution, in the sense that cutting planes proof systems allow proofs that are shorter by an exponential factor. Results on propositions that require exponential size resolution and constant-depth Frege proofs, but admit polynomial size Frege proofs and cutting-plane proofs, include propositional representations of Ramsey theorems and the pigeonhole theorems.

2.11.3 Chvátal Rank of Integer Hull

Early cutting plane algorithms were inefficient, and so cutting plane algorithms were largely abandoned for integer programming, in favor of branch-and-bound algorithms. This was justified by the empirical observation that cutting plane algorithms seemed to require a very large number of cuts, and the tableau tended to grow beyond the available memory. There are now some reasons to suspect that the number of cutting planes required to solve an satisfiability problem may not be very large.

The Chvátal rank of a polyhedron $P = \{x \in R^N | Ax \leq b\}$, was introduced by Chvátal (1973) to measure the compexity of integer linear programs. Let P' denote the polytope satisfying all cutting planes that can be derived directly from P. Let $P^{(0)} = P$, $P^{(I+1)} = (P^{(I)})'$. The Chvátal rank of P is the smallest k such that $P^{(k)} = P_I$, where P_I is the integer hull of P.

Recently, Bockmayr and Eisenbrand (1997) has showed that the Chvátal rank of the polytope of an integer program representing an unsatisfiable SAT is surprisingly small. They give the following lemma:

Theorem 4 (Bockmayr and Eisenbrand 1997). *The Chvátal rank of polytopes $P \subseteq [0,1]^n$ with $P_I = \emptyset$ is at most n.*

Since the polytope of a linear program in N variables has dimension at most N, the Chvátal rank of the linear program representing an unsatisfiable SAT problem in N variables is at most N. This result does not indicate that only N cutting planes are required. This results does indicate that only N generations of cutting planes are required, where at each generation all possible cutting planes are generated. The theorem limits the tree-height of the proof tree, rather than the size of the proof tree.

2.11.4 Summary

Taken together, Goerdt (1991) and Clote (1995) provide a bi-simulation result that the Frege proof system and the CP^+ proof systems are polynomially equivalent. Clote's CP^+ proof system is an extension of Goerdt's CP proof system, so Goerdt's proof that CP p-simulates Frege also shows that CP^+ also p-simulates Frege. The bi-simulation result shows that $CP+$ proof systems are polynomially equivalent to Frege proof systems, in that any proof in either system can be translated to an equivalent proof in the other, with at most a polynomial increase of the proof size.

The lower bound results for cutting plane proofs Krajíček (1997) and Pudlák (1997) seem at odds with the simulation results of Goerdt (1991) and Clote (1995). The lower bounds results depend on having monotone formula within the problem, such that the proof of the monotone subproblem must be monotone. The proof systems forbid the introduction of negated coefficients for monotone variables. Lower bounds for monotone circuits then apply to a circuit that simulates the proof.

The results of Clote, Goerdt and Barth are particularly important when viewed in the light of Cook and Reckhow (1974). Cook showed that Frege systems are polynomially equivalent to several very powerful proof systems, including natural deduction. Clote showed that cutting-plane systems can polynomially simulate Frege systems. Barth provided an example of this power, by showing that cutting plane proofs allow short proofs of unsatisfiability for pigeonhole problems. Hence, cutting-plane proof systems are very powerful propositional proof systems, that also allow a nice regular tabular format.

2.12 Summary

Many different algorithms have been proposed for the SAT problem. Tree search algorithms use reasoning to avoid searching some some sub-trees, but must still search an exponentially sized portion of the complete tree. Algebraic rewriting cannot solve SAT efficiently, because there can be no compact canonical form. Approaches for some special subclasses of SAT are efficient, but are only applicable to a vanishingly small fraction of all problems. A wide variety of heuristic or randomized search methods have been tried, but cannot provide complete proof of unsatisfiability. Linear relaxations allow the application of integer programming techniques, but the common branch-and-bound techniques for integer programming are essentially just tree-search algorithms.

Lower bounds results show that algorithms based on weak proof systems such as resolution cannot be efficient for SAT. Specific problems such as the pigeonhole problems have been found that require exponential size resolution proofs, but allow short Frege proofs. Specific algorithms have been constructed to show that Frege and cutting-plane proof systems allow short proofs of pigeonhole problems. The results on pigeonhole problems show a sequence of proof systems and proof-construction algorithms. Each algorithm is specifically designed to take advantage of the structure of the pigeonhole problem. Other hard problems clearly do exist, for which these known algorithms do not find short proofs. Unspecialized algorithms that find short proofs of various problems are conspicuously absent from the literature in this area.

The relative lack of literature on cutting plane algorithms for SAT indicates that the cutting-plane approach has not been fully explored. Historically, this appears to be because early cutting-plane algorithms for integer programming were observed to be slower than early tree-search algorithms, and so very little research has been

concentrated on cutting-plane algorithms. The lower-bounds on cutting plane proof systems depend on details of the definitions. It might be possible to give a different definition that avoids the lower-bound. We also have some evidence that cutting-plane algorithms are polynomially equivalent to Frege systems, and that strong lower bounds cannot be proved for Frege systems using current methods. The apparent contradictions in the literature need to be resolved. For all of these reasons, the cutting-plane approach to SAT needs further research.

Some formalization of cutting plane proofs that allows introduction of negative coefficients, and also allows dag-like proofs, is clearly needed. Such a definition would avoid the known lower bounds that apply to the existing cutting plane proof systems. An open theoretical problem is to show that a short refutation exists for every false proposition, or that short refutations cannot exist for some particular class of propositions. A related practical problem is to find some algorithm for constructing a short cutting plane refutation when one does exist for a given proposition. Even if some SAT problems cannot be solved efficiently, an algorithm that finds short refutations for a large class of propositions would be of practical use. A number of minor open questions also exist for cutting plane algorithms, such as how to construct strong cutting planes, which heuristics are most effective, and how to most efficiently implement the algorithms. The abundance of open questions indicate that the cutting-plane approach to SAT should be fertile ground for further research.

Chapter 3

A New SAT Algorithm

In this chapter, we will develop the theory of a new complete algorithm for satisfiability. The design of the algorithm is based on the conditions that are known to be necessary if a complete SAT algorithm is to be efficient. In chapter 2, we identified a number of papers showing that certain conditions must be fulfilled by any efficient algorithm. In this chapter, we will give an algorithm and show that it fulfills each of those necessary conditions.

The new algorithm is based on extended cutting plane proofs, which use slack variables to express the constraints as equalities. The algorithm uses Gomory cutting planes and a primal-dual simplex method, and so is somewhat similar to some cutting plane methods for integer programming. Several sections of this chapter develop algorithms to find cutting planes, and show that various necessary conditions are satisfied by these classes of cutting planes.

Section 3.1 studies the convex hull of the integer solutions of SAT, and gives a describes the facet inequalities of the SAT polytope. Various authors have studied methods for deriving facet inequalities using canonical hyperplanes for various problems, but have not fully described the integer hull of SAT (Sassano, 1989; Cornuéjols

& Sassano, 1989; Hooker, 1992; Barth, 1993, 1996). Balas and Jeroslow (1972) defined a canonical hyperplane on the unit hypercube as a particular form of linear equation, having coefficients $\in \{-1, 0, 1\}$. We show that the convex hull of SAT can be written as a conjunction of canonical hyperplanes.

Section 3.2 shows that cutting plane algorithms using slack variables provides a stronger proof system than the cutting plane proof systems defined by Chvátal (1984), Goerdt (1991), and Clote (1995). We show that a lower-bound on the length of cutting plane proofs, due to Krajíček (1997) and Pudlák (1997), does not hold for cutting plane algorithms with slack variables. It is necessary that any efficient SAT algorithm must avoid that lower bound.

In section 3.3, we develop a theory of lifting for inequalities in the SAT polytope. Lifting gives us a method for deriving strong valid inequalities of an n-dimensional SAT polytope from valid inequalities of lower-dimensional subproblems of that SAT problem. Several of the known necessary conditions are that the algorithm must be able to solve a particular kind of "hard" problem efficiently. The canonical lifting algorithm finds valid inequalities that lead to efficient solution of pigeonhole problems, which were nominated by Cook and Reckhow (1979) as being "hard" for a wide variety of known algorithms. Several authors have given special purpose algorithms that solve pigeonhole problems efficiently (Buss, 1987; Bibel, 1990; Urquhart & Fu, 1996; Barth, 1996).

In section 3.4 we give a new lifting algorithm that can be applied even when some of the variables are not constrained to be Boolean. That is, when some variables may take on integer values, the proof of the canonical lifting lemmas fail. We provide new integer-lifting lemmas and a new lifting algorithm that holds even with integer variables. These algorithms can be applied to Chvátal and Gomory cutting planes, even when the nonbasic variables include some slack variables. Hence, this new

lifting algorithm using cutting planes and slack variables has the potential to avoid the known lower bound on monotone cutting plane proofs (Krajíček, 1997; Pudlák, 1997).

We hypothesize that this new lifting algorithm allows short refutations of some hard unsatisfiable propositions that are not efficiently refuted by previous algorithms. The measure of efficiency in this context should be the number of steps that are required in the refutation proof, as measured by the number of cutting planes.

To use the cutting-plane algorithms based on linear programming relaxations of integer problems, it is necessary to have some algorithms to sequence and control the other algorithms. In sections 3.5 and 3.6, we consider algorithms for choosing the vertex to cut. The choice of vertex determines the cutting planes that are available to be lifted. Modification of the objective function is proposed as one method of choosing a vertex. Local search in the neighborhood of the optimal vertex is also considered. In section 3.7, we provide a cutting-plane strength measure that is independent of any particular objective function, and show that even small difference in the strength of cutting planes may have a large effect on the length of refutation proofs.

3.1 A Characterization of the SAT Polytope

A critical problem in a cutting-plane algorithm is to find tight cuts, so as to minimize the number of cuts that are needed. Facet cuts are cutting planes that are facets of the convex hull of the integer solutions to the problem. In order to find facet cuts, it is essential to have some information about the facets of the integer hull.

A full description of the convex hull of SAT is an open problem. Several authors have characterized subsets of the facets of the closely-related set-cover polytope (Balas & Jeroslow, 1972; Peled, 1977; Sassano, 1989; Cornuéjols & Sassano, 1989; Balas &

Ng, 1989b, 1989a; Nobili & Sassano, 1989). Each characterized some subset of the facets, but no complete characterization is known. Balas and Jeroslow (1972) give a good set of definitions, and an approach which seems to come close.

Balas and Jeroslow considered the convex hull of subsets of the vertices of the unit hypercube, which is exactly the convex hull of SAT. In this section, we will extend the result of Balas and Jeroslow (1972) to obtain a theorem characterizing the convex hull of SAT.

3.1.1 Canonical Hyperplanes

The n-dimensional *unit hypercube* is the set $K = \{x \in R^n | 0 \leq x_j \leq 1, j \in N\}$ where $N = \{1, \ldots, n\}$. A *vertex* of K is a point $x \in K$ such that exactly n of the inequalities are tight. Let V be the set of vertices. Two vertices $x, y \in V$ are *adjacent* if they differ in exactly one component.

Let $x, y \in V$ be adjacent vertices, then $[x, y] = \{z \in R^n | z = \lambda x + (1 - \lambda)y, 0 \leq \lambda \leq 1\}$ is an *edge* of K, and $(x, y) = \{z \in R^n | z = \lambda x + (1 - \lambda)y, 0 < \lambda < 1\}$ is an *open edge* of K. An edge is sometimes called a *closed edge* to emphasize that it is not an open edge. Two distinct edges of K are *adjacent* if and only if they have a common endpoint.

A *k-dimensional face* F^k of K is a set of points $x \in K$ such that exactly $n-k$ of the $2n$ inequalities (1) are tight. A k-dimensional face F^k can be written as a conjunction of $n-k$ literals. Let F be a k-dimensional face. let $N(F)^+ = \{j \in N | \forall x \in F \cap V, x_j = 1\}$, $N(F)^- = \{j \in N | \forall x \in F \cap V, x_j = 0\}$, and $N(F)^0 = N - N(F)^+ - N(F)^-$. Then a k-dimensional face F^k can be written as:

$$(\bigwedge_{i \in N(F)^+} x_i) \wedge (\bigwedge_{i \in N(F)^-} \overline{x_i})$$

Using DeMorgans law, a vertex $v \in V$ is on a k-dimensional face if the disjunction

of the negations of those literals is false. Hence, the equation of a k-dimensional face may be written in a linear form:

$$\sum_{i \in N(F)^+} (1 - x_i) + \sum_{i \in N(F)^-} x_i = 0$$

Two distinct k-dimensional faces of K are *adjacent* if and only if they have a common (k-1)-dimensional face. The *edge distance* $d(x, y)$ between two vertices $x, y \in V$ is the number of indices such $k \in N$ such that $x_k \neq y_k$. The *edge distance* $d(x, F)$ between a vertex x and a k-dimensional face F of K is $d(x, F) = \min(\{d(x, y) | y \in F \cap V\})$. The euclidean distance between two vertices $x, y \in V$ is just the square root of the edge distance, and that the edge-distance satisfies the axioms that are expected of a distance measure.

Using the definitions of a k-dimensional face and the edge distance, they then defined a canonical hyperplane as the set of vertices that are all at some fixed distance d from a given face. A *canonical hyperplane* associated with a face F and denoted $H(F)_d$ is defined by:

$$\sum_{i \in N(F)^+} (1 - x_i) + \sum_{i \in N(F)^-} x_i = d$$

for a constant d. The *order* of a canonical hyperplane $H(F)_d$ is the number of zero coefficients in the equation, $|N(F)^0|$. A *canonical inequality* is a half-space bounded by a canonical hyperplane:

$$\sum_{i \in N(F)^+} (1 - x_i) + \sum_{i \in N(F)^-} x_i \geq d$$

3.1.2 Canonical Facets

Balas and Jeroslow showed that canonical hyperplanes have a property in common with the convex hulls of vertices in the unit hypercube. In particular, If $V' \subset V$ is a subset of the vertices V of the unit hypercube, and $C(V')$ is the convex hull of V',

then $C(V')$ contains a point of an open edge (x, y) if and only if it contains the whole edge $[x, y]$. Similarly, a canonical hyperplane H contains a point of an open edge (x, y) of K if and only if it contains the whole edge $[x, y]$.

This common property suggests that the facets of the convex hull of SAT might be closely related to canonical hyperplanes. In particular, it suggests that the facets of the convex hull might be canonical defined by canonical inequalities. It turns out that this is not quite the case. In this section, we prove that the convex hull of any subset of the vertices of the unit hypercube can be written as canonical inequalities.

Recall that a facet of an n-dimensional polytope is an $n-1$-dimensional polytope having n affinely independent exact solutions. Several authors have studied the facets of the SAT and set-cover polytopes. Balas and Jeroslow identified that some canonical hyperplanes can be facets of the SAT polytope. The others characterized other classes of facets, on the assumption that a complete characterization of all facets would give a complete characterization of the convex hull.

The hyperplanes that contain n affinely independent integer points constitutes the set of hyperplanes that might be facets of some SAT problem. Peled (1977) found that there may exist some facets of the SAT polytope that are not canonical hyperplanes. Consider, for example:

$$2x_1 + 2x_2 + x_3 + x_4 = 3 \tag{3.1}$$

Equation (3.1) possesses four exact integer solutions that are affinely independent, and hence is a facet of some four-dimensional polytope. However, the intersection of $2x_1 + 2x_2 + x_3 + x_4 \geq 3$ with the unit hypercube cube possesses non-integral vertices (e.g. $(\frac{1}{2}, \frac{1}{2}, 1, 0)$), and hence could not be the convex hull of the integer solutions. For any SAT polytope having equation (3.1) as a facet, other facet inequalities must exist that eliminate the non-integral vertices.

It is easy to see that the inequality (3.1) can be expressed as a linear combination of canonical inequalities:

$$x_1 + x_2 \geq 1 \tag{3.2}$$

$$x_3 + x_4 \geq 1 \tag{3.3}$$

Two times (3.2) plus (3.3) gives the inequality (3.1). It is obvious that every facet inequality can be expressed as a linear combination of canonical inequalities, because every integer is the sum of some number of ones.

What is not obvious is that every facet inequality can be expressed as a linear combination of canonical facet inequalities. In theorem 5, we will assert that the convex hull may be written as a set of canonical inequalities, and that every facet inequality can be expressed as a linear combination of canonical facet inequalities. This is so, even though there may also exist some non-canonical inequalities that are facets of the convex hull.

Theorem 5. *The convex integer hull of SAT can be written as a conjunction of canonical inequalities.*

Proof. We need only show that, given any non-integer vertex of the polytope of a linear program, we can find a separating inequality that is a canonical inequality. Let X be a non-integer basic solution of a linear program, with a basic variable x_b and nonbasic variables x_j, for $j \in J$. Let

$$x_b + \sum_{j \in J} a_j x_j = b$$

be one of the equations in the solution of the linear integer program, where b is non-integer. By Gomory's method (Gomory, 1963), we obtain the separating cutting

plane:

$$\sum_{j \in J} f(a_j) x_j \geq f(b) \tag{3.4}$$

where $f(y) = y - \lfloor y \rfloor$ is just the fractional part of y. The inequality (3.4) is a valid inequality for the linear integer program, and the solution X does not satisfy (3.4). Taking the Chvátal cut of (3.4), we obtain the inequality:

$$\sum_{j \in J} \lceil f(a_j) \rceil x_j \geq \lceil f(b) \rceil \tag{3.5}$$

Now, inequality (3.5) is a canonical inequality, and is a valid inequality for the integer program. Further, the non-integer solution X is infeasible for (3.5), because every variable appearing in (3.5) is nonbasic in X. Hence, the valid inequality (3.5) separates the non-integer solution X from the convex hull of the integer program. This derivation was done for an arbitrary non-integer basic solution X, and the argument is valid for every particular non-integer basic solution. Hence, every non-integer basic solution of a linear program may be separated from the convex hull by a canonical inequality. By Chvátal's theorem, a finite number of such cutting planes is sufficient. Hence, it is possible to write the convex hull as a conjunction of canonical inequalities. □

Rather than characterizing all possible facets of SAT, we have characterized all possible SAT polytopes. Theorem 5 tells us that it is always possible to write the convex hull of a SAT problem as a conjunction of canonical inequalities. Other valid inequalities may exist, and some of those other valid inequalities may be facet inequalities. It follows immediately that every facet inequality can can be written as a linear combination of canonical facet inequalities.

Lemma 4. *Every facet inequality of the SAT polytope can be written as a canonical facet inequality.*

Proof. This follows immediately from theorem 5. □

The procedure used in the proof of theorem 5 to construct a separating canonical inequality gives weak cutting planes. Using only this method of finding cutting planes, a cutting plane algorithm may require a large number of cuts to solve the linear integer program. In the sequel we will give other methods for deriving canonical cutting planes that construct stronger cuts.

3.1.3 Canonical Polytopes

In general, a polytope can be described by as system of linear equations $Ax \leq B$. The linear relaxation of a satisfiability problem gives a polytope in the 0/1 cube. The linear relaxation of any given SAT problem can be written in the form:

$$\begin{aligned} \sum_i x_i + \sum_j (1 - x_j) \geq 1 \qquad & \text{, for each clause } \bigvee_i x_i \quad \vee \quad \bigvee_j \overline{x_j} \\ 0 \leq x_i \leq 1 \qquad & \text{, for each variable } x_i \end{aligned} \tag{3.6}$$

By theorem 5, the convex hull of the satisfying solutions can be written using only canonical inequalities. Hence, the convex hull of the satisfying solutions of a satisfiability problem can be written in the form:

$$\begin{aligned} \sum_j a_{ij} x_j \geq b_i \text{ , for each constraint index } i \\ 0 \leq x_j \leq 1 \qquad \text{, for each variable index } j \end{aligned} \tag{3.7}$$

where each $a_j \in \{-1, 0, 1\}$ and each $b \in \mathbb{I}$.

Assuming that we can find a suitable set of canonical inequalities, the solution to a satisfiability problem can be computed by a linear program using only canonical inequalities as constraints. That is a rather strong assumption. A major difficulty is that often the complete description of the convex hull requires an impractically large set of inequalities.

Because the coefficients a_j are all in $\{-1, 0, 1\}$, the canonical hyperplanes of the convex hull have a nice property. The intersection of a canonical hyperplane with a two-dimensional face of the unit hypercube is either parallel to one of the dimensional axis, or diagonal to both of them. Further, because the constant b is integer, the intercept of the hyperplane with each axis occurs at an integer point on the axis. Each canonical hyperplane intercepts some corners of the unit hypercube, and does not intercept the edges other than at the corners.

For convenience, we will define a canonical polytope as any polytope determined by an intersection of canonical half spaces. It is clear that the polytope (3.6) of every SAT problem is a canonical polytope. Any constraint of (3.6) that includes both x_i and $-x_1$ for any i can be discarded, because every $x \in [0, 1]^n$ satisfies every such constraint. Each remaining constraint of the SAT problem can be written as a canonical inequality.

3.2 Avoiding a Lower Bound

Krajíček (1997) and Pudlák (1997) showed lower bounds on the proof length of cutting plane proofs, based on known lower bounds on the size of monotone circuits. The lower bounds proofs depend critically on a lemma that a cutting plane proof system cannot introduce a negative coefficient for a variable if the coefficients of all previous occurrences of that variable are non-negative. That lemma in turn depends critically on using a particular definition of cutting plane proof system, in which a line of a proof is an inequality. If a cutting plane proof has only non-negative (resp. non-positive) coefficients, then that proof may be simulated by a monotone circuit in a straight forward manner. Using that simulation, the lower bounds on the monotone circuit size imply lower bounds on the proof length for the proofs of certain propositions.

For a proposition having only non-negative (resp. non-positive) coefficients, and using a proof system that cannot introduce a negative (resp. positive) coefficient into a proof, every proof of that proposition must have only non-negative (resp. non-positive) coefficients. If a proof system can introduce a negative (resp. positive) coefficient into a proof, then the lower bound result does not hold for that proof system. In this section we show that cutting plane algorithms used in integer linear programming are not subject to the monotone lower bounds of Krajíček and Pudlák because those cutting plane algorithms can introduce a negative (resp. positive) coefficient into a proof.

For cutting plane proofs, as defined by Chvátal (1984), Goerdt (1991), and Clote (1995) the lower bound does hold because those proof systems cannot introduce a negative coefficient. In those proof systems, proof lines are inequalities and may be derived by two basic operations: a prior inequality may be multiplied by positive numbers; and multiple prior inequalities may be added together. Neither method of producing a new inequality can produce a new negative coefficient.

Cutting plane algorithms for integer programming use slack variables to convert the inequalities to equalities in a higher-dimensional vector space. Because the algorithms are working with equalities, row operations from Gaussian elimination may be applied. In particular, an equality may be multiplied by a negative value, and one row may be subtracted from another. Hence, cutting plane algorithms may introduce a negative (resp. positive) coefficient to a sequence of tableaus in which all previous occurrences of that variable have non-negative (resp. non-positive) coefficients.

3.2.1 An Example

We will now give an example showing how a cutting plane algorithm can introduce a negative coefficient. This will show that a cutting plane algorithm that uses slack variables can generate a constraint with a negative coefficient in a proof where all previous occurrences of that variable have positive coefficients.

We start the example by supposing that an equality constraint with no negative coefficients exists in a linear program.

$$S_1 + \frac{1}{2}x_1 + \frac{3}{2}x_2 = \frac{1}{2} \tag{3.8}$$

The coefficient of the basis variable S_1 is 1, so this equality constraint represents the inequality $\frac{1}{2}x_1 + \frac{3}{2}x_2 \leq \frac{1}{2}$. From this equality constraint, we generate a Gomory cutting plane.

$$\left(\left\lfloor\frac{1}{2}\right\rfloor - \frac{1}{2}\right)x_1 + \left(\left\lfloor\frac{3}{2}\right\rfloor - \frac{3}{2}\right)x_2 \leq \left(\left\lfloor\frac{1}{2}\right\rfloor - \frac{1}{2}\right) \tag{3.9}$$

$$-\frac{1}{2}x_1 - \frac{1}{2}x_2 \leq -\frac{1}{2} \tag{3.10}$$

Converting the cutting plane to an equality requires a new slack variable, S_2. The resulting equality is then be added to the problem, with the new slack variable as a basic variable. We then have the two constraints:

$$S_1 + \frac{1}{2}x_1 + \frac{3}{2}x_2 = \frac{1}{2} \tag{3.11}$$

$$S_2 - \frac{1}{2}x_1 - \frac{1}{2}x_2 = -\frac{1}{2} \tag{3.12}$$

We have shown that generating a Gomory cut can introduce negative coefficients. This is because generating the Gomory cutting plane requires subtracting the equation from the inequality.

Traditional cutting plane proofs use Chvátal cuts, rather than Gomory cuts, and do not allow the step of subtracting one inequality from another. To generate Gomory

cutting plane inequalities, it is necessary to have an equality rather than an inequality as the starting point. The conversion of each inequality to an equality requires the use of a slack variable. Slack variable are also not allowed in traditional cutting plane proofs.

Negative coefficients may also be generated by pivot operations. Suppose, we pivot on the $-\frac{1}{2}x_2$ in the new cutting plane. The pivot operation yields:

$$S_1 + 3S_2 - x_1 = -1 \tag{3.13}$$

$$-2S_2 + x_1 + x_2 = 1 \tag{3.14}$$

In this case, two new negative coefficients are generated in the first constraint. Again, the negative coefficients occur when a subtraction step is performed.

We have demonstrated that introduction of Gomory cutting planes with slack variables can generate negative coefficients, and that pivot operations can also generate negative coefficients. Hence, a necessary condition in the lower bounds proofs of Krajíček and Pudlák does not hold, and so the lower bound result does not hold for cutting plane algorithms.

Chvátal (1984) showed that Chvátal cuts are as powerful as Gomory cuts, in the sense that all of the same theorems can be proved. However, Chvátal 's proof says nothing about the length of the proofs. The results of Krajíček and Pudlák exhibit lower bounds on the proof length of cutting plane proofs, if slack variables are not used to express the constraints as equality constraints. Those results use Chvátal cutting planes. With the addition of slack variables, Gomory cutting planes may be used. As we have seen, the lower bounds results of Krajíček and Pudlák do not hold if we use slack variables to express constraints as equalities.

3.3 Canonical Lifting

The facets of polytopes for certain 0-1 programming problems are related to the facets of certain subproblems. The subproblems are called minors. Geometrically, a minor of a polytope is a cross-section of the polytope obtained by fixing values for some of the variables. The early results in this area were given by Padberg (1975) and Wolsey (1976).

Peled (1977) gives a result that each inequality that is a *facet* of a minor of a set-packing problem can be lifted to obtain a facet of the full set-packing problem. Nobili and Sassano (1989) gives a similar result for the set-cover problem. Several other authors give similar definitions and related results for both problems (Zemel, 1978; Balas & Zemel, 1984; Balas & Ng, 1989b, 1989a; Balas, 1990; Gu, Nemhauser, & Savelsbergh, 1995; Plaza, 1996). In this section, we will translate those results to obtain a lifting lemma for the SAT problem.

The set-cover problem is closely related to the SAT problem. By introducing a new variables $z_i = \overline{x_i} = (1 - x_i)$ for each 0-1 variable x_i in SAT, and adding constraints $x_i + z_i \geq 1$, any given SAT problem may be stated as a set-cover problem. The variables of the set-cover problem correspond to the literals of the SAT problem. The SAT problem in n variables is satisfiable if there is a cover having at most n nonzero variables. Hence, the translation of the lifting lemmas to the context of the SAT problem is not difficult.

3.3.1 Minors of a SAT Problem

In this section, we define a particular type of subproblem of SAT called a *minor* subproblem. A minor of a SAT problem is constructed by fixing some subset of the variables to 0, fixing another subset of the variables to 1, discarding any of the

clauses that are satisfied by any of the fixed variables, and finally discarding the fixed variables.

Definition 4. *Let $P = \{x : Ax \geq B\}$ be the polytope of an SAT problem. Let $N = \{0, \ldots, n\}$ index the variables, $F \subset N$, $T \subset N$, and $N' \subset N$ be subsets of the index set N such that $F \cap T = \emptyset$ and $N' = N \setminus (F \cap T)$. Then the* minor P_F^T *of P is the polytope $\{x : Ax \geq P, x_j = 0$ for $j \in F, x_j = 1$ for $j \in T\}$.*

If S is the set of feasible solutions of some 0-1 programming problem, S_F^T can be naturally interpreted as a subproblem. In the SAT interpretation, the clauses that contain literals in T are satisfied by setting those literals true, and the literals in F are removed from the problem by setting those literals false. The minor of a SAT problem is a projection of the n-dimensional polytope onto an $(n - |F \cup T|)$-dimensional face of the unit hypercube.

The conditions $Ax \geq B$ are the usual inequalities constructed from the SAT problem as in the formulation (3.6). The projection simply fixes values to some of the variables, leaving a smaller number of unfixed variables. The minor of the SAT polytope is the lower-dimensional polytope of the resulting subproblem.

3.3.2 The Lifting Lemmas

A lifting lemma gives that for each facet of a minor sub-problem, a facet of the larger problem exists. The sequential lifting lemma gives that there is a sequence of minor sub-problems, such that the variables may be added one at a time rather than all at once.

Let P be the polytope of a satisfiability problem with variable index set N. Let

P_F^T be a minor of P and assume that the inequality

$$\sum_{v \in N'} a_v x_v \geq a_0 \tag{3.15}$$

is a valid inequality for P_F^T.

We may assume without loss of generality that $a_v \geq 0$ for all v in (3.15). For any $a_v < 0$, we use a change of variables based on the equality $\overline{x}_v = 1 - x_v$ ($x_v = 1 - \overline{x}_v$). That is, we complement the variables that have negative coefficients, to obtain an equivalent symmetric mirror-image polytope having all non-negative coefficients in the selected valid inequality.

For each $j \in T$, we know that the inequality

$$\sum_{v \in N'} a_v x_v + a_j x_j \geq a_0 + a_j \tag{3.16}$$

is a valid inequality for P_F^T, because the variable x_j is fixed to value 1. It is natural to ask if (3.16) might be valid for $P_F^{T \setminus \{j\}}$. Clearly, (3.16) is valid for $P_F^{T \setminus \{j\}}$ if and only if it is valid for both P_F^T and $P_{F \cup \{j\}}^{T \setminus \{j\}}$. If (3.16) is valid when $x_j = 1$, and also when $x_j = 0$, then it is valid for all values in the domain of x_j. Hence, (3.16) is valid for $P_F^{T \setminus \{j\}}$ if and only if:

$$\min_{x \in P_{F \cup \{j\}}^{T \setminus \{j\}}} (\sum_{v \in N'} a_v x_v) - a_0 \geq a_j \tag{3.17}$$

For the complements of the variables, we have the complement to the same argument. For each $j \in F$, we know that the inequality

$$\sum_{v \in N'} a_v x_v + a_j \overline{x}_j \geq a_0 + a_j \tag{3.18}$$

is a valid inequality for P_F^T, because the variable x_j is fixed to value 0, so the changed variable $\overline{x}$ has value 1. It is natural to ask if (3.18) might be valid for $P_{F \setminus \{j\}}^T$. Clearly, (3.18) is valid for $P_{F \setminus \{j\}}^T$ if and only if it is valid for both P_F^T and $P_{F \setminus \{j\}}^{T \cup \{j\}}$. If (3.18) is

valid when $x_j = 0$, and also when $x_j = 1$, then it is valid for all values in the domain of x_j. Hence, (3.18) is valid for $P^T_{F\setminus\{j\}}$ if and only if:

$$\min_{x \in P^{T\cup\{j\}}_{F\setminus\{j\}}} (\sum_{v\in N'} a_v x_v) - a_0 \geq a_j \tag{3.19}$$

Of course, we want to obtain the strongest possible valid inequality for $P_F^{T\setminus\{j\}}$ (resp. $P^T_{F\setminus\{j\}}$). It seems plausible that the strongest inequality might occur when (3.17) (resp. (3.19)) is satisfied exactly with equality. It also seems plausible that the resulting inequality might be stronger if the starting inequality (3.15) is a facet inequality for P_F^T. These intuitive considerations motivate the the lifting lemmas for the SAT polytope.

Lemma 5 (Canonical Inequality Lifting). *Let P be the polytope of a satisfiability problem with variable index set N. Let P_F^T be a minor of the polytope P and assume that the inequality*

$$\sum_{v\in N'} a_v x_v \geq a_0 \tag{3.20}$$

defines a valid inequality of $Q(P_F^T)$. Then:

1. *For each $j \in T$, and coefficient*

$$a_j = \min_{x \in P^{T\setminus\{j\}}_{F\cup\{j\}}} (\sum_{v\in N'} a_v x_v) - a_0 \tag{3.21}$$

the inequality:

$$\sum_{v\in N'} a_v x_v + a_j x_j \geq a_0 + a_j \tag{3.22}$$

defines a valid inequality of the polytope $Q(P_F^{T\setminus\{j\}})$.

2. *For each $j \in F$, and coefficient*

$$a_j = \min_{x \in P^{T\cup\{j\}}_{F\setminus\{j\}}} (\sum_{v\in N'} a_v x_v) - a_0 \tag{3.23}$$

the inequality:

$$\sum_{v \in N'} a_v x_v - a_j x_j \geq a_0 \tag{3.24}$$

defines a valid inequality of the polytope $Q(P^T_{F\setminus\{j\}})$.

Proof. We give a detailed proof of statement 1. The proof of statement 2 is complementary, with one additional step to note that $\sum_{v\in N'} a_v x_v + a_j \overline{x}_j \geq a_0 + a_j$ implies $\sum_{v\in N'} a_v x_v - a_j x_j \geq a_0$.

Let $a_j = \min_{x \in P^{T\setminus\{j\}}_{F\cup\{j\}}} (\sum_{v\in N'} a_v x_v) - a_0$. If $Q(P) \cap \{x : x_j = 0\} = \emptyset$, then $x_j \geq 1$ is valid for $Q(P^{T\setminus\{j\}}_F)$. In this case, for every a_j:

$$\sum_{v \in N'} a_v x_v \geq a_0 \tag{3.25}$$

$$\sum_{v \in N'} a_v x_v + a_j \geq a_0 + a_j \tag{3.26}$$

$$\sum_{v \in N'} a_v x_v + a_j x_j \geq a_0 + a_j \tag{3.27}$$

Otherwise, $Q(P)\cap\{x : x_j = 0\} \neq \emptyset$. For $x_j = 0$, we have $P^{T\setminus j}_F \cap \{x = 0\} = P^{T\setminus\{j\}}_{F\cup j}$. Then $\sum_{v\in N'} a_v x_v \geq a_0 + a_j$ for a_j such that:

$$\min_{x \in P^{T\setminus\{j\}}_{F\cup\{j\}}} (\sum_{v \in N'} a_v x_v) \geq a_0 + a_j \tag{3.28}$$

or equivalently:

$$a_j \leq \min_{x \in P^{T\setminus\{j\}}_{F\cup\{j\}}} (\sum_{v \in N'} a_v x_v) - a_0 \tag{3.29}$$

Hence, (3.22) is valid for $Q(P^{T\setminus\{j\}}_F)$. □

Lemma 5 gives a method for deriving valid inequalities, using inequalities that are valid for lower-dimension subproblems. To show that the method can be used to derive facet inequalities, we need an additional condition.

Lemma 6 (Sequential Facet Lifting). *Let P be the polytope of a satisfiability problem with variable index set N. Let P_F^T be a minor of the polytope P and assume that the inequality*

$$\sum_{v \in N'} a_v x_v \geq a_0 \tag{3.30}$$

defines a nontrivial facet of $Q(P_F^T)$. Then:

1. *For each $j \in T$, If $a_j = \min_{x \in P_{F \cup \{j\}}^{T \setminus \{j\}}} (\sum_{v \in N'} a_v x_v) - a_0$, the inequality:*

$$\sum_{v \in N'} a_v x_v + a_j x_j \geq a_0 + a_j \tag{3.31}$$

   defines a facet of the polytope $Q(P_F^{T \setminus \{j\}})$.

2. *For each $j \in F$, For $a_j = \min_{x \in P_{F \setminus \{j\}}^{T \cup \{j\}}} (\sum_{v \in N'} a_v x_v) - a_0$, the inequality:*

$$\sum_{v \in N'} a_v x_v - a_j x_j \geq a_0 \tag{3.32}$$

   defines a facet of the polytope $Q(P_{F \setminus \{j\}}^T)$.

Proof. We give a detailed proof of statement 1. The proof of statement 2 is complementary. Because (3.30) is a facet of $Q(P_F^T)$, (3.30) is a valid inequality of $Q(P_F^T)$. Hence, by lemma 5, (3.31) is a valid inequality for $Q(P_F^{T \setminus \{j\}})$. It remains to show that (3.31) defines a facet of the polytope $Q(P_F^{T \setminus \{j\}})$. To show this, we will show the existence of $k+1$ affinely independent vectors that satisfy (3.31) at equality.

Since (3.30) defines a facet of $Q(P_F^T)$, there must be $k = |N'| = |N \setminus (F \cup T)|$ affinely independent vectors x^i for $i \in \{1, \ldots, k\}$ that satisfy (3.30) at equality. Since $x_j^i = 0$, each vector x^i satisfies (3.31) at equality. To construct the one additional affinely independent vector, let $x^* \in Q(P_{F \cup j}^{T \setminus \{j\}})$ with $x_j^* = 1$. For $a_j = \min_{x \in P_{F \cup \{j\}}^{T \setminus \{j\}}} (\sum_{v \in N'} a_v x_v) - a_0$, x^* satisfies (3.31) at equality. Since $x_j^* = 1$,

x^* cannot be written as a combination of the k vectors x^i, so the $k+1$ vectors $\{x^*, x^1, \ldots, x^k\}$ are affinely independent. Hence, there exist $k+1$ affinely independent vectors in $Q(P_F^{T\setminus\{j\}})$ that satisfy (3.31) at equality, so (3.31) defines a facet of the polytope $Q(P_F^{T\setminus\{j\}})$. □

Lemma 7 (Lifting Lemma). *Let P be the polytope of a satisfiability problem with variable index set N. Let P_F^T be a minor of P and let $Q(P_F^T)$ be the convex hull of P_F^T. Suppose that the inequality*

$$\sum_{v\in N'} a_v x_v \geq a_0$$

defines a nontrivial facet of $Q(P_F^T)$ and that $Q(P_F^T)$ is full dimensional. Then there exist coefficients b_v, for $v \in N \setminus N'$ and b_0 such that the inequality

$$\sum_{v\in N'} a_v x_v + \sum_{v\in N\setminus N'} b_v x_v \geq a_0 + b_0$$

defines a facet of $Q(P)$.

Proof. This follows by induction from repeated application of lemma 6. At each application, the cardinality of $N \setminus N'$ is reduced. □

The sequential lifting lemma tells us that given a facet of a proper minor of a SAT problem, we may compute a facet of another minor having one additional variable. The sequential lifting lemma 6 requires the calculation of a minimum $a_j = \min_{x\in P_{F\cup\{j\}}^{T\setminus\{j\}}}(\sum_{v\in N'} a_v x_v) - a_0$ over the 0-1 domain of the variables. Hence, the method requires the solution of an integer program. To compute a facet of the full problem, the needed coefficients are in general not easy to compute.

However, the solution of that integer program is bounded by the solution of the linear relaxation. Given a facet of a minor, if the solution of the minimum (3.21) as a linear program produces a positive value $a_j > 0$, then the minimum integer value of a_j

is at least $\lceil a_j \rceil$. Hence, we may find some lifts without solving integer programming subproblems.

The sequence in which the individual coefficients are computed may be significant. It may be that only one sequence leads to the desired coefficients. Lemma 7 gives us only the existence of some sequence.

Fortunately, we do not need a complete set of facet inequalities. We need only enough strong inequalities so that the solution of the linear program obtains an integer solution. Rather than trying to find only facet inequalities, we settle for finding strong cutting planes that are canonical inequalities. By finding strong cutting planes that are canonical inequalities, we will eventually find the facet inequalities. If the methods for finding strong cutting planes are good, then facet inequalities might be found quickly.

3.3.3 Canonical Lifting on a Polytope

Suppose we have a set of canonical inequalities P that define the polytope of a SAT problem. We do not know a-priori if any one of those inequalities defines a facet of a minor of the problem. Obviously, if we have a canonical inequality

$$\sum_{v \in N'} a_v x_v \geq b \tag{3.33}$$

for some set of variables S, then it is possible that there may exist a minor subproblem P_F^T for which the inequality (3.33) is a facet.

If the inequality (3.33) is a facet of a minor of the SAT problem, then by the sequential lifting lemma there must exist an index $j \notin N'$, a literal $L_j \in \{x_j, (1-x_j)\}$, and a a_j such that

$$\sum_{v \in N'} a_v x_v + a_j L_j \geq b + a_j \tag{3.34}$$

is a facet of one of the minor subproblems $P_F^{T\setminus\{j\}}$ or $P_{F\setminus\{j\}}^T$. The literal L_j may be a positive occurrence of a variable x_j, or a negated occurrence $\overline{x}_j = 1 - x_j$. The two cases of the sequential lifting lemma 6 correspond to the two possibilities.

Suppose for a particular variable x_j, we use an algorithm to find $a_j = \min_{x \in P_{\{j\}}^{\emptyset}} (\sum_{v \in N'} a_v x_v) - a_0$. If $a_j > 0$ then the inequality

$$\sum_{v \in N'} a_v x_v + a_j x_j \geq b + a_j \tag{3.35}$$

is a valid inequality, and dominates the original inequality (3.33). Because (3.35) implies (3.33), we may remove (3.33) from the problem, replacing it with (3.35). As a bonus, if the inequality (3.33) is a facet of $P_{\emptyset}^{\{j\}}$, then the inequality (3.35) is a facet of P.

In the complementary case, suppose for a particular complemented variable $\overline{x}_j = (1 - x_v)$, we use an algorithm to find $a_j = \min_{x \in P_{\emptyset}^{\{j\}}} (\sum_{v \in N'} a_v x_v) - a_0$. If $a_j > 0$ then the inequality

$$\sum_{v \in N'} a_v x_v - a_j x_j \geq b \tag{3.36}$$

is a valid inequality, and dominates the original inequality (3.33). Because (3.36) implies (3.33), we may remove (3.33) from the problem, replacing it with (3.36). As a bonus, if the inequality (3.33) is a facet of $P_{\{j\}}^{\emptyset}$, then the inequality (3.35) is a facet of P.

The polytope $P_{\{j\}}^{\emptyset}$ (resp. $P_{\emptyset}^{\{j\}}$) is just a polytope described by some set of linear inequalities, and the sum of the weights $\sum_{v \in N'} a_v x_v$ is just a linear function. Hence, a linear programming algorithm can be used to find a lower bound for the minimum value of a_j.

The remaining difficulty is to find the good choice of the literal L_j. A nondeterministic algorithm could make every choice for the inequality (3.33), and every choice

for L_j, at each step. However, to simulate such a nondeterministic algorithm on a deterministic machine would require exponential time, and the resulting set of facet inequalities may be exponentially large. Fortunately, the sequential lifting lemma allows us to modify the problem each time a lifted inequality is found, and so a deterministic algorithm is possible.

3.3.4 The Canonical Lifting Algorithm

The abstract presentation of the previous sections described the lifting algorithm on an arbitrary canonical polytope. For a practical implementation, the polytopes is described by a set of linear inequalities. In this section, we will give algorithms that use linear-programming subproblems to obtain strong canonical inequalities. The algorithms are designed to find cutting planes that are canonical inequalities, and that satisfy some necessary conditions that must be satisfied by all facets of the convex hull.

The lifting algorithm lifts a given inequality by the addition of a single literal, to find when one exists a stronger valid inequality that implies the original valid inequality. All of the valid inequalities that are found by Barth's procedure (Barth, 1996, ch. 8) are also found by this new algorithm, which is much less complex than Barth's algorithm. Using the inequalities found by this method, the pigeonhole problems and a variety of hard SAT problems can be solved efficiently.

For a particular literal L_j, canonical lifting requires some means of finding the minimum value of a_j such that (3.34) follows from P. In this section we give an algorithm that uses a linear programming subproblem and a single Chvátal cut to find a lower bound on this value.

Suppose that we are given a set of valid inequalities

$$\sum_{j \in J} a_{ij} x_j \geq b_i \text{ , for each } i \in U \tag{3.37}$$

where all of the variables are restricted to the domain $\{0, 1\}$. The linear relaxation of the domain restriction gives the domain inequalities

$$\begin{aligned} x_j &\geq 0 \qquad \text{, for each } j \\ -x_j &\geq -1 \qquad \text{, for each } j \end{aligned} \tag{3.38}$$

for each the 0-1 variables in J.

Suppose also that the inequality (3.33) is one of the given inequalities, and that $S \subset J$.

We first choose one of the inequalities to be lifted. It is sufficient to try the algorithm for each available inequality, and for each available literal. Suppose that the inequality:

$$\sum_{v \in N'} a_{ij} x_j \geq b \tag{3.39}$$

is the selected inequality (3.37). If the inequality (3.39) is a facet of a minor of the SAT problem, then by the sequential lifting lemma there must exist an index $j \notin N'$, and a literal $L_j \in \{x_j, (1 - x_j)\}$ such that

$$\sum_{v \in N'} a_v x_v + L_j \geq b + 1 \tag{3.40}$$

We want to find a literal $L_j \in \{x_i, (1 - x_i) | i \in N \setminus N'\}$ and a strong canonical inequality (3.34) that dominates (3.33), if one exists.

Now, to find the minimum value of $\sum_{v \in N'} L_v + L_j$, we use a linear program.

$$\begin{aligned} \text{minimize: } & \sum_{v \in N'} a_v x_v + L_j \\ \text{such that: } & \sum_{j \in J} a_{ij} x_j \geq b_i \qquad \text{, for } i \in U \\ & x_j \geq 0 \qquad \text{, for each } j \\ & -x_j \geq -1 \qquad \text{, for each } j \\ \text{and: } & L_n = 0 \end{aligned} \tag{3.41}$$

If the minimum value of this linear program is greater than b, then by the sequential lifting lemma we may deduce the inequality (3.40). The inequality (3.40) dominates the original inequality (3.39). Hence, we may remove (3.39) from the problem and replace it with (3.40).

The remaining issue is how to choose the inequality (3.33) and the literal L_j. We assert that it suffices to search the available literals. That is, for each inequality in the problem, and for each literal in the problem, lifting may be attempted. Hence, the number of linear programming subproblems to consider is the product of the number of variables and the number of inequalities. If any one of the lifting attempts succeeds, then the integer program may be modified by adding new cutting plane and removing the inequality that was lifted.

3.3.5 Canonical Lifting Subsumes Diagonal Sums

Barth (1996, ch. 7) gives a method of solving the pigeonhole problems by finding diagonal sum cuts. Diagonal sums were first defined by Hooker (1992). Essentially, a *diagonal sum cut* is an inference rule in which $a_1x_1 + a_2x_2 + ... + a_nx_n \geq b$ may be inferred from the set of inequalities: $(a_1 - 1)x_1 + a_2x_2 + ... + a_nx_n \geq b - 1$, $a_1x_1 + (a_2 - 1)x_2 + ... + a_nx_n \geq b - 1$,..., $a_1x_1 + a_2x_2 + ... + (a_n - 1)x_n \geq b - 1$.

The name *diagonal sum* derives from the fact that the coefficients on the diagonal are each reduced by one in the antecedents.

Hooker's *generalized resolution* rule is the special case in which only 0-1 coefficients are present:

Definition 5. *An canonical inequality* $DS := \sum_{1 \leq i \leq n} L_i \geq d + 1$ *is a diagonal sum of a set of canonical inequalities if:*

1. *for all* $1 \leq j \leq n$ *the canonical inequality* $DS_j := \sum_{1 \leq i \neq j \leq n} L_i \geq d$ *dominated by an extended clause in* S

2. *there is no canonical inequality in* S *that dominates* DS*;*

3. $Var(L_i) \neq Var(L_j)$ *for all* $i \neq j$ *(that is, no two literals have the same variable).*

Barth's contribution was to give an algorithm for finding certain generalized resolution inferences that are present in the pigeonhole problems. The search algorithm looks for diagonal sum cuts in a certain sequence that corresponds to a lifting procedure. However, Barth's algorithm for finding diagonal sum cuts is a search algorithm that finds only certain linear combinations of inequalities called diagonal sum cuts. We show that our lifting method of solving subproblems to find canonical inequalities also solves the pigeonhole problem efficiently, by finding the same inequalities that are found by Barth's method.

In Barth's procedure, each canonical inequality in n variables with is found by first finding n canonical inequalities in $n - 1$ variables. The procedure recurses to develop the inequalities with $n-1$ variables. Ultimately, the canonical inequalities in three variables are each constructed from three inequalities in two variables. In the

case with three variables, Barth's procedure finds:

$$x_1 + x_2 \geq 1 \tag{3.42}$$

$$x_1 + x_3 \geq 1 \tag{3.43}$$

$$x_2 + x_3 \geq 1 \tag{3.44}$$

then constructs a single Chvátal cut of the sum of those inequalities to derive:

$$x_1 + x_2 + x_3 \geq 2 \tag{3.45}$$

To find a single canonical inequality in n variables, Barth's procedure may require searching for as many as $sum_{i=0}^{n/2}\binom{n}{i}$ goal clauses.

Now, to show that our procedure finds the same canonical inequalities in n variables. Rather than finding n canonical inequalities in $n-1$ variables, we require only one. The formula $\sum_{1\leq i\leq n} L_i \geq d = n-1$ is called a clique inequality, and is a very important special case. When a clique inequality is encoded into conjunctive normal form, it generates $\binom{n}{2}$ two-literal clauses $L_i \vee L_j$, for each $i \neq j$. The solution of the pigeonhole problem requires that the algorithm find clique inequalities.

To find a clique inequality, our procedure requires that we have previously found only one inequality in $n-1$ variables. Without loss of generality, we assume a convenient labeling of the variables. Suppose that we have found:

$$\sum_{1\leq i\leq n-1} L_i \geq d = n-2 \tag{3.46}$$

and that the problem also includes inequalities that dominate each of the two-literal inequalities that includes L_n. In the simple case where each of the two-literal inequalities is directly present, we have the set of inequalities:

$$\sum_{1\leq i\leq n-1} L_i \geq n-2 \tag{3.47}$$

$$L_j + L_n \geq 1 \qquad \text{, for } 1 \leq j \leq n-1 \tag{3.48}$$

Our procedure will attempt to lift the first inequality by the literal L_n to obtain

$$\sum_{1 \le i \le n-1} L_i + L_n \ge d = n - 1 \tag{3.49}$$

The linear program is then:

$$\begin{aligned} \text{minimize:} \quad & \sum_{1 \le i \le n} L_i \\ \text{such that:} \quad & \sum_{1 \le i \le n-1} L_i \ge d - 1 = n - 2 \\ & L_j + L_n \ge 1 \qquad \text{, for } 1 \le j \le n-1 \\ \text{and:} \quad & L_n = 0 \end{aligned} \tag{3.50}$$

the minimum value is then found to be $n-1$, which is greater than $n-2$, and hence by the lifting lemma the inequality 3.49 is justified.

In the case of finding a large clique inequality, our method requires solving as many as $n-1$ linear programs. This is potentially much more efficient than Barth's method, which requires a recursive procedure to find each of $sum_{i=0}^{n/2}\binom{n}{i}$ smaller inequalities.

The derivation for non-clique inequalities, sometimes called counting inequalities, is similar. The only difference is that the value of d is smaller, and the initial set of inequalities is larger. When a counting constraint

$$\sum_{1 \le i \le n} L_i \ge d = n - k \tag{3.51}$$

is encoded into conjunctive normal form, the resulting proposition has $\binom{n}{k+1}$ clauses of size $\binom{n}{k}$. Our procedure requires finding only one constraints with $n-1$ literals, from which the inequality (3.51) can be derived directly by solving one linear program.

$$\begin{aligned} \text{minimize:} \quad & \sum_{1 \le i \le n} L_i \\ \text{such that:} \quad & \sum_{1 \le i \le n-1} L_i \ge d - 1 = n - k - 1 \\ & \sum_{j \in S} L_i \ge 1 \text{ , for } S \subset \{1, \ldots, n\}, |S| = k \\ \text{and:} \quad & L_n = 0 \end{aligned} \tag{3.52}$$

After finding that the minimum is greater than $d-1$, the lifting lemma gives the result. Barth's procedure again requires a combinatorial number of intermediate inequalities to be built up before the counting constraint (3.51) can be recovered from the proposition.

3.4 Integer Lifting for Cutting Planes

The canonical lifting algorithm given in section 3.3.4 lifts canonical inequalities to discover stronger canonical inequalities that are also valid for the 0-1 program. Hooker and Barth gave similar algorithms, and Barth showed that his algorithm could solve pigeonhole problems efficiently (Barth, 1996; Hooker, 1992).

However, those lifting algorithms do not apply after some pivots have been performed in a linear program, because the proofs each depend on having only 0-1 variables. After some pivots have been performed, some of the variables in the inequalities are slack variables of the original problem, which are not constrained to take only 0-1 values. The use of slack variables is necessary to avoid the lower bound of Krajíček (1997) and Pudlák (1997), as discussed in section 3.2. Hence, to incorporate both lifting and slack variables in a single algorithm, new lifting lemmas and corresponding algorithms are needed.

In this section, we give a modification of the lifting lemmas, and show that cutting planes can be lifted in the context of a linear program, even if some non-Boolean slack variables are nonbasic. These new lifting lemmas use a modification of the definition of a minor, which accommodates the presence of some slack variables in a natural way.

3.4.1 Integer Minors

Recall the definition 4 of a minor of a SAT problem. That definition assumes that all of the variables present in the problem are Boolean variables, restricted to the domain $\{0, 1\}$. Before giving the lifting lemmas for the case in in which some non-Boolean integer variables are present in the problem, it is necessary to extend the definition of a minor to allow variables with non-Boolean domains. The slack variables have domain in the positive integers, so we must provide a definition of minor that allows such variables.

For the statements and proofs of integer lifting theorems, it will be convenient to redefine some notation. We define the minor of an integer programming problem using the same notation as definition 4. The reuse of the notation should not be confusing, because the new definition is equivalent to the previous definition for the case with all Boolean variables.

Definition 6. *Let $P = \{x : Ax \geq B\}$ be the polytope of an integer programming problem. Let $N = \{0, \ldots, n\}$ index the variables, $F \subset N$, $T \subset N$, and $N' \subset N$ be subsets of the index set N such that $F \cap T = \emptyset$ and $N' = N \setminus (F \cap T)$. Then the* integer minor P_F^T *of P is the polytope $\{x : Ax \geq P, x_j \leq 0$ for $j \in F, x_j \geq 1$ for $j \in T\}$.*

The difference between definition 4 and definition 6 is subtle. In definition 4, the restriction on the variables in T is $x_j = 1$. In definition 6, that restriction is altered to $x_j \geq 1$. For variables that are restricted to domain $\{0, 1\}$, there is no difference. The difference only affects variables having domain in the positive integers. Hence, for problems in which all variables are restricted to domain $\{0, 1\}$, the minor as defined by definition 6 is identical to the minor of definition 4. The two definitions are different only if some of the variables are not Boolean.

Definition 6 accommodates slack variables in a natural way. In a pure integer

program, or in the integer program relaxation of a SAT problem, the slack variables are restricted to the domain of positive integers. For a SAT problem having N Boolean variables and M clauses, M slack variables are introduced when the integer program relaxation is formed. Thus, the polytope of the integer program has $M+N$ variables. In the minors of that polytope, any of those variables may be restricted by either of the inequalities $x_j \leq 0$ or $x_j \geq 1$.

3.4.2 Integer Lifting Lemma

In this section, we give a modification of the lifting lemma, and show that cutting planes can be lifted in the context of a linear program, even if some non-Boolean slack variables are nonbasic. The difference is that this new version of lifting does not depend on having only 0-1 variables in the problem. Instead, this version depends on having only integer variables. The slack variables of an integer program are integer variables, so a 0-1 linear program meets this more general condition even after some pivots.

Lemma 8 (Integer Inequality Lifting). *Let P be the polytope of an integer programming problem with variable index set N. Let P_F^T be an integer minor of the polytope P and assume that the inequality*

$$\sum_{v \in N'} a_v x_v \geq a_0 \tag{3.53}$$

defines a valid inequality of $Q(P_F^T)$. Then:

1. *For each $j \in T$, and $a_j = \min_{x \in P_{F \cup \{j\}}^{T \setminus \{j\}}} (\sum_{v \in N'} a_v x_v) - a_0$, the inequality:*

$$\sum_{v \in N'} a_v x_v + a_j x_j \geq a_0 + a_j \tag{3.54}$$

   defines a valid inequality of the polytope $Q(P_F^{T \setminus \{j\}})$.

2. *For each* $j \in F$, *and* $a_j = \min_{x \in P_{F\setminus\{j\}}^{T\cup\{j\}}}(\sum_{v\in N'} a_v x_v) - a_0$, *the inequality:*

$$\sum_{v\in N'} a_v x_v - a_j x_j \geq a_0 \tag{3.55}$$

defines a valid inequality of the polytope $Q(P_{F\setminus\{j\}}^{T})$.

Proof. Proof of statement 1:

Let $a_j = \min_{x \in P_{F\cup\{j\}}^{T\setminus\{j\}}}(\sum_{v\in N'} a_v x_v) - a_0$. If $Q(P) \cap \{x : x_j = 0\} = \emptyset$, then $x_j \geq 1$ is valid for $Q(P_F^{T\setminus\{j\}})$. In this case, for every a_j:

$$\sum_{v\in N'} a_v x_v + a_j x_j \geq a_0 + a_j \tag{3.56}$$

Otherwise, $Q(P) \cap \{x : x_j = 0\} \neq \emptyset$. For $x_j = 0$, we have $P_F^{T\setminus j} \cap \{x : x = 0\} = P_{F\cup j}^{T\setminus\{j\}}$. Then $\sum_{v\in N'} a_v x_v \geq a_0 + a_j$ for a_j such that:

$$\min_{x\in P_{F\cup\{j\}}^{T\setminus\{j\}}}(\sum_{v\in N'} a_v x_v) \geq a_0 + a_j \tag{3.57}$$

or equivalently:

$$a_j \leq \min_{x\in P_{F\cup\{j\}}^{T\setminus\{j\}}}(\sum_{v\in N'} a_v x_v) - a_0 \tag{3.58}$$

Hence, (3.54) is valid for $Q(P_F^{T\setminus\{j\}})$.

Proof of statement 2:

Let $a_j = \min_{x \in P_{F\setminus\{j\}}^{T\cup\{j\}}}(\sum_{v\in N'} a_v x_v) - a_0$. If $Q(P) \cap \{x : x_j \geq 1\} = \emptyset$, then $x_j = 0$ is valid for $Q(P_{F\setminus\{j\}}^{T})$. In this case, for every a_j:

$$\sum_{v\in N'} a_v x_v - a_j x_j \geq a_0 \tag{3.59}$$

Otherwise, $Q(P) \cap \{x : x_j \geq 1\} \neq \emptyset$. For $x_j \geq 1$, we have $P_{F\setminus j}^{T} \cap \{x : x \geq 1\} = P_{F\setminus\{j\}}^{T\cup\{j\}}$. Then $\sum_{v\in N'} a_v x_v + \geq a_0 + a_j$ for a_j such that:

$$\min_{x\in P_{F\setminus\{j\}}^{T\cup\{j\}}}(\sum_{v\in N'} a_v x_v) \geq a_0 + a_j \tag{3.60}$$

or equivalently:

$$a_j \leq \min_{x \in P_{F\setminus\{j\}}^{T\cup\{j\}}} (\sum_{v \in N'} a_v x_v) - a_0 \tag{3.61}$$

Hence, (3.55) is valid for $Q(P_{F\setminus\{j\}}^T)$. □

Lemma 8 gives a method for deriving valid inequalities, using inequalities that are valid for lower-dimension subproblems. To show that the method can be used to derive facet inequalities, we need an additional condition.

Lemma 9 (Integer Facet Lifting). *Let P be the polytope of an integer programming problem with variable index set N. Let P_F^T be an integer minor of the polytope P and assume that the inequality*

$$\sum_{v \in N'} a_v x_v \geq a_0 \tag{3.62}$$

defines a nontrivial facet of $Q(P_F^T)$. Then:

1. *For each $j \in T$, If $a_j = \min_{x \in P_{F\cup\{j\}}^{T\setminus\{j\}}} (\sum_{v \in N'} a_v x_v) - a_0$, the inequality:*

$$\sum_{v \in N'} a_v x_v + a_j x_j \geq a_0 + a_j \tag{3.63}$$

   defines a facet of the polytope $Q(P_F^{T\setminus\{j\}})$.

2. *For each $j \in F$, For $a_j = \min_{x \in P_{F\setminus\{j\}}^{T\cup\{j\}}} (\sum_{v \in N'} a_v x_v) - a_0$, the inequality:*

$$\sum_{v \in N'} a_v x_v - a_j x_j \geq a_0 \tag{3.64}$$

   defines a facet of the polytope $Q(P_{F\setminus\{j\}}^T)$.

Proof. We give a detailed proof of statement 1. The proof of statement 2 is complementary. Because (3.62) is a facet of $Q(P_F^T)$, (3.62) is a valid inequality of $Q(P_F^T)$. Hence, by lemma 8, (3.63) is a valid inequality for $Q(P_F^{T\setminus\{j\}})$. It remains to show

that (3.63) defines a facet of the polytope $Q(P_F^{T\setminus\{j\}})$. To show this, we will show the existence of $k+1$ affinely independent vectors that satisfy (3.63) at equality.

Since (3.62) defines a facet of $Q(P_F^T)$, there must be $k = |N'| = |N \setminus (F \cup T)|$ affinely independent vectors x^i for $i \in \{1, \ldots, k\}$ that satisfy (3.62) at equality. Since $x_j^i = 0$, each vector x^i satisfies (3.63) at equality. To construct the one additional affinely independent vector, let $x^* \in Q(P_{F\cup j}^{T\setminus\{j\}})$ with $x_j^* \geq 1$. For $a_j = \min_{x\in P_{F\cup\{j\}}^{T\setminus\{j\}}}(\sum_{v\in N'} a_v x_v) - a_0$, x^* satisfies (3.63) at equality. Since $x_j^* \geq 1$, x^* cannot be written as a combination of the k vectors x^i, so the $k+1$ vectors $\{x^*, x^1, \ldots, x^k\}$ are affinely independent. Hence, there exist $k+1$ affinely independent vectors in $Q(P_F^{T\setminus\{j\}})$ that satisfy (3.63) at equality, so (3.63) defines a facet of the polytope $Q(P_F^{T\setminus\{j\}})$. □

Lemma 10 (Integer Lifting Lemma). *Let P be the polytope of a satisfiability problem with variable index set N. Let P_F^T be a minor of P and let $Q(P_F^T)$ be the convex hull of P_F^T. Suppose that the inequality*

$$\sum_{v\in N'} a_v x_v \geq a_0$$

defines a nontrivial facet of $Q(P_F^T)$ and that $Q(P_F^T)$ is full dimensional. Then there exist coefficients b_v, for $v \in N \setminus N'$ and b_0 such that the inequality

$$\sum_{v\in N'} a_v x_v + \sum_{v\in N\setminus N'} b_v x_v \geq a_0 + b_0$$

defines a facet of $Q(P)$.

Proof. This follows by induction from repeated application of lemma 9. At each application, the cardinality of $N \setminus N'$ is reduced. □

The use of lifting algorithm to find strong inequalities that dominate Chvátal -Gomory cutting planes gives an algorithm that is potentially more efficient than previous cutting plane algorithms because it can discover stronger valid inequalities.

3.4.3 Integer Lifting on a Polytope

Suppose we have a set of canonical inequalities P that define the polytope of an integer programming problem. We do not know a-priori if any one of those inequalities defines a facet of an integer minor of the problem. Obviously, if we have a canonical inequality

$$\sum_{v \in N'} a_v x_v \geq b \tag{3.65}$$

for some set of variables S, then it is possible that there may exist an integer minor subproblem P_F^T for which the inequality (3.65) is a facet.

If the inequality (3.65) is a facet of an integer minor of the problem, then by the sequential lifting lemma there must exist an index $j \notin N'$, a literal $L_j \in \{x_j, (1-x_j)\}$, and an a_j such that

$$\sum_{v \in N'} a_v x_v + a_j L_j \geq b + a_j \tag{3.66}$$

is a facet of one of the minor subproblems $P_F^{T\setminus\{j\}}$ or $P_{F\setminus\{j\}}^T$. The literal L_j may be a positive occurrence of a variable x_j, or a negated occurrence $\overline{x}_j = 1 - x_j$. The two cases of lemma 9 correspond to the two possibilities.

Suppose for a particular variable x_j, we use an algorithm to find $a_j = \min_{x \in P_{\{j\}}^{\emptyset}} (\sum_{v \in N'} a_v x_v) - a_0$. If $a_j > 0$ then the inequality

$$\sum_{v \in N'} a_v x_v + a_j x_j \geq b + a_j \tag{3.67}$$

is a valid inequality, and dominates the original inequality (3.65). Because (3.67) implies (3.65), we may remove (3.65) from the problem, replacing it with (3.67).

In the complementary case, suppose for a particular complemented variable $\overline{x}_j = (1 - x_v)$, we use an algorithm to find $a_j = \min_{x \in P_{\emptyset}^{\{j\}}} (\sum_{v \in N'} a_v x_v) - a_0$. If $a_j > 0$ then the inequality

$$\sum_{v \in N'} a_v x_v - a_j x_j \geq b \tag{3.68}$$

is a valid inequality, and dominates the original inequality (3.65). Because (3.68) implies (3.65), we may remove (3.65) from the problem, replacing it with (3.68).

The polytope $P^{\emptyset}_{\{j\}}$ (resp. $P^{\{j\}}_{\emptyset}$) is just a polytope described by some set of linear inequalities, and the sum of the weights $\sum_{v \in N'} a_v x_v$ is just a linear function. Hence, a linear programming algorithm can be used to find a lower bound for the minimum integer value of a_j.

3.4.4 Integer Lifting Algorithm

In section 3.4.2, the lifting lemmas were modified to show that cutting planes can be lifted in the context of an integer program. In section 3.4.3, we gave an an algorithm for lifting of an integer program defined by a given polytope. In this section, we give an algorithm for lifting an integer program defined by an SAT problem.

For a particular integer variable x_j, or its complement $1-x_j$, integer lifting requires some means of finding the minimum value of a_j such that (3.66) follows from P. In this section we give an algorithm that uses a linear programming subproblem to find a lower bound on this value. Using this lower bound, and knowing that the facets of SAT are canonical hyperplanes, we may use the integer ceiling of that minimum value to obtain a stronger inequality.

Suppose that we are given a set of valid inequalities

$$\sum_{j \in J} a_{ij} x_j \geq b_i \text{ , for each } i \in U \tag{3.69}$$

where all of the variables are restricted to the domain of nonnegative integers. Some of the variables may be further restricted to the domain $\{0, 1\}$. As before, the linear relaxation of any such domain restrictions gives the domain inequalities:

$$\begin{aligned} x_j &\geq 0 \qquad \text{, for each } j \\ -x_j &\geq -1 \qquad \text{, for each } j \end{aligned} \tag{3.70}$$

for each the Boolean variables. Suppose also that the inequality (3.65) is one of the given inequalities, and that $S \subset J$.

We first choose one of the inequalities to be lifted. Suppose that the inequality:

$$\sum_{v \in N'} a_v x_v \geq b \tag{3.71}$$

is the selected inequality. Suppose further that the selected inequality is a Gomory cutting plane generated from some linear combination of inequalities with only integer coefficients.

If the inequality (3.71) is a facet of an integer minor of the integer program, then by the sequential lifting lemma there exists an index $j \in N$, and a literal $L_j \in \{x_j, (1 - x_j)\}$ such that

$$\sum_{v \in N'} a_v x_v + L_j \geq b + 1 \tag{3.72}$$

We want to find a literal $L_j \in \{x_i, (1 - x_i)\}$ and a strong canonical inequality (3.66) that dominates (3.65), if one exists.

Now, to find the minimum value of $\sum_{v \in N'} a_v x_v + L_j$, we use a linear program:

$$\begin{aligned} \text{minimize: } & \sum_{v \in N'} a_v x_v + L_j \\ \text{such that: } & \sum_{j \in J} a_{ij} x_j \geq b_i \qquad \text{, for } i \in U \\ & x_j \geq 0 \qquad \text{, for each } j \\ & -x_j \geq -1 \qquad \text{, for each } j \\ \text{and: } & L_j \leq 0 \end{aligned} \tag{3.73}$$

Because the selected inequality (3.71) is a Gomory cutting plane algorithm from a set of linear inequalities having only integer coefficients, we also know that:

$$\sum_{v \in N'} a_v x_v \equiv b (mod 1) \tag{3.74}$$

Hence, if the minimum value of the linear program (3.73) is greater than b, then by the sequential lifting lemma we may deduce the inequality (3.72). The inequality (3.72) dominates the original inequality (3.71). Hence, we may remove (3.71) from the problem and replace it with (3.72).

The use of Chvátal or Gomory cutting planes with this lifting algorithm gives an algorithm that is potentially more efficient than the previous cutting plane algorithms because it can discover stronger canonical cutting planes. The derivation of a strong inequality that dominates a cutting plane gives a stronger inequality, and so fewer inequalities may be required to solve the problem.

3.5 Choosing a Vertex to Cut

For SAT, the objective function is not given by the problem. We are free to use any objective function, and even to ignore the objective function when it is advantageous to do so. When it is necessary to construct cutting planes, it may be that a different vertex of the polytope will allow us to construct a stronger cutting plane. A local search of some subset of the vertices, to find a vertex that allows us to generate a stronger cutting planes, may be advantageous.

In this section, we discuss a method of using a local search to find strong cutting planes. The basic idea is to measure the strength of the strongest cutting plane at each vertex, and to take any pivots that allow us to generate a stronger cutting plane.

3.5.1 Multi-Start Local Search Algorithms

The idea of multi-start local search is not new. Many operations research problems have been addressed by local search (Fiduccia & Mattheyses, 1982; Aarts & Lenstra, 1997). Much of artificial intelligence can be viewed as developing various algorithms

for local search. Numerous authors have applied multi-start local search to develop a variety of model-search algorithms for finding satisfying solutions SAT problems. (See section 2.3.4.) Those algorithms seek to find a satisfying solution to an SAT problem.

A multi-start local search algorithm searches a graph to find an local-optimum vertex. The basic algorithm uses iterations, with two basic steps at each iteration. In the first step, a starting node of the graph is selected. Second, while an adjacent nodes is better than the current node, select one of the better adjacent nodes. A "greedy" version always selects the best adjacent node, while a "randomized" version selects randomly from among the adjacent nodes. A variation of the randomized version selects the best adjacent node from a random sample of the adjacent nodes.

The two basic design decisions for such an algorithm are the method of selecting starting nodes, and the method of determining the quality of each node. For local search algorithms on convex polytopes, the nodes of the graph are the vertices of the polytope, which of course are just the basic solutions of the linear equations. The arcs of the graph are just the edges of the polytope, which are the possible pivots of the system of linear equations. The simplex algorithm is an example of a local-search algorithm, in which the local optimum is also a global optimum.

3.5.2 Finding a Stronger Cut

At each iteration of a cutting plane algorithm, an objective function is constructed, and the linear program is solved. In the usual algorithm, one or more cutting planes are added and the next iteration starts. This section outlines a modification of the usual algorithm to include a local search to find a stronger cutting plane.

Rather than immediately generating a cutting plane, we search for a nearby vertex

that allows a stronger cutting plane. The vertex the is the solution of the linear program is used as the starting point of the search. The adjacent vertices of the polytope, determined by the feasible pivots of the tableau, are examined to see if any of them gives a stronger cutting plane. If any pivot gives a tableau that allows a stronger cutting plane than is allowed by the current tableau, the pivot leading to the strongest such cutting plane is taken.

The two basic design decisions for a multi-start local search algorithm are the selection of starting nodes and the method of measuring the quality of each node. Our method of selecting the starting vertex is to solve the linear program using an objective function. In section 3.6, we develop an objective function that can be used with a simplex algorithm to select a basic solution as a starting vertex. Our method of measuring the quality of the nodes is by measuring the strength of the cuts that may be generated at a vertex. In section 3.7, we develop a method of measuring the quality of a vertex, by measuring the strength of the cutting planes that can be derived at that vertex.

3.5.3 Reducing the Denominator

Gomory (1963) presents the original method of cutting planes for integer programming. In Gomory's algorithm, a linear relaxation is first solved to optimality, then cutting planes and dual pivots are used to reduce the optimal solution to integers. In the traditional cutting-plane algorithm, all of the values in the tableau may be expressed using a single common denominator. The common denominator is explicit in Gomory's integer-only method, which keeps it separately (Gomory, 1963, Example 3). In the traditional algorithm, the common denominator may become quite large.

In the linear program solutions it is common to have coefficients with value zero

in the objective row of the optimal tableau. In such cases, multiple distinct solutions of the linear program give the same objective value. It may be that some optimal solutions have a smaller denominator than others. When this is the case, it is possible to choose the solution having the smaller denominator while preserving the optimality of the solution.

In our algorithm, we want to find the strongest cutting plane, and to reduce the denominator only when it does not cost us any cutting-plane strength. When the local search algorithm does not find a stronger cut, it may still find a pivot that gives the same strength of cut and a reduced denominator.

The value of reducing the common denominator is twofold. The first reason is to optimize the arithmetic. Multiple-precision arithmetic functions run faster if they are operating on smaller numbers, and the memory usage to store large numbers can become a problem. The second reason is aesthetic. Solutions expressed with smaller numbers are aesthetically more pleasing than solutions expressed with large numbers. As a heuristic, it seems plausible that simplex tableau with small common denominators may tend to contain fewer fractions.

3.5.4 Complexity of the Local Search

The complexity of local search for pivots that allow stronger cutting planes is polynomial. Clearly, for a simplex tableau having N columns and M variables, at most MN pivots exist that preserve primal feasibility. For each such pivot, the computation of the resulting tableau requires $O(MN)$ arithmetic operations. Assuming for the moment that the algorithm to measure the strength of a cutting plane is linear in N, the complexity of examining a tableau to determine the strength of the strongest cutting plane and the magnitude of the smallest pivot element is also $O(MN)$. Hence, the

total complexity of one iteration of the local search algorithm is at most a fourth-degree polynomial in N. In practice, far fewer than MN pivots exist that preserve primal feasibility.

The number of iterations required by the local search is also a low-degree polynomial. Tovey (1997) gives an analysis of search algorithms for finding local maxima/minima in graphs. Tovey the unit hypercube as a model of a discrete structure to be searched, but generalizes the theorems in later sections. The expected number of iterations for both greedy and randomized local search algorithms is shown to be logarithmic in the number of nodes in the graph, assuming some connectivity conditions. Convex polytopes, in particular, are mentioned as an example of graphs that meet the conditions. Hence, the expected complexity of our entire local search algorithm is polynomial.

The use of a subroutine that has fourth-degree polynomial complexity could contribute greatly to the running time of a SAT algorithm. An approximation is to use sampling, rather than exhaustive search. The algorithm may be reduced to cubic complexity by reducing one dimension of the search. We may choose to sample of the possible pivots in each column, or to measure the strength of only a sample of the cutting planes for each possible pivot. The resulting limited local search very frequently finds cutting planes that are as strong as the full local search, but is much less expensive. In any case, even the full local search is polynomially bounded.

3.6 Choosing an Objective Function

We have considered in the previous sections several cutting plane algorithms for SAT. To apply a cutting-plane algorithm to the linear relaxation of SAT, we must also select a basic solution from which to construct a cutting plane. In the usual operations

research applications of integer programming algorithms, the objective function is normally given as part of the problem, and the objective function determines the choice of vertex.

For SAT, the objective function is not given by the problem. We are free to use any objective function, and even to ignore the chosen objective function when it is advantageous to do so. The literature provides essentially no guidance in choosing an appropriate objective function. In this section, we will identify a good choice of objective function for SAT problems. This choice is of course purely a heuristic. As such, it may assist in the solution of some problems.

We also give a modification of the dual simplex algorithm to improve the objective function as cutting planes are added to the problem. This is justified by the fact that for SAT the choice of objective function is purely arbitrary. If the vertex that is optimal for one objective function is difficult to find, then it may be advantageous to modify the objective function.

3.6.1 Facets Containing a Vertex

An interesting question is to estimate the number of cutting planes that are required to find one of a small number of satisfying solutions, or to show that none exist. It is clear that the number of cutting planes depends in some sense on the number of facet inequalities that must be found. It is also clear that the number of facet inequalities that must be found depends on the number of facets that contain the optimal vertex, which in turn depends on the choice of objective function.

Hence, the number of cutting planes that are required depends on the choice of objective function. As a heuristic, we want to choose an objective function such that only a small number of facets of the convex hull contain the vertex that is optimal for

that objective function. If only a small number of facets are required, it seems likely that only a small number cutting planes are needed to find those facets.

In the following, we will consider unspecified SAT problems in n variables that have a set V' of satisfying solutions. Let $K = [0,1]^n$ be the unit hypercube. Let $V = \{0,1\}^n$ be the vertices of the unit hypercube. Let $V' \subseteq V$ be the set of satisfying solutions of an unspecified SAT problem. Also let $Q(V')$ be the convex hull of V', and H be the set of canonical hyperplanes that define facets of $Q(V')$.

We may easily construct SAT problems such that $Q(V')$ has $O(2^{n-1})$ facets. Consider, for example, a proposition representing a *parity* function which has value 1 when an even number of variables have value 1. However, for such problems the number of satisfying solutions $|V'|$ is also very large. Observe that, for the parity function, every vertex is contained on exactly n of the facets of the convex hull. Such problems tend to be easily solved by various search algorithms.

We are much more interested in hard SAT problems, which have either no satisfying solutions, or only a small number of satisfying solutions. For these problems the set V' is small, so we have $|V'|/|V| = O(p(n)/2^n)$. That is, the number of satisfying solution is a polynomial of the number of variables.

To find integer solutions by using a cutting plane method, it is usually necessary to find some number of cutting planes that each contain the optimal solution. Cutting planes are defined by valid inequalities, and in the limit the cutting planes are the facets of the polytope. It is clear that the minimum number of valid inequalities that is required is determined by the dimension of the convex hull $Q(V')$.

Proposition 4. *For each vertex $V_i \in V'$, if $Q(V')$ is full dimensional, then the minimum number of facets of the convex hull $Q(V')$ that contain V_i is n.*

Proof. A point in R^n is uniquely determined by the intersection of exactly n (n-1)-

dimensional and affinely independent hyperplanes. A facet is an (n-1)-dimensional and affinely independent hyperplane. Hence, at least n facets are needed to determine the point. □

To minimize the number of cutting planes that are required, we need to avoid certain vertices. In particular, we need to somehow avoid the vertices that are contained in large numbers of facets, because it may be necessary to find cutting planes that define all of those facets. The existence of such bad polytopes is demonstrated easily.

Proposition 5. *There exists a polytope P and a vertex $V_i \in V'$ such that the number of facets of the convex hull $Q(V')$ that contain V_i is 2^{n-1}.*

Proof. Consider the polytope with vertices consisting of the single point $(0, 1, \ldots, 1)$ and each of the 2^{n-1} points $(1, x_2, \ldots, x_n)$. For this polytope, the number of vertices is $|V| = 1 + 2^{n-1}$. There is an edge between the point $(0, 1, \ldots, 1)$ and each of the 2^{n-1} points $(1, x_2, \ldots, x_n)$. Hence, there are 2^{n-1} facets containing the point $(0, 1, \ldots, 1)$. □

Observe that for each such bad vertex, there must exist a large number of adjacent vertices that are not each contained in a large number of facets. This occurs because the number of facets containing a given vertex V_i cannot exceed the number of other vertices that are adjacent to V_i. When $|V'|$ is small, this gives a particularly nice limit on the number of facets containing each vertex.

Proposition 6. *If $|V'| = p(n)$ for some polynomial p, then for each vertex $V_i \in V'$, at most $p(n) - 1$ facets contain V_i.*

Proof. The simplex graph of the convex hull contains at most $|V'| - 1$ edges having V_i as an endpoint, because there are only $|V'| - 1$ other vertices. Each facet containing V_i is determined by two of those edges. There are at most $|V'| - 1$ such facets. □

Even for SAT problems having a large number of satisfying solutions, we observe that most vertices of the convex hull have only a small number of adjacent vertices, and are contained in only a small number of facets.

Proposition 7. *For every vertex $V_i \in V'$, V_i is determined by the intersection of only N facets.*

Proof. Indeed, in N dimensions, the intersection of any N affinely independent $N-1$ dimensional hyperplanes (facets) is a single point. □

For the polytopes of satisfiable SAT problems, most vertices are contained in only a small number of incident facets. *If any satisfying solution exists for a SAT problem in N dimensions, there must be a satisfying solution that is contained in the intersection of only N facets.* Hence, we may restrict the search for a solution to those vertices that are contained in a small number of facets without loss of completeness. If there is no satisfying solution that is contained in the intersection of only N facets, then there is no satisfying solution at all.

Now, consider the description of the polytope as a set of linear inequalities. After adding slack variables, the resulting linear matrix equation is a simplex tableau of the polytope. The number of linear inequalities, and the number of basic variables, is equal to the number of facets of the polytope. Hence, the restriction on the number of facets that must be constructed also restricts the number of facet inequalities that must be constructed. If we can find facet inequalities efficiently, then we should be able to solve the SAT problem efficiently.

3.6.2 Maximizing the Slack Variables

We have shown that we may restrict the search for a solution to those vertices that are contained in only a small number of facets. We will now give one method of doing

so. This method is based on choosing the objective function so that any vertices that are contained in large numbers of facets are suboptimal.

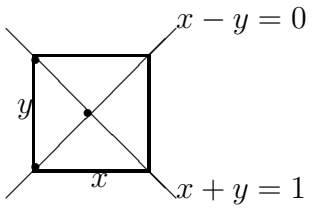
In the linear programming formulation, every cutting plane has an associated slack variable. For a vertex that is contained in a large number of facets, a large number of slack variables must have value 0. To avoid these vertices, we need only choose an objective function so that any vertex with a large number of 0-valued slack variables is not an optimal solution.

In particular, we may choose the objective function to *maximize the sum of the slack variables*. This choice of objective function also has the effect of finding integer-valued solutions for many problems very quickly, and using very few or no cutting planes. The choice of objective function allows the linear programming algorithm to solve some SAT problems without requiring any cutting planes.

In a trivial two-dimensional example (figure 3.1), the inequalities $x + y \leq 1$ and $x - y \leq 0$ are converted to equalities by adding slack variables s_1 and s_2, giving $s_1 + x + y = 1$ and $s_2 + x - y = 0$. The point $(1/2, 1/2)$ satisfies both inequalities with no slack. The maximum slackness occurs at either of the points $(0, 0)$ or $(0, 1)$. This illustrates a case in which the choice of objective function allows an integer solution to be found with a minimum number of cutting planes.

In higher numbers of dimensions, of course, the maximum slackness does not always occur at an integer vertex. (If it did, then that fact alone would solve SAT.) However, as a heuristic, maximizing the slackness may be helpful. Of course, the objective of maximizing the slackness requires that the objective function must be modified whenever a new slack variable is added to the problem with a new cutting plane.

Figure 3.1: A two-dimensional unit hypercube with two canonical constraints, $x+y \leq 1$ and $x - y \leq 0$. The three points mark the extrema of the polytope.



3.7 Strength of a Cutting Plane

To search for stronger cutting planes, it is necessary to have some function to measure the strength of a cutting plane. In the usual optimization-based methods used for integer programming, the strength of a cutting plane is often considered as the distance from the current basic solution. Various distance measures can be used, giving various strength measures. However, for our purposes we need a measure of the strength of a cutting plane that is independent of any particular basic solution.

Barth (1995) gives a function to determine the *weakness* of a valid inequality that is independent of any particular solution. Barth's weakness measure is conceptually the negation of a strength measure, and so can be used in place of a strength function. Barth's method simply counts the number of literals that may be false, while the inequality remains satisfied.

A corresponding strength measure would simply count the number of literals that must be true if the inequality is to be satisfied. However, such a measure captures very little information from the inequality. Consider, for example, the case of Chvátal cutting planes. For inequalities of the form $\sum x_i \geq 1$, the inequality is true whenever any one of the literals is true. Hence, every Chvátal cutting plane would have strength

1 by this measure. Clearly, we require a more discriminating measure.

We propose instead a measure that depends on the coefficients, rather than just the count of literals. Several authors have suggested that the distance from the current basic solution to the hyperplane as the measure of cutting-plane strength is a good measure of strength (Franco & Gelder, 1998). We will first show the derivation of such a measure, and then show a modification that we will use.

Suppose we have an inequality:

$$\sum_{i \in I} f_i x_i + \sum_{j \in J} g_j x_j \geq h \tag{3.75}$$

such that $f_i \geq 0$, $g_j \leq 0$, and $I \cap J$ is empty. To find the minimum distance from the origin to a given hyperplane, we will consider the case in which all of the $x_j = 0$. (Nonzero x_j could only increase the distance.) Then we have the hyperplane:

$$\sum_{i \in I} f_i x_i = h \tag{3.76}$$

Now, the parametric equations of the line through the origin and orthogonal to the hyperplane (3.76) are:

$$x_i = f_i t \qquad \text{, for each } i \in I \tag{3.77}$$

Substituting the equations of the orthogonal line (3.77) into the equation of the hyperplane (3.76), we obtain the parameter t of the intersection of the line and the hyperplane:

$$t = \frac{h}{\sum_{i \in I} f_i^2} \tag{3.78}$$

Now, at $t = 1$, the distance from the origin to the line is just $d_1 = \sqrt{\sum_{i \in I} f_i^2}$, so at $t = \frac{h}{\Sigma_{i \in I} f_i^2}$, the distance from the origin to the intersection of the hyperplane with the line is just:

$$d_t = \frac{h\sqrt{\sum_{i \in I} f_i^2}}{\sum_{i \in I} f_i^2} = \frac{h}{\sqrt{\sum_{i \in I} f_i^2}} \tag{3.79}$$

Now for our modification. Observe that a cutting plane with only two variables may have a distance at most $\sqrt{2}$, while a cutting plane with, say, nine variables may have a distance of $\sqrt{9} = 3$. But the two-variable cutting plane with distance $\sqrt{2}$ fixes both of those variables, while a nine-variable cutting plane with distance 2 does not fix any variables. Hence, the cutting plane with fewer variable is actually stronger, in the sense that it fixes more variables. This observation motivates our definition.

Then we define the *strength* of (3.75) as the distance from the origin to the cutting plane, expressed as a fraction of the maximum distance that is possible with the given number of variables.

Definition 7. *The* strength *of a cutting plane (3.75)*

$$\sum_{i \in I} f_i x_i + \sum_{j \in J} g_j x_j \leq h$$

such that $f_i \leq 0$, $g_i \geq 0$, $h \leq 0$, *and* $I \cap J = \emptyset$, *is given by*

$$\sqrt{\frac{h^2}{|I| \sum_{i \in I} f_i^2}} \tag{3.80}$$

Finally, to avoid computing the square root, algorithms may consider the square of the distance. Whenever one distance is greater than another, the same ordering holds between the squares of the distances.

3.7.1 Power of Strong Cuts

To justify both this strength measure and the local search for strong cutting planes, we consider the expected number of cutting planes that will be needed to solve the SAT problem. Consider a non-integer basic solution of the LP relaxation, with some set of N nonbasic variables. We may consider the N-dimensional hypercube of the nonbasic dimensions. Each Boolean nonbasic variable determines one dimension of

the hypercube. Each integer nonbasic slack variable determines a sum of Boolean variables, so increasing the value of a slack variable by one directly forces a Boolean variable to be increased by the same amount. Each cutting plane that may be constructed from the tableau has the effect of cutting off a polytope containing some fraction of the feasible points of the hypercube, including one or more vertices.

We are interested in determining the fraction of the feasible points that is cut off. The fraction of the feasible points that is cut off can be considered as the probability that a random point in the hypercube is eliminated by the cut. Alternatively, that fraction may be thought of as a reduction of the hypervolume of the cube. Because hypervolume is difficult to visualize in more than about three dimensions, we will use the probability approach.

Power of Chvátal Cuts

We first consider the simple case of Chvátal cutting planes. Suppose we have a Chvátal cutting plane $\sum_{i \in S} -x_i \leq -1$, for some subset S of the nonbasic variables. Suppose we have $2^{|S|}$ random points uniformly distributed in the $|S|$-dimensional hypercube defined by those nonbasic variables. Each such cutting plane cuts off $\frac{1}{2^{|S|-1}}$ of the hypervolume of the $|S|$-dimensional hypercube defined by those variables, properly containing exactly one vertex of the hypercube. We expect that $\frac{1}{2^{|S|-1}}$ of the randomly distributed points are excluded by the cutting plane. With high probability, at least $\frac{1}{2^{|S|}}$ of the randomly distributed points are properly excluded by the cutting plane.

The $|S|$-dimensional hypercube is a projection of the higher-dimensional N-dimensional hypercube defined by all of the non-basic variables. Each point (resp. vertex) of the $|S|$-dimensional hypercube is the projection of $2^{N-|S|}$ points (vertices) of the N-dimensional hypercube. Hence, $\frac{1}{2^{|S|}}$ of the points and vertices of the N-

dimensional hypercube are properly excluded by the cutting plane. The number of remaining points is reduced by a factor of at least $\frac{2^{|S|}-1}{2^{|S|}}$.

Power of Canonical Cuts

Next, consider the case of canonical cutting planes. Suppose we have a canonical cutting plane $\sum_{i \in S} -x_i \leq -k$, for some set S of nonbasic variables. Each such cutting plane is satisfied by $\sum_{j=k}^{|S|} \binom{|S|}{j}$ of the vertices of the $|S|$-dimensional hypercube. The fraction of the vertices of the N-dimensional hypercube of the nonbasic variables that are properly excluded by the cutting plane is just $\frac{\Sigma_{j=0}^{k-1}\binom{|S|}{j}}{2^{|S|}}$. Suppose we have $2^{|S|}$ random points uniformly distributed in the $|S|$-dimensional hypercube. With high probability, at least $\frac{\Sigma_{j=0}^{k-1}\binom{|S|}{j}}{2^{|S|}}$ of the random points are properly excluded by the cutting plane.

As before, the $|S|$-dimensional hypercube is a projection of the higher-dimensional N-dimensional hypercube defined by all of the non-basic variables. Each point (resp. vertex) of the $|S|$-dimensional hypercube is the projection of $2^{N-|S|}$ points (vertices) of the N-dimensional hypercube. Hence, we may expect that at least $\frac{\Sigma_{j=0}^{k-1}\binom{|S|}{j}}{2^{|S|}}$ of the points ($\frac{\Sigma_{j=0}^{k-1}\binom{|S|}{j}}{2^{|S|}}$ of the vertices) of the N-dimensional hypercube are excluded by the cutting plane. The number of remaining points is reduced by a factor of $\frac{2^{|S|}-\Sigma_{j=0}^{k-1}\binom{|S|}{j}}{2^{|S|}}$.

Power of Gomory Cuts

To extend this result to non-canonical Gomory cutting planes, we cannot easily use such binomial expressions to exactly count the number of vertices or the probability that a random point is excluded. Suppose we have a Gomory cutting plane $\sum_{i \in S} -f_i x_i \leq -h$, for some set S of nonbasic variables. To estimate the power of the Gomory cutting plane, we use the fact that the strength of an inequality is just the fraction the variables x_i that must be true to satisfy the inequality.

Suppose the Gomory cutting plane has strength $d = \sqrt{\frac{h^2}{|S| \sum_{i \in S} f_i^2}}$. Consider again $2^{|S|}$ random points uniformly distributed in the $|S|$-dimensional hypercube. If the cutting plane is orthogonal to the diagonal of the hypercube, then the expected fraction of random points that are excluded by the Gomory cutting plane is at least

$$\frac{\Sigma_{j=0}^{\lfloor d|S|-1 \rfloor} \binom{|S|}{j}}{2^{|S|}}. \tag{3.81}$$

If the cutting plane is not orthogonal to the diagonal, then fraction of points that are excluded is even larger.

Correspondence of Strength with Power

The strength of a cutting plane corresponds with the fraction of the volume of the hypercube that is excluded by the cutting plane. To see this, compare the definition of cutting plane strength (definition 7) with the probability that a random point is excluded (equation 3.81). It is easy to verify that for constant $|S|$:

$$\frac{h_1}{|S|} > \frac{h_2}{|S|} \iff \frac{\Sigma_{j=0}^{h_1-1} \binom{|S|}{j}}{2^{|S|}} > \frac{\Sigma_{j=0}^{h_2-1} \binom{|S|}{j}}{2^{|S|}}. \tag{3.82}$$

Similarly, for constant h and different subsets S_1, S_2 of the variables:

$$\frac{h}{|S_1|} > \frac{h}{|S_2|} \iff \frac{\Sigma_{j=0}^{h-1} \binom{|S_1|}{j}}{2^{|S_1|}} < \frac{\Sigma_{j=0}^{h-1} \binom{|S_2|}{j}}{2^{|S_2|}}. \tag{3.83}$$

Hence, we see that the stronger of two cutting planes excludes a larger fraction of the hypervolume in each case. By a diagonal argument, in which at each step either the numerator of the strength is increased or the denominator of the strength is decreased, it can be shown that the strength of a Chvátal or canonical cutting plane is *measure* of the probability that a random point in the hypercube is excluded by that cutting plane (Halmos, 1950).

3.7.2 Expected Number of Cutting Planes

A natural question to ask is how many cutting planes are needed to eliminate all of the vertices of a hypercube. For any specific set if inequalities and specific sequence of cutting planes, we could obviously compute the exact number of cutting planes. It is more interesting to ask how many cutting planes are needed in the average case.

We first consider a simple case. Suppose that we have a hypercube of N dimensions, and that we have a procedure to find cutting planes with strength $\frac{1}{2}$. Each such cutting plane excludes an estimated

$$\frac{\Sigma_{j=0}^{\lfloor \frac{N}{2}-1 \rfloor} \binom{N}{j}}{2^N}$$

vertices. That is, each cutting plane with strength 1/2 excludes nearly 1/2 of the vertices. If the cutting planes are independent, then each cutting plane excludes nearly 1/2 of the remaining vertices that were not excluded by previous cutting planes. Hence, we may expect that only $O(N)$ such cutting planes are required.

To estimate the number of cutting planes that are required for strength $\neq 1/2$, we will consider the average strength of the cutting planes. Suppose we have a procedure to find cutting planes that have strength $\frac{h}{k} < 1$. We estimate the fraction of the remaining vertices that are excluded by each cut as in equation (3.81). Assuming independence, we need a number c of cutting planes such that:

$$\left(1 - \frac{\Sigma_{j=0}^{\lfloor \frac{h}{k}N-1 \rfloor} \binom{N}{j}}{2^N}\right)^c \leq \frac{1}{2^N} \tag{3.84}$$

This formula is entirely too complex to easily visualize the relationship between h, k, and c for a parameter N. The case with $h/k = 1/2$ was considered above, requiring an expected $O(N)$ cutting planes. It is easy to see that as $\frac{h}{k}$ is reduced, the number of cutting planes c increases. To illustrate the rate at which the number of cuts increases

Cut Strength	Number of Cuts	Cut Strength	Number of Cuts
0.50	59	0.30	74019
0.48	84	0.28	226602
0.46	126	0.26	768312
0.44	324	0.24	2.90488e+06
0.42	565	0.22	1.23465e+07
0.40	1050	0.20	5.95714e+07
0.38	2093	0.18	3.30278e+08
0.36	4499	0.16	2.13688e+09
0.34	10484	0.14	1.64649e+10
0.32	26619	0.12	1.55352e+11

Table 3.1: Estimated number of cutting planes required to exclude 2^{50} random points from a problem having 50 variables, for various values of cut strength.

as $\frac{h}{k}$ is reduced, we calculated the number of cuts required for various values of cutting plane strength, for a problem with only 50 variables. The results of that calculation are given in table 3.1. It is clear that small differences of cutting plane strength are associated with large differences in the number of cuts required.

3.8 Summary

In this chapter, we have developed the theory of a new complete algorithm for satisfiability. The design of the algorithm is based on the conditions that are known to be necessary if a complete SAT algorithm is to be efficient. The new algorithm is based on extended cutting plane proofs, using slack variables to express the constraints as equalities. Cutting plane proof systems are known to be able to simulate general Frege proof systems, and so are among the most powerful available proof systems for propositional logic.

Section 3.2 shows that cutting plane algorithms using slack variables provides a stronger proof system than the cutting plane proof systems defined by Chvátal (1984), Goerdt (1991), and Clote (1995). We show that a lower-bound on the length of cutting plane proofs, due to Krajíček (1997) and Pudlák (1997), does not hold for cutting plane algorithms with slack variables.

Section 3.1 studies the convex hull of the integer solutions of SAT, and describes the facet inequalities of the SAT polytope. We show that the convex hull of SAT can be written as a conjunction of canonical hyperplanes. In section 3.3, we develop a theory of lifting for inequalities in the SAT polytope. In section 3.4 we give a theory of lifting for inequalities that may include slack variables, and hence which may be used with cutting plane algorithms.

3.8.1 Algorithms

The algorithm uses Gomory cutting planes and a primal-dual simplex method, and so is somewhat similar to some cutting plane methods for integer programming. The new algorithm uses exact arbitrary-precision integer arithmetic to implement the integer-only simplex method as was proposed by Gomory (1963). Exact arbitrary-precision was not implemented by Gomory, and is rarely used in traditional integer programming codes. The use of exact arithmetic is necessary to avoid round-off errors which prevent rapid convergence (Jeroslow & Kortanek, 1971).

Several sections of this chapter develop subalgorithms to find strong cutting planes, and show that various necessary conditions are satisfied by these classes of cutting planes. The methods for finding and using strong cutting planes are the key to the new algorithm. The integer lifting lemmas, which allow lifting of Chvátal and Gomory cutting planes with some nonbasic integer-domain variables, are new. Previous lifting algorithms were applicable only if all nonbasic variables were Boolean. Lifted cutting planes are stronger than previous cutting planes, and should converge to facet-defining cuts in fewer iterations.

Several of the known necessary conditions are that any efficient algorithm must be able to solve a particular kind of "hard" problem efficiently. We show that cutting planes found by our algorithms can be used to efficiently solve those hard problems. Our first lifting algorithm finds valid canonical inequalities that lead to efficient solution of pigeonhole problems, which were nominated by Cook and Reckhow (1979) as being "hard" for a wide variety of known algorithms. Our second lifting algorithm generalizes the first to find strong valid inequalities that dominate Gomory cutting planes, and can avoid the known lower bound on monotone cutting plane proofs (Krajíček, 1997; Pudlák, 1997).

Finally, we discuss some algorithms to control and sequence the linear programming algorithms. In section 3.6, we consider the choice of objective function for the linear program. Three important heuristics are incorporated into our algorithm. First, the objective function maximizes the slack variables. Second, extra pivots are used to minimize the denominator when the optimal objective function is degenerate. Finally, extra pivots are used to obtain basic solutions that yield stronger cutting planes than would be obtained from the optimal solution. The use of extra pivots to find stronger cutting planes is new. The local search for stronger cutting planes is polynomially bounded, and depends on having a measure of cutting plane strength that is independent of any particular optimal linear solution. One such strength measure is provided, though other measures of cutting plane strength are clearly possible.

3.8.2 Complexity

The complexity of the individual sub-algorithms is clearly polynomial average time. It is well known that linear programs can be solved in polynomial time, and that the average-case complexity of the simplex method is polynomial. SAT polytopes do not includes the types of polytopes that are known to elicit super-polynomial behavior from simplex-type algorithms, and the use of a simplex method is not an essential part of the larger algorithm. The construction of the objective function, solving of linear programs, generation of cutting planes, and the local search heuristics, can all clearly be done in polynomial time.

The only possible source of super-polynomial complexity is the number of cutting planes that may be required. In proposition 7, we give an argument that if the problem is satisfiable, then there must exist a solution that can be found using only a polynomial number of facet inequalities. We have not provided demonstrated that

the presented algorithms for generating cutting planes can find such a solution. We have not demonstrated that the presented algorithms can prove unsatisfiability using only a polynomial number of lifted cutting planes. Any such demonstration would imply $P = NP$.

3.8.3 Finale

We have presented a complete algorithm for SAT that meets all conditions that are known to be necessary if a complete SAT algorithm is to be efficient. The new algorithm is based on extended cutting plane proofs, using slack variables to express the constraints as equalities. The algorithm uses two new methods for finding strong cutting planes: lifting, and local search. Both methods are specialized for the SAT polytope, but could potentially be adapted for use in general integer programming.

We have demonstrated that the new algorithm may generate facet cutting planes for SAT, and that a polynomial number of such facet cutting planes are sufficient to determine a model for any satisfiable SAT problem. We have not demonstrated that the new algorithm always does generate only facet cutting planes, or that any specific number of such cutting planes is sufficient to show unsatisfiability. The complexity of the algorithm depends on the number of cutting planes that are required to be generated. An analytic determination of that complexity seems to be out of reach at this time.

Chapter 4

Methodology

In chapter three, we provide some new algorithms to find strong canonical and lifted Gomory cutting planes. A proof of completeness was given, showing that the convex integer hull of any SAT problem may be described by a finite number of canonical cutting planes. The completeness of lifted Gomory cutting planes follows directly from Gomory's original proof (Gomory, 1963). We also suggested several heuristics, including a method of comparing the strength of cutting planes that is independent of any particular objective function, and a local search of adjacent vertices to find a strong cut

The theory to describe the complexity of a cutting plane algorithm is elusive. The complexity of finding one cutting plane is certainly polynomial, because it requires only solving one linear program and applying some heuristics. However, we have not found a subexponential bound for the number of such cutting plane iterations that might be needed to solve a given SAT problem. To gain some insight as to the number of cutting planes that are required, we conducted a computational experiment.In this chapter, we describe the experiment.

The purpose of the experiment is to verify our hypothesis:

> *Short cutting-plane proofs of unsatisfiability exist for many hard SAT problems. Algorithms and heuristics for finding strong cutting planes can be used to find such short refutation proofs.*

The experiment consisted of using algorithms and heuristics identified in chapter three to attempt a number of hard unsatisfiable SAT problems. The number of cutting planes in the resulting refutation proofs was determined. If the hypothesis is true, there should be a polynomial correlation between the number of cutting planes and the size of the SAT problem, for some identifiable class of SAT problems.

4.1 Implementation of Algorithms

A test-bed program was written to input SAT problems and to output the solutions and descriptive statistics in the DIMACS suggested format (DIMACS Center for Mathematics and Theoretical Computer Science, 1993a). The CutSat program was designed to allow easy implementation of various particular cutting-plane algorithms, and to experimentation with various design choices and parameter settings. In addition, the program includes instrumentation, to gather the results as indicated in section 4.3.3, below.

The algorithms identified in chapter 3 were implemented, including integer lifting of Gomory cutting planes. Additional algorithms and modifications of these algorithms were also implemented and tested on a selection of small problems. The program version that performed well for the small test problems was used for the large experiment.

Because the cutting plane algorithms require the use of exact arithmetic, no suitable program code could be found. None of the available source codes for linear programming implements exact rational or integer-only algorithms. Instead, Go-

mory's (1963) integer-only algorithm was implemented to solve the linear programming problems using arbitrary-precision integer arithmetic. The arbitrary precision integer arithmetic was implemented using the NTL package due to Shoup (1999). A compile time option also allows use of the native hardware arithmetic for small problems. Gomory's integer-only simplex algorithm was adapted to use a "bigM" implementation of the perturbation method to avoid degeneracy and cycling (Chvátal, 1983, pp. 138).

For the solution of the linear programming subproblems, an algorithm with guaranteed polynomial complexity could in principle have been implemented. However, the simplex type algorithms for linear programming are much less difficult to program than the alternative programs for solving linear programs, and are more practical. In practice, simplex type algorithms exhibit strongly polynomial expected time behavior.

All of the algorithms that are used to generate one lifted cutting plane have polynomial average complexity. The local search algorithms for cut strengthening and denominator reduction are limited by the size of the tableau, and thus have polynomial complexity. The unknown factor in the complexity of the compete algorithm was the number of cutting planes that are needed to solve a given SAT problem.

4.2 The Input Data

The CutSat program was used to attempt refutation of a number of unsatisfiable test problems. Several sources of test problems were used. Section 4.2.1 discusses the selection of problem sources. The choice of problem parameters, particularly the ratio of clauses to variables, may affect the results. Section 4.2.2 discusses the selection of the clause-variable ratio for our test problems. Finally, three sections present the three problem sets that were used. The DIMACS challenge problems were used to

provide a basis for comparisons with other algorithms working on that common set of problems. Two sets of randomly generated problems were also used.

4.2.1 Sources of Test Problems

To make an empirical determination of the average performance of the new algorithm over a range of problem size parameters, we require a sample of test problems. In addition, we will distinguish between unsatisfiable problems and satisfiable problems, because an algorithm may perform differently for those two classes. Unsatisfiable problems are known to be the most difficult, because any proof of unsatisfiability must be exhaustive.

We would have preferred to use an established collection of SAT problems as test data. The only such published collection is the DIMACS challenge problem set (DIMACS Center for Mathematics and Theoretical Computer Science, 1993b). The DIMACS collection is a collection of challenge problems for SAT solvers. It includes many problems with known answers, and many problems that have proven particularly hard to for previous SAT solver programs to solve.

However, the DIMACS collection contains only a few distinct sizes of randomly generated SAT problems, and very few of the problems are unsatisfiable. Satisfiable and unsatisfiable randomly generated 3-SAT problems are provided in the DIMACS collection, but only in sizes with 50, 100, and 200 variables, and only a few of each size. No other published collection of SAT problems is available.

Due to the absence of a published collection containing a sufficient number of input problems of useful sizes, we used randomly generated test problems. The use of randomly generated problems for testing of SAT solvers is commonly accepted (Gu et al., 1997; Asahiro, Iwama, & Miyano, 1996; Franco & Gelder, 1998). Randomly

generated 3-SAT problems are known to be difficult even for the best algorithms. To allow comparison with previous results, published random SAT problem generators were used to generate the test problems.

Several models of random problem generation have been proposed for SAT (Gu et al., 1997). In each case, the parameter m controls the number of clauses, n the number of variables. The ratio $r = m/n$, and the constant k denotes the number of literals per clause. The constant-density model attempts to generated problems with a constant number of occurrences of each literal by elementary means. Older constant-density generators have been criticized because these problems may have structural features that allow them to be easily solved (Franco & Ho, 1986; Franco, 1989). A generator of unsatisfiable problems due to Asahiro et al. (1996) has been proved to generate hard problems. Finally, the $\mathcal{M}^k_{m,n}$ model consists of sampling clauses with replacement from the $2^k\binom{n}{k}$ distinct clauses of size k in n variables, such that no two literals in one clause have the same variable.

4.2.2 Sizes of Test Problems

The size measures of the input problems include the the number of variables and the number of clauses. There is general agreement that random 3-SAT instances should be generated with clause/variable ratio approximately 4.25, because that clause-variable ratio gives hard 3-SAT problems (Asahiro et al., 1996). That ratio is known to generate problems that are about half satisfiable and half unsatisfiable.

It is also known that smaller ratios yield easier problems, and larger ratios yield harder problems. That is, the larger the ratio, the harder it is to find refutations for the problems that are unsatisfiable. For any fixed $r \geq 2^k \ln 2$, the shortest resolution refutation of a $\mathcal{M}^k_{m,n}$ random formula has super-polynomial number of steps with

high probability (Franco & Gelder, 1998). Hence, to examine the length of refutation proofs, it is appropriate to use input problems with clause/variable ratio greater $r \geq 2^k \ln 2$. For $k = 3$, the ratio should r be greater than 5.545.

4.2.3 DIMACS Challenge Problems

The unsatisfiable problems included in the DIMACS challenge benchmark set were attempted (DIMACS Center for Mathematics and Theoretical Computer Science, 1993b). The DIMACS challenge set of unsatisfiable problems includes a number of SAT problems that are thought to be hard for all known algorithms. Since 1993, some algorithms have been found that find short refutations for some of those problems, but for others there is still no known short refutation (Barth, 1994). However, the number of unsatisfiable problems in the DIMACS test suite is small, and so this test set does not provide a sufficient basis from which to draw any conclusions.

The size and parameters of the first problem set were fixed by the set of unsatisfiable problems present in the DIMACS problems.

4.2.4 Generated NSAT Problems

A set of test problems were generated by the NSAT random SAT problem generator due to Asahiro et al. (1996), with clause/variable ratio 4.25. The NSAT generator was previously used to generate some of the problems in the DIMACS challenge benchmark set (DIMACS Center for Mathematics and Theoretical Computer Science, 1993b). The problems generated by the NSAT problem generator are known to be unsatisfiable by construction, and the security of the NSAT generator has been proved. Because of the verified security, problems generated by NSAT are believed to be at least as hard as random SAT problems generated by any other generator.

4.2.5 Generated $\mathcal{M}_{m,n}^k$ Problems

A set of random test problems were generated using the $\mathcal{M}_{m,n}^k$ model with clause/variable ratio 6. The $\mathcal{M}_{m,n}^k$ model consists of sampling clauses with replacement from the $2^k\binom{n}{k}$ distinct possible clauses, such that no two literals in one clause have the same variable. It is known that for $r \geq 2^k \ln 2$, almost all formulas in the $\mathcal{M}_{m,n}^k$ model are unsatisfiable, and that *no known polynomial-time algorithm verifies unsatisfiability with high probability* (Franco & Gelder, 1998; Chvátal & Szemerédi, 1988).

A pseudo random generator program, named MSAT, was written to generate problems using the $\mathcal{M}_{m,n}^k$ model. $\mathcal{M}_{m,n}^k$ problems were generated for several values of n covering a range of sizes. We used $k = 3$ to generate 3-SAT problems. We used clause/variable ratio 6, to generate problems that are almost certainly unsatisfiable and are known to be hard for previous algorithms.

4.3 Test Procedure

After the test program was written, and a version was developed that performed satisfactorily on a small number of test problems, the program was tested on fresh input data. The test procedure consisted of generating fresh test problems, and then running the program on those test problems.

4.3.1 Test Problem Generation

The DIMACS test problems were obtained from the DIMACS ftp host, and stored locally. The files which contain unsatisfiable problems were identified and stored in a separate directory. Those problem files were used without alteration.

For each model of generated test problems, problems were generated with a number of variables ranging from 30 variables, in increments of 5 variables, up to 100 variables. For each number of variables, 10 problems were generated. In total, we generated 150 problems of the NSAT model, and at 150 problems of the $\mathcal{M}^k_{m,n}$ model.

4.3.2 Running the Test Problems

For the DIMACS problems, we ran the CutSat program to attempt each of the problems and record the result. At most one hour of cpu time was be allowed for any one problem. The computer used for this experiment is a Macintosh G4/450, so one hour was a relatively generous allowance of cpu time.

For the NSAT problems, the sequence of problems was arranged so that the smaller problems were attempted first. For each size of problem, we ran the CutSat program to attempt each of the problems of that size and record the result. At most one hour of cpu time was allowed for any one problem. The sequence was to be stopped, and larger problems not attempted, if less than half of the problems of one size were completed in one hour each.

For the $\mathcal{M}^k_{m,n}$, we used the same procedure as for the NSAT problems.

4.3.3 Measures to be Observed

For each test problem, data describing the size of the problem problem, the result, the number of lifted cutting planes used, and the computational effort expended was gathered. We were particularly interested in the number of cutting planes required. The computation to generate one cutting plane is known to be polynomial, so the number of cutting planes required is the interesting statistic.

For each problem, several observations were gathered:

1. The number of variables in the problem.

2. The number of clauses in the problem.

3. Whether or not the algorithm terminated before the time limit.

For each problem, if the CutSat program solved the problem within the one hour time limit, we also gathered:

1. Whether or not the problem is satisfiable (the answer).

2. The number of cutting planes that were used.

3. The number of seconds of cpu time that were used.

4.3.4 Analysis of the Results

For each problem set, we used Mathematica to fit both a polynomial function and an exponential function to the experimental data. For both functions, the size of the problem was measured as the number of variables in the problem, and the experimental data of interest was the number of cutting planes used by the CutSat algorithm. The quality of the correlation between each function and the experimental data is measured as the sum of the square error. The function that best correlates with the experimental data was then determined by comparing the sum-square error between the fitted functions and the experimental data.

If the particular algorithms in the CutSat program are indeed useful for finding refutations, refutations would be found for enough problems to enable the correlation to be calculated. We expected that if the hypothesis was true, then the number of cutting planes required in a refutation should best correlate with a polynomial of the size of the problem. Similarly, if the hypothesis was false, the number of cutting

planes required in a refutation should best correlate with an exponential function of the size of the problem.

4.3.5 Criteria for Success

We determined to deem that the experiment supports the hypothesis, if there is a correlation between the number of lifted cutting planes and some low-degree polynomial in the size of the problems, for some tested class of SAT problems.

4.4 Summary

We gave in chapter three a new and complete characterization of the convex integer hull for SAT, and several algorithms to compute strong cutting planes. The characterization of the convex integer hull of SAT by canonical hyperplanes is new. Two algorithms to find strong cutting planes are also new. The new cutting plane algorithm is a complete algorithm that is capable of proving unsatisfiability. While the theory is correct, and the family of cutting planes is complete, the performance of the new algorithms was unknown.

In this chapter, we have described an experiment to measure the performance of the new algorithms, and to gain the experience of using the new cutting plane algorithms on some real SAT problems.

Chapter 5

Results

Several programs were successfully prepared to implement the algorithms described in chapter 3, and the test problem generators described in section 4.2.1.

In section 5.1, we describe the implementation of the cutting plane algorithms algorithms. In section 5.3, we describe the implementation of the test problem generators. In section 5.4, we give the results obtained by running the program with the described input data.

5.1 The CutSat Test Program Implementation

A program was written in `C++` to implement the algorithms discussed in chapter 3. The CutSat program is designed to enable testing of various cutting plane algorithms for SAT problems. The implementation has several significant features that distinguish it from previous implementations of cutting plane algorithms.

From a software-engineering point of view, the program is object-oriented. The measure of cutting plane strength is encapsulated in two classes, `CutMeasure` and `TableauMeasure`. The integer-only simplex algorithms are encapsulated in an

`IntegerSimplex` class. The cutting plane algorithms, including the integer lifting, is encapsulated in a `CuttingAlgorithm` class.

Exact integer arithmetic is fully implemented with integer-only simplex methods similar to those reported by (Gomory, 1963). We are not aware of any other implementation of integer-only simplex aside from the one reported by Gomory, and Gomory's original implementation was restricted to the natural word size of the machine.

After finding one optimal solution, a denominator-reduction method applies extra pivots to reduce the common denominator. The denominator-reduction algorithm is similar to the linear programming algorithms, in that it selects pivot positions and applies pivots.

LP representations of SAT problems are often highly degenerate. For many choices of objective function, the linear program encodings most SAT problems have numerous optimal solutions. Hence, SAT problems require cycle-avoiding algorithms to assure finite termination of the linear programming algorithms. The implementation uses a randomized perturbation method to assure finite termination by eliminating ties in pivot selection for both the linear programming algorithms and the denominator-reduction algorithms.

A local-search algorithm uses extra pivot operations to find an improved cutting plane. At each step, adjacent feasible solutions are examined to see of one of them gives a stronger cutting plane than the current feasible solution. The local search is randomized in the sense that only a random sample of the cutting planes given by each adjacent feasible solution are examined.

Integer lifting is used to further improve the best cutting plane before it is added to the tableau. During the integer lifting procedure, some variables may be fixed to zero by observing that restricting the variable to a nonzero value yields an infeasible tableau.

Each of these features is presented in detail in the following sections.

5.1.1 Exact Integer Arithmetic

Two modes of integer arithmetic are implemented, controlled by a conditional compilation switch. If a macro `USE_EXACT_INTEGER_ARITHMETIC` is defined, the program uses arbitrary-precision integers for the simplex and cutting plane calculations. Otherwise, the program uses "`long long`" integers, which are 64 bit signed integers. The file "`Integer.h`" defines the type "`Integer`" according to the definition of `USE_EXACT_INTEGER_ARITHMETIC`. The code is available in Appendix A.4.

The implementation of arbitrary precision arithmetic uses a library due to Victor Shoup (Shoup, 1999). A few minor modifications were needed to correct a memory leak, and to optimize memory allocation. Exact arithmetic is rather slow, but never overflows unless heap memory is exhausted.

The difference in execution time between the two integer arithmetic implementations is a logarithmic factor of the largest integers encountered. For problems of practical size, the largest integers encountered fit in a 64 bit integer, and so the difference is essentially a constant factor. An arithmetic operation that requires a single instruction on native 64 bit integers requires a few dozen instructions in the exact arithmetic package.

The `pivot` method of the `IntegerSimplex` class includes code to detect overflow if it should occur. That overflow-detection code is enabled if a macro `CHECK_64_INTEGER_OVERFLOW` is defined. The denominator-reduction features of the algorithm help to avoid overflow. In tests with SAT problems of practical size, overflow was not observed.

5.1.2 Pseudo-Random Number Generator

During the development, the pseudo-random number generators provided with the CodeWarrior compiler was found to be defective. The sequences of generated numbers failed to pass commonly accepted statistical tests of randomness. To guard against possible error due to poor quality random numbers, a known good pseudo random number generator was implemented. The R250 generator was selected because it is very fast, the source code is available, and R250 is guaranteed to give very high quality pseudo random numbers (Carter Jr., 1994; Kirkpatrick & Stoll, 1981). The wrapper code that was needed to adapt the R250 generator to this purpose is presented in Appendix A.2.

5.1.3 The Integer Simplex Tableau

The class `IntegerSimplex` encapsulates the data structures of the tableau, together with the various linear-programming algorithms that operate on that tableau structure. The data structure includes the matrix of coefficient numerators, the common denominator, the two vectors of Variable names, and a few miscellaneous items.

Figure 5.1 shows the essential data structure of the simplex tableau. The data structure is a full (non-sparse) matrix representation. The matrix elements are the numerators of the linear coefficients, and the common denominator is stored separately. The objective function coefficients are stored in row zero of the matrix. The constants are located in column zero of each row vector. The objective value is located in column zero of row zero.

The `IntegerSimplex` class has the usual constructors and destructors. In addition, it provides methods to set the problem into the tableau, and to read the results from the tableau, and to print parts of the tableau. The prototypes of these methods

```
public:
    // The tableau structure
    int                 rows;           // size of the tableau
    int                 cols;
    Integer             denominator;    // The common denominator
    vector<vector<Integer> > mat;       // The matrix of numerators
    vector<Variable>    basic_vars;     // primal basis variables
    vector<Variable>    nonbasic_vars;  // dual basis variables
```

Figure 5.1: Data Structures of the Simplex Tableau

are shown in figure 5.2.

The `setObjective` method sets the given `vector<Integer>` into row zero of the matrix, and saves a second copy for later use by the `readPrimalSolution` and `readDualSolution` methods. The `addConstraint` method checks for duplicate constraints, and adds the given `vector<Integer>` to the matrix as a new row only if it is not identical to a previous constraint.

5.1.4 The Simplex Methods

The real work of linear programming is performed in just a few methods of the `IntegerSimplex` class. Both primal and dual solution algorithms are provided, and also a primalDual method. All of these methods use the same pivot method, which applies one pivot to the tableau. The `primalSimplex` method implements a version of the usual linear programming algorithm. The `dualSimplex` method implements the dual simplex algorithm. The `primalDualSimplex` method uses dual pivots to find a feasible solution, then calls `primalSimplex`. We give detailed explanation of the `primalSimplex` method only. The other two are similar.

The `primalSimplex` method iteratively selects a pivot column and row, and pivots the tableau, while the tableau is not optimal. The pivot selection method searches all

```
public:
    // Methods for setting the problem into the tableau,
    void    setObjective(const vector<Integer>& obj);
    bool    addConstraint(const vector<Integer>& constraint);

    // Methods for reading the results out of the tableau.
    LPStatus readPrimalSolution(Integer& objective,
                    vector<Integer>& primalVariables,
                    vector<Integer>& primalSlacks,
                    Integer& denominator);

    LPStatus readDualSolution(Integer& objective,
                    vector<Integer>& dualVariables,
                    vector<Integer>& dualSlacks,
                    Integer& denominator);

    // Method to print one inequality
    void printRow(ostream& os, int row);

    // Method to print the vector of nonbasic variables
    void printNonbasicVariables(ostream& os);
```

Figure 5.2: Input/Output methods of `IntegerSimplex`

```
public:
    // Simplex algorithms
    LPStatus    primalDualSimplex();
    LPStatus    dualSimplex();
    LPStatus    primalSimplex();

    // The pivot operation
    void        pivot(int  row, int  column);
```

Figure 5.3: Methods of `IntegerSimplex` that implement basic linear programming algorithms

possible pivot columns to find the primal pivot that gives the largest increase of the objective function. The method to select a primal pivot in a given column is slightly complicated by the need to avoid cycling.

Cycle Elimination

SAT problems are often highly degenerate. For many SAT problems, the linear program has numerous optimal solutions. This common feature of SAT problems requires that the simplex algorithm be implemented with additional cycle-avoiding algorithms to assure finite termination. This implementation uses a "perturbation" method, and breaks any remaining ties randomly. Both methods are discussed by Chvátal (1983, pp. 138). Essentially, each element of column zero is perturbed by a small amount, and the perturbances serve to break ties during pivot selection. By reducing the incidence of ties during pivot selection, the perturbations avoid degeneracy and cycling. If each possible basis solution has a distinct objective value, then the algorithm must terminate at a single optimal solution. However, the perturbation method does not guarantee the complete elimination of all ties. Any remaining ties are broken randomly.

The perturbation values are actually stored separately, in a `vector<Integer>` named `colZero`. Each element of `colZero` is effectively the low-order part of the corresponding `Integer` in column zero of the matrix. The two values are combined *as if* the matrix element in column zero was multiplied by a large multiplier M and added to the corresponding element of the `colZero` vector. The value of M is never made explicit, but is assumed to be sufficiently large so that the largest value in `colZero` is very small compared to M. The technique is similar to the "big-M" method for finding feasible solutions (Schrijver, 1986; Chvátal, 1983). An example that illustrates the technique is given in Figure 5.4. The pivot selection algorithm uses

`choosePrimalPivotRow` to select the pivot row within one column. The algorithm selects the smallest quotient `mat[i][0]/mat[i][col]`, and breaks ties by selecting the smallest quotient `colZero[i]/mat[i][col]`.

The use of random perturbations fails, and may allow cycling, if ties occur between the perturbation values themselves. The second element of the cycle-avoiding algorithm detects when these ties occur, and chooses randomly. Finite termination of the methods is assured by these random choices (Schrijver, 1986, pp. 138). The perturbations serve to minimize the number of random choices that are needed.

The only other element of the `primalSimplex` method that is particularly unusual is due to the all-integer representation of the tableau. The matrix contains only the numerators of each value, and the common denominator is stored separately. The quotients that are needed for pivot selection are formed using only the numerators, since the common denominators cancel each other out in each case. Similar logic appears in the pivot method. The dual simplex method is just a mirror image of the primal simplex method, and the primal dual method is also very similar.

5.1.5 The Denominator-Reduction Algorithm

Gomory (1963) presents the original method of cutting planes for integer programming and the original integer-only method. In Gomory's algorithm, all of the values in the tableau are expressed using a single common denominator, which may become quite large. Gomory mentions that one may choose when to add cutting planes, noting that each pivot in a new cutting plane has the effect of reducing the common denominator because the coefficients of a cutting plane are less than one.

In the linear program solutions of SAT problems, it is common to have coefficients with value zero in the objective row of the optimal tableau. In such cases, multiple

```
int
IntegerSimplex::choosePrimalPivotRow(int &col)
{
    int         best_row = -1;
    Quotient    best_M_q;   // Major part of best quotient
    Quotient    best_u_q;   // micro part of best quotient

    for ( int i=1; i<=rows; i++ )
    {
        if ((sign(mat[i][col]) > 0) && (sign(mat[i][0]) >= 0))
        {
            // Compute (Major,micro) quotient for this column.
            Quotient M_q = Quotient(mat[i][0],mat[i][col]);
            Quotient u_q = Quotient(colZero[i],mat[i][col]);

            // Keep the row with smallest (Major,micro) quotient,
            // And break ties randomly.
            if ( (best_row == -1) ||
                 (M_q < best_M_q) ||
                 ( (M_q == best_M_q) &&
                   ( (u_q < best_u_q) ||
                     ( (u_q == best_u_q) &&
                       randomBit())))
            {
                best_M_q = M_q;
                best_u_q = u_q;
                best_row = i;
            }
        }
    }
    return best_row;
}
```

Figure 5.4: Pivot selection within one column.

distinct solutions of the linear program give the same objective value. It may be that some optimal solutions have a smaller denominator than others. When this is the case, it is possible to choose the solution having the smaller denominator while preserving the optimality of the solution.

The `IntegerSimplex` class provides a `denominatorReduction` method that looks for feasible pivots which both do not change the objective value and also reduce the common denominator. Such pivots are either: a primal pivots with a zero coefficient in row zero; a dual pivot with a zero coefficient in column zero. When several such pivots are available, the one giving the least common denominator is taken. This method does not necessarily find the optimal vertex that has the minimal common denominator. Instead, it finds a local minima such that none of the adjacent optimal vertices has a lower common denominator.

5.1.6 The Cutting Plane Algorithm

The cutting plane algorithm is implemented as the `operator()` method of class `CuttingAlgorithm`. Figure 5.5 presents an outline of the code for the top level of the algorithm. Some details are omitted to fit the display on one page. The complete code is available in Appendix A.7. This `CuttingAlgorithm` method is a variant of the usual cutting plane algorithm. Every cutting plane algorithm iteratively solves the linear program, checks for fractions, and adds one or more cutting planes. This algorithm contains two additional steps, and the method of adding cutting planes is unusual. The three differences between `CuttingAlgorithm` and the usual cutting plane algorithm are:

1. The use of a `denominatorReduction` method, which searches for adjacent optimal and feasible solutions and may modify the tableau by pivoting to an

```
LPStatus CuttingAlgorithm::operator()(IntegerSimplex& tab)
{
    LPStatus status;
    while (1)
    {
        // Solve the LP to optimal, and test unsatisfiability
        status = tab.primalDualSimplex();
        if (status == INFEASIBLE) break;

        // If there are multiple optimal solutions,
        // choose one with a small common denominator.
        tab.denominatorReduction();

        // Use pivots to improve the best cutting plane.
        tab.primalCutImproving();

        // Show the basic solution.
        cout << "THE NONBASIC VARIABLES:" << endl;
        tab.printNonbasicVariables(cout);

        // Look for fractions in the solution
        if (tableau.isFractionalSolution())
        {
            // Generate and lift one or more cutting planes
            status = applyCuts(tableau);
        }
        else
        {
            status = FEASIBLE;
            break; // satisfied and integer
        }
    }
    return status;
```

Figure 5.5: Outline of Cutting Algorithm. Some details are omitted to fit the code on one page. The complete code is available in Appendix A.7.

adjacent feasible solution that has a smaller common denominator.

2. The use of the `primalCutImproving` method, which searches adjacent feasible solutions for better Gomory cutting planes and may modify the tableau by pivoting to an adjacent feasible solution that contains a better cutting plane.

3. The details of the `applyCuts` method, which generates a cutting plane and applies a lifting algorithm to the generated cut before adding it to the tableau.

The `denominatorReduction` method has been discussed above, in section 5.1.5. The `primalCutImproving` method and the `applyCuts` method will be discussed in the following sections.

5.1.7 Measuring the Cutting Planes

Any cutting plane algorithm has at its core some method of choosing from the available cutting planes. In this algorithm, we search adjacent basic solutions, and hence also must choose also the basic solution that contains the chosen cutting plane. To make these choices, it is necessary to have some function to measure the desirability of a cutting plane, or at least to compare two cutting planes to choose between them. In the usual optimization-based methods used for integer programming, the strength of a cutting plane is often considered as the distance from the current basic solution. We consider also some alternative measures.

A *measure* on a set is a an additive monotonic set function. An *outer measure* on a set is a sub-additive monotonic set function (Doob, 1993; Halmos, 1950). The sub-additive property is a essentially just generalization (to higher dimensions) of the familiar geometric triangle inequality. Either kind of measure determines an induced partial order of the subsets of the set. An *approximate measure* is just a set function

that approximates a measure.

The `CutMeasure` class encapsulates the methods and data to implement an approximate measure of the subset of the hypercube that is cut off by a cutting plane. The `TableauMeasure` class encapsulates the methods and data to implement a corresponding approximate measure of entire `IntegerSimplex` objects, with additional criteria to decide between tableaus when there is a tie between cutting planes. Each is intended to implement a subadditive partial ordering of the subsets of a hypercube, where the subsets are defined by cutting plane inequalities. The numerical value of any particular measure function is unimportant. The implementation of `CutMeasure` determines a partial ordering of cutting planes, and hence can be used to compare two cutting planes. Similarly, the implementation of `TableauMeasure` induce a partial ordering of `IntegerSimplex` objects.

The public interface of both `CutMeasure` and `TableauMeasure` include comparison operators. The comparison operators (`>`, `<`, `==`, `>=`, `<=`) of `CutMeasure` implement the induced partial order on cutting planes. The comparison operators internally use the `value()` method of `CutMeasure`, which returns a floating-point number that approximately represents the abstract subadditive measure of the cut. The comparison operators of `TableauMeasure` induce a partial order on entire tableaus by first comparing the `CutMeasure` of the best cut in each tableau, and then using other criteria to break ties.

`CutMeasure` provides an assortment of geometric measure functions, including the `distance`, the `strength` (fraction of diagonal), and the `power` (volume). These functions are based on Euclidean geometric measures of the hypercube and the cutting plane, such as distance to the cutting plane, and the volume excluded by the cutting plane. The geometric interpretation makes it very easy to see that the various functions are indeed (approximate) outer measure functions. The `power` method to

Value Function	Explanation of Function
`distance()`	Distance from origin to the cutting plane
`diagonolDistance()`	Distance from origin to intersection of cutting plane with the diagonal of the hypercube.
`strength()`	Distance as a fraction of the diagonal of the hyperplane
`power()`	Fraction of the volume eliminated by the cut, using an orthogonality assumption to estimate the volume
`logPower()`	Logarithm (base 2) of power
`distance() / (count * count * count)`	A polynomial that is easy to evaluate.
`(log2(strength()*count) - count`	A fast approximation of logPower

Table 5.1: Cutting plane measure functions defined in `Parameters.h`

compute the volume suffers from arithmetic overflow, and so a `logPower` method is defined to compute the logarithm of the power. Because the logarithm function is monotonic increasing, the `logPower` function induces the same partial order as the power method.

The selection of which measure function to use for to solve a SAT problem may be modified by editing the code of the `value()` method of class `CutMeasure`. The present implementation defines the value of a cutting plane according to a macro `CUT_VALUE_FUNCTION` defined in the header file `Parameters.h`. Table 5.1 lists the several definitions that are currently listed in the parameters file. Several of these

definitions simply invoke other methods of `CutMeasure`. The `power` and `logPower` methods are computationally expensive, and the `power` method tends to suffer arithmetic overflow. The values of a large number of cutting planes must be evaluated for each step in the proof, according to the size of the sample of cutting planes that is chosen. Hence, the efficiency of the value function is a consideration. The polynomial `(distance() / (count * count * count))` is easy to evaluate, and approximates the ordering induced by the `power` function over a range of problems sizes. The expression `(log2(strength()*count) - count)` is also easy to evaluate, and approximates the `logPower` function over a wider range of problem size.

Preliminary experiments using a small sample of test problems indicated that the last expression provides fast evaluation, and also uses nearly as few cutting planes as the power function. Based on the preliminary result, we used the expression `(log2(strength()*count) - count)` as the value of a `CutMeasure` for the large experiment.

5.1.8 Searching for a Stronger Cut

The method of using a local search to find strong cutting planes is implemented by a method of `IntegerSimplex` named `primalCutImproving`. The choice to implement this algorithm as a method of class `IntegerSimplex` is purely a convenience. The algorithm could have been implemented in as subclass of `IntegerSimplex`, or as a method of the `CuttingAlgorithm` class.

The primal cut-improving method is structurally similar to a primal simplex method, and uses the internal representation of the underlying `IntegerSimplex`. The pivot selection algorithms used by `primalCutImproving` are similar to the pivot selection algorithms in the simplex methods.

The basic idea is to search the possible pivots that give feasible solutions, to find a pivot that gives cutting plane that is better than any available in the current feasible solution. The implementation uses the `TableauMeasure` class to measure the strength of a sample of cutting planes at each adjacent feasible basic solution, and takes the pivot that gives the greatest `TableauMeasure`. The principle difference between the cut improving method and the primal simplex method is the way in which the value of a pivot is determined. The primal simplex method seeks to maximize the value of the objective function. The primal cut-improving method seeks to maximize a different measure of the tableau.

The key to the `primalCutImproving` method is the pivot selection. Rather than selecting pivots that improve the objective function, the cut improving method selects pivots that give better cutting planes. The `primalCutImproving` method uses `measureCuts` to measure the initial tableau using all of the Gomory cutting planes in the current tableau. That initial measure provides a baseline against which improvements can be compared. For each possible pivot, method `measureCutsAfterPivot` is used to measure the tableau that would result if that pivot were taken, using a sample of the cutting planes that would be available in the resulting tableau.

The order of the search is randomized. The columns are searched in random order. For each column, method `choosePrimalCutImprovingPivotRow` is used to search the pivot rows within that column. The rows are searched also in random order. The pivot giving the greatest `TableauMeasure` is selected. When there is a tie among the tableaus that would result from several different pivot positions, the first one is kept. Because the search order is random, the ties are broken randomly.

Method `measureCutsAfterPivot` uses `TableauMeasure` to measure the tableau that would result if a proposed pivot were taken, using a sample of the cutting planes that would be available in the resulting tableau. The sample size is controlled by

symbol `CUT_SAMPLE_SIZE`, which is defined in file `Parameters.h`. The for each possible pivot, the sample of cutting planes is drawn by considering the rows in random order without replacement, until the required number of cutting planes has been examined or all rows have been considered. For each row, if the pivot would result in a fractional basis variable in that row of the tableau, then the Gomory cutting plane is generated and measured.

5.1.9 The Integer Lifting Algorithm

In section 3.4.4, we gave an algorithm for lifting an integer program defined by an SAT problem. That algorithm uses the solution of a linear program to justify increasing the coefficient of a literal by one. The algorithm is implemented in method `cutLifting` of class `CuttingAlgorithm`.

After choosing an inequality (3.71) represented by a row of a basic solution to the linear program, the algorithm seeks to justify the derivation of a lifted inequality (3.66) that dominates (3.65). In the implementation, we choose the literal L_j to be the negation of one of the literals that appears in the inequality. Reversing the sense of the inequality sign, the inequality (3.71) may be written as:

$$\sum_{v \in N'} -a_v x_v \leq -b \tag{5.1}$$

and the inequality (3.66) may be written as:

$$\sum_{v \in N'} -a_v x_v - L_j \leq -b - 1 \tag{5.2}$$

Now, by choosing the literal L_j to be $\overline{x_{v'}} = (1 - x_{v'})$, for some $v' \in N'$, we may rewrite

the inequality (5.2) as:

$$\sum_{v \in N'} -a_v x_v - (1 - x_{v'}) \leq -b - 1 \tag{5.3}$$

$$\sum_{v \in N'} -a_v x_v - 1 + x_{v'} \leq -b - 1 \tag{5.4}$$

$$\sum_{v \in N'} -a_v x_v + x_{v'} \leq -b \tag{5.5}$$

Hence, the lifting operation may be thought of as increasing the coefficient of some $x_{v'}$ in the inequality by one. It is important to modify the coefficients only by integer quantities, because the all-integer simplex method depends on preserving a unimodular transform from the initial tableau. The introduction of a non-integer modification would cause the all-integer pivot algorithm to fail. For Gomory cutting planes, the values of all of the coefficients $-a_v$ are in the range $(-1, 0]$, so increasing any coefficient by one yields a non-negative coefficient.

There are two parts to the implementation lifting algorithm. First, we must choose the subset and sequence of the variables for which the lift will be attempted. Second, we must construct and solve the linear programs to actually test the lifts.

Choosing the Variables to Lift

It is known that different sequences of variables give different linear lifting results (Nobili & Sassano, 1989; Gu et al., 1995). That is, the choice of the order in which the possible lifts are tried has an effect on the result. The heuristic used in `cutLifting` to make the choice uses two factors. The variables with the largest (absolute value) coefficients are tried first. Ties are broken by choosing the variable such that setting the variable to value one causes the largest total infeasibility of other inequalities. This is similar to the heuristic suggested by (Gu et al., 1995).

The heuristic of choosing first the variables with large negative coefficients is

justified by considering the effect of setting the variable to value one. A large negative coefficient causes that literal to make a large contribution to the required value of the sum. The heuristic of choosing first the variable that gives the largest infeasibility of other constraints is justified by considering that the infeasibility of other constraints contributes to forcing other nonbasic variables to nonzero values. The lift is justified if some other nonbasic variables in the inequality 5.1 are forced to nonzero values.

Because the testing of one lift involves solving a linear programming subproblem, it is a rather expensive operation. Hence, the implementation tests the lifting of only some subset of the variables in a cutting plane inequality. In particular, after some number of lift tests fail to find a lift, the lifting of remaining variables in that cutting plane may be skipped entirely. As a heuristic, this is justified by the observation that the lifting tests which are skipped are very unlikely to be successful, and so the cost of performing the additional tests is very likely to be wasted.

Testing One Lift

As developed in sections (3.4) through (3.4.4), the mathematical justification for lifting and the algorithm for testing one lift do not appear to be particularly easy. However, the implementation is actually not difficult. The key is to observe that there is an alternate proof of the lifting lemma that can be derived immediately from logic which appears in Gomory's 1963 paper. To show:

$$\sum_{v \in N'} -a_v x_v - L_{v'} \leq -b - 1 \tag{5.6}$$

It is required to show that the maximum value of:

$$\begin{aligned} \text{max: } & \sum_{v \in N'} -a_v x_v - L_{v'} \\ \text{such that: } & \sum_{j \in J} -a_{ij} x_j \leq -b_i \qquad \text{, for } i \in U \\ & -x_j \leq 0 \qquad \text{, for each } j \\ & x_j \leq 1 \qquad \text{, for each } j \\ \text{and: } & L_{v'} \leq 0 \end{aligned} \tag{5.7}$$

is less than b. Now, for choices of $L_{v'}$ that are nonbasic variables, this is not the case because the nonbasic solution has already the required value of that literal. Hence, it is enough to consider choices of $L_{v'}$ that are negations of nonbasic variables. For those literals, we may write $L_{v'} = (1 - x_{v'})$. The constraint $L_{v'} \leq 0$ gives the constraint $x_{v'} \geq 1$. So the problem is to test the maximum value of:

$$\begin{aligned} \text{max: } & \sum_{v \in N'} -a_v x_v + x_{v'} \\ \text{such that: } & \sum_{j \in J} -a_{ij} x_j \leq -b_i \qquad \text{, for } i \in U \\ & -x_j \leq 0 \qquad \text{, for each } j \\ & x_j \leq 1 \qquad \text{, for each } j \\ \text{and: } & -x_{v'} \leq -1 \end{aligned} \tag{5.8}$$

We can see that the last constraint is infeasible, so at least one pivot is needed to find a feasible solution. Now, if the maximum value of $\sum_{v \in N'} -a_v x_v + x_{v'}$ is strictly less than $(1 - b)$, we have:

$$\sum_{v \in N'} -a_v x_v + x_{v'} < -b + 1 \tag{5.9}$$

Gomory's (1963) insight was that the entire system is a module with modulus 1. Hence, the maximum value of the linear program (5.8) must be congruent to $-b$

(mod 1), so the strict inequality of (5.9) gives us the desired result. Inequality (5.9) and

$$\sum_{v \in N'} -a_v x_v + x_{v'} \equiv -b \pmod 1 \tag{5.10}$$

immediately implies:

$$\sum_{v \in N'} -a_v x_v + x_{v'} \leq -b \tag{5.11}$$

Now, the inequation (5.9) is true if and only if $\sum_{v \in N'} -a_v x_v < -b$. Hence, it is enough to test whether or not the maximum value of the linear program:

$$\begin{aligned}
\text{max: } & \sum_{v \in N'} -a_v x_v \\
\text{such that: } & \sum_{j \in J} -a_{ij} x_j \leq -b_i \qquad \text{, for } i \in U \\
& -x_j \leq 0 \qquad \text{, for each } j \\
& x_j \leq 1 \qquad \text{, for each } j \\
\text{and: } & -x_{v'} \leq -1
\end{aligned} \tag{5.12}$$

is strictly less than $-b$.

The implemented code in method `cutLifting` constructs the linear program (5.12) using the `IntegerSimplex` class. The tableau is constructed using the current basic solution together with all of the Gomory cutting planes that can be derived immediately from that basic solution. The objective function is just a copy of the cutting plane that we are attempting to lift, and one constraint is added to force the selected nonbasic variable to value at least one.

Modifying the Tableau

If a lift is found to be justified by testing the linear program, then the tableau representing the problem may be modified. There are two cases:

1. If the linear program (5.12) is infeasible, then the variable $x_{v'}$ cannot take on any nonzero value, and that literal may be eliminated from all constraints.

2. If the linear program (5.12) is feasible with maximum value strictly less than $-b$, then the constraint (5.1) may be replaced by:

$$\sum_{v \in N'} -a_v x_v + x_{v'} \leq -b \tag{5.13}$$

In the first case, rather than adding a constraint $x_{v'} \leq 0$, the implementation zeros the coefficients of the variable $x_{v'}$ in every row of the simplex tableau. Setting the coefficients to zero constrains that variable to remain nonbasic for all future iterations, and hence fixes the value of that variable to zero. In the second case, the coefficient of the variable $x_{v'}$ in the Gomory cut is modified, giving a stronger "lifted" cutting plane.

5.1.10 Applying Cutting Planes to the Tableau

Method `applyCuts` of class `CuttingAlgorithm` generates all Gomory cutting planes for multiplier $\lambda = 1$, selects which of those cutting planes is to be used, applies the `cutLifting` method to the selected cut(s), and adds the resulting lifted cutting plane(s) to the tableau. The `applyCuts` method is essentially administrative, in that it does not actually do much of the work. There are two important heuristic implemented in `applyCuts`.

First, the present implementation adds at most one cutting plane to the tableau at each basic solution. That is, when multiple distinct cutting planes are available, only one is selected. This is in keeping with the goal of finding a short proof using the fewest number of cutting planes. A modification to use multiple of the available cutting planes could be useful, if the goal were to use the fewest number of iterations

or basic solutions.

Second, the present implementation does not necessarily use even one cutting plane at every iteration. If symbol `AVOID_USING_LARGE_CUTS` is defined, then cutting plane inequalities having a large number of literals are avoided. The avoidance algorithm is randomized, so that large cutting planes are less likely to be used than smaller cutting planes, but are not entirely excluded.

By avoiding the use of large cutting planes, we merely avoid the computational expense of computing the extra rows of the tableau during following iterations. Cutting planes with large numbers of literals do not reduce the volume of the polytope by much at all, so they contribute little to the proof. By excluding them, we avoid some computation and we avoid adding a step to the proof that is not likely to be helpful.

5.2 Reading the Output as Proof

The usual kind of cutting-plane proof of unsatisfiability consists of a sequence of inequalities, where some number of inequalities represent the initial clauses of the SAT problem, the others are derived from previous inequalities prescribed operations, with the last line of the proof being an inequality of the form $0 \leq b$ for some $b < 0$. The prescribed operations require that each inequality must be a positive linear combination of previous inequalities, or must be a cutting plane derived from one such inequality.

Because it uses simplex tableaus with slack variables, this algorithm is concerned with entire basic solutions of the linear system. The initial simplex tableau is constructed by adding slack variables to the initial inequalities. Subsequent simplex tableaus are derived from previous simplex tableaus by prescribed operations, with

the last tableau of the proof containing an equality of the form $s + 0 = b$, where s is a non-negative slack variable, for some $b < 0$.

The prescribed operations require that each tableau is derived from the previous tableau by a sequence of pivot operations, or by the addition of a lifted cutting plane derived from the previous tableau. The pivot operation constructs each row of the new tableau from a linear combination of two rows of the previous tableau. Because the rows represent equalities, the linear combination need not be a positive linear combination. Because linear combination is transitive, any sequence of pivot operations yields a tableau where each row is a linear combination of rows of the initial tableau. The derivation of a lifted cutting plane is a slightly more complex operation.

The derivation of a lifted cutting plane from a given tableau includes several steps. First, all of the distinct Gomory cutting planes are derived, and one of them is chosen to be lifted. Temporary subproblems are used in the sequential lifting algorithm, as presented in section (5.1.9). The sequential lifting algorithm uses solutions of the linear programming subproblems to show that the lifted cutting plane inequality is valid. The new tableau is then derived from the given tableau by adding the new lifted cutting plane inequality as a new constraint, with a new slack variable. Because the cutting plane inequality is a valid equality, all of the integer solutions that are valid for the given tableau are also valid for the new tableau.

The verification of such a proof requires only verifying the derivation of each tableau from the previous tableau. This, in turn, requires that either the tableaus or enough information to efficiently reconstruct the tableaus must be printed. The proof must include either the complete tableaus, or other information that is sufficient reconstruct the complete tableaus. In addition, the proof must include the derivation of the lifted cutting planes, or other information that is sufficient to reconstruct those

derivations.

5.2.1 Parameters to Print the Proof

A full proof of unsatisfiability that explicitly includes all of the tableaus and the derivations of all of the cutting planes is quite bulky. It is possible to consider a smaller amount of output as a proof. The implemented program has compile-time parameters that control how much detail should be shown in the proof.

The initial simplex tableau is easy to construct from the SAT problem. The initial tableau of an SAT problem has one line for each variable and one for each clause in the SAT problem. Because the initial simplex tableau is determined by the given SAT problem, it is not necessary to explicitly print the tableau. The implemented program has a conditional compilation switch to control printing of the initial tableau. If symbol `SHOW_INITIAL_TABLEAU` is defined, the CutSat program prints the initial tableau.

A full proof includes the derivations of each Gomory cutting plane used in the proof. These derivations are bulky, and are usually not interesting. However, either those derivations (or equivalent information) are needed to check a proof. If the symbol `SHOW_CUT_DERIVATIONS` is defined, the CutSat program prints the detailed cut derivations so that they may be inspected. Each detailed cut derivation prints the equality from which the Gomory cutting plane inequality is derived, and then the cutting plane inequality.

Alternatively, it is possible to verify that each asserted Gomory cutting plane inequality is in fact valid by reproducing the derivation. The proof checker may reproduce the derivation from the simplex tableau containing the basic solution from which the cutting plane inequality was derived, and then deriving the Gomory cutting

planes from that tableau.

Hence, both the tableaus and the cutting plane derivations may be checked if the proof-checker has sufficient information to reproduce the basic solutions. The basis solution is uniquely identified by giving the complete tableau, but it is also be uniquely identified by the vector of basic variables, or by the vector of non-basic variables. The most concise way to present this information is to identify the vector of non-basic variables. If the symbol `SHOW_NONBASIC_VARIABLES` is defined, the CutSat program prints the vector of non-basic variable names of each simplex tableau used in the proof.

It may be desired to inspect every simplex tableau used in the proof. If the symbol `SHOW_EVERY_TABLEAU` is defined, the CutSat program prints every simplex tableau that is used in the proof. The simplex tableaus are large, and there are as many of them as there are steps in the proof, so this option outputs a very large number of lines. There is not an option to show the simplex tableau after every pivot operation, because that amount of output would be impractically large for all but the smallest SAT problems.

The number of optimal tableaus is linear in the number of lifted cutting plane steps that are used in the proof, and the size of those tableaus is polynomial in the number of input clauses, the number of variables, and the number of lifted cutting planes. Hence, even if all of the printing parameters are enabled, the required print size would be merely impractical, rather than exponential in the number of lifted cutting-plane steps. If the number of lifted cutting planes is polynomial in the input size, then the proof size is also polynomial in the input size, for any choice of these parameters.

```
THE NONBASIC VARIABLES:
 s126 s15 s152 s67 s157 s146 s109 s154 s60 s113 s87 s139
 x20 s131 s85 s42 s94 s44 s117 s144 s32 s142 s128 s82
 x14 s127 s134 s121 s119 s48
```

Figure 5.6: Example of output showing the basic variables.

5.2.2 Reading and Checking the Proofs

We assume that the most concise presentation has been chosen. That is, the simplex tableaus are not normally printed, and the nonbasic variables are printed by defining `SHOW_NONBASIC_VARIABLES`. With these settings, the initial tableau can be reconstructed by the proof checker from the SAT problem, and so the initial tableau is known.

At each major step of the proof the program prints the nonbasic variables. This output is labeled by printing "`THE NONBASIC VARIABLES:`", followed by the list of the basic variable names. Figure 5.6 shows an example of a display of nonbasic variables. Knowing the list of nonbasic variables, it is possible for the proof checker to reconstruct the tableau at this point of the proof by applying a number of pivot operations to the previous tableau. The number of pivot operations is at most equal to the number of columns in the tableau, and hence the computation needed to reconstruct the tableau is polynomial in the number of variables.

Given the basic solution, the proof checker may check the cutting planes by deriving the Gomory cuts. Alternatively, if the symbol `SHOW_CUT_DERIVATIONS` is defined, the program prints the cut derivations. Figure 5.7 shows an example of program output that gives the derivation of one Gomory cutting plane inequality. Using this output, the derivation of the cut may be checked by two steps. First, the proof checker should verify that the equality is indeed present in the tableau. Second, the proof checker should verify that the Gomory cut of the given equality is indeed the asserted

```
The equality:
s76 -2s126 +1s15 -4s152 +3s157 +1s113 -2s87 -3x20 +3s85 +3s42
 +2s142 +4s121 = 2
 with denominator 3 generates the candidate Gomory cut:
-1s126 -1s15 -2s152 -1s113 -1s87 -2s142 -1s121 <= -2
```

Figure 5.7: Example of output showing the derivation of one Gomory cutting plane inequality.

```
THE CANDIDATE CUT:
-1s126 -1s15 -2s152 -1s113 -1s87 -2s142 -1s121 <= -2
```

Figure 5.8: Example of output showing the Gomory cutting plane inequality that is selected for lifting.

inequality. Both of these steps are trivial. Checking for the presence of the equality in the tableau requires time at most proportional to the size of tableau. Checking that the Gomory cut of the equality is the given inequality requires time proportional to the number of literals in the equality.

Given that the proof checker has verified the current simplex tableau and the current set of Gomory cutting planes that may be derived from that simplex tableau, the proof checker must be able to verify the sequential lifting operations. The first step of lifting is to select a Gomory cutting plane to be lifted. The program indicates the selected Gomory cut by printing the label "`THE CANDIDATE CUT:`" followed by the selected cut. Figure 5.8 shows an example of output indicating the selection of a cut for lifting. To verify that the candidate cut is indeed one of the Gomory cutting plane inequalities that exists in the current set of Gomory cuts is trivial, and requires time at most proportional to the size of the current simplex tableau.

The sequence of successful lifting operations is indicated by the program by printing a line for each successful lift. There are two cases of successful lifting. In one, a literal may be added to the cutting plane if the optimum solution of a certain lin-

```
FOUND A LIFT.  The term: -2s152 is lifted to: 1s152
```

Figure 5.9: Example of output indicating a successful lift.

ear programming subproblem is negative. The construction of that LP subproblem requires the previous simplex tableau, the current set of Gomory cutting plane inequalities, and the selection of one variable. If the lift is successful, the coefficient of a that variable in the candidate cut may be increased by one. Because of the all-integer format of the tableau, the coefficient is actually increased by the denominator of the tableau. When this case is detected, the program outputs a line with label "`FOUND A LIFT`." Figure 5.9 shows an example of output indicating that a lift has been found. To verify that the asserted lift is justified, the proof checker may construct the linear programming subproblem and solve it. The construction of that LP subproblem requires the previous simplex tableau, the current set of Gomory cutting plane inequalities, and the selection the variable. All of that information is available to the proof checker, and so it is possible to reconstruct the LP subproblem. By solving the LP subproblem, the proof checker may verify that the lift is justified. Solving the linear program requires time proportional to a low-degree polynomial of the size of the simplex tableau.

If the linear programming subproblem for a lift is not feasible, then a variable may be fixed. The LP subproblem is formed from the previous simplex tableau, the current set of Gomory cutting plane inequalities, and one additional constraint that forces the selected variable to be ≥ 1. If the addition of the a constraint setting that variable ≥ 1 yields a linear program that has no feasible solution, then the selected variable must have value 0. When this case is detected, the program prints a line labeled "`FOUND A FIXED VARIABLE:`" indicating that a variable may be fixed. Figure 5.10 shows an example of output indicating that a fixed variable has been found. To

```
FOUND A FIXED VARIABLE: s142 = 0
```

Figure 5.10: Example of output indicating that a variable is fixed to zero.

```
THE LIFTED CUT:
-1s126 +1s152 -1s87 <= -2
```

Figure 5.11: Example of output showing a fully lifted cutting plane inequality.

verify that the variable can indeed be fixed, it is sufficient for the proof checker to construct the linear programming subproblem that was used to test the lift of that variable and attempt to solve it. The output indicating a fixed variable also indicates the variable, so all of the information needed to construct the subproblem is available to the proof checker. The construction of the linear program and the solution of that linear program are the same as in the previous case of verifying a lift.

Finally, after all lifting steps have been verified, the lifted cut is displayed. This inequality is preceded by a line containing the label "`THE LIFTED CUT`." Figure 5.11 shows an example of output indicating that the sequential lifting of a Gomory cutting plane inequality is completed, and giving the the fully lifted inequality.

After some number of iterations, the program either finds a feasible integer solution or determines that none exists. If a SAT problem is unsatisfiable, the program prints a line "`NO FEASIBLE SOLUTION EXISTS.`" At this point, it is easy for the proof checker to verify that there is indeed no feasible solution to the current simplex tableau. Because every step in the derivation of the current simplex tableau has been verified, the nonexistence of any feasible solution for the current simplex tableau implies the nonexistence of any feasible integer solution for the original simplex tableau, and the unsatisfiability of the original SAT problem.

Alternatively, if the program finds a solution, it prints a line "`FOUND A FEASIBLE SOLUTION.`" In this case, the program also prints the feasible solution, and so it also

can be checked easily.

5.3 Test Problem Generators

The `rsat` and `nsat` random problem generators, due to Asahiro et al. (1996), were used with only minimal alteration. Some minor alteration was required to successfully compile the code using an ANSI standard compiler. Source code for the `rsat` and `nsat` generators is available at the DIMACS challenge web site (DIMACS Center for Mathematics and Theoretical Computer Science, 1993b).

The implementation of the `msat` random problem generator is trivially easy. The source code of the generator is presented in appendix B. To draw each independent 3-clause from all possible clauses, the code simply places all of the variables in a `vector<Variable>`, applies the `std::shuffle` library procedure to the vector, and uses the variables in the first three positions of the resulting shuffled vector. To apply the negation operators independently, each literal of a clause is negated with probability one-half.

5.4 Computational Results

The CutSat program was run with three sets of input problems. The DIMACS problems, a set of problems generated by the nsat pseudorandom generator, and a set of problems generated by the msat pseudorandom generator were run. This section reports the computational results obtained from those runs.

For each set of problems, the raw results are presented in a table. The results of curve-fitting calculations are presented, and a graph of the raw data with the fitted curve is displayed.

5.4.1 Unsatisfiable DIMACS Problems

The observations obtained by running the CutSat program using unsatisfiable DIMACS problems as input are given in Table (5.2). The table has one row for each problems in the problem set. The column labeled "Problem" just identifies the problem by name. The columns labeled "Vars" and "Clauses" give the number of variables and clauses, respectively, that appear in the problem. The column labeled "Solved" indicates whether or not the CutSat program solved the problem within a one-hour time limit. The column labeled "Satisfiable" indicates the solution. The column labeled "Seconds" indicates the time required to solve the problem. Finally, the column labeled "Cutting Planes" indicates the number of lifted cutting planes used in the proof of unsatisfiability.

Table 5.2: Computational results obtained by running the CutSat program on unsatisfiable problems from the DIMACS problem set.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
aim-100-1_6-no-1	100	160	yes	no	21	23
aim-100-2_0-no-1	100	200	yes	no	24	12
aim-100-2_0-no-2	100	200	yes	no	40	19
aim-100-2_0-no-3	100	200	yes	no	28	13
aim-100-2_0-no-4	100	200	yes	no	64	29
aim-200-2_0-no-1	200	400	yes	no	814	52
aim-200-2_0-no-2	200	400	yes	no	1225	57
aim-200-2_0-no-3	200	400	yes	no	1255	66
aim-200-2_0-no-4	200	400	yes	no	509	33
aim-50-2_0-no-1	50	100	yes	no	2	8
aim-50-2_0-no-2	50	100	yes	no	3	11
aim-50-2_0-no-3	50	100	yes	no	7	16
aim-50-2_0-no-4	50	100	yes	no	1	7
dubois20	60	160	yes	no	68	67
dubois21	63	168	yes	no	116	73
dubois22	66	176	yes	no	158	93
dubois23	69	184	yes	no	95	79

Table 5.2: Continued.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
dubois24	72	192	yes	no	171	90
dubois25	75	200	yes	no	373	151
dubois26	78	208	yes	no	291	142
dubois27	81	216	yes	no	284	138
dubois28	84	224	yes	no	274	129
dubois29	87	232	yes	no	472	164
dubois30	90	240	yes	no	600	178
dubois50	150	400	yes	no	2841	241
hole6	42	133	yes	no	33	99
hole7	56	204	yes	no	102	155
hole8	72	297	yes	no	436	284
hole9	90	415	yes	no	1352	437
hole10	110	561	no			
jnh2	100	850	yes	no	64	6
jnh3	100	850	yes	no	1260	73
jnh4	100	850	yes	no	424	26
jnh5	100	850	yes	no	112	12
jnh6	100	850	yes	no	663	39
jnh8	100	850	yes	no	187	13
jnh9	100	850	yes	no	75	6
jnh10	100	850	yes	no	108	12
jnh11	100	850	yes	no	260	20
jnh13	100	850	yes	no	191	15
jnh14	100	850	yes	no	107	10
jnh15	100	850	yes	no	229	19
jnh16	100	850	no			
jnh18	100	850	yes	no	860	57
jnh19	100	850	yes	no	162	10
jnh20	100	850	yes	no	504	30
jnh202	100	800	yes	no	34	3
jnh203	100	800	yes	no	380	33
jnh206	100	800	yes	no	728	55
jnh208	100	800	yes	no	539	40
jnh211	100	800	yes	no	147	13
jnh214	100	800	yes	no	405	32
jnh215	100	800	yes	no	134	13
jnh216	100	800	yes	no	958	59
jnh219	100	800	yes	no	842	48

Table 5.2: Continued.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
jnh302	100	900	yes	no	15	1
jnh303	100	900	yes	no	813	40
jnh304	100	900	yes	no	37	3
jnh305	100	900	yes	no	113	10
jnh306	100	900	yes	no	2713	109
jnh307	100	900	yes	no	37	4
jnh308	100	900	yes	no	582	28
jnh309	100	900	yes	no	44	5
jnh310	100	900	yes	no	54	3
pret150_25	150	400	yes	no	2812	215
pret150_40	150	400	yes	no	1906	185
pret150_60	150	400	yes	no	521	57
pret150_75	150	400	yes	no	2582	219
pret60_25	60	160	yes	no	148	94
pret60_40	60	160	yes	no	114	72
pret60_60	60	160	yes	no	104	73
pret60_75	60	160	yes	no	125	88

A graph of the relation between the number of variables and the number of lifted cutting planes is shown in Figure 5.12.

Using Mathematica, two regression calculations were carried out (Wolfram, 1999). The curves were fitted to determine the number of lifted cutting planes as a function of the number n of variables. The relation between the number of variables and the number of lifted cutting planes correlates with both a low-order polynomial and with an exponential function. Formula (5.14) gives the polynomial. Formula (5.15) gives the exponential function with the exponent being a low-order polynomial.

$$-4.48557 * 10^{-9} n^6 + 2.21128 * 10^{-6} n^5 - 0.000380681 n^4 + 0.0266796 m^3 - 0.620576 n^2 - 0.0343057 n - 0.00118877 \quad (5.14)$$

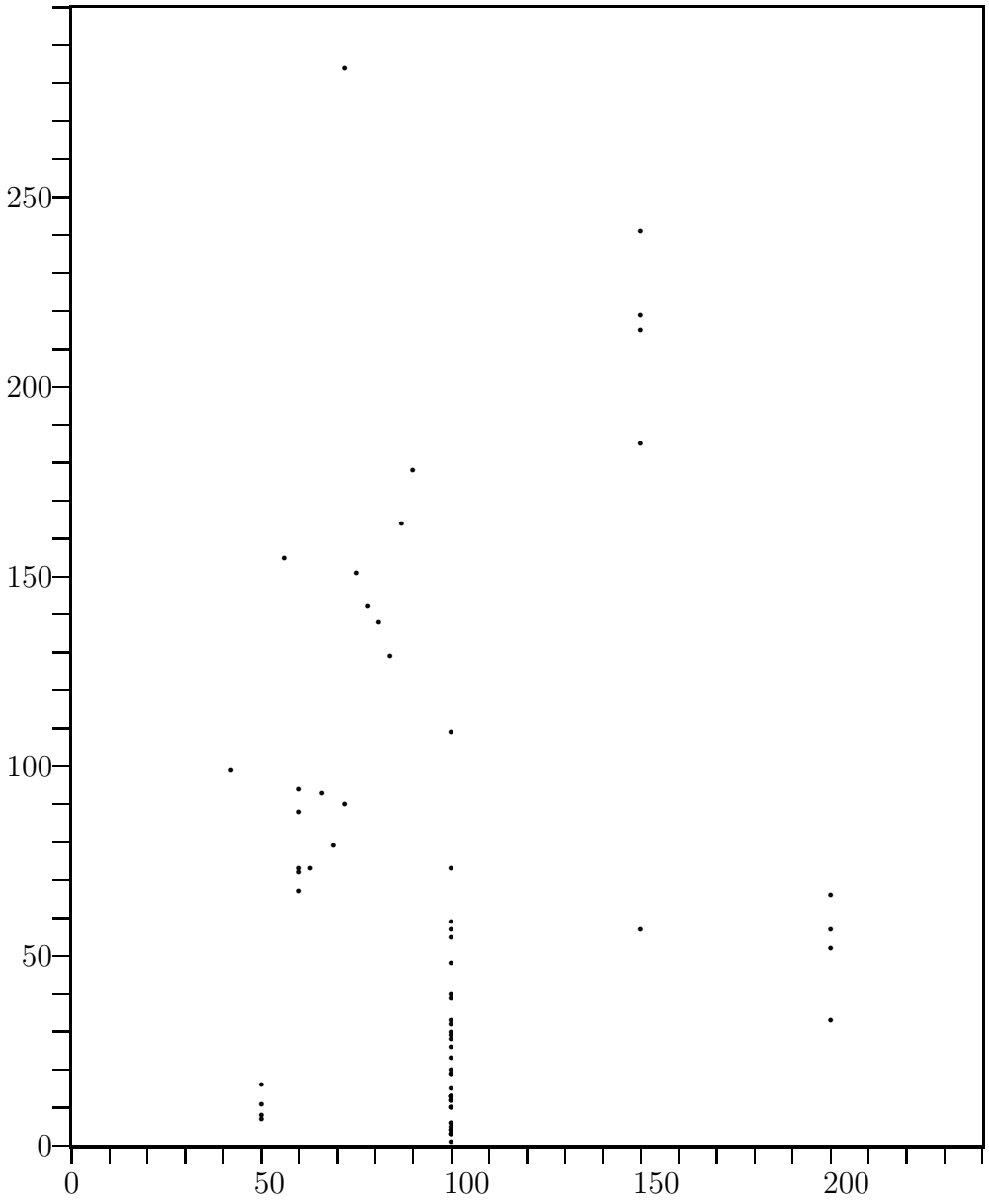


Figure 5.12: Plot of Number of Variables vs. Number of Cutting Planes for unsatisfiable DIMACS problems. The horizontal scale plots the number of variable in the problem. The vertical scale plots the number of lifted cutting planes used by CutSat program to solve the problem.

$$e^{f(m)}, \text{ where } f(m) =$$
$$-5.76453 * 10^{-11} n^6 + 2.75109 * 10^{-8} n^5 - 4.46904 * 10^{-6} n^4$$
$$+ .000277831 n^3 - 0.00455965 n^2 - 0.000252154 n - 8.73875 * 10^{-6} \quad (5.15)$$

It is easy to observe that the coefficients of the exponential formula (5.15) are extremely small, and that the coefficients in the polynomial formula (5.14) seem more reasonable.

The quality of a curve fitting is determined by the sum of the squared errors. The polynomial formula (5.14) approximates the data with sum of squared error approximately 434912.16. The exponential formula (5.15) approximates the data with sum of squared error approximately 514846.02. We can say that the polynomial formula (5.14) is a better fit to the data than the exponential formula. Neglecting the terms with low-order coefficients, it appears that the experimental data correlates with a cubic polynomial of the number of variables. Of course, the number of data points is quite small, and the DIMACS test problem set includes problems with special structure, so this correlation cannot be said to be significant.

5.4.2 Random NSAT Problems

The observations obtained by running the CutSat program using randomly generated NSAT problems as input are given in Table (5.3). The table has one row for each problems in the problem set. The column labels are the same as for the previous table. For several of the problems having 100 variables, the CutSat program suffered 64-bit arithmetic overflow. Because of the overflow, the finite-precision version of the program did not complete those problems in less than one hour.

Table 5.3: Computational results obtained by running the CutSat program on unsatisfiable problems from the NSAT problem set.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
nsat-30-4.25-1	30	127	yes	no	2	4
nsat-30-4.25-2	30	127	yes	no	4	9
nsat-30-4.25-3	30	127	yes	no	4	8
nsat-30-4.25-4	30	127	yes	no	5	12
nsat-30-4.25-5	30	127	yes	no	7	12
nsat-30-4.25-6	30	127	yes	no	5	11
nsat-30-4.25-7	30	127	yes	no	1	2
nsat-30-4.25-8	30	127	yes	no	5	10
nsat-30-4.25-9	30	127	yes	no	4	12
nsat-30-4.25-10	30	127	yes	no	2	7
nsat-35-4.25-1	35	148	yes	no	9	8
nsat-35-4.25-2	35	148	yes	no	5	8
nsat-35-4.25-3	35	148	yes	no	10	16
nsat-35-4.25-4	35	148	yes	no	8	12
nsat-35-4.25-5	35	148	yes	no	13	16
nsat-35-4.25-6	35	148	yes	no	7	9
nsat-35-4.25-7	35	148	yes	no	8	7
nsat-35-4.25-8	35	148	yes	no	8	13
nsat-35-4.25-9	35	148	yes	no	11	10
nsat-35-4.25-10	35	148	yes	no	9	12
nsat-40-4.25-1	40	170	yes	no	3	4
nsat-40-4.25-2	40	170	yes	no	11	12
nsat-40-4.25-3	40	170	yes	no	13	15
nsat-40-4.25-4	40	170	yes	no	7	8
nsat-40-4.25-5	40	170	yes	no	12	16
nsat-40-4.25-6	40	170	yes	no	13	15
nsat-40-4.25-7	40	170	yes	no	7	7
nsat-40-4.25-8	40	170	yes	no	14	14
nsat-40-4.25-9	40	170	yes	no	8	8
nsat-40-4.25-10	40	170	yes	no	8	11
nsat-45-4.25-1	45	191	yes	no	21	19
nsat-45-4.25-2	45	191	yes	no	15	15
nsat-45-4.25-3	45	191	yes	no	13	14
nsat-45-4.25-4	45	191	yes	no	27	21
nsat-45-4.25-5	45	191	yes	no	23	18
nsat-45-4.25-6	45	191	yes	no	7	7

Table 5.3: Continued.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
nsat-45-4.25-7	45	191	yes	no	23	20
nsat-45-4.25-8	45	191	yes	no	16	17
nsat-45-4.25-9	45	191	yes	no	9	7
nsat-45-4.25-10	45	191	yes	no	11	10
nsat-50-4.25-1	50	212	yes	no	28	20
nsat-50-4.25-2	50	212	yes	no	7	6
nsat-50-4.25-3	50	212	yes	no	16	9
nsat-50-4.25-4	50	212	yes	no	11	8
nsat-50-4.25-5	50	212	yes	no	21	14
nsat-50-4.25-6	50	212	yes	no	46	29
nsat-50-4.25-7	50	212	yes	no	5	5
nsat-50-4.25-8	50	212	yes	no	21	17
nsat-50-4.25-9	50	212	yes	no	33	21
nsat-50-4.25-10	50	212	yes	no	9	6
nsat-55-4.25-1	55	233	yes	no	47	26
nsat-55-4.25-2	55	233	yes	no	51	26
nsat-55-4.25-3	55	233	yes	no	50	28
nsat-55-4.25-4	55	233	yes	no	43	22
nsat-55-4.25-5	55	233	yes	no	35	21
nsat-55-4.25-6	55	233	yes	no	43	20
nsat-55-4.25-7	55	233	yes	no	50	24
nsat-55-4.25-8	55	233	yes	no	35	24
nsat-55-4.25-9	55	233	yes	no	51	22
nsat-55-4.25-10	55	233	yes	no	29	16
nsat-60-4.25-1	60	255	yes	no	32	18
nsat-60-4.25-2	60	255	yes	no	60	24
nsat-60-4.25-3	60	255	yes	no	38	17
nsat-60-4.25-4	60	255	yes	no	26	14
nsat-60-4.25-5	60	255	yes	no	54	25
nsat-60-4.25-6	60	255	yes	no	49	25
nsat-60-4.25-7	60	255	yes	no	25	13
nsat-60-4.25-8	60	255	yes	no	25	13
nsat-60-4.25-9	60	255	yes	no	57	25
nsat-60-4.25-10	60	255	yes	no	18	14
nsat-65-4.25-1	65	255	yes	no	81	23
nsat-65-4.25-2	65	255	yes	no	59	24
nsat-65-4.25-3	65	255	yes	no	50	20
nsat-65-4.25-4	65	255	yes	no	52	23

Table 5.3: Continued.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
nsat-65-4.25-5	65	255	yes	no	42	16
nsat-65-4.25-6	65	255	yes	no	56	18
nsat-65-4.25-7	65	255	yes	no	64	25
nsat-65-4.25-8	65	255	yes	no	78	27
nsat-65-4.25-9	65	255	yes	no	25	13
nsat-65-4.25-10	65	255	yes	no	57	21
nsat-70-4.25-1	70	297	yes	no	108	32
nsat-70-4.25-2	70	297	yes	no	112	32
nsat-70-4.25-3	70	297	yes	no	60	17
nsat-70-4.25-4	70	297	yes	no	62	22
nsat-70-4.25-5	70	297	yes	no	203	48
nsat-70-4.25-6	70	297	yes	no	59	22
nsat-70-4.25-7	70	297	yes	no	93	27
nsat-70-4.25-8	70	297	yes	no	51	16
nsat-70-4.25-9	70	297	yes	no	88	29
nsat-70-4.25-10	70	297	yes	no	78	20
nsat-75-4.25-1	75	318	yes	no	92	23
nsat-75-4.25-2	75	318	yes	no	70	16
nsat-75-4.25-3	75	318	yes	no	107	28
nsat-75-4.25-4	75	318	yes	no	96	23
nsat-75-4.25-5	75	318	yes	no	198	45
nsat-75-4.25-6	75	318	yes	no	67	20
nsat-75-4.25-7	75	318	yes	no	37	9
nsat-75-4.25-8	75	318	yes	no	134	31
nsat-75-4.25-9	75	318	yes	no	110	27
nsat-75-4.25-10	75	318	yes	no	149	33
nsat-80-4.25-1	80	340	yes	no	137	28
nsat-80-4.25-2	80	340	yes	no	184	35
nsat-80-4.25-3	80	340	yes	no	161	30
nsat-80-4.25-4	80	340	yes	no	301	48
nsat-80-4.25-5	80	340	yes	no	111	25
nsat-80-4.25-6	80	340	yes	no	120	25
nsat-80-4.25-7	80	340	yes	no	147	31
nsat-80-4.25-8	80	340	yes	no	109	25
nsat-80-4.25-9	80	340	yes	no	147	28
nsat-80-4.25-10	80	340	yes	no	167	30
nsat-85-4.25-1	85	361	yes	no	130	22
nsat-85-4.25-2	85	361	yes	no	266	35

Table 5.3: Continued.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
nsat-85-4.25-3	85	361	yes	no	81	16
nsat-85-4.25-4	85	361	yes	no	111	25
nsat-85-4.25-5	85	361	yes	no	440	45
nsat-85-4.25-6	85	361	yes	no	227	35
nsat-85-4.25-7	85	361	yes	no	190	26
nsat-85-4.25-8	85	361	yes	no	178	30
nsat-85-4.25-9	85	361	yes	no	239	33
nsat-85-4.25-10	85	361	yes	no	189	30
nsat-90-4.25-1	90	382	yes	no	212	26
nsat-90-4.25-2	90	382	yes	no	271	26
nsat-90-4.25-3	90	382	yes	no	251	20
nsat-90-4.25-4	90	382	yes	no	133	20
nsat-90-4.25-5	90	382	yes	no	141	20
nsat-90-4.25-6	90	382	yes	no	464	47
nsat-90-4.25-7	90	382	yes	no	59	11
nsat-90-4.25-8	90	382	yes	no	55	13
nsat-90-4.25-9	90	382	yes	no	228	24
nsat-90-4.25-10	90	382	yes	no	342	42
nsat-95-4.25-1	95	403	yes	no	3394	40
nsat-95-4.25-2	95	403	yes	no	446	37
nsat-95-4.25-3	95	403	yes	no	300	31
nsat-95-4.25-4	95	403	yes	no	211	23
nsat-95-4.25-5	95	403	yes	no	165	23
nsat-95-4.25-6	95	403	yes	no	409	29
nsat-95-4.25-7	95	403	yes	no	262	23
nsat-95-4.25-8	95	403	yes	no	323	38
nsat-95-4.25-9	95	403	yes	no	429	42
nsat-95-4.25-10	95	403	yes	no	487	40
nsat-100-4.25-1	100	425	yes	no	472	36
nsat-100-4.25-2	100	425	yes	no	509	43
nsat-100-4.25-3	100	425	yes	no	339	39
nsat-100-4.25-4	100	425	yes	no	656	43
nsat-100-4.25-5	100	425	yes	no	466	25
nsat-100-4.25-6	100	425	yes	no	718	30
nsat-100-4.25-7	100	425	no	-	-	-
nsat-100-4.25-8	100	425	no	-	-	-
nsat-100-4.25-9	100	425	no	-	-	-
nsat-100-4.25-10	100	425	yes	no	1476	30

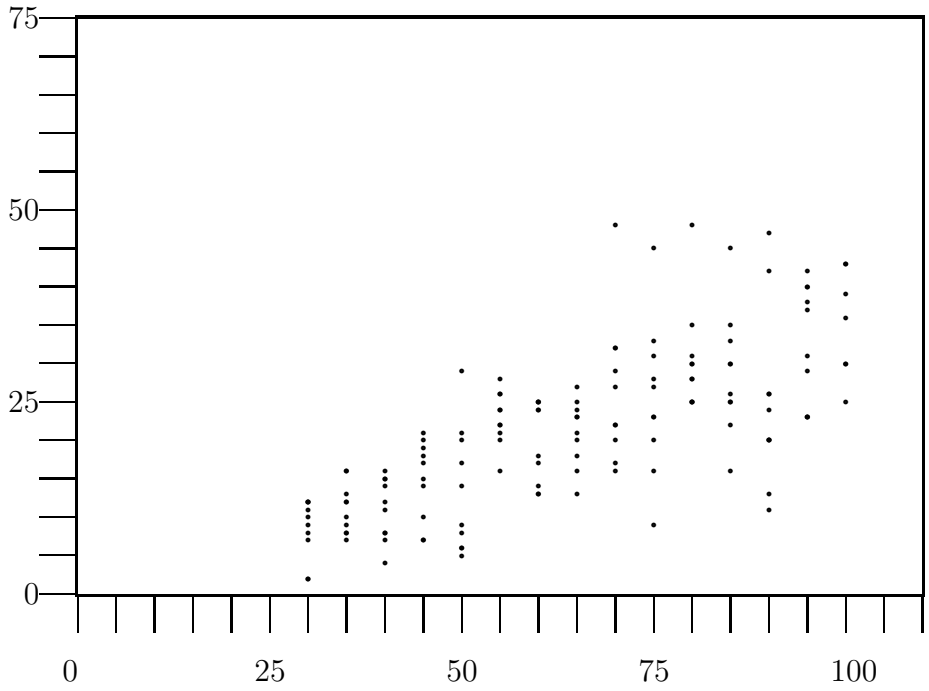


Figure 5.13: Plot of Number of Variables vs. Number of Cutting Planes for unsatisfiable problems generated by the NSAT generator. The horizontal scale plots the number of variable in the problem. The vertical scale plots the number of lifted cutting planes used by CutSat program to solve the problem.

A graph of the relation between the number of variables and the number of lifted cutting planes is shown in Figure 5.13.

Using Mathematica, two regression calculations were carried out (Wolfram, 1999). The curves were fitted to determine the number of lifted cutting planes as a function of the number n of variables. The relation between the number of variables and the number of lifted cutting planes correlates with both a low-order polynomial and with an exponential function. Formula (5.16) gives the polynomial. Formula (5.17) gives the exponential function with the exponent being a low-order polynomial.

$$2.79101 * 10^{-9}n^6 - 8.24924 * 10^{-7}n^5 + 0.0000926481n^4 - 0.00494967m^3 + 0.127805n^2 - 0.990449n - 0.107548 \quad (5.16)$$

$$e^{f(m)}, \text{ where } f(m) = 1.28558 * 10^{-10}n^6 - 3.79313 * 10^{-8}n^5 + 4.22018 * 10^{-6}n^4 - 0.000215812n^3 + 0.00436818n^2 + 0.0443921n + 0.00477311 \quad (5.17)$$

It is easy to observe that the coefficients of the exponential formula (5.17) are extremely small, and that the coefficients in the polynomial formula (5.16) seem more reasonable.

The quality of a curve fitting is determined by the sum of the squared errors. The polynomial formula (5.16) approximates the data with sum of squared error approximately 7348.73. The exponential formula (5.17) approximates the data with sum of squared error approximately 82770.59. We can say that the polynomial formula (5.16) is a better fit to the data than the exponential formula. Neglecting the terms with low-order coefficients, it appears that the experimental data correlates with a quadratic polynomial of the number of variables.

5.4.3 Random MSAT Problems

The observations obtained by running the CutSat program using as input a set of problems problems generated by the MSAT pseudorandom SAT problem generator are given in Table (5.4). The table has one row for each problems in the problem set. The column labels are the same as for the previous table. For problems with 85

variables, the first 7 of the 10 problems each required more than one hour of cpu time, and so the experiment was terminated. Problems having more than 85 variables were not attempted.

Table 5.4: Computational results obtained by running the CutSat program on randomly generated MSAT problems.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
msat-30-6.0-1	30	180	yes	no	9	10
msat-30-6.0-2	30	180	yes	no	3	4
msat-30-6.0-3	30	180	yes	no	6	11
msat-30-6.0-4	30	180	yes	no	9	11
msat-30-6.0-5	30	180	yes	no	9	12
msat-30-6.0-6	30	180	yes	no	7	12
msat-30-6.0-7	30	180	yes	no	7	11
msat-30-6.0-8	30	180	yes	no	12	17
msat-30-6.0-9	30	180	yes	no	9	12
msat-30-6.0-10	30	180	yes	no	9	13
msat-35-6.0-1	35	210	yes	no	24	18
msat-35-6.0-2	35	210	yes	no	20	17
msat-35-6.0-3	35	210	yes	no	20	15
msat-35-6.0-4	35	210	yes	no	5	5
msat-35-6.0-5	35	210	yes	no	32	24
msat-35-6.0-6	35	210	yes	no	7	8
msat-35-6.0-7	35	210	yes	no	4	5
msat-35-6.0-8	35	210	yes	no	12	9
msat-35-6.0-9	35	210	yes	no	12	12
msat-35-6.0-10	35	210	yes	no	14	11
msat-40-6.0-1	40	240	yes	no	17	12
msat-40-6.0-2	40	240	yes	no	41	23
msat-40-6.0-3	40	240	yes	no	18	16
msat-40-6.0-4	40	240	yes	no	24	15
msat-40-6.0-5	40	240	yes	no	22	13
msat-40-6.0-6	40	240	yes	no	18	14
msat-40-6.0-7	40	240	yes	no	34	21
msat-40-6.0-8	40	240	yes	no	25	13
msat-40-6.0-9	40	240	yes	no	26	17
msat-40-6.0-10	40	240	yes	no	24	10
msat-45-6.0-1	45	270	yes	no	43	16

Table 5.4: Continued.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
msat-45-6.0-2	45	270	yes	no	72	29
msat-45-6.0-3	45	270	yes	no	49	26
msat-45-6.0-4	45	270	yes	no	101	44
msat-45-6.0-5	45	270	yes	no	41	19
msat-45-6.0-6	45	270	yes	no	54	23
msat-45-6.0-7	45	270	yes	no	48	19
msat-45-6.0-8	45	270	yes	no	23	13
msat-45-6.0-9	45	270	yes	no	44	19
msat-45-6.0-10	45	270	yes	no	43	16
msat-50-6.0-1	50	300	yes	no	81	24
msat-50-6.0-2	50	300	yes	no	100	35
msat-50-6.0-3	50	300	yes	no	98	23
msat-50-6.0-4	50	300	yes	no	42	15
msat-50-6.0-5	50	300	yes	no	62	24
msat-50-6.0-6	50	300	yes	no	89	26
msat-50-6.0-7	50	300	yes	no	56	24
msat-50-6.0-8	50	300	yes	no	87	29
msat-50-6.0-9	50	300	yes	no	46	18
msat-50-6.0-10	50	300	yes	no	32	14
msat-55-6.0-1	55	330	yes	no	196	39
msat-55-6.0-2	55	330	yes	no	135	29
msat-55-6.0-3	55	330	yes	no	122	27
msat-55-6.0-4	55	330	yes	no	152	37
msat-55-6.0-5	55	330	yes	no	199	42
msat-55-6.0-6	55	330	yes	no	294	55
msat-55-6.0-7	55	330	yes	no	105	27
msat-55-6.0-8	55	330	yes	no	185	42
msat-55-6.0-9	55	330	yes	no	175	36
msat-55-6.0-10	55	330	yes	no	123	33
msat-60-6.0-1	60	360	yes	no	202	34
msat-60-6.0-2	60	360	yes	no	269	46
msat-60-6.0-3	60	360	yes	no	115	22
msat-60-6.0-4	60	360	yes	no	52	14
msat-60-6.0-5	60	360	yes	no	231	36
msat-60-6.0-6	60	360	yes	no	207	32
msat-60-6.0-7	60	360	yes	no	49	11
msat-60-6.0-8	60	360	yes	no	288	44
msat-60-6.0-9	60	360	yes	no	174	27

Table 5.4: Continued.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
msat-60-6.0-10	60	360	yes	no	162	29
msat-65-6.0-1	65	390	yes	no	288	38
msat-65-6.0-2	65	390	yes	no	328	40
msat-65-6.0-3	65	390	yes	no	645	68
msat-65-6.0-4	65	390	yes	no	552	64
msat-65-6.0-5	65	390	yes	no	653	73
msat-65-6.0-6	65	390	yes	no	383	47
msat-65-6.0-7	65	390	yes	no	332	41
msat-65-6.0-8	65	390	yes	no	699	77
msat-65-6.0-9	65	390	yes	no	474	59
msat-65-6.0-10	65	390	yes	no	400	54
msat-70-6.0-1	70	420	yes	no	944	80
msat-70-6.0-2	70	420	yes	no	238	25
msat-70-6.0-3	70	420	yes	no	754	77
msat-70-6.0-4	70	420	yes	no	698	77
msat-70-6.0-5	70	420	yes	no	430	41
msat-70-6.0-6	70	420	yes	no	964	86
msat-70-6.0-7	70	420	yes	no	557	56
msat-70-6.0-8	70	420	yes	no	416	43
msat-70-6.0-9	70	420	yes	no	331	35
msat-70-6.0-10	70	420	yes	no	874	79
msat-75-6.0-1	75	450	yes	no	2170	122
msat-75-6.0-2	75	450	yes	no	1057	68
msat-75-6.0-3	75	450	yes	no	1347	90
msat-75-6.0-4	75	450	yes	no	2661	144
msat-75-6.0-5	75	450	yes	no	1490	101
msat-75-6.0-6	75	450	yes	no	1182	78
msat-75-6.0-7	75	450	yes	no	792	51
msat-75-6.0-8	75	450	yes	no	1293	81
msat-75-6.0-9	75	450	yes	no	1300	79
msat-75-6.0-10	75	450	yes	no	1919	107
msat-80-6.0-1	80	480	yes	no	1536	80
msat-80-6.0-2	80	480	yes	no	1191	66
msat-80-6.0-3	80	480	yes	no	1318	75
msat-80-6.0-4	80	480	yes	no	1281	67
msat-80-6.0-5	80	480	yes	no	2390	123
msat-80-6.0-6	80	480	yes	no	1827	92
msat-80-6.0-7	80	480	yes	no	1259	84

Table 5.4: Continued.

Problem	Vars	Clauses	Solved	Satisfiable	Seconds	Cutting Planes
msat-80-6.0-8	80	480	yes	no	1934	105
msat-80-6.0-9	80	480	yes	no	939	62
msat-80-6.0-10	80	480	yes	no	3458	153
msat-85-6.0-1	85	520	no	-	-	-
msat-85-6.0-2	85	520	no	-	-	-
msat-85-6.0-3	85	520	no	-	-	-
msat-85-6.0-4	85	520	no	-	-	-
msat-85-6.0-5	85	520	no	-	-	-
msat-85-6.0-6	85	520	no	-	-	-
msat-85-6.0-10	85	520	no	-	-	-

A graph of the relation between the number of variables and the number of lifted cutting planes is shown in Figure 5.14.

Using Mathematica, two regression calculations were carried out (Wolfram, 1999). The curves were fitted to determine the number of lifted cutting planes as a function of the number n of variables. The relation between the number of variables and the number of lifted cutting planes correlates with both a low-order polynomial function and with an exponential function. Formula (5.18) gives the polynomial. Formula (5.19) gives the exponential function with the exponent being a low-order polynomial.

$$-1.14124 * 10^{-7} n^6 + 0.0000307964 n^5 - 0.00323486 n^4 + 0.165352 n^3 - 4.09936 n^2 + 39.5697 n + 4.83655 \quad (5.18)$$

$$e^{f(m)}, \text{ where } f(m) = -2.7707 * 10^{-9} n^6 + 7.691665 * 10^{-7} n^5 - 0.0000834537 n^4 + 0.00442236 n^3 - 0.114759 n^2 + 1.23383 n + 0.150731 \quad (5.19)$$

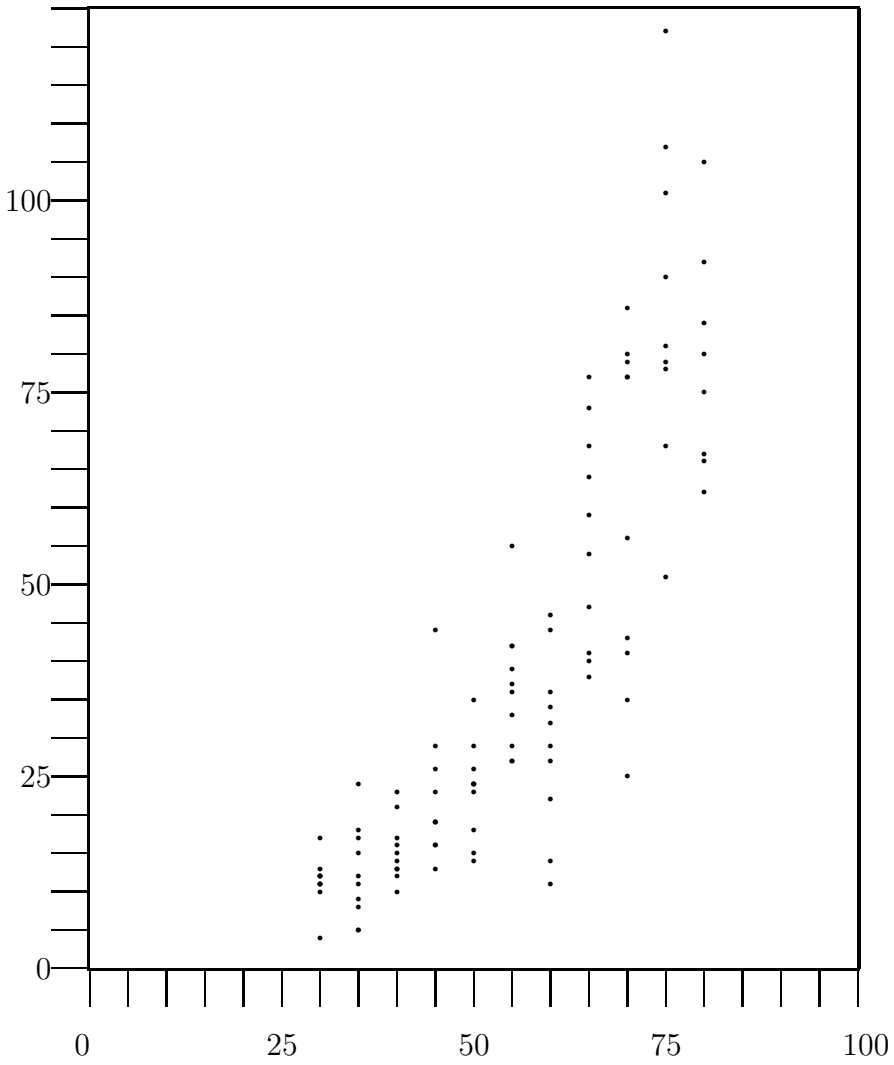


Figure 5.14: Plot of Number of Variables vs. Number of Cutting Planes for randomly generated MSAT problems. The horizontal scale plots the number of variable in the problem. The vertical scale plots the number of lifted cutting planes used by CutSat program to solve the problem.

The quality of a curve fitting is determined by the sum of the squared errors. The polynomial formula (5.18) approximates the data with sum of squared error approximately 34981.4. The exponential formula (5.19) approximates the data with sum of squared error approximately 29279.5. In this case the exponential function is a better fit than the polynomial function.

The MSAT test problem set includes problems having no special structure that could be used to advantage by an algorithm. The number of lifted cutting planes generated by the CutSat program for problems in this problem set correlates with a first-degree exponential function of the number of variables. This experimental result supports an inference that the CutSat program solves these problems using a super-polynomial number of steps.

5.5 New Proofs of Pigeonhole Unsatisfiability

The CutSat program discovers and uses cliques in a way that is very different from previous programs, such as that of Barth (1996). Previous programs explicitly search for diagonal sum cuts representing cliques in the graph of the problem. A diagonal sum cut is formed from a set of k clauses in k literals where each clause has $k-1$ of the literals.

The CutSat program discovers and uses the same cliques, but finds and represents them in a very different manner. A cutting plane equivalent to a diagonal sum cut is represented as a lower bound on the feasible integer values of the slack variables, rather than on the original variables. An example from the `hole6` problem, one of the pigeonhole problems in the DIMACS problem set, illustrates the algorithm.

```
s65 -1x2 -1x14 = -1
s66 -1x2 -1x20 = -1
s75 -1x14 -1x20 = -1
```

Figure 5.15: Some inequalities of the `hole6` problem.

```
THE NONBASIC VARIABLES:
 x1 s169 x3 x4 x5 s162 x7 x8 x9 x10 s170 x12
 x13 s171 s101 x16 x17 s65 x19 s172 s75 s98 x23 x24
 s135 x26 x27 x28 s173 x30 x31 x32 s174 x34 x35 x36
 x37 s66 x39 x40 x41 s175
THE CANDIDATE CUT:
-1s65 -1s75 -1s66 <= -1
THE LIFTED CUT:
-1s65 -1s75 -1s66 <= -1
```

Figure 5.16: The first proof step of the `hole6` problem.

The `hole6` problem contains the following clauses:

$$(x_2 x_{14})(x_2 x_{20})(x_{14} x_{20}) \tag{5.20}$$

Among the equalities constructed in the initial tableau, we find equalities (5.21) representing the clauses (5.20):

$$\begin{aligned} s_{65} - x_2 - x_{14} &= -1 \\ s_{66} - x_2 - x_{20} &= -1 \\ s_{75} - x_{14} - x_{20} &= -1 \end{aligned} \tag{5.21}$$

When printed in the format used by CutSat, these inequalities are as shown in figure 5.15. In the first step of the proof constructed by CutSat during one of the runs of the `hole6` problem, we find the output shown in figure 5.16.

These lines of the proof indicate that the CutSat program constructed the equality shown in figure 5.17. This construction was verified by enabling the `SHOW_EVERY_TABLEAU` feature of the program, and could also be verified by reconstructing the tableau with the given non-basic variables. The tableau containing the

```
2s2 +1s76 -1s67 -1s65 = 1
```

Figure 5.17: One line of the first tableau constructed for the `hole6` problem.

```
-1s65 -1s75 -1s66 <= -1
```

Figure 5.18: A Gomory cutting plane constructed for the `hole6` problem.

equation in figure 5.17 has common denominator 2, so the cutting plane can be derived as in figure 5.18 Indeed, we have seen in figure 5.16 that the cutting plane of figure 5.18 is derived by the CutSat program.

Now, when the cutting plane in figure 5.18 is added to the equations (5.21), we find that the inequality given in figure 5.19 is a valid linear combination of rows of the tableau.

But the inequality of figure 5.19 is just the diagonal sum cut that could be derived from the clauses (5.20) or from inequalities representing those clauses.

Hence, the diagonal sum cut is a linear combination of the original tableau and the cutting plane. Rather than deriving the diagonal sum cut directly, the CutSat program finds a lower bound on the slack variables which implies the diagonal sum cut arithmetically. Every solution which does not satisfy the diagonal sum cut also does not satisfy the tableau with the cutting plane of figure 5.18. Hence, after the cutting plane is added, the diagonal sum cut is implicitly present in the tableau.

By finding cutting planes of this type, the CutSat program quickly finds a sufficient number of cutting planes and completes the proof of unsatisfiability for the pigeonhole problem. The cutting planes used are equivalent to diagonal sum cuts, and hence the

```
-2x2 -2x14 -2x20 <= -4
```

Figure 5.19: A Gomory cutting plane constructed for the `hole6` problem.

resulting proof is equivalent to previous proofs using diagonal sum cuts. The difference is that the CutSat program does not explicitly look for these cuts, and finds them using a general mechanism that is also useful for finding other kinds of cutting planes.

5.6 Summary

In this chapter, we have described a computer program, named CutSat, that implements the algorithms described in chapter 3. We have also briefly described the implementation of the pseudo-random problem generators. The results of running the CutSat program using as input the unsatisfiable DIMACS problems and two sets of randomly generated problems are presented. For each problem set, both polynomial and exponential form expressions were fitted to the data to minimize the sum-square error. Finally, we show that the CutSat program finds cutting planes that are equivalent to diagonal sum cutting planes without specifically looking for them. The representation of the diagonal sum cuts is very different because CutSat uses the slack variables, but the effect is exactly equivalent.

Chapter 6

Conclusions and Recommendations

In chapter three, we provided some new algorithms to find strong canonical and lifted Gomory cutting planes. The new algorithms are based on cutting plane methods for solving systems of integer linear inequalities, and use heuristics and random search to find good cutting planes. Because the algorithms themselves are difficult, the theory to describe the complexity of the algorithms is elusive. In chapter four, we describe an experiment which was conducted to obtain some information about the complexity of the cutting plane algorithm. In chapter five, we describe the implementation of the algorithm as a computer program, and the results of the computational experiment using that implemented program.

In this chapter, we review those results to form conclusions.

6.1 The Cutting Plane Algorithm

The successful implementation of the CutSat program, which uses integer-only simplex tableau and lifted cutting planes, demonstrates that it is possible to successfully implement such complex algorithms. We believe that the integer-only simplex

tableau implementation is the first published implementation of an integer-only simplex method. The implementation of the lifting algorithms is much less difficult that the theory would seem to indicate, partially because additional theory was developed to allow the simplified implementation.

6.2 Discussion of Computational Results

For two of the three classes of test problems, the CutSat program found proofs of unsatisfiability with proof size polynomial in the number of variables.

For the DIMACS problems, the problems are known to have specific structures, and the CutSat program found short cutting plane proofs that take advantage of the problem structures. The fact that all but two of the unsatisfiable DIMACS problems were proved using very short proofs is encouraging, since the DIMACS problems were presented as challenge problems and are widely believed to be very hard problems indeed.

For the NSAT problems, the problems were not known to have any special structure, but the CutSat problem found many small cutting planes consisting of only two literals. The abundance of such small cutting planes suggests that there is an underlying structure. The algorithm used in the nsat generator increases the number of clauses by "splitting" a previous clause using the rule $c \rightarrow (x \wedge c) \vee (\neg x \wedge c)$ (Asahiro et al., 1996). This derivation process may provide the problem structure which was found by the CutSat program. Previous SAT algorithms have not found or made use of any underlying structure for NSAT problems.

For the MSAT problems, the CutSat program did not do as well. For these problems, the CutSat program was able to find many cutting planes having only small numbers of literals. However, the small cutting planes were found less often in

larger problems. This may be due to a genuine absence of small cutting planes in large random problems, or it may be due to the inability of the particular randomized algorithm to find them efficiently.

In the proofs that were produced for the problems in this experiment, the great majority of the proof steps found a cutting plane having a number of literals bounded by the square root of the number of variables in the problem. This result indicates that small cutting planes may be very common.

6.3 Conclusions

Based on the results of the computational experiment and the criteria set in Chapter 4, we may deem the experiment a success. There is a correlation between the number of cutting planes and some low-degree polynomial in the size of the problems for two of the three sets of test problems. This meets the criteria set in section 4.3.5. Hence, we may conclude that our thesis is supported:

> *Short cutting-plane proofs of unsatisfiability do exist for many hard SAT problems. Algorithms and heuristics for finding strong cutting planes can be useful to find such short refutation proofs.*

The experiment demonstrates the effective use of algorithms and heuristics identified in chapter three to solve a number of hard unsatisfiable SAT problems. The number of lifted cutting plane steps in the resulting refutation proofs was determined to be quite small. In two of the three sets of test problems, there is a polynomial correlation between the number of cutting planes and the size of the SAT problem.

Randomized local search for good cutting planes does succeed surprisingly often. That is, for the great majority of the proof steps, the CutSat program successfully

found a small cutting plane. The successful finding of small cutting planes accounts for the successful finding of short proofs. This observation confirms the theory developed in section 3.7.

6.4 Implications

This research has demonstrated that short cutting plane proofs do exist for many hard SAT problems, and that those proofs may often be constructed using algorithms and heuristics for finding strong cutting planes. It is well known that cutting plane proof systems are stronger than resolution or model-search proof systems, in the sense that shorter proofs are possible for at least some problems. We have provided an approach to actually constructing such short proofs for a variety of problems.

Specific algorithms and heuristics used in this study have been shown to be useful. Search for strong cutting planes which may occur at non-optimal vertices of the polytope is useful, in the sense that the extended search often finds stronger cutting planes than are available at the optimal vertices. Modification of the objective function to direct the cutting plane search is useful in the sense that such modification helps to find strong cutting planes. Integer lifting of Gomory cutting planes is also useful because it allows construction of stronger cutting planes from those that are found by the search algorithms.

Many of the cutting plane proofs constructed by this algorithm are shorter than proofs constructed by other algorithms for the same problems. For the DIMACS problems set, direct comparisons are possible. The relative success in finding short plane proofs indicates that cutting plane methods may be more useful than is commonly believed. The relative success in finding small, strong cutting planes suggests that such cutting planes may be common, and relatively easy to find.

6.5 Recommendations

Implementation of improved cut-lifting algorithms, perhaps based on the recent work of Balas et al. (1996), should allow shorter proofs than those found by the CutSat program. The work of Balas *et. al.* is oriented to solving integer programming problems, rather than SAT problems, but it seems that their cutting plane derivations could be used to find cutting planes for SAT problems.

The analysis of search algorithms for model-search and in other areas such as artificial intelligence is quite advanced, but very little work has been on search algorithms that search for strong cutting planes. We are not aware of any other work that considers altering the objective function to assist in the search for small cutting planes. Also, we are not aware of any other work that considers pivoting to adjacent vertices to find strong cutting planes. The theoretical performance of such cutting-plane search algorithms is an open question.

The design of optimal search strategies for finding strong cutting planes is also open. Strategies similar to those previously used for model search, such as tabu search and multi-start search, may be helpful. A great variety of search strategies is possible beyond the several considered in this thesis. The ideal solution to this problem would give an algorithm to solve for the strongest possible cutting plane without having to search.

A key factor that affects the success of any algorithm for finding small cutting planes is the number of small cutting planes that exist. An important question is to determine how common are small cutting planes? That is, do all or most hard SAT problems possess small, strong cutting planes? If so, are there enough such strong cutting planes so that an efficient search algorithm may have a high probability of finding one? The frequency of occurrence of strong cutting planes in

random problems, or in any certain class of practical problems, is an open question.

6.6 Summary

We gave in chapter three a characterization of the convex integer hull for SAT, and several algorithms to find strong cutting planes. The new cutting plane algorithm is a complete algorithm that is capable of proving unsatisfiability. In chapter four and five, we describe an experiment to estimate the proof complexity of the new cutting plane algorithms. We found that cutting plane algorithms do find short proofs of unsatisfiability for a large class of SAT problems, and that strong cutting planes are very common.

The algorithms presented in this thesis are different from almost all previously presented algorithms for SAT. The common types of SAT algorithms are based on either tree-search or resolution. Both of those types cannot be efficient, because the proof size doubles with each added variable. Cutting plane algorithms have the property that no branching is needed. Because no branching is needed, it is not known that small cutting plane proofs are not possible. This thesis suggests that small cutting plane proofs are indeed possible for many SAT problems.

However, the questions suggested by this result are far more numerous than the questions settled. The theoretical analysis of why these cutting plane algorithms are effective for many problems is open. The development of improved cutting plane search algorithms is also open. The study of specialized search algorithms for finding strong cutting-planes for SAT problems is barely begun. Because of the importance of propositional logic to computer science and other fields, we may expect cutting plane algorithms to provide fertile ground for research for many years to come.

Appendix A

CutSat Program Code

The source code of the CutSat program includes a number of files. The files that implement specific algorithms and which are not available elsewhere are reproduced here.

A.1 Parameters

Various algorithms are controlled by parameters which are defined at compile time. The Parameters.h file contains those definitions.

A.1.1 Parameters.h

```
//
// Parameter file for CUTSAT implementation.
//
// These parameters control various features of the
// implementation, and may be varied for particular
// classes of problems.
//
// Copyright (c) 2000 Stephen Lee Hansen
//
#ifndef PARAMETERS_H_INCLUDED
#define PARAMETERS_H_INCLUDED
```

```
//////////////////////////////////////////////////////
//
// PARAMETERS CONTROLLING NUMBER REPRESENTATION
//
//////////////////////////////////////////////////////
//
// Two modes of integer arithmetic are available:
//
// If USE_EXACT_INTEGER_ARITHMETIC is defined, the
// implementation uses arbitrary-precision integers.
// This is based on (Victor Shoup, "NTL  -- a library
// for doing number theory --  version 3.7a")
// Exact arithmetic is very slow, but never
// overflow unless heap memory is exhausted.

//#define USE_EXACT_INTEGER_ARITHMETIC

// If USE_EXACT_INTEGER_ARITHMETIC is not defined,
// the implementation uses "long long" integers, which
// are 64 bits on most machines.  64 bit integers are
// long enough to handle most SAT problems of practical
// size, but can overflow.  Overflow may be detected,
// at substantial computational cost, by defining
// the symbol CHECK_64_INTEGER_OVERFLOW.

//#define CHECK_64_INTEGER_OVERFLOW


//////////////////////////////////////////////////////
//
// PARAMETERS CONTROLLING THE DISPLAY OF THE PROOF
//
//////////////////////////////////////////////////////
//
// A full proof of unsatisfiability includes the initial
// simplex tableau. That tableau is bulky, and is easy to
// construct from the SAT problem, and so it not very
// interesting.  If you want to see the initial simplex
// tableau, define SHOW_INITIAL_TABLEAU.

//#define SHOW_INITIAL_TABLEAU

// A full proof includes the derivations of each cut
// used in the proof.  These derivations are bulky, and
// are usually not interesting.  However, either those
// derivations (or equivalent information) are needed to
// check a proof.  If you want to see the detailed cut
// derivations, define SHOW_CUT_DERIVATIONS.

//#define SHOW_CUT_DERIVATIONS
```

```
// Alternatively, it is possible to verify that each
// asserted cut is in fact a cut by reproducing the simplex
// tableau representing the basic solution from which the cut
// is derived, and then deriving the cuts from that tableau.
// Hence, we may allow the proof to be checked by enabling the
// proof-checker to reproduce the basic solutions.
// The most consise way to do this is to identify the vector
// of non-basic variables.

#define SHOW_NONBASIC_VARIABLES

// To check that the asserted cut derivations are correct,
// it may be desired to inspect the simplex tableau from which
// each cut is derived. The tableaus are very large, and there
// are as many of them as there are cuts, so this option
// outputs an enormous number of lines.  If you want to see
// all of the simplex tableaus, define SHOW_EVERY_TABLEAU.

//#define SHOW_EVERY_TABLEAU

//////////////////////////////////////////////////////
//
//
//
//////////////////////////////////////////////////////
//
// CuttingAlgorithm::operator() performs a search for
// cut-improving pivots, starting from the current
// optimal solution, at each iteration.
//
// The primalCutImproving() method simulates a Markov
// process that uses pivot steps that give stronger cuts.
// The method searches for a basic solution that gives
// a local-maximum of cut measure, as measured by the
// CutMeasure class.
//
// Multiple calls to primalCutImproving() from the
// same starting tableau may find different local optimum
// basic solutions, because the randomized choices in
// primalCutImproving may take different paths.
//
// If CUT_IMPROVEMENT_RESTARTS is defined, the
// primalCutImproving() algorithm is restarted from the
// optimal tableau a number of times controlled by the value.
//

//#define  CUT_IMPROVEMENT_RESTARTS 3


//////////////////////////////////////////////////////
//
```

```
// The randomized algorithm to search for cut-improving
// pivots takes a sample of the cuts from each resulting
// feasible tableau.  CUT_SAMPLE_SIZE controls the sample
// size.  CUT_SAMPLE_SIZE cutting planes are measured
// for each adjacent feasible vertex of the polytope.
//
// Values may be defined in terms of the variables
// 'rows', 'cols', 'tabMeasure'.
// For full (derandomized) search, use 'rows'.
//

// for rsat, msat problems:
#define  CUT_SAMPLE_SIZE  25

// for nsat problems:
//#define  CUT_SAMPLE_SIZE (1500/cols)

// for dimacs problems:
//#define  CUT_SAMPLE_SIZE  15


////////////////////////////////////////////////////////
//
// CUT_RESAMPLE_COUNT controls re-sampling.
// To protect against too-small values of CUT_SAMPLE_SIZE,
// the cut-improving algorithm may be configured to draw
// as many as 1+CUT_SAMPLE_COUNT independent samples.
//
// This doesn't seem to help if the sample size is
// at all adequate.
//

//#define  CUT_RESAMPLE_COUNT 0


////////////////////////////////////////////////////////
//
// The objective function maximizes the slacks of prior
// cuts.  Using only cuts of optimal vertices has the
// effect of choosing cuts having minimal overlap with
// prior cuts.  It may be helpful to limit the search
// to only those vertices.  To use only objective-maximal
// vertices, define USE_ONLY_MAXIMAL_PIVOTS
//

//#define USE_ONLY_MAXIMAL_PIVOTS


////////////////////////////////////////////////////////
//
// CuttingAlgorithm::ApplyCuts adds cuts to the tableau.
```

```
// Large cuts don't eliminate much volume from the
// remaining feasible polytope.  If DONT_USE_LARGE_CUTS
// is defined, and the best available cut is large/weak,
// ApplyCuts does not add the large cut to the tableau.
//
// The decision is  randomized.  There is nonzero
// probability that each cut will be used.  That assures
// eventual termination even when all available cuts
// are large.
//
// In any case, the objective function  modified by the
// cut.  That allows the next LP solution to find a
// different optimal vertex, so the next cut-improving
// search can find a different cutting plane.
//

#define AVOID_USING_LARGE_CUTS


////////////////////////////////////////////////////////
//
// CutMeasure provides comparison operators (>, <, ==, >=, <=)
// that implement the induced partial order on cutting planes.
// The comparison operators use the value() of the CutMeasure,
// which is a floating-point number approximately representing
// the abstract  measure.
//
// An assortment of geometric measures, including the distance,
// strength (i.e. fraction of diagonol), and power (i.e. volume)
// of the cutting plane are provided.  The selection of
// which to use is done by editing the value() function.
//
// To choose the value() of a cut, define CUT_VALUE_FUNCTION.
// The definition may use the private methods and data members
// of CutMeasure.
//
// #define  CUT_VALUE_FUNCTION distance()
// #define  CUT_VALUE_FUNCTION diagonolDistance()
// #define  CUT_VALUE_FUNCTION strength()
// #define  CUT_VALUE_FUNCTION power()
//#define   CUT_VALUE_FUNCTION logPower()
//#define   CUT_VALUE_FUNCTION (distance() / (count * count * count))
//

#define     CUT_VALUE_FUNCTION (log2(strength()*count) - count)


////////////////////////////////////////////////////////
//
//  To limit the amount of time spent on any one problem
// during testing runs, define MAX_RUNNING_TIME in seconds.
```

```
//

#define MAX_RUNNING_TIME 3600


#endif // def PARAMETERS_H_INCLUDED
```

A.2 Pseudo-Random Generator

The pseudo-random generators provided in the CodeWarrior development system were found to be defective. A good quality pseudo-random generator function was used instead (Carter Jr., 1994; Kirkpatrick & Stoll, 1981). A little function is provided to initialize and call the R250 generator.

A.2.1 random.cc

```
#include "r250.hpp"
#include "random.h"



// A usable pseudo random generator
long random()
{
    static R250*  gener = 0;
    if (gener == 0)
    {
        // Allocate a new generator, seed it,
        // and warm it up.

        long t = time(0);
        gener = new R250(abs(t));

        for (int i = 1; i < 1747; i++) gener->rani();
    }
    return  (unsigned int) gener->rani();
}

// A usable pseudo-random bit generator
long randomBit()
{
    static long r = 0;
```

```
    static long b = 0;

    if (b == 0)
    {
        b = 1;
        r = random();
    }
    else
    {
        b = b << 1;
    }

    return ((r & b) == 0);
}
```

A.3 CNF Term Structure

The internal representation of a CNF Term is used only to read the problem from the DIMACS format input file.

A.3.1 CnfTerm.h

```
#ifndef CNFTERM_INCLUDED
#define CNFTERM_INCLUDED

#include <ctype.h>
#include <vector>
#include <iostream>

// This SAT solver is specialized to CNF propositions.
// General propositions would require a different set of
// classes to reflect the different term structure.
//
// Literals are coded per DIMACS format.
// The absolute value of a literal is the subscript of
// the variable.  The sign of the literal is
// negative iff the literal is negated.
// Variable number 0 is not used.
// operator() tests values[abs(variable)]

class Literal
{
public:
    Literal();
    Literal( int i);
```

```
    int        variable;

    bool    operator()(vector<bool>& values);
};

// A clause is a vector of Literals

class Clause : public vector<Literal>
{
public:
    Clause() : vector<Literal>() {}
    bool    operator()(vector<bool>& values);
};

// Read DIMACS format of a clause
istream& operator>>(istream& is, Clause& clause);

// A CNF term is a vector of Clauses

class CnfTerm : public vector<Clause>
{
  public:
    CnfTerm() : vector<Clause>()
        {variables = clauses = 0;}

    int     variables;    // number of variables
    int     clauses;      // number of clauses
    bool    operator()(vector<bool>& values);

    double fitness(vector<bool>& values);
};

// Read DIMACS format of a CNF problem.
istream& operator>>(istream& is, CnfTerm& cnf);

#endif // CNFTERM_INCLUDED
```

A.3.2 CnfTerm.cc

```
#include <iterator>
#include "CnfTerm.h"

Literal::Literal()
{
    variable = 0;
}

Literal::Literal(int i)
```

```
{
    variable = i;
}

bool
Literal::operator()(vector<bool>& values)
{
    if (variable > 0)
        return values[ variable ];
    else
        return ! values[ -variable ];
}

bool
Clause::operator()(vector<bool>& values)
{
    for (iterator i = begin(); i!=end(); ++i)
    {
        if ( (*i)(values) ) return true;
    }
    return false;
}

istream&
operator>>(istream& is, Clause& clause)
{
    int i;
    while ( 1 )
    {
        is >> i;
        if (!is.good())     break;
        else if (i == 0)    break;
        else    clause.push_back(Literal(i));
    }
    return is;
}

bool
CnfTerm::operator()(vector<bool>& values)
{
    for (iterator i = begin(); i!=end(); ++i)
    {
        if ( ! (*i)(values) ) return false;
    }
    return true;
}

istream&
operator>>(istream& is, CnfTerm& cnf)
{
    while (1)
```

```
    {
        char c;
        is >> c;
        if ( !is.good() )
        {
            break;
        }
        else if (c == 'p')
        {
            // "Problem" line
            string format;
            is >> format >> cnf.variables >> cnf.clauses;
            break;
        }
        else if ( c == 'c' )
        {
            // "Comment" line
            string line;
            getline(is, line);
            continue;
        }
        else
        {
            // Ignore debug/trace output of problem generator
            string line;
            getline(is, line);
            continue;
        }
    }
    for (int clauses = 1; clauses <= cnf.clauses; ++clauses)
    {
        Clause clause;
        is >> clause;
        cnf.push_back(clause);
    }
    return is;
}
```

A.4 Number Types

The CutSat program uses either arbitrary precision integers or fixed precision integers, according to the definition of type `Integer`.

A.4.1 Integer.h

```
#ifndef Integer_h
#define Integer_h

#include <stdlib.h>
#include <iostream.h>

#include "Parameters.h"

////////////////////////////////////////////////////
#ifdef USE_EXACT_INTEGER_ARITHMETIC
////////////////////////////////////////////////////

// Use Victor Shoup's integer arithmetic package

#define NTL_LONG_LONG long long
#include "NTL/ZZ.h"

#define Integer ZZ

// Define some Integer constants
const Integer MINUSONE = ZZ(INIT_VAL, -1);
const Integer ZERO = ZZ(INIT_VAL, 0);
const Integer ONE = ZZ(INIT_VAL, 1);
const Integer TWO = ZZ(INIT_VAL, 2);

////////////////////////////////////////////////////
#else // NOT #ifdef USE_EXACT_INTEGER_ARITHMETIC
////////////////////////////////////////////////////

#include <cmath>

//#define CHECK_64_INTEGER_OVERFLOW

#define Integer  long long
#define to_Integer (long long)

// Define some Integer constants
const Integer MINUSONE = -1;
const Integer ZERO = 0;
const Integer ONE = 1;
const Integer TWO = 2;

#define IsZero(x) (x == 0)
#define sign(x) (x)
Integer GCD(const Integer& a, const Integer& b);
inline double to_double(Integer x) {return x;}

////////////////////////////////////////////////////
#endif // #ifdef USE_EXACT_INTEGER_ARITHMETIC
```

```
//////////////////////////////////////////////////////

// In both cases,
#define NonZero(x) (! IsZero(x))

// In both cases, mod() is the same.
inline Integer mod(const Integer& x, const Integer& y)
{
    Integer r = (x % y);
    if (r >= 0) return r;
    else return (r+y);
}
inline int mod(const int& x, const int& y)
{
    int r = (x % y);
    if (r >= 0) return r;
    else return (r+y);
}

// In both cases, lg() is the same.
const Integer TWO16 = (ONE<<16);
const Integer TWO8 = (ONE<<8);
const Integer TWO4 = (ONE<<4);
const Integer TWO2 = (ONE<<2);
const Integer TWO1 = (ONE<<1);

inline size_t
lg(const Integer& x)
{
    Integer xa = abs(x);
    size_t lx = 0;
    while (xa >= TWO16) {lx += 16; xa = xa >> 16;}
    if  (xa >= TWO8) {lx += 8; xa = xa >> 8;}
    if  (xa >= TWO4) {lx += 4; xa = xa >> 4;}
    if  (xa >= TWO2) {lx += 2; xa = xa >> 2;}
    if  (xa >= TWO1) {lx += 1; xa = xa >> 1;}
    return lx;
}

// A routine to help check for overflow when
// we are using finite precision arithmetic.

inline Integer
multiply_and_check_overflow(const Integer& x, const Integer& y)
{
//  if (IsZero(x) || IsZero(y)) return ZERO;
#ifdef CHECK_64_INTEGER_OVERFLOW
    if (lg(x) + lg(y) > 62)
        cout << "POSSIBLE 64 BIT OVERFLOW" << endl;
#endif
    return (x * y);
```

```
}

#endif // #ifndef Integer_h
```

A.4.2 Integer.cc

```
#include "Integer.h"


#ifdef USE_EXACT_INTEGER_ARITHMETIC


#else // not #ifdef USE_EXACT_INTEGER_ARITHMETIC

// Use native arithmetic
#include <cmath>

#include "Integer.h"

// This cannot be inline, because it is recursive.
// Recursive inline functions cause some compilers to fail.

Integer GCD(const Integer& a, const Integer& b)
{
    if (a < 0) return GCD(-a, b);
    else if (b < 0) return GCD(a, -b);
    if (IsZero(b)) return a;
    else return GCD(b, mod(a,b) );
}


#endif // #ifdef USE_EXACT_INTEGER_ARITHMETIC
```

A.4.3 Quotient.h

```
#ifndef Quotient_h
#define Quotient_h

#include "Integer.h"
#include "ostream.h"

class Quotient
{
public:
    Quotient()
        {num = ZERO; den = ONE;}

    Quotient(Integer n, Integer d)
```

```
        {
            if (sign(d) < 0) {num = -n; den = -d;}
            else {num = n; den = d;}
        }

    double asDouble() const; // returns value in [0,1]

    inline Quotient& operator=(const Quotient& a)
        {num = a.num; den = a.den;
        return *this;}

    inline operator==(const Quotient& a) const
        {return ((num*a.den) == (a.num*den));}

    inline operator>(const Quotient& a) const
        {return ((num*a.den) > (a.num*den));}

    inline operator>=(const Quotient& a) const
        {return ((num*a.den) >= (a.num*den));}

    inline operator<(const Quotient& a) const
        {return ((num*a.den) < (a.num*den));}

    inline operator<=(const Quotient& a) const
        {return ((num*a.den) <= (a.num*den));}

    Integer num;
    Integer den;
};

ostream& operator<<(ostream& os, const Quotient q);

// Define some Rational constants
const Quotient ONEHALF = Quotient(ONE, TWO);

#endif
```

A.4.4 Quotient.cc

```
#include "Quotient.h"


ostream& operator<<(ostream& os, const Quotient q)
{ return (os << q.num << '/' << q.den);}


double
Quotient::asDouble() const
```

```
{
    return (to_double(num) / to_double(den));
}
```

A.5 Cutting Plane Measure

The CutSat program uses class CutMeasure and class TableauMeasure to induce a partial order of cutting planes. The greatest cutting planes, according to the definition of this order, are used in the proofs.

A.5.1 CutMeasure.h

```
//
// Copyright (c) 2000 Stephen Lee Hansen
//

#ifndef CutMeasure_h
#define CutMeasure_h

#include <limits.h>
#include <vector>
#include <set>

#include "Integer.h"
#include "Quotient.h"
#include "Parameters.h"

//
// A Measure on a set is a an additive monotonic set function.
// An Outer Measure o a set is a sub-additive monotonic set
// function \cite(Doob1993, Halmos1950). Either defines a
// partial order of the subsets of a set.
//
// Mathematically, a CutMeasure is a subadditive partial ordering
// of the subsets of a hypercube, where the subsets are defined by
// cutting plane inequalities.  The sub-additive property is
// a generalization (to higher dimensions) of the familiar
// geometric triangle inequality.
//
// CutMeasure provides comparison operators (>, <, ==, >=, <=)
// that implement the induced partial order on cutting planes.
// The comparison operators use the value() of the CutMeasure,
// which is a floating-point number approximately representing
// the abstract  measure.
```

```
//
// An assortment of geometric measures, including the distance,
// the strength (fraction of diagonol), and the power (volume)
// of the cutting plane are provided.  The selection of
// which to use is done by editing the value() function.

// Several variants of CutMeasure all are based on geometric
// measures such as distance and volume.  This makes it very
// easy to see that the various measures are indeed (approximate)
// outer measures.
//
//

class CutMeasure
{
public:
    CutMeasure()
        : num(ZERO), den(ONE), sum(ZERO),
         count(1), cols(0), _value(-1)
    {}

    CutMeasure(vector<Integer>& cut)
        : num(cut[0]*cut[0]), den(ZERO), sum(ZERO),
         count(0), cols(cut.size()-1),  _value(-1)
    {
        for (int col = 1; col < cut.size(); col++)
        {
            if (cut[col] < ZERO)
            {
                sum += cut[col];
                den += cut[col]*cut[col];
                count++;
            }
        }
    }

    CutMeasure(const CutMeasure& a)
        : num(a.num), den(a.den), sum(a.sum),
         count(a.count), cols(a.cols),  _value(-1)
    {}

    inline CutMeasure& operator=(const CutMeasure& a)
    {
        num = a.num;
        den = a.den;
        count = a.count;
        cols = a.cols;
        sum = a.sum;
        _value = a._value;
        return *this;
    }
```

```
    // The public interface of a measure is the ordering,
    // which is encapsulated as comparison operators.
    // The numerical value of any measure is unimportant.
    // The important part is that the measure can be used
    // to compare two objects.
    //
    // Which cut is better or more valuable ?
    // Operator > compares the values of two cuts.

    inline  operator>(const CutMeasure& a) const
    {return (value() > a.value());}

    inline  operator>=(const CutMeasure& a) const
    {return ( !(a > *this) );}

    inline  operator<(const CutMeasure& a) const
    {return (a > *this);}

    inline operator<=(const CutMeasure& a) const
    {return (! (*this > a));}

    inline  operator==(const CutMeasure& a) const
    {return ( !(*this > a) && !(a > *this) );}

    inline  operator!=(const CutMeasure& a) const
    {return (! (*this == a));}


    // Predicates to answer questions about the cut.

    // If a tableau contains no fractional basic variables,
    // the best cut has (num == ZERO).  We can use this to
    // test for integer-valued solutions.
    inline bool isIntegral()
    {
        return IsZero(num);
    }

    // If a tableau has fractional basic variables, but no
    // fractional coeficients in one of those rows, then
    // we have a refutation.
    inline bool isRefutation()
    {
        return (NonZero(num) && IsZero(den));
    }

//private:
    // Internally, the ordering of the measure depends on a
    // choice of geometric measures.  These are private.
    // The only purpose of these geometric measures is to
```

```
// be used by the comparison operators.
//
// For ease of modification, we provide methods
// to compute various geometric measures of the cut.

// Distance from origin to the cutting plane.
inline double distance() const
{
    return sqrt(to_double(num) / to_double(den));
}

// Distance, as a fraction of the diagonol.
inline double strength() const
{
    return sqrt(to_double(num) /
                (to_double(den) * count));
}

// Power of the cut, computed by interpolation
// of fraction of the points in the hypercube that are
// eliminated by the cut.
double power() const;

// Unfortunately, the power function overflows very easily
// and is expensive to compute.  So we want some other
// function that induces approximately the same partial ordering
// that is induced by the power function.
double logPower() const;

// Cosine of angle between the vector orthogonol to the
// cutting plane and the vector that is the diagonol
// of the hypercube. (Ref. e.g. (Shilov1971, )
inline double cos() const
{
    return ( - to_double(sum) / (sqrt(to_double(den) * count)));
}

// Sin of the same angle
inline double sin() const
{
    // sin() == sqrt(1 - cos()*cos())
    // The following just avoids an extra sqrt() call.
    double s = to_double(sum);
    return sqrt(1.0 - (s*s)/(to_double(den) * count));
}

// Distance from the origin to intersection of the
// cutting plane with the diagonol of the hypercube.
inline double diagonolDistance() const
{
    // diagonolDistance = distance() / sin()
```

```
        // The following just avoids the extra sqrt() call.
        double d2 = to_double(num) / to_double(den);
        // d2 == square of distance()
        double s = to_double(sum);
        double s2 = 1.0 - (s*s)/(to_double(den) * count);
        // s2 == square of sin()
        return sqrt(d2 / s2);
    }

    // The measure of a cut depends on a choice of
    // definitions.  There are several reasonable
    // ways to define the ordering, based on the various
    // geometric measures.
    //
    // For this "test-bed" program, we make it easy
    // to modify.  To modify, just choose one value calculation
    // or define a new one, un-comment the chosen definition,
    // and comment-out the others.
    inline double value() const
    {
        if (_value == -1)
        {
            // cast-away const-ness, so we can cache the value.
            CutMeasure* self = (CutMeasure*)this;

            // compute the value, using the parameterized
            // choice of function.
            self->_value = CUT_VALUE_FUNCTION;
        }
        return _value;
    }

//private:

    // Direct observations from the cut inequality
    Integer     num;        // square of constant (column 0)
    Integer     den;        // sum of squares of negative coeficients
    Integer     sum;        // sum of coeficients
    int         count;      // count of negative coeficients
    int         cols;       // count of columns (nonbasic vars)

    // Computed value is cached to avoid repeated computation.
    double      _value;

    // Allow relatives to access the variables
    friend  class TableauMeasure;
    friend  ostream& operator<<(ostream& os, const CutMeasure q);
};
ostream& operator<<(ostream& os, const CutMeasure q);
```

```
class TableauMeasure
{
public:
    TableauMeasure()
        : cuts(0), best(), cutRows(0),
            sumCounts(0),  ties(0),
            denominator(), objective()
    {}

    inline TableauMeasure& operator=(const TableauMeasure& a)
    {
        cutRows = a.cutRows;
        best = a.best;
        sumCounts = a.sumCounts;
        cuts = a.cuts;
        ties = a.ties;
        denominator = a.denominator;
        objective = a.objective;
        return *this;
    }

    inline void
    measureCut(vector<Integer>& cut)
    {
        // Measure the cut, and keep the strongest measurement
        CutMeasure cutMeasure = CutMeasure(cut);
        if (cuts == 0)
        {
            best = cutMeasure;
            ties = 1;
        }
        else if (cutMeasure > best)
        {
            best = cutMeasure;
            ties = 1;
        }
        else if (cutMeasure == best)
        {
            ties++;
        }

        cuts++;
        sumCounts += cutMeasure.count;
    }

    // Which tableau has the best cut?
    // (Alternatively:  Which vertex of the polytope
    // has the best cutting plane?)
    // From the perspective of the entire tableau,
```

```
    // the decision may consider not only the cut but
    // also other attributes of the tableau.
    // The decision is encapsulated in the comparison
    // operators of TableauMeasure.

    inline operator>(const TableauMeasure& a) const
    {
        return (
                // Keep the strongest cut
               ((best > a.best)
                ||
                ((best == a.best)
                 &&
                 // break ties by choosing: smallest denominator
                 ((denominator < a.denominator)
                  ||
                  ((denominator == a.denominator)
                   &&
                   // break ties by choosing: fewest fractional coeficients
                   (((sumCounts/cuts) < (a.sumCounts/a.cuts))
                    ||
                    (((sumCounts/cuts) == (a.sumCounts /a.cuts))
                     &&
                     // break ties by choosing: greatest objective
                     ((objective > a.objective)
                      ||
                      ((objective == a.objective)
                       &&
                           false)))))))));
    }

    inline  operator>=(const TableauMeasure& a) const
    {return ( !(a > *this) );}

    inline  operator<(const TableauMeasure& a) const
    {return (a > *this);}

    inline operator<=(const TableauMeasure& a) const
    {return ( !(*this > a) );}

    inline  operator==(const TableauMeasure& a) const
    {return ( !(*this > a) && !(a > *this) );}

    inline  operator!=(const TableauMeasure& a) const
    {return ( !(*this == a) );}

//protected:

    int         firstCut;       // initialization flag
    CutMeasure  best;           // greatest cut in the tableau
```

```
    // Cumulative Observations from all cuts
    double      cuts;           // count of cuts
    double      sumCounts;      // count of nonzero coeficients in all cuts
    double      ties;           // count of cuts tied for best

    // Observations from the tableau
    int         cols;           // count of columns in the tableau
    double      cutRows;        // count of cut rows in the tableau
    double      integralRows;   // count of non-cut rows in the tableu
    Integer     denominator;    // denominator of the tableau
    Quotient    objective;      // objective value of the tableau
};
ostream& operator<<(ostream& os, const TableauMeasure q);




#endif
```

A.5.2 CutMeasure.cc

```
/*
 *   2000 Stephen Hansen
 */

#include "ostream.h"

#include "CutMeasure.h"
#include "Integer.h"

#include <iomanip>

ostream& operator<<(ostream& os, const CutMeasure q)
{
//  os << exp2(q.value()) << "(";
    os << q.value() << "(";
    os << q.num << '/' << q.den << "," << q.count ;
    os << ")" ;

    return os;
}

ostream& operator<<(ostream& os, const TableauMeasure q)
{
    os << q.best << " (" << q.ties << " of " << q.cuts << " cuts)";
    if (q.denominator != ZERO)
        os << " den=" << q.denominator;

    return os;
```

```
}

#include <iostream.h>
#include <vector.h>
#include <math.h>


long double binom( long  n, long  k);
long double cumBinom( long  n, long  k);
long double pow2(long n);
long double cutPower(long n, double strength);

// Binomial (cached)
long double binom( long  n, long  k)
{
   static vector< vector< long double > > values;
   static vector<long double>  row;

   if ((k == 0) || (k == n)) return 1;
   if ((k < 0) || (k > n)) return 0;

   for (long i = values.size(); i <= n; i++)
   {
      while (row.size() <= i+1) row.push_back(0);
      values.push_back(row);
   }
   if (values[n][k] == 0)
   {
      values[n][k] = binom(n-1, k-1) + binom(n-1, k);
   }
   return values[n][k];
}

// Cumulative Binomial (cached)
long double cumBinom( long  n, long  k)
{
   static vector< vector< long double > > values;
   static vector<long double>  row;

   if (k == 0) return 1;
   if (k < 0) return 0;
   if (k > n) k = n;

   for (long i = values.size(); i <= n; i++)
   {
      while (row.size() <= i+1) row.push_back(0);
      values.push_back(row);
   }
   if (values[n][k] == 0)
   {
      values[n][k] = cumBinom(n-1, k-1) + cumBinom(n-1, k);
```

```
   }
   return values[n][k];
}

// Powers of 2 (cached)
long double pow2(long n)
{
    static vector<long double> values;
    for (long i = values.size(); i <= n; i++)
    {
        if (i == 0) values.push_back(1.0);
        else values.push_back(2.0 * values[i-1]);
    }
    return values[n];
}


// Compute the volume of a hypercube that is cut off by
// of a cut of given strength.  We approximate the
// volume by assuming that the cut is orthogonol to the
// diagonol of the hypercube.  The assumption allows an
// efficient calculation.

double CutMeasure::power() const
{
    // How many variables need to be true?
    double s = strength() * count;

    // How many ways can that number of variables
    // be true?  (This uses linear interpolation to
    // allow for fractional numbers of variables.)
    long sup = ceil(s);
    long inf = floor(s);
    double f = (s - inf);
    double ip = cumBinom(count, inf)
        + (f * (binom(count, sup) + binom(count, inf)) / 2);

    // divide by the power of 2
    return (ip / pow2(count));
}

// Same as above, but use logarithms to avoid using
// the pow2 function.
double CutMeasure::logPower() const
{
    // How many variables need to be true?
    double s = strength() * count;

    // Compute the volume of the cut, assuming
    // that it is orthogonol to the diagonol.
    double sup = ceil(s);
```

```
    double inf = floor(s);
    double f = (s - inf);

    double ip = cumBinom(count, inf)
        + (f * (binom(count, sup) + binom(count, inf)) / 2);

    // return log_{2}(ip/2^{count)}
    return (log2(ip)  - count);
}
```

A.6 The Integer Simplex Tableau

The IntegerSimplex class, and related minor classes, define the data structures and methods implementing various linear programming algorithms, and some cutting plane algorithms that are closely related to those linear programming algorithms.

A.6.1 Simplex.h

```
#ifndef SIMPLEX_H_INCLUDED
#define SIMPLEX_H_INCLUDED

#include <vector.h>
#include "Integer.h"
#include "Quotient.h"
#include "CutMeasure.h"

// Define LP Status
typedef enum
{
    CONTINUE    = 0,
    OPTIMAL     = 1,
    UNBOUNDED   = 2,
    FEASIBLE    = 3,
    INFEASIBLE  = 4
} LPStatus;


// A class for the name of a variable
class Variable
{
public:
    Variable(const string n, const int s) : name(n)
    { subscript = s; }
```

```
    Variable(const char* n, const int s) : name(n)
    { subscript = s; }

    Variable(const Variable& v): name(v.name)
    { subscript = v.subscript; }

    Variable()
    { name = "x"; subscript = 0; }

    string  name;
    int     subscript;

    bool isSlack()
    { return (name == "s"); }

    inline operator==(const Variable& v)
    {return (subscript == v.subscript && name == v.name);}

};

ostream& operator<<(ostream& os, const Variable& s);



// Define a tableau data structure for the simplex method,
// with utilities, simplex methods, and some variants.
class IntegerSimplex
{
public:

    // Constructors
    IntegerSimplex(int columns);
    IntegerSimplex(const IntegerSimplex& tableau);

    ~IntegerSimplex();

    // Methods for setting the problem into the tableau,
    void    setObjective(const vector<Integer>& obj);
    bool    addConstraint(const vector<Integer>& constraint);

    // and for removing redundant constraints,
    void    eraseConstraint(int index);

    // Simplex algorithms
    LPStatus    primalDualSimplex();
    LPStatus    dualSimplex();
    LPStatus    primalSimplex();

    // The pivot operation
    void        pivot(int  row, int  column);
```

```
    // Variant methods for SAT problems
    // (These could be defined in a sub-class)
    LPStatus    primalDualCutLifting();
    LPStatus    denominatorReduction();

    // Methods to measure and improve cutting planes
    TableauMeasure  primalCutImproving();
    TableauMeasure  measureCuts();

    // Look to see if the current solution is fractional
    bool        isFractionalSolution();

    // Methods for reading the results out of the tableau.
    LPStatus readPrimalSolution(Integer& objective,
                    vector<Integer>& primalVariables,
                    vector<Integer>& primalSlacks,
                    Integer& denominator);

    LPStatus readDualSolution(Integer& objective,
                    vector<Integer>& dualVariables,
                    vector<Integer>& dualSlacks,
                    Integer& denominator);

    // Methods to print rows with variable names
    void printObjective(ostream& os) const;
    void printEquality(ostream& os, int row) const;
    void printInequality(ostream& os, int row) const;

    // Method to print the vector of nonbasic variables
    void printNonbasicVariables(ostream& os) const;

protected:
//public: // could be protected/private, but public for rapid development

    // Submethods for simplex algorithms
    LPStatus    findFeasibleSolution();
    int         chooseDualPivotColumn(int &row);
    LPStatus    chooseDualPivot(int& row, int& column);
    int         choosePrimalPivotRow(int &column);
    LPStatus    choosePrimalPivot(int& row, int& column);

    // Method for selecting pivots that give small denominator.
    LPStatus    chooseDenominatorReducingPivot(int &pivot_row, int &pivot_col);
    int         chooseDenominatorReducingPivotRow(int col);
    int         chooseDenominatorReducingPivotCol(int col);

    // Methods for selecting pivots that give strong cuts.
    TableauMeasure  measureCutsAfterPivot(int row, int col, int sampleSize);
    TableauMeasure  choosePrimalCutImprovingPivot(int &pivot_row, int &pivot_col,
                        TableauMeasure init_strength);
```

```
    int             choosePrimalCutImprovingPivotRow(const int col,
                        TableauMeasure& strength);

    // The artificial objectives (with random purturbations)
    // for tie-breaking during pivot-selection (to avoid cycling)

    vector<Integer> rowZero;
    void            initializePrimalRandomness();
    void            dropPrimalRandomness();

    vector<Integer> colZero;
    void            initializeDualRandomness();
    void            dropDualRandomness();


public:
    // The tableau structure
    int                 rows;           // size of the tableau
    int                 cols;           //
    Integer             denominator;    // The common denominator
    vector<vector<Integer> > mat;       // The matrix of numerators
    vector<Variable>    basic_vars;     // primal basis variables
    vector<Variable>    nonbasic_vars;  // dual basis variables

    // Miscelaneous data
    LPStatus            status;     //
    int                 nextVar;    // Next slack variable number
    vector<Integer>     originalObjective;

    int     pivotCounter;           // How many pivots were used?
};

ostream& operator<<(ostream& os, const LPStatus& s);
ostream& operator<<(ostream& os, const IntegerSimplex& v);


// Operators that are useful for vectors, but are
// not defined by STL vector class:

template<class T>
ostream& operator<<(ostream& os, const vector<T>& v)
{
    for (vector<T>::const_iterator i = v.begin(); i != v.end(); i++)
    {
            os << *i << "\t";
    }
    return os;
}

// Sum of two vectors
template <class T>
```

```
vector<T> operator+(const vector<T> &A,
    const vector<T> &B)
{
    int N = A.size();
    assert(N==B.size());
    vector<T> tmp(N);
    int i;

    for (i=0; i<N; i++)
            tmp[i] = A[i] + B[i];

    return tmp;
}

// Difference of two vectors
template <class T>
vector<T> operator-(const vector<T> &A,
    const vector<T> &B)
{
    int N = A.size();
    assert(N==B.size());
    vector<T> tmp(N);
    int i;

    for (i=0; i<N; i++)
            tmp[i] = A[i] - B[i];

    return tmp;
}

// Multiplication (dot-product) of two vectors
template <class T>
vector<T> operator*(const vector<T> &A,
    const vector<T> &B)
{
    int N = A.size();

    assert(N==B.size());

    vector<T> tmp(N);
    int i;

    for (i=0; i<N; i++)
            tmp[i] = A[i] * B[i];

    return tmp;
}

// Scalar multiplication
template <class T>
vector<T> operator*(const T &A, const vector<T> &B)
```

```
{
    int N = A.size();

    assert(N==B.size());

    vector<T> tmp(N);
    int i;

    for (i=0; i<N; i++)
            tmp[i] = A * B[i];

    return tmp;
}

// Scalar multiplication
template <class T>
vector<T> operator*(const vector<T> &B, const T &A)
{
    return operator*(A, B);
}

#endif /* ifndef SIMPLEX_H_INCLUDED */
```

A.6.2 Simplex.cc

```
//
// The IntegerSimplex class implements an "integer only"
// primal-dual simplex algorithm, per (Gomory 1963).
//
// An perturbed objective row (column) with random
// perturbations is used to avoid degenerate cycling.
// Any remaining ties are broken randomly.
//
// Calculations are performed using variables of class
// "Integer".  Integer objects are assumed to implement
// integer arithmetic operations with precision appropriate
// for the problem.
//
// Copyright (c) 2000 Stephen Lee Hansen
//

#include <vector>
#include <list>
#include <strstream>
#include <string>
#include <stdlib.h>
#include <assert.h>
#include "Parameters.h"
#include "Integer.h"
```

```
#include "Simplex.h"
#include "random.h"
extern void keepMacHappy();

IntegerSimplex::IntegerSimplex(int columns)
    : mat(), denominator(ONE),
        originalObjective(1+columns, ZERO),
        basic_vars(),
        nonbasic_vars(1+columns),
        rows(0),
        cols(columns),
        rowZero(),
        colZero()
{
    // provide an empty row for the objective function
    vector<Integer> objective = vector<Integer>(1+columns, ZERO);
    mat.push_back(objective);
    basic_vars.push_back(Variable("x",0));

    // assign names to the primal nonbasic (dual basic) variables
    nextVar = 1;
    for (int j = 1; j<= cols; j++)
    {
        nonbasic_vars[j] = Variable("x", nextVar++);
    }

    // Initialize number of first slack variable
    nextVar = 1;
}

// Copy constructor
IntegerSimplex::IntegerSimplex(const IntegerSimplex& st)
    : mat(st.mat), denominator(st.denominator),
        basic_vars(st.basic_vars),
        nonbasic_vars(st.nonbasic_vars),
        originalObjective(st.originalObjective),
        rows(st.rows),
        cols(st.cols),
        nextVar(st.nextVar),
        status(st.status),
        rowZero(st.rowZero),
        colZero(st.colZero)
{}

// Destructor -- just need to invoke instance-variable
// destructors.  (Some compilers do not invoke them for
// classes that do not have explicit destructors.)
IntegerSimplex::~IntegerSimplex()
{}

bool
```

```
IntegerSimplex::addConstraint(const vector<Integer>& constraint)
{
    assert(constraint.size() == 1+cols);
    assert(mat.size() == 1+rows);
    assert(basic_vars.size() == 1+rows);

    // Do not add duplicate constraints
    for (int row = 1; row <= rows; row++)
        if (constraint == mat[row])
            return false;

    // Add the new row to the matrix
    rows++;
    mat.push_back(constraint);

    // and give it a slack (dual) (primal basis) variable
    basic_vars.push_back(Variable("s",nextVar));
    nextVar++;

    return true;
}

void
IntegerSimplex::eraseConstraint(int row)
{
    assert(row <= rows);

    mat.erase(&mat[row]);
    basic_vars.erase(&basic_vars[row]);
    if (colZero.size() >= row) colZero.erase(&colZero[row]);
    --rows;
}

void
IntegerSimplex::setObjective(const vector<Integer>& obj)
{
    // objective coeficients are stored in mat[0]
    // mat[0,0] is the objective value.
    assert (obj.size() == 1+cols);
    originalObjective = mat[0] = obj;
}

// Methods to manage the perturbation row and column.
void
IntegerSimplex::initializePrimalRandomness()
{
    // Initialize elements of column zero for each row.
    Integer randomness;
    colZero = vector<Integer>(1+rows);
    for (int row = 0; row <= rows; row++)
    {
```

```
        randomness = random() & 0xffffff;
        colZero[row] =  randomness * denominator;
    }
}

void
IntegerSimplex::initializeDualRandomness()
{
    Integer randomness;
    rowZero = vector<Integer>(1+cols);
    for (int col = 0; col <= cols; col++)
    {
        randomness = random() & 0xffffff;
        rowZero[col] = randomness * denominator;
    }
}

void
IntegerSimplex::dropPrimalRandomness()
{
    colZero = vector<Integer>();
}

void
IntegerSimplex::dropDualRandomness()
{
    rowZero = vector<Integer>();
}

// This shuffle() function is adapted from std::random_shuffle.
// The random number generator used by random_shuffle is
// really bad.  This version uses a decent RNG.

template <class RandomAccessIterator>
inline void
shuffle(RandomAccessIterator first, RandomAccessIterator last)
{
    if (first != last)
    for (RandomAccessIterator i = first + 1; i != last; ++i)
    {
        long r =  random() % ((i - first) + 1);
        iter_swap(i, first + r);
    }
}

////////////////////////////////////////////////////////////
//
// PRIMAL-DUAL SIMPLEX METHOD
//
// This is actually a DUAL-PRIMAL method.
// We first find a feasible solution, then find
```

```
// an optimal feasible solution.

LPStatus
IntegerSimplex::primalDualSimplex()
{
    findFeasibleSolution();
    if (status == INFEASIBLE) return status;

    primalSimplex();
    return status;
}

// To find a feasible starting solution, we use a dual
// simplex method with a dummy objective row.  The real
// objective row is carried along in mat[rows+1]
LPStatus
IntegerSimplex::findFeasibleSolution()
{
    int j;

    // Search for negative coeficients in the objective
    Integer m = ZERO;
    for (j = 0; j <= cols ; j++)
        if (mat[0][j] < m) m = mat[0][j];

    // If no negative coeficients, we can just
    // use a dual simplex method.
    if (m >= 0) return (dualSimplex());

    // Otherwise, we must use a modified objective.

    // Stash the real objective at the end.
    int realRows = rows;
    mat.push_back(mat[0]);
    rows = realRows+1;

    // Add an integer to each coeficient in the objective
    // to obtain a non-negative objective row.
    m =  mod(m, denominator) - m;
    for (j = 0; j <= cols ; j++)
        mat[0][j] = mat[0][j] + m;

    // Use dual pivots to find a feasible solution
    initializeDualRandomness();
    pivotCounter = 0;
    while(true)
    {
        int leaving_col, entering_row;

        // Choose a pivot among the real constraint rows,
        // but carry the real objective through the pivots.
```

```
        rows = realRows;
        chooseDualPivot(entering_row, leaving_col);
        rows = realRows+1;
        if (status != CONTINUE) break;

        // Otherwise, this is just like dualSimplex.
        pivot(entering_row, leaving_col);
    }
    dropDualRandomness();
    cout <<   pivotCounter << " dual pivots were used " << endl;

    // restore the real objective function, as modified
    // by the pivots.
    rows = realRows;
    mat[0] = mat[rows+1];
    mat.erase(&mat[rows+1]);

    // cout << "After pivots, the real objective row is:" << endl;
    // cout << mat[0] << endl;

    return status;
}

//////////////////////////////////////////////////////
//
// DUAL SIMPLEX METHODS
//
//////////////////////////////////////////////////////
//
// Choose the dual pivot that gives the greatest
// decrease in the objective function.  This choice
// tends to require fewer pivots than the traditional
// steepest-descent method.
int
IntegerSimplex::chooseDualPivotColumn(int &row)
{
    // Allow the GUI OS to get control occasionally
    keepMacHappy();

    // Choose the col with the smallest (absolute value) negative ratio

    int         best_col = -1;  // Best column
    Quotient    best_M_q;   // Major part of best quotient
    Quotient    best_u_q;   // micro part of best quotient

    for ( int j=1; j<=cols; j++ )
    {
        if (( sign(mat[row][j]) < 0) && (sign(mat[0][j]) >= 0))
        {
            // Compute (Major,micro) quotient for this column.
            Quotient M_q = Quotient(mat[0][j],-mat[row][j]);
```

```
            Quotient u_q = Quotient(rowZero[j],-mat[row][j]);

            // Keep the column with smallest (Major,micro) quotient,
            // And break ties randomly.
            if ( (best_col == -1)
                 ||
                 (M_q < best_M_q)
                 ||
                 ( (M_q == best_M_q)
                   &&
                   ( (u_q < best_u_q)
                     ||
                     ( (u_q == best_u_q)
                       &&
                       randomBit())))
            {
                best_M_q = M_q;
                best_u_q = u_q;
                best_col = j;
            }
        }
    }

//cout << "chooseDualPivotCol, row=" << row <<" col=" << best_col
//     << " q =" << q <<  endl;

    return best_col;
}

LPStatus
IntegerSimplex::chooseDualPivot(int &pivot_row, int &pivot_col)
{
    int         row;
    int         col;
    int         best_row = -1;
    int         best_col = -1;
    Quotient    best_M_value;
    Quotient    best_u_value;

    /* preset, no infeasible constraint rows remain */
    status = FEASIBLE;

    for (row=1; row<=rows; row++ )
    {
        if (sign(mat[row][0]) >= 0) continue;

        // found an infeasible constraint. Now look for
        // a pivot to eliminate that infeasibility.
        col = chooseDualPivotColumn(row);
        if  (col == -1)
        {
```

```
            // no feasible pivot exists in an infeasible row
            best_row = row;
            status = INFEASIBLE;

// cout << "INFEASIBLE row:" << best_row << endl;
// cout << "INFEASIBLE row:" << mat[row] << endl;
// cout << "        mat[0]:" << mat[0] << endl;
// cout << "   denominator:" << denominator << endl;
            break;
        }

        // Keep the pivot that gives the greatest objective decrease.
        // And break ties randomly.
        Quotient M_value = Quotient(-mat[0][col]*mat[row][0], mat[row][col]);
        Quotient u_value = Quotient(-rowZero[col]*mat[row][0], mat[row][col]);

        if ((best_row == -1)
            ||
            (M_value < best_M_value)
            ||
            ( (M_value == best_M_value)
              &&
              ( (u_value < best_u_value)
                ||
                ( (u_value == best_u_value)
                  &&
                  randomBit())))
        {
            best_row = row;
            best_col = col;
            best_M_value = M_value;
            best_u_value = u_value;
            status = CONTINUE;

//  cout << "The pivot position (" << best_row << "," << best_col << ")"
//       << " has value " << best_M_value << endl;
        }
    }

    pivot_row = best_row;
    pivot_col = best_col;
    return status;
}

LPStatus
IntegerSimplex::dualSimplex()
{
    int row,col;

    initializeDualRandomness();
    pivotCounter = 0;
```

```
    while (true)
    {
        chooseDualPivot( row, col);
        if (status != CONTINUE) break;
        pivot( row, col);
    }
    dropDualRandomness();
    cout <<   pivotCounter << " dual pivots were used " << endl;
    return status;
}

////////////////////////////////////////////////////
//
// CUT LIFTING
//
////////////////////////////////////////////////////
//
// The lifting algorithms do not care about the objective.
// value -- only whether or not it is less than zero.
// So we provide a specialized version that returns
// a decision of whether or not the optimal value is
// less than zero.  This gives the lifting algorithm
// a speedup.
//
// Note that this is a DUAL-PRIMAL method.
//

LPStatus
IntegerSimplex::primalDualCutLifting()
{
    findFeasibleSolution();
    if (status == INFEASIBLE) return status;
    if (sign(mat[0][0]) >= 0) return status;

    // Then apply primal pivots until we obtain
    // a non-negative value, or can determine that
    // the optimal value is negative.

    initializePrimalRandomness();
    while (sign(mat[0][0]) < 0)
    {
        int row,col;
        choosePrimalPivot( row, col);
        if (status != CONTINUE) break;

        pivot( row, col);
    }
    dropPrimalRandomness();
    return status;
}
```

```
////////////////////////////////////////////////////
//
// PRIMAL SIMPLEX METHODS
//
////////////////////////////////////////////////////

// Choose the dual pivot that gives the greatest
// decrease in the objective function.  This choice
// tends to require fewer pivots than the traditional
// steepest-descent method.
int
IntegerSimplex::choosePrimalPivotRow(int &col)
{
    // Allow the GUI OS to get control occasionally
    keepMacHappy();

    // Choose the row with the smallest positive ratio
    int         best_row = -1;
    Quotient    best_M_q;   // Major part of best quotient
    Quotient    best_u_q;   // micro part of best quotient

    for ( int i=1; i<=rows; i++ )
    {
        if (( sign(mat[i][col]) > 0) && ( sign(mat[i][0]) >= 0))
        {
            // Compute (Major,micro) quotient for this column.
            Quotient M_q = Quotient(mat[i][0],mat[i][col]);
            Quotient u_q = Quotient(colZero[i],mat[i][col]);

            // Keep the row with smallest (Major,micro) quotient,
            // And break ties randomly.
            if ( (best_row == -1)
                 ||
                 (M_q < best_M_q)
                 ||
                 ( (M_q == best_M_q)
                   &&
                   ( (u_q < best_u_q)
                     ||
                     ( (u_q == best_u_q)
                       &&
                       randomBit())))
            {
                best_M_q = M_q;
                best_u_q = u_q;
                best_row = i;
            }
        }
    }
    return best_row;
}
```

```
LPStatus
IntegerSimplex::choosePrimalPivot(int &pivot_row, int &pivot_col)
{
    int         row;
    int         col;
    int         best_col = -1;
    int         best_row = -1;
    Quotient    best_M_value;   // Major part of best pivot value
    Quotient    best_u_value;   // micro part of best pivot value

    // preset, no infeasible constraint rows remain
    status = OPTIMAL;

//  // To randomize the order of visiting the columns:
//  vector<int> columns = vector<int>(cols);
//  for (col = 1; col <= cols; col++) columns[col-1] = col;
//  shuffle(columns.begin(), columns.end());
//  for (int j=0; j<cols; j++ )
//  {
//      col = columns[j];

    for (col = 1; col <= cols; col++)
    {
        if (sign(mat[0][col]) >= 0) continue;

        // Found a suboptimal column.
        row = choosePrimalPivotRow(col);
        if  (row == -1)
        {
            // no finite pivot exists in a suboptimal column
            best_col = col;
            status = UNBOUNDED;
            break;
        }

        // Keep the pivot that gives the greatest objective increase.
        // Break ties randomly
        Quotient M_value = Quotient(-mat[0][col]*mat[row][0], mat[row][col]);
        Quotient u_value = Quotient(-mat[0][col]*colZero[row], mat[row][col]);

        if ((best_row == -1)
            ||
            (M_value > best_M_value)
            ||
            ( (M_value == best_M_value)
              &&
              ( (u_value > best_u_value)
                ||
                ( (u_value == best_u_value)
                  &&
```

```
                    randomBit())))
        {
            best_row = row;
            best_col = col;
            best_M_value = M_value;
            best_u_value = u_value;
            status = CONTINUE;

//cout << "The pivot position (" << best_row << "," << best_col << ")"
//<< " has value " << pivot_value << endl;
         }
    }

    pivot_row = best_row;
    pivot_col = best_col;
    return status;
}

LPStatus
IntegerSimplex::primalSimplex()
{
  int row,col;
  initializePrimalRandomness(); // init the tiebreaker
  pivotCounter = 0;

  while(true)
  {
    choosePrimalPivot( row, col);
    if (status != CONTINUE) break;

    pivot(row, col);
  }

  dropPrimalRandomness();
  cout <<   pivotCounter << " primal pivots were used " << endl;
  return status;
}



//////////////////////////////////////////////////////
//
// DENOMINATOR REDUCTION METHODS
//
//////////////////////////////////////////////////////

int
IntegerSimplex::chooseDenominatorReducingPivotRow(int col)
{
    // Allow the GUI OS to get control occasionally
    keepMacHappy();
```

```
    // Choose the row with the smallest ratio,
    // and the smallest (primal) pivot element
    int         best_row = -1;
    Quotient    best_quotient;
    Integer     bestElement;

    for ( int row=1; row<=rows; row++ )
    {
        if ( sign(mat[row][col]) > 0)
        {
            Quotient quotient = Quotient(mat[row][0],mat[row][col]);

            if ( (best_row == -1) ||
                 (quotient < best_quotient) ||
                 ( (quotient == best_quotient) && (mat[row][col] < bestElement))
            )
            {
                bestElement = mat[row][col];
                best_quotient = quotient;
                best_row = row;
            }
        }
    }

//cout << "choosePrimalPivotRow, col=" << col << " row=" << best_row
//     << " quotient =" << quotient  << endl;

    return best_row;
}

int
IntegerSimplex::chooseDenominatorReducingPivotCol(int row)
{
    // Allow the GUI OS to get control occasionally
    keepMacHappy();

    // Choose the column with the smallest ratio,
    // and the smallest (dual) pivot element
    int         best_col = -1;
    Quotient    best_quotient;
    Integer     bestElement;

    for ( int col=1; col<=cols; col++ )
    {
        if (sign(mat[row][col]) < 0)
        {
            Quotient quotient = Quotient(mat[0][col],mat[row][col]);

            if ((best_col == -1) ||
                (quotient > best_quotient) ||
```

```
                ((quotient == best_quotient) && (mat[row][col] > bestElement))
            )
            {
                bestElement = mat[row][col];
                best_quotient = quotient;
                best_col = col;
            }
        }
    }
    return best_col;
}

LPStatus
IntegerSimplex::chooseDenominatorReducingPivot(int &pivot_row, int &pivot_col)
{
    int         row;
    int         col;
    int         best_col = -1;
    int         best_row = -1;
    Integer     best_denominator = denominator;

    // preset, no denominator-reducing pivots remain
    status = OPTIMAL;

    // Randomize the order of visiting the columns
    vector<int> columns = vector<int>(cols);
    for (col = 1; col <= cols; col++) columns[col-1] = col;
    shuffle(columns.begin(), columns.end());
    for (int j=0; j<cols; j++ )
    {
        col = columns[j];

//#define   REDUCE_DENOMINATOR_WITHOUT_OBJECTIVE
#ifndef REDUCE_DENOMINATOR_WITHOUT_OBJECTIVE
        // Use only pivots that do not change the objective
        if (sign(mat[0][col]) != 0) continue;
#endif
        // find a pivot in this column
        row = chooseDenominatorReducingPivotRow(col);
        if  (row == -1) continue;

        // Keep the pivot that gives the denominator nearest ONE.
        if (abs(mat[row][col]) < best_denominator)
        {
            best_row = row;
            best_col = col;
            best_denominator = abs(mat[row][col]);
            status = CONTINUE;

//cout << "The pivot position (" << best_row << "," << best_col << ")"
//<< " has value " << pivot_value << endl;
```

```
        }
    }

    for (row=1; row<=rows; row++ )
    {
        // Use only dual pivots that preserve primal feasibility
        if (sign(mat[row][0]) != 0) continue;

        // find a pivot in this column
        col = chooseDenominatorReducingPivotCol(row);
        if  (col == -1) continue;

        // Keep the pivot that gives the smallest denominator.
        if (abs(mat[row][col]) < best_denominator)
        {
            best_row = row;
            best_col = col;
            best_denominator = abs(mat[row][col]);
            status = CONTINUE;

//cout << "The pivot position (" << best_row << "," << best_col << ")"
//<< " has value " << pivot_value << endl;
        }
    }

    pivot_row = best_row;
    pivot_col = best_col;
    return status;
}

//
// A method that can be applied to OPTIMAL (primal
// and dual feasible) tableau to reduce the denominator.
//
// This gives an equlvalent optimal solution, that
// is better only in the sense that it is expressed
// using a lower denominator.
//
// When several alternative solutions have the same objective
// value, the simplex method using an artificial objective
// chooses one of them at random.  This function has the
// effect of modifying the  artificial objective to choose
// the optimal solution with the lesser denominator.
//

LPStatus
IntegerSimplex::denominatorReduction()
{
  int row,col;

    // This applies only to optimal tableau!
```

```
    if (status != OPTIMAL) return status;

    while(true)
    {
        // Choose a zero-cost pivot that gives
        // a smaller denominator, if one is available.
        chooseDenominatorReducingPivot( row, col);
        if (status != CONTINUE) break;

cout << "Denominator Reducing pivot is:" << mat[row][col] << endl;

        // Take that pivot
        pivot(row, col);
    }

    status = OPTIMAL;
    return status;
}

//////////////////////////////////////////////////////
//
//  CUT STRENGTHENING METHOD
//
//////////////////////////////////////////////////////]
//
// Methods to do a local search, to find pivots that reduce the
// number of fractions in the tableau.
//


// Look to see if the current solution is fractional
bool
IntegerSimplex::isFractionalSolution()
{
    for (int row = 1; row <= rows; row++)
    {
        // If fractional basic variable value, we need a cut.
        if (NonZero( mod(mat[row][0], denominator)))
        {
            return true;
        }
    }
    return false;
}


// Measure the available cuts in the current tableau
TableauMeasure
IntegerSimplex::measureCuts()
{
    // Preset that no cuts are possible
```

```
    TableauMeasure tabMeasure = TableauMeasure();
    vector<Integer> cut = vector<Integer>(1+cols);
    int countCutRows = 0;
    int countIntegralRows = 0;

    for (int row = 1; row <= rows; row++)
    {
        if (sign(mod(mat[row][0], denominator)) != 0)
        {
            // Form the Gomory cut
            for (int col = 0; col <= cols; col++)
                cut[col] = - mod(mat[row][col], denominator);

            // Measure the cut, and keep the best/worst cut measures
            tabMeasure.measureCut(cut);

            ++countCutRows;
        }
        else
        {
            ++countIntegralRows;
        }
    }

    // Capture observations about the tableau.
    tabMeasure.objective = Quotient(mat[0][0], denominator);
    tabMeasure.denominator = denominator;
    tabMeasure.cols = cols;
    tabMeasure.cutRows = countCutRows;
    tabMeasure.integralRows = countIntegralRows;

cout <<"measureCuts: " <<   tabMeasure << endl;

    return  tabMeasure;
}

// Measure the cut measure that would result after a given pivot
TableauMeasure
IntegerSimplex::measureCutsAfterPivot(int row, int col, int sampleSize)
{
    TableauMeasure tabMeasure = TableauMeasure();

    int countCutRows = 0;
    int countIntegralRows = 0;

    vector<Integer> cut = vector<Integer>(1+cols);
    int firstCut = 1;
    int i,j;

    // Compute some common factors
    Integer element = mat[row][col];
```

```
Integer s = ONE;
Integer sd = denominator;
Integer abselement = element;
if (sign(element) < 0)
{
    abselement = -element;
    s =  MINUSONE;
    sd = -denominator;
}

// Always measure the cut of the pivot row
if ( NonZero(mod(s * mat[row][0], abselement)))
{
    // Count the non-integral rows in the tableau
    ++countCutRows;

    // Limit the sample of cuts
    sampleSize--;

    // Form the cut of the pivot row
    for (j = 0; j <= cols; j++)
        if (j != col)
        {
            if (IsZero(mat[row][j]))
                cut[j] = ZERO;
            else
                cut[j] = - mod(s * mat[row][j], abselement);
        }
    cut[col] = - mod(sd, abselement);

    // Measure the cut
    tabMeasure.measureCut(cut);
}
else
{
    ++countIntegralRows;
}

// Consider the nonpivot rows in a uniform random order.
vector<int> Rows(rows);
for ( i=0; i < rows; i++) Rows[i] = i+1;

for (int rowIdx = 0; rowIdx < rows; rowIdx++ )
{
    // Choose a random row from remaining rows.
    unsigned long r = random() % (rows - rowIdx);
    if (r != 0) swap(Rows[rowIdx], Rows[rowIdx+r]);
    i = Rows[rowIdx];

    // not the pivot row
    if (i == row) continue;
```

```
        // Compute the constant (column 0) that would
        // result after the pivot.
        // (We compute the partial products separately,
        // to avoid some of the arithmetic.)

        Integer P1 = ZERO;
        if (NonZero(mat[i][0]))
            P1 = mat[i][0] * element;
        Integer P2 =  ZERO;
        if (NonZero(mat[row][0]) && NonZero(mat[i][col]))
            P2 = mat[row][0] * mat[i][col];
        cut[0] =  - mod( (P1 - P2) / sd , abselement);

        // If it is integral, this row is not a cut row.
        if (IsZero(cut[0]))
        {
            ++countIntegralRows;
            continue;
        }

        // Count the non-integral rows in the tableau
        ++countCutRows;

        // Limit the sample of cuts
        if (sampleSize <= 0) break;
        sampleSize--;

        // Compute the Gomory cut of the selected row.
        for (j = 1; j <= cols; j++)
        if (j == col)
        {
            // Pivot column
            cut[col] =  - mod ( -s * mat[i][col], abselement);
        }
        else
        {
            // Non-pivot column
            Integer P1 = ZERO;
            if (NonZero(mat[i][j]))
                P1 = mat[i][j] * element;
            Integer P2 = ZERO;
            if (NonZero(mat[row][j]) && NonZero(mat[i][col]))
                P2 = mat[row][j] * mat[i][col];
            cut[j] =  - mod( (P1 - P2) / sd , abselement);
        }

        // Measure the cut
        tabMeasure.measureCut(cut);
    }
```

```
    // Record some observations about the tableau.
    // Save the objective value, denominator, and column count
    // for this resulting tableau.
    tabMeasure.objective =
        Quotient(((mat[0][0]*mat[row][col])-(mat[row][0]*mat[0][col]))
                        / denominator ,  mat[row][col]);
    tabMeasure.denominator = abs(mat[row][col]);
    tabMeasure.cols = cols;
    tabMeasure.cutRows = countCutRows;
    tabMeasure.integralRows = countIntegralRows;

    return tabMeasure;
}

int /*row*/
IntegerSimplex::choosePrimalCutImprovingPivotRow(const int col,
    TableauMeasure& tabMeasure)
{
    // Allow the GUI OS to get control occasionally
    keepMacHappy();

    // Choose a row with the smallest ratio, similar to the
    // selection made by primal pivot selection, so that all satisfied
    // constraints remain satisfied after the pivot.

    Quotient    quotient;
    Quotient    best_quotient;

    vector<int> pivotRows = vector<int>();
    int         pivotRowCount = 0;

    // Find the possible primal pivot rows in this column

    // Take a quick look for ZERO ratios
    for ( int row=1; row <= rows; row++ )
    {
        if ( IsZero(mat[row][0]) && NonZero(mat[row][col]) )
        {
            pivotRows.push_back(row);
            pivotRowCount++;
        }
    }

    // If no ZERO ratios, we must compute the other
    // ratios to find the minimum non-negative ratio.
    if (pivotRowCount == 0)
    {
#ifdef USE_ONLY_MAXIMAL_PIVOTS
        if (IsZero(mat[0][col]))
#endif
        for ( int row=1; row <= rows; row++ )
```

```
    {
        if ( sign(mat[row][col]) > 0)
        {
            // Found a possible pivot row,
            // check the quotient for the primal pivot
            quotient = Quotient(mat[row][0],mat[row][col]);

            if ((pivotRowCount == 0) || (quotient < best_quotient))
            {
                best_quotient = quotient;
                pivotRows = vector<int>();
                pivotRows.push_back(row);
                pivotRowCount = 1;
            }
            else if (quotient == best_quotient)
            {
                pivotRows.push_back(row);
                pivotRowCount++;
            }
        }
    }
}

// In either case, we now have a list of the possible
// pivot elements.  There might not be any.
if (pivotRowCount == 0) return -1;

// How many pivots should we try in this column?
// We want to find a good pivot when one exists, and
// there usually are not very many eligible pivot rows.
int pivotSampleSize = pivotRowCount;

// For each of those pivots, how many cuts should we examine?
// We want a good probability of finding a stronger cut when
// one exists, without spending a lot of effort to do so.
int cutSampleSize = CUT_SAMPLE_SIZE;

// Search the possible primal pivots in this column,
// to find one that gives maximum cut measure.
int         best_row = -1;
shuffle(pivotRows.begin(), pivotRows.end());
for (int i = 0; i < pivotSampleSize; i++ )
{
    int row = pivotRows[i];

    // Measure a sample of the Gomory cuts that
    // would be generated after taking this pivot.
    TableauMeasure pivotTabMeasure =
            measureCutsAfterPivot( row, col, cutSampleSize);

    // If the pivot gives an integer solution,
```

```
        // or a refutation, we're done!
        if ( pivotTabMeasure.best.isIntegral() ||
             pivotTabMeasure.best.isRefutation() )
        {
            tabMeasure = pivotTabMeasure;
            best_row = row;
            break;
        }

        // Keep the pivot that gives the best cut,
        // the least worst cut,
        // or the smallest denominator.
        if (pivotTabMeasure > tabMeasure)
        {
            tabMeasure = pivotTabMeasure;
            best_row = row;
//cout << "col=" << col << " row=" << best_row << " " << tabMeasure << endl;
        }
    }

//if (best_row != -1)
//cout << "choosePrimalCutImprovingPivotRow: col=" << col << " row=" << best_row
//     << " tabMeasure =" << tabMeasure
//     << endl;

    return best_row;
}

TableauMeasure
IntegerSimplex::choosePrimalCutImprovingPivot(int &pivot_row, int &pivot_col,
    TableauMeasure initTableauMeasure)
{
    // Preset no cut-improving pivots exist
    TableauMeasure  tabMeasure = initTableauMeasure;
    pivot_col = -1;
    pivot_row = -1;
    status = OPTIMAL;

    int         row;
    int         col;

//cout << "Look for cut-improving pivot.  Initial denominator is " <<
// denominator << " and initial  measure is " << tabMeasure << endl;

    // Search the columns in a random order
    vector<int> columns = vector<int>(cols);
    for (col = 1; col <= cols; col++) columns[col-1] = col;
    shuffle(columns.begin(), columns.end());
    for (int j=0; j<cols; j++ )
    {
        col = columns[j];
```

```
        // find a cut-improving pivot in this column
        TableauMeasure pivotTabMeasure = tabMeasure;
        row = choosePrimalCutImprovingPivotRow(col, pivotTabMeasure);
        if  (row == -1) continue;

        // If a cut gives an integer solution,
        // or a refutation, we're done
        if ( pivotTabMeasure.best.isIntegral() ||
             pivotTabMeasure.best.isRefutation() )
        {
            pivot_row = row;
            pivot_col = col;
            status = CONTINUE; // One more pivot
            break;
        }

        // There are several approaches that might be reasonable
        // to select from among several cut-improving pivots.
        // The algorithm to decide between several cut-improving
        // pivots is encapsulated in class TableauMeasure.
        if (pivotTabMeasure > tabMeasure)
        {
            // Found an improvement
            pivot_row = row;
            pivot_col = col;
            tabMeasure = pivotTabMeasure;
            status = CONTINUE;
        }
    }
    return tabMeasure;
}

// A method that can be applied to an OPTIMAL (primal
// and dual feasible) tableau to improve the measure
// of the available Gomory cuts.

TableauMeasure
IntegerSimplex::primalCutImproving()
{
    int     row,col;
    int     samples = 0;

    // Measure the best cut in the current tableau.
    TableauMeasure  tabMeasure = measureCuts();
    if (status != OPTIMAL) return tabMeasure;

    while(true)
    {
        // If no feasible solution, we're done.
        if (tabMeasure.best.isRefutation()) break;
```

```
        // If no cuts exist, we're done.
        if ( tabMeasure.best.isIntegral() ) break;

        // Choose a  pivot
        TableauMeasure pivotTabMeasure =
            choosePrimalCutImprovingPivot(row, col, tabMeasure);

        // If no feasible pivot, we're done.
        if (status != CONTINUE) break;

        // If we found a strictly stronger cut, take it
        if  (pivotTabMeasure.best > tabMeasure.best)
        {
            // Take that pivot
            samples = 0;
            pivot(row, col);
cout << "Cut-improving pivot." << "\tCutMeasure=" << pivotTabMeasure
// << " row=" << row << " col=" << col
 << endl << flush;
            tabMeasure = pivotTabMeasure;
            continue;
        }

        // Pivots that give lower denominators (or other
        // tableau improvement, per the TableauMeasure)
        // are also taken.
        if  ((pivotTabMeasure.best == tabMeasure.best)
             &&
             (pivotTabMeasure.denominator < tabMeasure.denominator) )
        {
            // Take that pivot
            samples= 0;
            pivot(row, col);

cout << "Denominator-reducing pivot." << "\tmeasure=" << pivotTabMeasure
// << " row=" << row << " col=" << col
 << endl << flush;
            tabMeasure = pivotTabMeasure;
            continue;
        }

#ifdef CUT_RESAMPLE_COUNT
        // (Optionally) repeat the search for a
        // cut-improving pivot a few times before
        // giving up.  This protects against a too-small
        // sample size in the probabilistic search.
        if (samples < CUT_RESAMPLE_COUNT) // Pick a limit
        {
            samples++;
            continue;
```

```
        }
#endif

        // Else, we have the best cut we're going to find.
        break;
    }
    return tabMeasure;
}

//////////////////////////////////////////////////////
//
// ALL-INTEGER PIVOT -- Per (Gomory 1963, example 3)
//
//////////////////////////////////////////////////////

// Function to pivot the all-integer matrix, with optional
// perturbation objectives  (rowZero and colZero).
void
IntegerSimplex::pivot(int  row, int  col)
{
    int     i,j;

    // Allow the GUI to get control occasionally
    keepMacHappy();

    // Swap the variable names
    swap(basic_vars[row], nonbasic_vars[col]);

    // Compute some common factors
    Integer     element = mat[row][col];
    Integer     sd = denominator;
    int         s = 1;
    if (sign(element) < 0)
    {
        sd = -denominator;
        s = -1;
    }

    // Do the purturbation row, if it is present
    if (rowZero.size() > 0)
    {
        for (j = 0; j <= cols; j++) if (j != col)
        {
            rowZero[j] =
                (multiply_and_check_overflow(rowZero[j],element) -
                 multiply_and_check_overflow(mat[row][j],rowZero[col]))
                / sd;
        }
        if (s > 0) rowZero[col] =  -rowZero[col];
    }
```

```
    // Do the purturbation column, if it is present
    if (colZero.size() > 0)
    {
        for ( i=0; i <= rows; i++) if (i != row)
        {
            colZero[i] =
                (multiply_and_check_overflow(colZero[i],element) -
                 multiply_and_check_overflow(colZero[row],mat[i][col]))
                / sd;
        }
        if (s < 0) colZero[row] =  -colZero[row];
    }

    // Do the nonpivot rows.
    for ( i=0; i <= rows; i++) if (i != row)
    {
        // Do the non-pivot columns in the non-pivot rows
        for (j = 0; j <= cols; j++) if (j != col)
        {
            mat[i][j] =
                (multiply_and_check_overflow(mat[i][j], element) -
                 multiply_and_check_overflow(mat[row][j], mat[i][col]))
                / sd;
        }
    }

    // Do the pivot row
    if (s < 0)
        for (j = 0; j <= cols; j++) if (j != col)
            mat[row][j] = - mat[row][j];

    // Do the pivot column
    if (s > 0)
        for (i = 0; i <= rows; i++) if (i != row)
            mat[i][col] = - mat[i][col];

    // Do the pivot element
    mat[row][col] = sd;

    // Store the new common denominator
    denominator = abs(element);

    // And count the pivot
    pivotCounter++;
}


//////////////////////////////////////////////////////
//
// Methods for reading and printing
//
```

```
//////////////////////////////////////////////////

LPStatus
IntegerSimplex::readPrimalSolution(Integer& objective,
                    vector<Integer>& primalVariables,
                    vector<Integer>& primalSlacks,
                    Integer& den)
{
    primalVariables = vector<Integer>(1+cols, ZERO);
    primalSlacks = vector<Integer>(1+rows, ZERO);

    // Iterate through the primal basis variables
    // reading off the primal solution.
    for (int i=1; i<= rows; i++)
    {
        Variable var = basic_vars[i];

        if ( ! var.isSlack() )
        {
            primalVariables[var.subscript] =  mat[i][0];
        }
        else // if ( var.isSlack() )
        {
            primalSlacks[var.subscript] = mat[i][0];
        }
    }

    // Then re-compute the primal solution
    Integer sum = ZERO;
    for (int j = 1; j <= cols; j++)
    {
        // Accumulate the objective value.
        sum += originalObjective[j] * primalVariables[j];
    }
    objective = sum;

    // Return 1-based vectors of variable values
    primalVariables.erase(&primalVariables[0]);
    primalSlacks.erase(&primalSlacks[0]);

    den = denominator;

    return status;
}

LPStatus
IntegerSimplex::readDualSolution(Integer& objective,
                    vector<Integer>& dualVariables,
                    vector<Integer>& dualSlacks,
                    Integer& den)
{
```

```
    dualVariables = vector<Integer>(1+rows, ZERO);
    dualSlacks    = vector<Integer>(1+cols, ZERO);

    // Iterate through the primal basis variables
    // reading off the primal solution.
    for (int i=1; i<= cols; i++)
    {
        Variable var = nonbasic_vars[i]; // (aka  dual basic vars)
        if (var.isSlack())
        {
            // Primal slack is dual variable
            dualVariables[var.subscript] =  mat[0][i];
        }
        else // if (!var.isSlack())
        {
            // Primal variable is dual slack
            dualSlacks[var.subscript] =  mat[0][i];
        }
    }

    // Then re-compute the primal solution value
    Integer sum = ZERO;
    for (int j = 1; j <= cols; j++)
    {
        // Accumulate the objective value.
        sum += originalObjective[j] * dualSlacks[j];
    }
    objective = sum;

    // Return 1-based vectors of variable values
    dualVariables.erase(&dualVariables[0]);
    dualSlacks.erase(&dualSlacks[0]);

    den = denominator;

    return status;
}

void
IntegerSimplex::printObjective(ostream&os) const
{
    int nonzeros = 0;
    for (int col = 1; col <= cols; col++)
    {
        if (sign(mat[0][col]) != ZERO)
        {
            // Second and subsequent terms must have an operator
            if (nonzeros > 0)
            {
                if (sign(mat[0][col]) >= ZERO)
                    os << " +" ;
```

```
                else
                    os << " "; // The '-' will be printed by the Integer
            }

            // output the coeficient and the variable
            os << mat[0][col] << nonbasic_vars[col];
            nonzeros++;

            // put 10 per line
            if ((nonzeros % 10) == 0) os << endl;
        }
    }
    // If there are no terms,
    if (nonzeros == 0) os << "0";

    if ((nonzeros % 10) != 0) os << endl;
}

// Method to print one row of the tableau, as
// an inequality form.
void
IntegerSimplex::printInequality(ostream&os , int row) const
{
    int nonzeros = 0;
    for (int col = 1; col <= cols; col++)
    {
        if (sign(mat[row][col]) != ZERO)
        {
            // Second and subsequent terms must have an operator
            if (nonzeros > 0)
            {
                if (sign(mat[row][col]) >= ZERO)
                    os << " +" ;
                else
                    os << " "; // The '-' will be printed by the Integer
            }

            // output the coeficient and the variable
            os << mat[row][col] << nonbasic_vars[col];
            nonzeros++;

            // put 10 per line
            if ((nonzeros % 10) == 0) os << endl;
        }
    }

    // If there are no terms,
    if (nonzeros == 0) os << "0";

    // the \le sign and the constant
    os  << " <= " << mat[row][0] << endl ;
```

```
}

// Method to print one row of the tableau, as
// an equality form.  This form prints an equation
// that includes a primal basis variable.
void
IntegerSimplex::printEquality(ostream&os , int row) const
{
    os << denominator << basic_vars[row];

    int nonzeros = 1;
    for (int col = 1; col <= cols; col++)
    {
        if (sign(mat[row][col]) != ZERO)
        {
            if (sign(mat[row][col]) > ZERO)
                os << " +" ;
            else
                os << " ";

            // output the coeficient and the variable
            os << mat[row][col] << nonbasic_vars[col];
            nonzeros++;

            // put 10 per line
            if ((nonzeros % 10) == 0) os << endl;
        }
    }

    // If there are no terms,
    if (nonzeros == 0) os << "0";

    // the \le sign and the constant
    os  << " = " << mat[row][0] << endl ;
}

// print the list of nonbasic variables
void
IntegerSimplex::printNonbasicVariables(ostream&os) const
{
    int columns = 0;
    for (int j = 1; j <= cols; j++)
    {
        os << " " <<  nonbasic_vars[j] ;
        columns++;

        // put 12 per line
        if ((j < cols) && ((columns % 12) == 0))
           os << endl;
    }
    os << endl;
```

```
}

ostream& operator<<(ostream& os, const LPStatus& s)
{
    switch ( s )
    {
        case CONTINUE:
            os << "CONTINUE";
            break;
        case OPTIMAL:
            os << "OPTIMAL";
            break;
        case UNBOUNDED:
            os << "UNBOUNDED";
            break;
        case FEASIBLE:
            os << "FEASIBLE";
            break;
        case INFEASIBLE:
            os << "INFEASIBLE";
            break;
        default:
            os << "(" << (int)s << ")";
            break;
    }
    return os;
}

ostream& operator<<(ostream& os, const Variable& s)
{
    return (os << s.name << s.subscript);
}

// A stream output operator for the entire tableau.
ostream& operator<<(ostream& os, const IntegerSimplex& v)
{
    os << "Maximize:" << endl;
    v.printObjective(os);

    os << "Such that:" << endl;
    for ( int row = 1; row <= v.rows; row++)
    {
        v.printEquality(os, row);
    }
    return os;
}
```

A.7 The Cutting Algorithm

The CuttingAlgorithm class and related minor classes, define the data structures and methods implementing the cutting plane algorithms for solution of SAT problems. This class is procedure-centric, having very little data, and has the primary purpose of providing an environment for implementation of cutting plane algorithms.

A.7.1 CuttingAlgorithm.h

```
#ifndef CUTTINGALGO_H_INCLUDED
#define CUTTINGALGO_H_INCLUDED

#include <set.h>

#include "CnfTerm.h"
#include "Integer.h"
#include "Simplex.h"



class CuttingAlgorithm
{
public:

    CuttingAlgorithm(ostream&  proof)
        : PROOF(proof), term(0), tableau(0), seconds(0) {}
    ~CuttingAlgorithm()
        {if (tableau) delete tableau;}


    bool    operator()(CnfTerm* term);
    void    report(ostream& answerFile);

    void    constructTableau(CnfTerm* term);

protected:

    int     originalRowCount;

    CnfTerm*        term;
    IntegerSimplex* tableau;

    LPStatus    status;

    clock_t     startTicks;     // Starting time, in ticks.
```

```
    long        seconds;        // Running time, in seconds.

    ostream&    PROOF;          // File to write the proof.

    LPStatus solve(IntegerSimplex& tableau);

    // Generate a set of cuts, and try to lift one of them.
    LPStatus applyCuts(IntegerSimplex& tableau);
    void constructCuts(int row, IntegerSimplex& tableau, IntegerSimplex& cuts);

    // Apply sequential integer lifting
    void cutLifting(int row, IntegerSimplex& cuts, IntegerSimplex& tableau);

    // Utility to add a cut to a tableau, suppressing duplicates
    int addCut(vector<Integer>& cut, IntegerSimplex& tableau);
};



#include <list.h>
template<class T>
ostream& operator<<(ostream& os, const list<T>& v)
{
    for (list<T>::const_iterator i = v.begin(); i != v.end(); i++)
    {
            os << *i << "\t";
    }
    return os;
}


#endif /* ifndef CUTTINGALGO_H_INCLUDED */
```

A.7.2 CuttingAlgorithm.cc

```
//
// Copyright (c) 2000 Stephen Lee Hansen
//

#include <assert.h>
#include <list.h>
#include <set.h>
#include <map.h>

#include "Parameters.h"
#include "CuttingAlgorithm.h"

extern void keepMacHappy();
extern unsigned int random();
```

```
bool
CuttingAlgorithm::operator()(CnfTerm* theTerm)
{
    term = theTerm;
    constructTableau(theTerm);
    originalRowCount = tableau->rows;

    startTicks = clock();

    // apply the lifting-enabled cutting plane algorithm
    // to generate and lift cutting planes.
    status = solve(*tableau);

    seconds =  (clock() - startTicks)/CLOCKS_PER_SEC;

    if (status == INFEASIBLE)
    {
        PROOF << "NO FEASIBLE SOLUTION EXISTS." << endl << flush;
    }
    else if (status == FEASIBLE)
    {
        PROOF << "FOUND A FEASIBLE SOLUTION." << endl << flush;
    }
    else if (status == CONTINUE)
    {
        PROOF << "MAXIMUM TIME EXPIRED." << endl << flush;
    }
    return status;
}

void
CuttingAlgorithm::constructTableau(CnfTerm* theTerm)
{
    term = theTerm;
    tableau = new IntegerSimplex(term->variables);
    vector<Integer> objective = vector<Integer>(1+term->variables, ZERO);
    vector<Integer> constraint = vector<Integer>(1 + term->variables,  ZERO);

    // For each Boolean variable in the term, construct an
    // inequality that limits the range of the variable.
    // (Non-negativity is enforced by the algorithm.)

    for (int var = 1; var <= term->variables; var++)
    {
        for (int j = 1; j <= term->variables; j++)
        {
             constraint[j] = ZERO;
        }
        constraint[0] = ONE;
        constraint[var] = ONE;
```

```
    tableau->addConstraint(constraint);
}

// For each clause in the term, construct a linear
// constraint from the clause and add it to the tableau

CnfTerm::iterator clause;
for (clause=term->begin(); clause!=term->end(); ++clause)
{
    // The linear inequality must be a "\le" form.
    // (\vee L_i) === (\sum -L_i \le -1 )
    // The RHS constant is stored in constraint[0]

    // Form the inequality corresponding to the empty clause
    constraint[0] = MINUSONE;
    for (int j = 1; j <= term->variables; j++)
        constraint[j] = ZERO;

    // Then add the literals to that inequality
    Clause::iterator  lit;
    for (lit = clause->begin(); lit != clause->end(); ++lit)
    {
        int var = lit->variable;

        if ((var < 0) && (constraint[-var] <= ZERO))
        {
            // If previous literals give:
            //    -\sum (others) \le -b
            // subtracting the new negated literal (1 - X_I) gives:
            //    x_i - \sum (others) \le -b + 1

            var = -var;
            constraint[var] += ONE;
            constraint[0] += ONE;
        }
        else if ((var > 0) && (constraint[var] >= ZERO))
        {
            // If previous literals give:
            //    -\sum (others) \le -b
            // subtracting the new literal x_i gives:
            //    -x_i - \sum (others) \le -b

            constraint[var] -= ONE;
        }
    }

    // objective is sum of slack variables
    // (negated sum of constraints)
    objective = objective +  constraint;
```

```
        tableau->addConstraint(constraint);
    }

    tableau->setObjective(objective);
}

void
CuttingAlgorithm::report(ostream& os)
{
    Integer             objective;
    vector<Integer>     primalVariables;
    vector<Integer>     primalSlacks;
    Integer             denominator;

    // Print the results, in DIMACS format

    os << "c" << endl;
    if (status == INFEASIBLE)
    {
        os << "c  The term is NOT satisfiable." << endl;
    }
    else if (status == FEASIBLE)
    {
        os << "c  The term IS satisfiable." << endl;
        os << "c" << endl;

        tableau->readPrimalSolution(objective,
                primalVariables, primalSlacks, denominator);
        for (int var=0; var<primalVariables.size(); var++)
            if (sign(primalVariables[var]) > ZERO)
                primalVariables[var] = ONE;
        os << "c The primal variable vector is: " <<
            primalVariables << endl;
    }
    else if (status == CONTINUE)
    {
        os << "c No Solution -- Time Expired." << endl << flush;
    }
    os << "c" << endl;

    int cuts = tableau->rows - originalRowCount;
    os  << "c " << cuts << " lifted cutting planes were used in "
        << seconds << " seconds." <<endl;
    os << "c" << endl;

    os << "c The DIMACS \"solution\" line:" << endl;
    os << "s cnf ";
    if (status == INFEASIBLE) os << "0";
    else if (status == FEASIBLE) os << "1";
    else if (status == CONTINUE) os << "?";
    os << " " << term->variables << " " << term->clauses << endl;
```

```
    os << "c" << endl;

    os << "c The DIMACS \"timing\" line:" << endl;
    os << "t cnf ";
    if (status == INFEASIBLE) os << "0";
    else if (status == FEASIBLE) os << "1";
    else if (status == CONTINUE) os << "?";
    os << " " << term->variables << " " << term->clauses
       << " " << seconds << " " << cuts << endl;

    os << flush;
}


//////////////////////////////////////////////////////////////////////
//
// CUTTING PLANE ALGORITHMS
//
//////////////////////////////////////////////////////////////////////
//

LPStatus
CuttingAlgorithm::solve(IntegerSimplex& tableau)
{
#ifdef SHOW_INITIAL_TABLEAU
    PROOF << "THE INITIAL SIMPLEX TABLEAU:" << endl;
    PROOF << tableau;
#endif

    while (1)
    {
#ifdef MAX_RUNNING_TIME
        seconds =  (clock() - startTicks)/CLOCKS_PER_SEC;
        if (seconds > MAX_RUNNING_TIME)
        {
            status = CONTINUE;
            break;
        }
#endif

        // Solve the LP to optimal
        status = tableau.primalDualSimplex();

        // Test for termination (unsatisfiable)
        if (status == INFEASIBLE) break;

        // If there are multiple optimal solutions,
        // choose one with a small common denominator.
        tableau.denominatorReduction();

        // Use pivots to increase the measure of the Gomory cuts.
```

```
#ifdef  CUT_IMPROVEMENT_RESTARTS
        // Multi-start local search for cut-improving pivots.
        IntegerSimplex bestTableau = IntegerSimplex(tableau);
        TableauMeasure  bestMeasure = bestTableau.measureCuts();

        for (int i = 1; i <= CUT_IMPROVEMENT_RESTARTS; i++)
        {
            IntegerSimplex aTableau = IntegerSimplex(tableau);
            TableauMeasure aMeasure = aTableau.primalCutImproving();
            if (aMeasure > bestMeasure)
            {
                bestTableau = aTableau;
                bestMeasure = aMeasure;
            }
        }
        tableau = bestTableau;
#else
        // Single local search for cut-improving pivots.
        tableau.primalCutImproving();
#endif


// To allow the reader to verify the proof, we must identify
// the basic solution from which the cuts are derived.  The
// most concise way to do this is to display the vector of
// nonbasic variables.  This makes it possible for the
// proof verifier to verify that each alleged cut is actually
// a valid cut.

#ifdef SHOW_EVERY_TABLEAU
        PROOF << "THE SIMPLEX TABLEAU:" << endl;
        PROOF << tableau;
#endif
#ifdef SHOW_NONBASIC_VARIABLES
        PROOF << "THE NONBASIC VARIABLES:" << endl;
        tableau.printNonbasicVariables(PROOF);
#endif

        // Look for fractions in the solution
        if (tableau.isFractionalSolution())
        {
            // Generate and lift one or more cutting planes
            status = applyCuts(tableau);
        }
        else // No fractions
        {
            status = FEASIBLE;
            break; // satisfied and integer
        }
    }
    return status;
```

```
}



// A function to apply cuts.
LPStatus
CuttingAlgorithm::applyCuts(IntegerSimplex& tableau)
{
// cout << "dualSimplexIteration" << endl;
    int cols = tableau.cols;
    Integer den = tableau.denominator;
    int row;

    //
    // Construct a set of available cuts.
    //

    // We will store the cuts in a tableau
    IntegerSimplex cuts = IntegerSimplex(tableau);

    // Construct Gomory cuts of the given rows
    // (Note that we may construct a cut of the objective,
    //  since the objective is a sum of constraints.)
    int firstCutRow = tableau.rows +1;
    for (row = 1; row <= tableau.rows; row++)
    {
        // If fractional basic variable value, we have a cut.
        if (mod(tableau.mat[row][0], tableau.denominator) != ZERO)
        {
            // Contruct one or more cuts from this row
            constructCuts(row, tableau, cuts);
        }
    }
cout << "count of Gomory cuts = " << cuts.rows - firstCutRow + 1 << endl;

    // Find the maximum of the measures of the cuts
    CutMeasure bestMeasure;
    int firstTime = 1;
    for (row = firstCutRow; row <= cuts.rows; row++)
    {
        CutMeasure measure = CutMeasure(cuts.mat[row]);
        if (firstTime || (measure > bestMeasure))
        {
            firstTime = 0;
            bestMeasure = measure;
        }

//      cout << "Candidate Gomory cut:" << endl;
//      cuts.printInequality(cout, row);
    }
```

```
    // For each candidate cut that has the required measure,
    // try to lift that cut to obtain a stronger cut.

    for (row = firstCutRow; row <= cuts.rows; row++)
    {
        // Choose only the strong cuts
        CutMeasure measure = CutMeasure(cuts.mat[row]);
        if (measure < bestMeasure) continue;

        PROOF << "THE CANDIDATE CUT: "<< endl;
        cuts.printInequality(PROOF, row);

        cout << "The candidate cut has measure: " << measure << endl;

#ifdef AVOID_USING_LARGE_CUTS
        // Large cuts do not reduce the volume of the polytope by
        // much at all, so they contribute little to the proof.
        // Also, we must carry every cut through all of the
        // succeeding pivots and searches, so there is a
        // computational cost.
        int sqrc = sqrt(cuts.cols);
        if ((measure.count <= sqrc) ||
            (mod(random(),cols) < (cols/(1+measure.count-sqrc))))
#endif
        {
            // NEGATIVE LIFTING
            // (Tests forcing a variable >= 1.)
            cutLifting(row, cuts, tableau);

            PROOF << "THE LIFTED CUT: " << endl ;
            cuts.printInequality(PROOF, row);

            measure = CutMeasure(cuts.mat[row]);
            cout << "The lifted cut has measure: " << measure << endl;

            // Add the new cut to the tableau, and add a line to the
            // proof indicating that the cut has been used.
            int r = addCut(cuts.mat[row], tableau);
            PROOF << " is added to the tableau with slack variable: "
                 << tableau.basic_vars[r] << endl;
        }

        // If the new cut is unsatisfiable, we have enough cuts
        if (measure.isRefutation()) break;

        // Modify the objective function for next time.
        tableau.mat[0] = tableau.mat[0] +  cuts.mat[row];
//      cout << " is added to the objective row." << endl;

        // Add only one strong cut per iteration
        break;
```

```
    }
    return CONTINUE;
}


void
CuttingAlgorithm::constructCuts(int row,
                                IntegerSimplex& tableau,
                                IntegerSimplex& cuts)
{
// cout << "constructCut" << endl;
// cout << " row = " << row << endl
// << " tableau.mat[row] =" << tableau.mat[row] << endl ;

    Integer     den = tableau.denominator;
    int         cols = tableau.cols;
    int         col;

    vector<Integer> vcut = vector<Integer>(1+cols, ZERO);

    // Construct Gomory cut of the selected row (with \lambda == 1)
    vector<Integer> cut = vector<Integer>(1+cols,  ZERO);
    for (col = 0; col <= cols ; col++)
        cut[col] = - mod(tableau.mat[row][col], den);

    // Add the plain Gomory cut to the tableau
    int cutRow = cuts.addConstraint(cut);
    if (cutRow)
    {
//#define SHOW_CUT_DERIVATIONS
#ifdef SHOW_CUT_DERIVATIONS
        PROOF << "The equality: " << endl;
        cuts.printEquality(PROOF, row);
        PROOF << " with denominator " << tableau.denominator;
        PROOF << " generates the candidate Gomory cut:" << endl;
        cuts.printInequality(PROOF, cuts.rows);
#endif

    }
}


void
CuttingAlgorithm::cutLifting(int row,
        IntegerSimplex& cuts, IntegerSimplex& tableau)
{
    int         col;
    LPStatus    status;
    int         cols = cuts.cols;
    Integer     den = cuts.denominator;
    int         liftAttempts = 0;
```

```
int         liftsFound = 0;

// Lift first the variables that have the largest negative
// coeficients in the cut row.  Within one coeficient-value,
// lift first the variables that give the greatest total infeasibility.
multimap<Integer,int> liftCols;
for (col = 1; col <= cols; col++)
{
    if (sign(cuts.mat[row][col]) < ZERO)
    {
        // Measure the infeasibility that would result from
        // forcing the variable in column col to one.
        Integer infeasibility = ZERO;
        for (int i = 1; i <= cuts.rows; i++)
        {
            Integer inf = cuts.mat[i][col] - cuts.mat[i][0];
            if (inf > ZERO) infeasibility += inf;
        }
        // If any, insert that variable into liftCols with
        // a weight determined by the coeficient and the
        // infeasibility.
        if (infeasibility > ZERO)
        {
            Integer weight = ( cuts.mat[row][col] * 1000000000)
                           + infeasibility;
            liftCols.insert(pair<const Integer,int>(weight , col));
        }
    }
}

// Attempt the lifts in the order determined by the weighting.
multimap<Integer,int>::iterator iter;
for (iter = liftCols.begin(); iter != liftCols.end(); iter++)
{
    col = iter->second;

    // Limit the amount of lifting that will be attempted.
    int noLifts = liftAttempts - liftsFound;
    if (noLifts * noLifts  > cuts.cols) break;
    liftAttempts++;

    // Try to lift by adding the inequality $(1 - x_col) \ge 1$
    // to the (\ge) cut inequality.  We know that this is true for
    // for $x_col \eq 0$, so we need to test for $x_col \ge 1$.
    //
    // Form an LP to test the lift of the cut.   We maximize
    // the variable side of the cut, subject to $x_col \ge 1$.
    // If the max is less than the rhs of the cut, then the
    // proposed cut is valid.

        // Initialize a tableau.
```

```
            IntegerSimplex t = IntegerSimplex(cuts);

            // Set the objective to minimize the cut.
            // (Note that  objective[0] == -cuts.mat[row][0],
            // so we may compare the max to ZERO, later.)
            for (int j = 0; j <= cols; j++)
                t.mat[0][j] = - cuts.mat[row][j];

            // Add the constraint:  $x_col \ge 1$
            vector<Integer> v = vector<Integer>(1+cols, ZERO);
            v[0] = v[col] = -den;
            t.addConstraint(v);

            // And force x_col into the basis.
            t.pivot(t.rows, col);

        // Solve for the maximum feasible solution
        status = t.primalDualCutLifting();

        // If the LP is infeasible, we can fix the variable
        // $(x_col \not\ge 1  \longrightarrow  x_col \le 0)$
        if (status == INFEASIBLE)
        {
#define BASISPREVENTION
#ifdef BASISPREVENTION
            // Prevent this variable from ever entering the basis.
            int i;
            for (i = 0; i <= cuts.rows; i++)
                cuts.mat[i][col] = ZERO;
            for (i = 0; i <= tableau.rows; i++)
                tableau.mat[i][col] = ZERO;
#else
            // Form a constraint to fix the variable to ZERO.
            vector<Integer> fix = vector<Integer>(1+cols, ZERO);
            fix[col] = den;
            addCut(fix, cuts);
            addCut(fix, tableau);
#endif
    PROOF << "FOUND A FIXED VARIABLE: "
         << tableau.nonbasic_vars[col] << " = 0" << endl ;
                liftsFound++;
        }
        else if (sign(t.mat[0][0]) < ZERO) // !
        {
            // LP is feasible, but we have a lift.
    PROOF << "FOUND A LIFT.  The term: "
        << cuts.mat[row][col] << tableau.nonbasic_vars[col]
        << " is lifted to: "
        << cuts.mat[row][col]+den << tableau.nonbasic_vars[col]
        <<  endl << flush;
```

```
            cuts.mat[row][col] += den;
            liftsFound++;
        }
    }
}


// Procedure to add a cut to a tableau, rejecting duplicates
int
CuttingAlgorithm::addCut(vector<Integer>& cut,
                         IntegerSimplex& tableau)
{
    int row;

    // Do not add duplicate cuts
    for (row = 1; row <= tableau.rows; row++)
        if (cut == tableau.mat[row])
            return row;

    // Add the cut to the tableau.
    tableau.addConstraint(cut);
    row = tableau.rows;

    return row;
}
```

Appendix B

Msat Generator Program Code

The pseudo-random SAT problem generator for the $\mathcal{M}_{m,n}^{k}$ model of random SAT problem is implemented in a "C" program named `msat`. The source file of the msat program is `msat.c`.

B.1 msat.c

```
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>

#include "vector.h"
#include "algo.h"

#include "stream.h"

extern "C"
{
long random();
int srandom( unsigned seed);
}



/*ARGSUSED*/
main(int argc,
     char* argv[],
     char* /* envp[] is not used */ )
```

```
{

   srandom(time(0));

   int variables;
   cerr << "variables?";
   cin >> variables;

   float ratio;
   cerr << "ratio?";
   cin >> ratio;

   int clauses = variables * ratio;

   vector<int> vars = vector<int>(variables);
   for (int variable = 0; variable < variables; variable++)
      vars[variable] = variable+1;

   cout << "p cnf " << variables << " " << clauses << endl;

   for (int clause = 1; clause <= clauses; clause++)
   {
      random_shuffle(vars.begin(), vars.end());

      for (int literal = 0; literal < 3; literal++)
      {
         if (random() & 1)
         {
            cout << -vars[literal] << " " ;
         }
         else
         {
            cout << vars[literal] << " ";
         }
      }
      cout << 0 << endl;
   }
}
```

Appendix C

A Complete Unsatisfiability Proof

For unsatisfiable input problems, the `CutSat` program produces a proof of unsatisfiability. The reader may be interested to see a small example. The example presented here has only 30 variables, and only 127 clauses.

C.1 Example Problem

This section displays the input file that was used to generate the proof given in section C.2, below. The input file was generated using the `RSAT` pseudo-random problem generator (Asahiro et al., 1996). One alteration was made to accommodate the paper width. The `RSAT` program does not insert carriage control characters into the display of the CNF predicate following the label "`### 3SAT CNF Predicate ###`". Newline characters were inserted manually, to enable display of the resulting very long line.

The `CutSat` program ignores extraneous input lines until it finds a line having the character "`p`" in column one. This behavior serves to eliminate the need to edit generated problem files, which commonly include extra lines preceding the actual DIMACS format SAT problem. The `RSAT` problem generator, and the other problem

file generators provided by Asahiro et al. (1996), each produce a considerable number of such extra lines. In this example, the actual DIMACS format SAT problem begins with the line that reads "`p cnf 30 127`".

```
*****  3SAT-Generator for Random CNF Predicates *****

How many variables ? 1 ... 500   30
Literal Distribution
   (0) Specify Literal Distribution
   (1) Flat Distribution
   (2) Normal Distribution
   (3) Input Ratio(Clause/Variable)
Select[ 0 - 3 ]:   3
Input the ratio(Clause/Variable)  4.25
Clauses except Random Ones: 0
Clause Remove: 0

@@@@@@@@@@@  OUTPUT  @@@@@@@@@@@

###   Literal Distribution  ###

N[ 1]= 6    N[ 1']= 6
N[ 2]= 6    N[ 2']= 6
N[ 3]= 7    N[ 3']= 6
N[ 4]= 6    N[ 4']= 6
N[ 5]= 6    N[ 5']= 6
N[ 6]= 7    N[ 6']= 7
N[ 7]= 6    N[ 7']= 6
N[ 8]= 7    N[ 8']= 6
N[ 9]= 6    N[ 9']= 7
N[10]= 7    N[10']= 7
N[11]= 6    N[11']= 6
N[12]= 6    N[12']= 6
N[13]= 6    N[13']= 6
N[14]= 6    N[14']= 6
N[15]= 7    N[15']= 6
N[16]= 8    N[16']= 6
N[17]= 6    N[17']= 7
N[18]= 7    N[18']= 6
N[19]= 6    N[19']= 6
N[20]= 6    N[20']= 6
N[21]= 6    N[21']= 6
N[22]= 6    N[22']= 8
N[23]= 6    N[23']= 7
N[24]= 6    N[24']= 7
N[25]= 7    N[25']= 6
N[26]= 6    N[26']= 8
N[27]= 6    N[27']= 6
```

```
N[28]= 6    N[28']= 7
N[29]= 6    N[29']= 6
N[30]= 6    N[30']= 7

Total of literals      :: 381
Number of clauses      :: 127



###  3SAT CNF Predicate  ###

(13'27'28')(10 18 22')(11'18'26')(5'10 22 )(16 16 17')(3'17'29 )(1'8 22 )
(10'19 26')(2'5 26 )(7 11 14')(2 6'10')(18 20 23')(20'21'27')(2'6'14 )
(12'19 30 )(5'21'25 )(1'19'22 )(6 9'16')(9'26'28 )(9'22'23')(9'17 25')
(13'17'21 )(2 18 29')(4'15'27 )(3 8'21')(10 20 30 )(15 18'25 )(11'27'28')
(3'18 27 )(19'25'30')(3'20 29')(1'2'30')(8'13'25 )(4'9 29 )(15 22'24 )
(1'16 16')(4 23 30')(10'15'19 )(6 7'23 )(5'10'28 )(3'18 29')(13'23'26 )
(2'6 24')(1'10 24')(10 15 23')(11'17'26')(12'23 29')(12'17'24')(3 9'12 )
(12 24'28 )(17'18'23 )(1 13 28 )(10'10'28 )(10 20 26 )(8'19'21 )
(12 16'18 )(6 13 27 )(1 8 24')(6 7'30')(15'19 28')(12 13'19 )(14'19'29 )
(3 10 21 )(15'20'28')(21 28'30')(4'6'9 )(22 26 29 )(15 22 26')(7 10'14')
(24 24'27 )(17 26 27 )(6'17 22')(3 14'21 )(12'18'25 )(11 14 25 )(5 8'30 )
(4 23'28')(16 16'27')(1 8 19 )(2'13 28')(3 7'17 )(6'11 14 )(7 18'30')
(5'23 27 )(15 16'22')(7'8'21')(2 16 22 )(4'21'29')(23 23'25')(4'12'29 )
(6 7 14 )(5'5'18')(7'16 25 )(5 26'26')(7'8'20 )(15'16 17')(11'13 30 )
(5 11'20')(1 1 24 )(4 9 24')(17 18 30')(11 13 20')(2'15 24 )(7 7 25')
(3'25'28 )(2 5 9 )(6'14 27')(9 13 24 )(4 12 21')(1 11'19')(9'14'22')
(11 26 29 )(2 11 14 )(4 8 30 )(5 8 22')(9 15'25 )(13'19'22')(3 21 24 )
(20 27'30 )(3'6 16 )(2 8 15 )(3 23'29')(6'12 25')(4 4'14')(12'17 20')
(1'16'26')(8 9'20')

c
c NOTE: Random
c
p cnf 30 127
-13 -27 -28 0
10 18 -22 0
-11 -18 -26 0
-5 10 22 0
16 16 -17 0
-3 -17 29 0
-1 8 22 0
-10 19 -26 0
-2 5 26 0
7 11 -14 0
2 -6 -10 0
18 20 -23 0
-20 -21 -27 0
-2 -6 14 0
-12 19 30 0
-5 -21 25 0
```

```
-1 -19 22 0
6 -9 -16 0
-9 -26 28 0
-9 -22 -23 0
-9 17 -25 0
-13 -17 21 0
2 18 -29 0
-4 -15 27 0
3 -8 -21 0
10 20 30 0
15 -18 25 0
-11 -27 -28 0
-3 18 27 0
-19 -25 -30 0
-3 20 -29 0
-1 -2 -30 0
-8 -13 25 0
-4 9 29 0
15 -22 24 0
-1 16 -16 0
4 23 -30 0
-10 -15 19 0
6 -7 23 0
-5 -10 28 0
-3 18 -29 0
-13 -23 26 0
-2 6 -24 0
-1 10 -24 0
10 15 -23 0
-11 -17 -26 0
-12 23 -29 0
-12 -17 -24 0
3 -9 12 0
12 -24 28 0
-17 -18 23 0
1 13 28 0
-10 -10 28 0
10 20 26 0
-8 -19 21 0
12 -16 18 0
6 13 27 0
1 8 -24 0
6 -7 -30 0
-15 19 -28 0
12 -13 19 0
-14 -19 29 0
3 10 21 0
-15 -20 -28 0
21 -28 -30 0
-4 -6 9 0
22 26 29 0
```

```
15 22 -26 0
7 -10 -14 0
24 -24 27 0
17 26 27 0
-6 17 -22 0
3 -14 21 0
-12 -18 25 0
11 14 25 0
5 -8 30 0
4 -23 -28 0
16 -16 -27 0
1 8 19 0
-2 13 -28 0
3 -7 17 0
-6 11 14 0
7 -18 -30 0
-5 23 27 0
15 -16 -22 0
-7 -8 -21 0
2 16 22 0
-4 -21 -29 0
23 -23 -25 0
-4 -12 29 0
6 7 14 0
-5 -5 -18 0
-7 16 25 0
5 -26 -26 0
-7 -8 20 0
-15 16 -17 0
-11 13 30 0
5 -11 -20 0
1 1 24 0
4 9 -24 0
17 18 -30 0
11 13 -20 0
-2 15 24 0
7 7 -25 0
-3 -25 28 0
2 5 9 0
-6 14 -27 0
9 13 24 0
4 12 -21 0
1 -11 -19 0
-9 -14 -22 0
11 26 29 0
2 11 14 0
4 8 30 0
5 8 -22 0
9 -15 25 0
-13 -19 -22 0
3 21 24 0
```

```
20 -27 30 0
-3 6 16 0
2 8 15 0
3 -23 -29 0
-6 12 -25 0
4 -4 -14 0
-12 17 -20 0
-1 -16 -26 0
8 -9 -20 0
```

C.2 Proof of the Example Problem

When the `CutSat` program is run, one of the output files created is a proof file. If the problem is unsatisfiable, the contents of the resulting proof file is a proof of unsatisfiable. A proof file that was produced from the input given in section C.1, above, is reproduced in this section. Interpretation and checking of the proof is discussed in section 5.2.

For this run, the symbol `SHOW_INITIAL_TABLEAU` was defined, so the initial tableau is displayed. This provides an example of a tableau. The reader may wish to compare this tableau with the problem representation in the input file. In addition, the symbol `SHOW_NONBASIC_VARIABLES` was defined, so the vector of nonbasic variable names is displayed at each iteration. This allows the reader to observe the much more concise proof format that results. The symbols `SHOW_EVERY_TABLEAU` and `SHOW_CUT_DERIVATIONS` were not defined for this run.

```
THE INITIAL SIMPLEX TABLEAU:
Maximize:
1x1 -1x3 -1x5 +1x7 -1x8 +1x9 -1x10 -1x15 -1x16 +1x17
 -1x18 +2x22 +1x23 +1x24 -1x25 +1x26 +1x28 +1x30
Such that:
1s1 +1x1 = 1
1s2 +1x2 = 1
1s3 +1x3 = 1
1s4 +1x4 = 1
1s5 +1x5 = 1
1s6 +1x6 = 1
```

```
1s7 +1x7 = 1
1s8 +1x8 = 1
1s9 +1x9 = 1
1s10 +1x10 = 1
1s11 +1x11 = 1
1s12 +1x12 = 1
1s13 +1x13 = 1
1s14 +1x14 = 1
1s15 +1x15 = 1
1s16 +1x16 = 1
1s17 +1x17 = 1
1s18 +1x18 = 1
1s19 +1x19 = 1
1s20 +1x20 = 1
1s21 +1x21 = 1
1s22 +1x22 = 1
1s23 +1x23 = 1
1s24 +1x24 = 1
1s25 +1x25 = 1
1s26 +1x26 = 1
1s27 +1x27 = 1
1s28 +1x28 = 1
1s29 +1x29 = 1
1s30 +1x30 = 1
1s31 +1x13 +1x27 +1x28 = 2
1s32 -1x10 -1x18 +1x22 = 0
1s33 +1x11 +1x18 +1x26 = 2
1s34 +1x5 -1x10 -1x22 = 0
1s35 -1x16 +1x17 = 0
1s36 +1x3 +1x17 -1x29 = 1
1s37 +1x1 -1x8 -1x22 = 0
1s38 +1x10 -1x19 +1x26 = 1
1s39 +1x2 -1x5 -1x26 = 0
1s40 -1x7 -1x11 +1x14 = 0
1s41 -1x2 +1x6 +1x10 = 1
1s42 -1x18 -1x20 +1x23 = 0
1s43 +1x20 +1x21 +1x27 = 2
1s44 +1x2 +1x6 -1x14 = 1
1s45 +1x12 -1x19 -1x30 = 0
1s46 +1x5 +1x21 -1x25 = 1
1s47 +1x1 +1x19 -1x22 = 1
1s48 -1x6 +1x9 +1x16 = 1
1s49 +1x9 +1x26 -1x28 = 1
1s50 +1x9 +1x22 +1x23 = 2
1s51 +1x9 -1x17 +1x25 = 1
1s52 +1x13 +1x17 -1x21 = 1
1s53 -1x2 -1x18 +1x29 = 0
1s54 +1x4 +1x15 -1x27 = 1
1s55 -1x3 +1x8 +1x21 = 1
1s56 -1x10 -1x20 -1x30 = -1
1s57 -1x15 +1x18 -1x25 = 0
```

```
1s58 +1x11 +1x27 +1x28 = 2
1s59 +1x3 -1x18 -1x27 = 0
1s60 +1x19 +1x25 +1x30 = 2
1s61 +1x3 -1x20 +1x29 = 1
1s62 +1x1 +1x2 +1x30 = 2
1s63 +1x8 +1x13 -1x25 = 1
1s64 +1x4 -1x9 -1x29 = 0
1s65 -1x15 +1x22 -1x24 = 0
1s66 -1x4 -1x23 +1x30 = 0
1s67 +1x10 +1x15 -1x19 = 1
1s68 -1x6 +1x7 -1x23 = 0
1s69 +1x5 +1x10 -1x28 = 1
1s70 +1x3 -1x18 +1x29 = 1
1s71 +1x13 +1x23 -1x26 = 1
1s72 +1x2 -1x6 +1x24 = 1
1s73 +1x1 -1x10 +1x24 = 1
1s74 -1x10 -1x15 +1x23 = 0
1s75 +1x11 +1x17 +1x26 = 2
1s76 +1x12 -1x23 +1x29 = 1
1s77 +1x12 +1x17 +1x24 = 2
1s78 -1x3 +1x9 -1x12 = 0
1s79 -1x12 +1x24 -1x28 = 0
1s80 +1x17 +1x18 -1x23 = 1
1s81 -1x1 -1x13 -1x28 = -1
1s82 +1x10 -1x28 = 0
1s83 -1x10 -1x20 -1x26 = -1
1s84 +1x8 +1x19 -1x21 = 1
1s85 -1x12 +1x16 -1x18 = 0
1s86 -1x6 -1x13 -1x27 = -1
1s87 -1x1 -1x8 +1x24 = 0
1s88 -1x6 +1x7 +1x30 = 1
1s89 +1x15 -1x19 +1x28 = 1
1s90 -1x12 +1x13 -1x19 = 0
1s91 +1x14 +1x19 -1x29 = 1
1s92 -1x3 -1x10 -1x21 = -1
1s93 +1x15 +1x20 +1x28 = 2
1s94 -1x21 +1x28 +1x30 = 1
1s95 +1x4 +1x6 -1x9 = 1
1s96 -1x22 -1x26 -1x29 = -1
1s97 -1x15 -1x22 +1x26 = 0
1s98 -1x7 +1x10 +1x14 = 1
1s99 -1x27 = 0
1s100 -1x17 -1x26 -1x27 = -1
1s101 +1x6 -1x17 +1x22 = 1
1s102 -1x3 +1x14 -1x21 = 0
1s103 +1x12 +1x18 -1x25 = 1
1s104 -1x11 -1x14 -1x25 = -1
1s105 -1x5 +1x8 -1x30 = 0
1s106 -1x4 +1x23 +1x28 = 1
1s107 -1x1 -1x8 -1x19 = -1
1s108 +1x2 -1x13 +1x28 = 1
```

```
1s109 -1x3 +1x7 -1x17 = 0
1s110 +1x6 -1x11 -1x14 = 0
1s111 -1x7 +1x18 +1x30 = 1
1s112 +1x5 -1x23 -1x27 = 0
1s113 -1x15 +1x16 +1x22 = 1
1s114 +1x7 +1x8 +1x21 = 2
1s115 -1x2 -1x16 -1x22 = -1
1s116 +1x4 +1x21 +1x29 = 2
1s117 +1x4 +1x12 -1x29 = 1
1s118 -1x6 -1x7 -1x14 = -1
1s119 +1x5 +1x18 = 1
1s120 +1x7 -1x16 -1x25 = 0
1s121 -1x5 +1x26 = 0
1s122 +1x7 +1x8 -1x20 = 1
1s123 +1x15 -1x16 +1x17 = 1
1s124 +1x11 -1x13 -1x30 = 0
1s125 -1x5 +1x11 +1x20 = 1
1s126 -1x1 -1x24 = -1
1s127 -1x4 -1x9 +1x24 = 0
1s128 -1x17 -1x18 +1x30 = 0
1s129 -1x11 -1x13 +1x20 = 0
1s130 +1x2 -1x15 -1x24 = 0
1s131 -1x7 +1x25 = 0
1s132 +1x3 +1x25 -1x28 = 1
1s133 -1x2 -1x5 -1x9 = -1
1s134 +1x6 -1x14 +1x27 = 1
1s135 -1x9 -1x13 -1x24 = -1
1s136 -1x4 -1x12 +1x21 = 0
1s137 -1x1 +1x11 +1x19 = 1
1s138 +1x9 +1x14 +1x22 = 2
1s139 -1x11 -1x26 -1x29 = -1
1s140 -1x2 -1x11 -1x14 = -1
1s141 -1x4 -1x8 -1x30 = -1
1s142 -1x5 -1x8 +1x22 = 0
1s143 -1x9 +1x15 -1x25 = 0
1s144 +1x13 +1x19 +1x22 = 2
1s145 -1x3 -1x21 -1x24 = -1
1s146 -1x20 +1x27 -1x30 = 0
1s147 +1x3 -1x6 -1x16 = 0
1s148 -1x2 -1x8 -1x15 = -1
1s149 -1x3 +1x23 +1x29 = 1
1s150 +1x6 -1x12 +1x25 = 1
1s151 +1x12 -1x17 +1x20 = 1
1s152 +1x1 +1x16 +1x26 = 2
1s153 -1x8 +1x9 +1x20 = 1
THE NONBASIC VARIABLES:
 s127 s119 s61 s105 s122 s132 s131 s3 s93 s95 s108 s146
 s110 s143 s68 x11 x17 s85 x22 s82 s2 s31 s89 s126
 s59 s70 s84 s47 s76 s96
THE CANDIDATE CUT:
-4s119 -1s61 -4s105 -3s122 -4s132 -3s131 -5s3 -2s108 -3s146 -5s2
```

```
 -2s31 -5s59 -6s70 <= -6
THE NONBASIC VARIABLES:
 s87 s119 s43 s105 s92 s109 s131 s121 s93 s95 s133 s104
 s110 s143 s40 s63 s102 s85 x22 s82 s129 s151 s20 s126
 x23 x3 s127 s47 s76 s96
THE CANDIDATE CUT:
-2s131 -2s104 -2s40 <= -2
THE LIFTED CUT:
-2s131 -2s104 -2s40 <= -2
 is added to the tableau with slack variable: s154
THE NONBASIC VARIABLES:
 s37 s119 s131 s124 s92 s109 s137 s121 s148 x3 s133 s98
 s41 s143 s112 s11 s125 s85 x2 s82 s63 s16 s151 s126
 s66 s34 s53 s118 s76 s96
THE CANDIDATE CUT:
-1s98 -1s41 -1x2 -1s118 <= -1
FOUND A LIFT.  The term: -1x2 is lifted to: 1x2
THE LIFTED CUT:
-1s98 -1s41 +1x2 -1s118 <= -1
 is added to the tableau with slack variable: s155
THE NONBASIC VARIABLES:
 s37 s119 s131 s80 s92 s109 s137 s93 s102 s55 s133 s90
 s13 s143 s112 s52 s125 s121 x2 s82 s15 s16 s141 s126
 s45 s34 x23 s48 x30 s96
THE CANDIDATE CUT:
-1s90 -1s13 -1s45 -1x30 <= -1
FOUND A LIFT.  The term: -1s90 is lifted to: 1s90
THE LIFTED CUT:
1s90 -1s13 -1s45 -1x30 <= -1
 is added to the tableau with slack variable: s156
THE NONBASIC VARIABLES:
 s37 s119 s55 s80 s92 s109 s137 s43 s102 s11 s133 s90
 s58 s143 x3 s83 s125 s121 s88 s82 s15 s16 s78 s126
 s66 s34 s85 s60 s54 s96
THE CANDIDATE CUT:
-2s92 -2s43 -3s11 -3s58 -3x3 -1s83 -4s125 -1s121 -3s82 <= -4
FOUND A LIFT.  The term: -4s125 is lifted to: 1s125
THE LIFTED CUT:
-2s92 -2s43 -3s11 -3s58 -3x3 -1s83 +1s125 -1s121 -3s82 <= -4
 is added to the tableau with slack variable: s157
THE NONBASIC VARIABLES:
 s37 s119 s55 s40 s39 s109 s129 s116 s102 s43 s133 s90
 s48 s143 s79 s114 s125 s121 s88 s47 s15 s16 s78 s126
 s66 s34 s82 s14 s54 s96
THE CANDIDATE CUT:
-3s129 -3s90 -3s79 -3s125 -3s47 -3s126 -3s34 -3s82 <= -3
THE NONBASIC VARIABLES:
 s37 s119 s55 s40 s39 s109 s129 s116 s102 s43 s133 s90
 s48 s143 s79 s114 s125 s121 s88 s47 s15 s16 s78 s126
 s66 s34 s82 s14 s54 s96
THE CANDIDATE CUT:
```

```
-3s129 -3s90 -3s79 -3s125 -3s47 -3s126 -3s34 -3s82 <= -3
FOUND A LIFT.  The term: -3s126 is lifted to: 3s126
FOUND A LIFT.  The term: -3s82 is lifted to: 3s82
THE LIFTED CUT:
-3s129 -3s90 -3s79 -3s125 -3s47 +3s126 -3s34 +3s82 <= -3
 is added to the tableau with slack variable: s158
THE NONBASIC VARIABLES:
 s142 s119 s55 x3 s67 s109 s83 s91 s106 s60 s133 x20
 s122 s41 s143 s100 s71 s77 s88 s156 s64 s16 s139 s126
 s66 s155 s127 s62 s54 s96
THE CANDIDATE CUT:
-6x3 -3s109 -3s41 -3s77 -3s88 -6s156 -3s126 -6s155 -3s62 <= -6
THE NONBASIC VARIABLES:
 s60 s119 s142 s136 s42 s109 s83 s91 s144 s87 s133 x20
 x8 s41 s143 s100 x29 s137 s101 s156 s37 s16 s15 s126
 s66 s36 s132 s128 s54 s96
THE CANDIDATE CUT:
-2s87 -2x8 -2s126 <= -2
FOUND A LIFT.  The term: -2s87 is lifted to: 2s87
THE LIFTED CUT:
2s87 -2x8 -2s126 <= -2
 is added to the tableau with slack variable: s159
THE NONBASIC VARIABLES:
 s8 s119 s127 s48 s114 s73 s83 s91 s16 s87 s133 s81
 s159 s54 s131 s100 s121 s137 s101 s93 s151 s89 x29 s85
 s66 s64 s122 s128 s36 s96
THE CANDIDATE CUT:
-1s127 -1s159 -1x29 -1s64 <= -1
FOUND A LIFT.  The term: -1x29 is lifted to: 1x29
FOUND A FIXED VARIABLE: s159 = 0
THE LIFTED CUT:
-1s127 +1x29 -1s64 <= -1
 is added to the tableau with slack variable: s160
THE NONBASIC VARIABLES:
 x1 s119 s160 s48 s114 s36 s83 s139 s111 s87 s137 s81
 s159 s54 s80 s100 s121 s84 s101 s93 s103 s138 s105 x2
 s66 s125 s122 s51 s63 s96
THE CANDIDATE CUT:
-1x1 -3s119 -1s114 -2s111 -4s137 -1s84 -2s105 -1s125 -1s122 <= -4
THE NONBASIC VARIABLES:
 s79 s119 s160 s48 s59 s94 s112 s139 s73 s87 s41 s81
 s159 s54 s128 s100 s121 s89 x19 x23 s49 s138 s34 x2
 s66 s117 s122 s51 s154 s96
THE CANDIDATE CUT:
-2s119 -2s160 -4s112 -3s54 -2s128 -3s100 -1s121 -2s89 -3x19 -3x23
 -2s49 -3s66 <= -4
THE NONBASIC VARIABLES:
 s79 s119 s160 s48 s39 s94 s103 s139 s73 s87 s154 s57
 s159 s134 s61 s129 s121 s89 s83 s88 s71 s115 s149 s27
 s50 s116 s122 s51 s109 s96
THE CANDIDATE CUT:
```

```
-4s73 -4s87 -4s154 -4s61 -4s129 -4s83 -4s71 -4s149 -4s122 <= -4
THE NONBASIC VARIABLES:
 s79 s119 s160 s48 s146 s94 s117 s139 s73 s87 s41 s38
 s159 s134 s36 s138 s121 s89 s124 s88 s120 s115 s129 s27
 s50 s58 s49 s51 s109 s96
THE CANDIDATE CUT:
-1s146 -1s124 -1s129 -1s27 <= -1
FOUND A LIFT.  The term: -1s129 is lifted to: 1s129
THE LIFTED CUT:
-1s146 -1s124 +1s129 -1s27 <= -1
 is added to the tableau with slack variable: s161
THE NONBASIC VARIABLES:
 s79 s119 s9 s48 s82 s103 s117 s139 x29 s87 s41 s66
 s159 s134 s36 s138 s121 s113 s114 s88 s107 s125 s77 s153
 s50 s63 s49 s51 s109 s96
THE CANDIDATE CUT:
-2s79 -2s119 -1s9 -1s103 -2s121 -1s77 -1s49 -1s51 <= -2
THE NONBASIC VARIABLES:
 s79 s119 s160 s48 s82 s94 x21 s139 x29 s87 s41 s66
 s159 s134 s36 s138 s121 s113 s78 s88 s107 s125 s77 s153
 s50 s63 s49 s51 s109 s96
THE CANDIDATE CUT:
-1s160 -1x29 -1s36 -1s78 -1s77 <= -1
FOUND A FIXED VARIABLE: s78 = 0
FOUND A LIFT.  The term: -1s36 is lifted to: 1s36
FOUND A FIXED VARIABLE: x29 = 0
FOUND A FIXED VARIABLE: s160 = 0
THE LIFTED CUT:
1s36 -1s77 <= -1
 is added to the tableau with slack variable: s162
THE NONBASIC VARIABLES:
 s79 s93 s160 s48 s155 s94 s106 s139 x29 s87 s41 s58
 s159 s101 s147 s138 s121 s85 s78 s120 s57 s89 s162 s153
 s136 s63 s128 s51 s46 s96
THE CANDIDATE CUT:
-1s48 -1s147 -1s162 <= -1
FOUND A FIXED VARIABLE: s48 = 0
FOUND A FIXED VARIABLE: s147 = 0
FOUND A FIXED VARIABLE: s162 = 0
THE LIFTED CUT:
0 <= -1
 is added to the tableau with slack variable: s163
NO FEASIBLE SOLUTION EXISTS.
c
c  The term is NOT satisfiable.
c
c 10 lifted cutting planes were used in 4 seconds.
c
c The DIMACS "solution" line:
s cnf 0 30 127
c
```

```
c The DIMACS "timing" line:
t cnf 0 30 127 4 10
```

C.3 Example CutSat Session Transcript

When the `CutSat` program is run, it interacts with the user via the "`cin`" and "`cout`" streams. The transcript of the session which produced the proof given in section C.2 is displayed in this section. The output produced to the `cout` stream during a session is not part of the formal proof. For formal purposes, it should be considered as commentary or ignored entirely.

```
Cnf Filename>rsat-30-4.25-1.cnf
Cnf Filename>go
read 127 clauses.
34 dual pivots were used
30 primal pivots were used
measureCuts: -21.1421(324/2170,22) (6 of 127 cuts) den=21
Cut-improving pivot.    CutMeasure=-12.3685(36/195,13) (2 of 25 cuts) den=7
count of Gomory cuts = 6
The candidate cut has measure: -12.3685(36/195,13)
0 dual pivots were used
24 primal pivots were used
Denominator Reducing pivot is:4
measureCuts: -14(4/56,14) (40 of 130 cuts) den=4
Cut-improving pivot.    CutMeasure=-3(4/12,3) (10 of 25 cuts) den=4
count of Gomory cuts = 3
The candidate cut has measure: -3(4/12,3)
14 dual pivots were used
4 dual pivots were used
21 dual pivots were used
The lifted cut has measure: -3(4/12,3)
8 dual pivots were used
15 primal pivots were used
Denominator Reducing pivot is:4
Denominator Reducing pivot is:2
measureCuts: -11(1/11,11) (92 of 92 cuts) den=2
Cut-improving pivot.    CutMeasure=-4(1/4,4) (25 of 25 cuts) den=2
count of Gomory cuts = 1
The candidate cut has measure: -4(1/4,4)
9 dual pivots were used
8 dual pivots were used
9 dual pivots were used
21 dual pivots were used
```

```
The lifted cut has measure: -3(1/3,3)
9 dual pivots were used
8 primal pivots were used
Denominator Reducing pivot is:4
Denominator Reducing pivot is:2
measureCuts: -13(1/13,13) (91 of 91 cuts) den=2
Cut-improving pivot.    CutMeasure=-4(1/4,4) (25 of 25 cuts) den=2
count of Gomory cuts = 1
The candidate cut has measure: -4(1/4,4)
22 dual pivots were used
15 dual pivots were used
12 dual pivots were used
16 dual pivots were used
The lifted cut has measure: -3(1/3,3)
6 dual pivots were used
8 primal pivots were used
Denominator Reducing pivot is:5
measureCuts: -12.2545(16/74,13) (25 of 131 cuts) den=5
Cut-improving pivot.    CutMeasure=-10.3847(16/75,11) (4 of 25 cuts) den=5
Cut-improving pivot.    CutMeasure=-8.39214(16/62,9) (4 of 25 cuts) den=5
count of Gomory cuts = 4
The candidate cut has measure: -8.39214(16/62,9)
17 dual pivots were used
12 dual pivots were used
10 dual pivots were used
17 dual pivots were used
19 dual pivots were used
11 dual pivots were used
17 dual pivots were used
The lifted cut has measure: -7.26178(16/46,8)
10 dual pivots were used
7 primal pivots were used
measureCuts: -8(9/72,8) (23 of 131 cuts) den=6
count of Gomory cuts = 5
The candidate cut has measure: -8(9/72,8)
0 dual pivots were used
0 primal pivots were used
measureCuts: -8(9/72,8) (23 of 131 cuts) den=6
count of Gomory cuts = 5
The candidate cut has measure: -8(9/72,8)
19 dual pivots were used
12 dual pivots were used
3 dual pivots were used
34 dual pivots were used
12 dual pivots were used
17 dual pivots were used
19 dual pivots were used
17 dual pivots were used
The lifted cut has measure: -6(9/54,6)
5 dual pivots were used
27 primal pivots were used
```

```
measureCuts: -21.7185(4096/15943,23) (2 of 154 cuts) den=67
Cut-improving pivot.    CutMeasure=-8.5(144/648,9) (2 of 25 cuts) den=18
Denominator-reducing pivot. measure=-8.5(36/162,9) (6 of 25 cuts) den=9
count of Gomory cuts = 8
The candidate cut has measure: -8.5(36/162,9)
0 dual pivots were used
20 primal pivots were used
Denominator Reducing pivot is:6
measureCuts: -7(9/63,7) (33 of 134 cuts) den=6
Cut-improving pivot.    CutMeasure=-3(9/27,3) (7 of 25 cuts) den=6
Denominator-reducing pivot. measure=-3(4/12,3) (13 of 25 cuts) den=4
count of Gomory cuts = 3
The candidate cut has measure: -3(4/12,3)
29 dual pivots were used
28 dual pivots were used
24 dual pivots were used
The lifted cut has measure: -2(4/8,2)
14 dual pivots were used
14 primal pivots were used
Denominator Reducing pivot is:-4
Denominator Reducing pivot is:2
measureCuts: -8(1/8,8) (91 of 91 cuts) den=2
Cut-improving pivot.    CutMeasure=-4(1/4,4) (25 of 25 cuts) den=2
count of Gomory cuts = 1
The candidate cut has measure: -4(1/4,4)
8 dual pivots were used
5 dual pivots were used
21 dual pivots were used
0 dual pivots were used
The lifted cut has measure: -2(1/2,2)
20 dual pivots were used
0 primal pivots were used
measureCuts: -21.2589(16/126,22) (28 of 141 cuts) den=5
Cut-improving pivot.    CutMeasure=-8.98174(16/39,10) (3 of 25 cuts) den=5
Cut-improving pivot.    CutMeasure=-8.039(16/38,9) (7 of 25 cuts) den=5
count of Gomory cuts = 4
The candidate cut has measure: -8.039(16/38,9)
0 dual pivots were used
19 primal pivots were used
measureCuts: -12.0405(16/55,13) (25 of 129 cuts) den=5
Cut-improving pivot.    CutMeasure=-11.3863(16/82,12) (3 of 25 cuts) den=5
count of Gomory cuts = 4
The candidate cut has measure: -11.3863(16/82,12)
0 dual pivots were used
24 primal pivots were used
measureCuts: -10.4312(196/980,11) (8 of 150 cuts) den=21
Cut-improving pivot.    CutMeasure=-9(16/144,9) (6 of 25 cuts) den=8
count of Gomory cuts = 7
The candidate cut has measure: -9(16/144,9)
0 dual pivots were used
14 primal pivots were used
```

```
measureCuts: -10(1/10,10) (98 of 98 cuts) den=2
Cut-improving pivot.    CutMeasure=-4(1/4,4) (25 of 25 cuts) den=2
count of Gomory cuts = 1
The candidate cut has measure: -4(1/4,4)
8 dual pivots were used
15 dual pivots were used
13 dual pivots were used
15 dual pivots were used
The lifted cut has measure: -3(1/3,3)
3 dual pivots were used
13 primal pivots were used
measureCuts: -15.5543(16/138,16) (20 of 138 cuts) den=5
Cut-improving pivot.    CutMeasure=-7.66096(4/20,8) (13 of 25 cuts) den=3
Cut-improving pivot.    CutMeasure=-7.54373(4/17,8) (12 of 25 cuts) den=3
count of Gomory cuts = 2
The candidate cut has measure: -7.54373(4/17,8)
0 dual pivots were used
2 primal pivots were used
Denominator Reducing pivot is:2
measureCuts: -9(1/9,9) (78 of 78 cuts) den=2
Cut-improving pivot.    CutMeasure=-5(1/5,5) (25 of 25 cuts) den=2
count of Gomory cuts = 1
The candidate cut has measure: -5(1/5,5)
3 dual pivots were used
15 dual pivots were used
21 dual pivots were used
13 dual pivots were used
0 dual pivots were used
The lifted cut has measure: -1(1/1,1)
17 dual pivots were used
0 primal pivots were used
measureCuts: -12(1/12,12) (109 of 109 cuts) den=2
Cut-improving pivot.    CutMeasure=-3(1/3,3) (25 of 25 cuts) den=2
count of Gomory cuts = 1
The candidate cut has measure: -3(1/3,3)
7 dual pivots were used
2 dual pivots were used
0 dual pivots were used
The lifted cut has measure: nan(1/0,0)
0 dual pivots were used
c
c  The term is NOT satisfiable.
c
c 10 lifted cutting planes were used in 4 seconds.
c
c The DIMACS "solution" line:
s cnf 0 30 127
c
c The DIMACS "timing" line:
t cnf 0 30 127 4 10
```

References

Aarts, E., & Lenstra, J. K. (1997). *Local search in combinatorial optimization.* John Wiley & Sons.

Abramsky, S., Gabbay, D. M., & Maibaum, T. S. E. (Eds.). (1992). *Handbook of logic in computer science: Volume 2 background: Computational structures.* Oxford: Clarendon Press.

Arora, S., Lund, C., Motwani, R., Sudan, M., & Szegedy, M. (1998). Proof verification and the hardness of approximation problems. *Journal of the Association for Computing Machinery*, *45*(3), 501–555.

Asahiro, Y., Iwama, K., & Miyano, E. (1996). Random generation of test instances with controlled attributes. In D. S. Johnson & M. A. Trick (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Cliques, coloring, and satisfiability: Second DIMACS implementation challenge october 11–13, 1993* (Vol. 26, pp. 377–393). Providence, RI: American Mathematical Society.

Balas, E. (1990). Finding out whether a valid inequality is facet defining. In R. Kannan & W. R. Pulleyblank (Eds.), *Proceedings of the 1st integer programming and combinatorial optimization conference* (pp. 45–60). Waterloo, ON, Canada: University of Waterloo Press.

Balas, E., Ceria, S., Cornuéjols, G., & Natraj, N. (1996). Gomory cuts revisited. *Operations Research Letters*, *19*(1), 1–9.

Balas, E., & Clausen, J. (Eds.). (1995). *Lecture notes in computer science 920: Integer programming and combinatorial optimization, 4th international IPCO conference* (Vol. 920). Copenhagen, Denmark: Springer Verlag.

Balas, E., & Jeroslow, R. (1972). Canonical cuts on the unit hypercube. *SIAM Journal on Applied Mathematics*, *23*(1), 61–69.

Balas, E., & Ng, S. M. (1989a). On the set covering polytope: II. Lifting the facets with coefficients in {0,1,2}. *Mathematical Programming*, *45*, 1–20.

Balas, E., & Ng, S. M. (1989b). On the set covering polytope: I. All the facets with coefficients in {0,1,2}. *Mathematical Programming*, *43*, 57–69.

Balas, E., & Zemel, E. (1984). Lifting and complementing yields all the facets of positive zero-one programming polytopes. In *Mathematical programming (rio de janeiro, 1981)* (pp. 13–24). Amsterdam: North-Holland.

Barendregt, H. P. (1984). *The lambda calculus: Its syntax and semantics* (2nd ed.). North Holand.

Barth, P. (1993). Linear 0-1 inequalities and extended clauses. In A. Voronkov (Ed.), *Lecture notes in computer science 698: Logic programming and automated reasoning, 4th international conference, LPAR'93, st. petersburg, russia, july 13-20, 1993, proceedings* (pp. 40–51). Petersburg, Russia: Springer Verlag.

Barth, P. (1994). Simplifying clausal satisfiability problems. In J. P. Jouannaud (Ed.), *Lecture notes in computer science 845: Constraints in computational logics. proceedings* (pp. 19–33). Munich, Germany: Springer Verlag.

Barth, P. (1995). *A Davis-Putnam based enumeration algorithm for linear pseudo-boolean optimization* (Tech. Rep. No. MPI-I-95-2-003). Im Stadtwald, D 66123 Saarbrücken, Germany: Max-Planck Institut für Informatik.

Barth, P. (1996). *Logic-based 0-1 constraint programming.* Boston: Klewer Academic.

Beame, P., Impagliazzo, R., Krajíček, J., Pitassi, T., Pudlak, P., & Woods, A. (1992). Exponential lower bounds for the pigeonhole principle. In N. Alon (Ed.), *Proceedings of the 24th annual ACM symposium on the theory of computing* (pp. 200–220). Victoria, B.C., Canada: ACM Press.

Beame, P., & Pitassi, T. (1996). Simplified and improved resolution lower bounds. In *Proceedings 37th annual symposioum on foundations of computer science* (pp. 274–282). Burlington, Vermont: IEEE Computer Society.

Bernays, P. (1991). *Axiomatic set theory.* New York: Dover Publications. (Original work published 1968)

Bibel, W. (1990). Short proofs of the pigeonhole formulas based on the connection method. *Journal of Automated Reasoning*, *6*(3), 287–297.

Bibel, W., & Eder, E. (1993). Methods and calculi for deduction. In D. M. Gabbay, C. J. Hogger, & J. A. Robinson (Eds.), *Handbook of logic in artificial intelligence and logic programming: Volume 1 logical foundations* (pp. 68–183). Oxford: Clarendon Press.

Blair, C. (1976). Two rules for deducing valid inequalities for 0-1 problems. *SIAM Journal on Applied Mathematics*, *31*(4), 614–617.

Bockmayr, A., & Eisenbrand, F. (1997). *On the chvátal rank of polytopes in the* 0/1 *cube* (Tech. Rep. No. MPI-I-97-2-009). Im Stadwald, 66123 Saarbrucken (Germany): Max-Planck-Institut Für Informatik. (Available: http://www.mpi-sb.mpg.de/∼eisen/)

Bonet, M. L., Esteban, J. L., Galesi, N., & Johannsen, J. (1998). Exponential separations between restricted resolution and cutting plane proof systems. *Electronic Colloquium on Computational Complexity*, *5*(TR98-035). (Available: www.eccc.uni-trier.de/eccc-local/Lists/TR-1998.html)

Book, R. V. (Ed.). (1991, April). *Lecture notes in computer science 488: Rewriting techniques and applications: 4th international conference, RTA-91.* Como, Italy: Springer Verlag.

Book, R. V. (Ed.). (1993, June). *Lecture notes in computer science 690: Rewriting techniques and applications: 5th international conference, RTA-93.* Montreal, Canada: Springer Verlag.

Boole, G. (1958). *An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities.* New York: Dover. (Original work published 1854)

Buss, S. R. (1987). Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, *52*(4), 916–927.

Carter Jr., E. F. (1994). *R250.* (Available: http://www.taygeta.com/random.xml)

Ceria, S., Cornuéjols, G., & Dawande, M. (1995). Combining and strengthing gomory cuts. In E. Balas & J. Clausen (Eds.), *Lecture notes in computer science 920: Integer programming and combinatorial optimization, 4th international IPCO conference* (Vol. 920, pp. 438–451). Copenhagen, Denmark: Springer Verlag.

Church, A. (1936a). A note on the entscheidungsproblem. *The Journal of Symbolic Logic*, *1*, 40–41. (Correction, ibid., pp. 101-102)

Church, A. (1936b). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, *58*, 345–363.

Church, A. (1941). *The calculi of lambda-conversion.* Princeton, NJ: Princton University Press.

Church, A. (1956). *Introduction to mathematical logic* (revised ed.). Princeton, NJ: Princeton University Press.

Church, A., & Rosser, J. B. (1936). Some properties of conversion. *Transactions of the American Mathematical Society*, *39*, 472–482.

Chvátal, V. (1973). Edmund polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, *4*, 305–337.

Chvátal, V. (1983). *Linear programmig.* New York: W. H. Freeman and Co.

Chvátal, V. (1984). *Cutting-plane proofs and the stability number of a graph* (Technical Report No. 84326). Rheinische Friedrich-Wilhelms-Universität, Bonn: Institut für Ökonometrie und Operations Research.

Chvátal, V., & Szemerédi, E. (1988). Many hard examples for resolution. *Journal of the Association for Computing Machinery*, *35*, 759–770.

Clote, P. (1995). Cutting plane and Frege proofs. *Information and Computation*, *121*(1), 103–122.

Comon, H., & Jouannaud, J.-P. (Eds.). (1993, May). *Lecture notes in computer science 909: Term rewriting: French spring school of theoretical computer science.* Font Romeux, France: Springer Verlag.

Cook, S., & Reckhow, R. (1974). On the lengths of proofs in the propositional calculus (preliminary version). In *Conference record of sixth annual ACM symposium on theory of computing* (pp. 135–148). Seattle, Washington: ACM.

Cook, S., & Reckhow, R. (1979). The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*(1), 36-50.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Conference record of third annual ACM symposium on theory of computing* (pp. 151–158). Shaker Heights, Ohio: ACM.

Cook, W., Coullard, C. R., & Turán, G. (1987). On the complexity of cutting plane proofs. *Discrete Applied Mathematics*, *18*(1), 25–38.

Cook, W. J. (1990). Cutting-plane proofs in polynomial space. *Mathematical Programming*, *47*(1), 11–18.

Cornuéjols, G., & Sassano, A. (1989). On the 0, 1 facets of the set covering polytope. *Mathematical Programming*, *43*(1), 45–55.

Crescenzi, P., & Kann, V. (1995). *A compendium of NP optimization problems* (Tech. Rep. No. SI/RR-95/02). Dipartimento di Scienze dell´Informazione, Università di Roma "La Sapienza". (Available: http://www.nada.kth.se/nada/theory/-problemlist.html)

Curry, H. B. (1977). *Foundations of mathematical logic.* Dover Publications, Inc. (Original work published 1963)

Dantsin, E. Y. (1997). Algorithmics of propositional satisfiability problems. In V. Kreinovich & G. Mints (Eds.), *Problems of reducing the exhaustive search* (pp. 5–22). Providence, RI: American Mathematical Society.

Dantzig, G. B. (1963). *Linear programming and extensions.* Princeton, NJ: Princeton University Press.

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem proving. *Communications of the ACM*, *5*, 394–397.

Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, *7*, 201–215.

Dershowitz, N. (1987). Termination of rewriting. In J.-P. Jouannaud (Ed.), *Rewriting techniques and applications* (pp. 69–115). London: Academic Press. (Reprinted from *Journal of Symbolic Computation*, 1987, *3*[1&2])

Dershowitz, N. (Ed.). (1989, April 3–5). *Lecture notes in computer science 355: Rewriting techniques and applications: 3rd international conference, RTA-89.* Chapel Hill, NC: Springer Verlag.

Dershowitz, N., & Jouannaud, J.-P. (1994). Rewrite systems. In J. V. Leeuwen (Ed.), *Handbook of theoretical computer science: Vol. B: Formal models and semantics* (pp. 243–320). Cambridge, MA: MIT Press.

Deskins, W. E. (1978). Basic concepts in algebra. In W. H. Beyer (Ed.), *CRC standard mathematical tables* (25th ed., pp. 16–22). West Palm Beach, Florida: CRC Press.

DIMACS Center for Mathematics and Theoretical Computer Science. (1993a, May 8). *Satisfiability suggested format.* (Available: ftp://dimacs.rutgers.edu/pub/-challenge/satisfiabiliy/doc/satformat.ps)

DIMACS Center for Mathematics and Theoretical Computer Science. (1993b). *DIMACS SAT benchmarks.* (Available: ftp://dimacs.rutgers.edu/pub/challenge/-satisfiability/benchmarks/cnf)

Doob, J. L. (1993). *Measure theory.* New York: Springer.

Dorfman, R., Samuelson, P. A., & Solow, R. M. (1987). *Linear programming and economic analysis.* New York: Dover. (Original work published 1958)

Dowling, W. F., & Gallier, J. H. (1984). Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, *1*(3), 267–284.

Du, D., Gu, J., & Pardalos, P. M. (Eds.). (1997). *DIMACS series on discrete mathematics and theoretical computer science: Satisfiability problem: Theory and applications* (Vol. 35). Providence, RI: American Mathematical Society.

Edmonds, J. (1965). Paths, trees, and flowers. *Canadian Journal of Mathematics (Journal Canadien de Mathématiques)*, *17*, 449-467.

Eisinger, N., & Ohlbach, H. J. (1993). Deduction systems based on resolution. In D. M. Gabbay, C. J. Hogger, & J. A. Robinson (Eds.), *Handbook of logic in artificial intelligence and logic programming: Volume 1 logical foundations* (pp. 183–271). Oxford: Clarendon Press.

Epstein, R. L. (1995). *The semantic foundations of logic: Propositional logics* (second ed.). New York: Oxford University Press.

Ershov, Y. E., & Palyutin, E. A. (1984). *Mathematical logic* (V. Shokurov, Trans.). Moscow: Mir Publishers. (Original work published 1979)

Evan, S. A., Itai, A., & Shamir, A. (1976). On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, *5*, 691–703.

Evans, T. (1951). On multiplicative systems defined by generators and relations. I. normal form theorems. *Proc. Cambridge Phil. Soc.*, *47*, 637–649.

Fages, F. (1983). *Formes canoniques dans les algèbres booléennes et application à la démonstration automatique en logique du premier ordre.* Thèse de 3ème cycle, University de Paris VI, Paris.

Fiduccia, C., & Mattheyses, R. (1982). A linear-time heuristic for improving network partition. 175–181.

Franco, J. (1989). *On the occurrence of null clauses in random instances of satisfiability* (Tech. Rep. No. TR 291). Bloomington, Indiana: Computer Science Department, Indiana University.

Franco, J. (1997). Relative size of certain polynomial time solvable subclasses of satisfiability. In D. Du, J. Gu, & P. M. Pardalos (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Satisfiability problem: Theory and applications* (Vol. 35, pp. 211–223). Providence, RI: American Mathematical Society.

Franco, J., Dunn, J. M., & Wheeler, W. H. (1992). Recent work at the interface of logic, combinatorics, and computer science. *Annals of Mathematics and Artificial Intelligence*, *6*, 1–16. (Special issue on Logic and Combinatorics. Available: http://www.ece.uc.edu/∼franco/content.html)

Franco, J., & Gelder, A. V. (1998). *A perspective on certain polynomial time solvable classes of satisfiability* (Tech. Rep.). University of Cincinnati, Cincinnati, OH 45221: Computer Science Department. (To appear in Journal of Global Optimization. Available: http://www.ece.uc.edu/∼franco/content.html)

Franco, J., & Ho, Y. C. (1986). *On the probabilistic performance of algorithms for the satisfiability problem* (Tech. Rep. No. TR 167). Bloomington, Indiana: Computer Science Department, Indiana University.

Freeman, J. W. (1995). *Improvements to propositional satisfiability search algorithms.* Unpublished doctoral dissertation, University of Pensylvania.

Freese, R., Jezek, J., & Nation, J. B. (1993). Term rewrite systems for lattice theory. *Journal of Symbolic Computation*, *16*(3), 279–288.

Gabbay, D. M. (1992). Elements of algorithmic proof. In S. Abramsky, D. M. Gabbay, & T. S. E. Maibaum (Eds.), *Handbook of logic in computer science: Volume 2 background: Computational structures* (pp. 311–413). Oxford: Clarendon Press.

Gabbay, D. M., Hogger, C. J., & Robinson, J. A. (Eds.). (1993). *Handbook of logic in artificial intelligence and logic programming: Volume 1 logical foundations.* Oxford: Clarendon Press.

Galil, Z. (1974). *The complexity of resolution procedures for theorem proving* (Tech. Rep. No. TR 75-223). Ithaca, New York: Dept. of Computer Science, Cornell University. (Available: http://cs-tr.cs.cornell.edu:80/Dienst/UI/2.0/Describe/-ncstrl.cornell%2fTR75-223)

Galil, Z. (1975a). *The complexity of resolution procedures for theorem proving in the propositional calculus.* Unpublished doctoral dissertation, Cornell University. (Also published as a technical report (Galil, 1975b))

Galil, Z. (1975b). *The complexity of resolution procedures for theorem proving in the propositional calculus* (Tech. Rep. No. TR 75-239). Ithaca, New York: Dept. of Computer Science, Cornell University. (Available: http://cs-tr.cs.cornell.edu:80/Dienst/UI/2.0/Describe/ncstrl.cornell%2fTR75-239)

Ganzinger, H. (Ed.). (1996, July). *Lecture notes in computer science 1103: Rewriting techniques and applications: 7th international conference, RTA-96.* New Brunswick, NJ: Springer Verlag.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-Completeness.* New York: W. H. Freeman and Company.

Gärtner, B., & Ziegler, G. M. (1994). Randomized simplex algorithms on Klee-Minty cubes. In *35th annual symposium on foundations of computer science* (pp. 502–510). Santa Fe, New Mexico. (Available: http://elib.zib.de:88/preprints/-shadows/SC-94-07.html)

Gödel, K. (1992). *On formally undecidable propositions of principia mathematica and related systems* (B. Meltzer, Trans.). New York: Dover Publications. (Reprinted from *Monatshefte Für Mathematic und Physik*, 1931, *38*, 173-198)

Goerdt, A. (1991). Cutting plane versus frege proof systems. In E. Boerger, H. K. Buening, M. M. Richter, & W. Schoenefeld (Eds.), *Lecture notes in computer science 533: Proceedings of computer science logic (CSL '90)* (Vol. 533, pp. 174–194). Berlin: Springer-Verlag.

Gomory, R. E. (1958). Outline of an algorithm for integer solutions to linear programs. *Bulletin American Mathematical Society*, *64*, 275–278.

Gomory, R. E. (1963). An algorithm for integer solutions to linear programs. In R. L. Graves & P. Wolfe (Eds.), *Recent advances in mathematical programming* (pp. 269–302). New York: McGraw Hill.

Gondran, M. (1973). Un outil pour la programmation en nombres entires. la méthode des congruences décroissantes. *Revue Française d'Automatique, Informatique, et Recherche Opérationelle*, *3*, 35–54.

Gu, J., Purdom, P., Franco, J., & Wah, B. (1997). Algorithms for satisfiability (SAT) problem: A survey. In D. Du, J. Gu, & P. M. Pardalos (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Satisfiability problem: Theory and applications* (Vol. 35, pp. 19–151). Providence, RI: American Mathematical Society.

Gu, Z., Nemhauser, G. L., & Savelsbergh, M. W. P. (1995). Sequence independent lifting of cover inequalities. In E. Balas & J. Clausen (Eds.), *Lecture notes in computer science 920: Integer programming and combinatorial optimization, 4th international IPCO conference* (Vol. 920, pp. 452–461). Copenhagen, Denmark: Springer Verlag.

Halmos, P. R. (1950). *Measure theory.* New York: D. van Nostrand Company, Inc.

Hankin, C. (1993). *Lambda calculi: A guide for computer scientists.* Oxford: Clarendon Press.

Håstad, J. (1997). Some optimal inapproximability results. In *Proceedings of the twenty-ninth annual ACM symposium on theory of computing* (pp. 1–10). El Paso, Texas.

Håstad, J. T. (1987). *Computational limitations for small-depth circuits.* Cambridge, MA: MIT Press. (ACM Doctoral Dissertation Award, 1986)

Heering, J., Mienke karl, Möller, B., & Nipkow, T. (Eds.). (1993, September). *Lecture notes in computer science 816: Higher-order algebra, logic, and term rewriting: First international workshop, HOA-93.* Amsterdam, The Netherlands: Springer Verlag.

Herbrand, J. (1967). Investigations in proof theory: The properties of true propositions (B. Dreben & J. van Heijenoort, Trans.). In J. van Heijenoort (Ed.), *From Frege to Godel: A source book in mathematical logic, 1879–1931* (pp. 525–581). Cambridge, MA: Harvard University Press. (Original work published 1930)

Hochbaum, D. S. (Ed.). (1995). *Approximation algorithms for NP-hard problems.* Boston, MA: PWS Publishing Company.

Hooker, J. N. (1992). Generalized resolution for 0-1 linear inequalities. *Annals of Mathematics and Artificial Intelligence*, *6*, 271–286.

Hsiang, J. (1982). *Topics in automated theorem proving and program generation* (Tech. Rep. No. R-82-1113). Urbana, IL: Department of Computer Science, University of Illinois.

Hsiang, J. (1983). *Topics in automated theorem proving and program generation.* Unpublished doctoral dissertation, University of Illinois at Urbana-Champaign. (Abstract DAI-B 43/12, p. 4056, Jun 1983. Also published as a technical report (Hsiang, 1982).)

Hsiang, J. (Ed.). (1995, April). *Lecture notes in computer science 914: Rewriting techniques and applications: 6th international conference, RTA-95.* Kaiserslautern, Germany: Springer Verlag.

Hsiang, J., & Huang, G. S. (1997). Some fundamental properties of Boolean ring normal forms. In D. Du, J. Gu, & P. M. Pardalos (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Satisfiability problem: Theory and applications* (Vol. 35). Providence, RI: American Mathematical Society. (Available: http://www.csie.ntu.edu.tw/∼hsiang/dimacs1.ps)

Jeroslow, R. G., & Kortanek, K. O. (1971). On an algorithm of gomory. *SIAM Journal on Applied Mathematics*, *21*(1), 55–60.

Johnson, D. S., & Trick, M. A. (Eds.). (1996). *DIMACS series on discrete mathematics and theoretical computer science: Cliques, coloring, and satisfiability: Second DIMACS implementation challenge october 11–13, 1993* (Vol. 26). Providence, RI: American Mathematical Society.

Jouannaud, J.-P. (Ed.). (1987). *Rewriting techniques and applications.* London: Academic Press. (Reprinted from *Journal of Symbolic Computation*, 1987, *3*[1&2])

Joy, S., Mitchell, J., & Borchers, B. (1997). A branch and cut algorithm for MAX-SAT and weighted MAX-SAT. In D. Du, J. Gu, & P. M. Pardalos (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Satisfiability problem: Theory and applications* (Vol. 35, pp. 519–536). Providence, RI: American Mathematical Society.

Jünger, M., Reinelt, G., & Thienel, S. (1995). Practical problem solving with cutting plane algorithms in combinatorial optimization. In J. E. Goodman, R. Pollack, & W. Steiger (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Combinatorial optimization: Papers from the DIMACS special year* (Vol. 20, pp. 111–152). Providence, RI: American Mathematical Society.

Kalai, G. (1992). A subexponential randomized simplex algorithm (extended abstract). In *Proceedings of the twenty-fourth annual ACM symposium on the theory of computing* (pp. 475–482). Victoria, British Columbia, Canada.

Karp, R. M. (1972). Reducibility among combinatorial problems. In R. E. Miller & J. W. Thatcher (Eds.), *Complexity of computer computations* (pp. 85–103). New York: Plenum Press.

Kirkpatrick, S., & Stoll, E. P. (1981). A very fast shift-register sequence random number generator. *Journal of Computational Physics*, *40*(2), 517–526.

Klop, J. W. (1992). Term rewriting systems. In S. Abramsky, D. M. Gabbay, & T. S. E. Maibaum (Eds.), *Handbook of logic in computer science: Volume 2 background: Computational structures* (pp. 1–116). Oxford: Clarendon Press.

Knuth, D. E., & Bendix, P. B. (1970). Simple word problems in universal algebras. In J. Leech (Ed.), *Computational problems in abstract algebra* (p. 263-297). Permagon Press.

Krajíček, J. (1997). Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *Journal of Symbolic Logic*, *62*(2), 457–486.

Kullmann, O. (1998). New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science.* (To appear. Available: http://mi.informatik.uni-frankfurt.de/people/kullmann/papers.html)

Le Chenadec, P. (1986). *Canonical forms in finitely presented algebras.* New York: John Wiley & Sons, Inc.

Lescanne, P. (Ed.). (1987, May). *Lecture notes in computer science 256: Rewriting techniques and applications.* Bordeaux, France: Springer Verlag.

Lewis, H. R. (1978). Renaming a set of clauses as a horn set. *Journal of the Association for Computing Machinery*, *25*, 134–135.

Løkkentangen, A., & Glover, F. (1997). Surrogate constraint analysis – new heuristics and learning schemes for satisfiability problems. In D. Du, J. Gu, & P. M. Pardalos (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Satisfiability problem: Theory and applications* (Vol. 35, pp. 537–572). Providence, RI: American Mathematical Society.

Mendelson, E. (1987). *Introduction to mathematical logic* (Third ed.). Monterey, California: Wadsworth & BrooksCole.

Minoux, M. (1986). *Mathematical programming: Theory and algorithms* (S. Vajda, Trans.). Chichester, UK: John Wiley & Sons. (Original work published 1983)

Motwani, R., & Raghavan, P. (1995). *Randomized algorithms.* Cambridge, UK: Cambridge University Press.

Nemhauser, G. L., & Wolsey, L. A. (1988). *Integer and combinatorial optimization.* New York: John Wiley & Sons.

Newman, M. H. A. (1942). On theories with a combinatorial definition of "equivalence". *Ann. Math.*, *43*, 223–243.

Nobili, P., & Sassano, A. (1989). Facets and lifting procedures for the set covering polytope. *Mathematical Programming*, *45B*(1), 111–137.

Padberg, M. W. (1975). A note on zero-one programming. *Operations Research*, *23*, 833-837.

Peled, U. (1977). Properties of the facets of binary polytopes. *Annals of Discrete Mathematics*, *1*, 435–456.

Plaza, J. A. (1996). Soundness and completeness versus lifting property. In J. Calmet, J. A. Campbell, & J. Pfalzgraf (Eds.), *Lecture notes in computer science 1138: Artificial intelligence and symbolic mathematical computation: Proceedings of the international conference, AISMC-3* (pp. 354–364). Steyr, Austria: Springer Verlag.

Pudlák, P. (1997). Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, *62*(3), 981–998.

Quine, W. V. O. (1951). *Mathematical logic* (Revised ed.). New York: Harper & Row.

Quine, W. V. O. (1952). The problem of simplifying truth functions. *American Math. Monthly*, *59*(8), 521–531.

Resende, M. G. C., & Feo, T. A. (1996). A GRASP for satisfiability. In D. S. Johnson & M. A. Trick (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Cliques, coloring, and satisfiability: Second DIMACS implementation challenge october 11–13, 1993* (Vol. 26, pp. 499–520). Providence, RI: American Mathematical Society.

Robinson, J. A. (1965). A machine oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, *12*, 23-41.

Russell, B. (1938). *Principles of mathematics* (second ed.). New York: W. W. Norton & Co. (Original work published 1902)

Salomaa, A. (1973). *Formal languages.* Orlando, Florida: Academic Press.

Sassano, A. (1989). On the facial structure of the set covering polytope. *Mathematical Programming*, *44*(2), 181–202.

Schrijver, A. (1986). *Theory of linear and integer programming.* Chichester, UK: John Whley & Sons.

Selman, B., Kautz, H., & Cohen, B. (1996). Local search strategies for satisfiability testing. In D. S. Johnson & M. A. Trick (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Cliques, coloring, and satisfiability: Second DIMACS implementation challenge october 11–13, 1993* (Vol. 26, pp. 521–531). Providence, RI: American Mathematical Society.

Selman, B., Levesque, H., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *Proceedings of AAAI '92* (pp. 440–446). AAAI Press.

Shoup, V. (1999). *Ntl: A library for doing number theory.* (Available: http://-www.shoup.net/ntl)

Smale, S. (1983). On the average number of steps in the simplex method of linear programming. *Mathematical programming*, *27*, 241–262.

Spears, W. M. (1996). Simulated annealing for hard satisfiability problems. In D. S. Johnson & M. A. Trick (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Cliques, coloring, and satisfiability: Second DIMACS implementation challenge october 11–13, 1993* (Vol. 26, pp. 533–555). Providence, RI: American Mathematical Society.

Stoll, R. R. (1961). *Sets, logic, and axiomatic theories.* San Francisco, CA: W. H. Freeman and Co.

Stoll, R. R. (1979). *Set theory and logic.* New York: Dover Publications. (Original work published 1963)

Suppes, P. (1972). *Axiomatic set theory.* New York: Dover Publications. (Original work published 1960)

Tovey, C. A. (1997). Local improvement in discrete structures. In *Local search in combinatorial optimization.* John Wiley & Sons.

Truemper, K. (1998). *Effective logic computation.* New York: John Wiley & Sons.

Turing, A. (1936–1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society, ser. 2*, *42*, 230–265. (Correction, ibid, vol. 43, pp. 544-546, 1937)

Urquhart, A., & Fu, X. (1996). Simplified lower bounds for propositional proofs. *Notre Dame Journal of Formal Logic*, *37*(4), 523–544.

Vizvári, B. (1989). Two algorithms to get strong Gomory cuts. *Optimization*, *20*(1), 117–126.

von Neumann, J. (1986). The general and logical theory fo automata. In W. Aspray & A. Burks (Eds.), (pp. 391–431). Cambridge, MA: The MIT Press. (Original work published 1948)

Wang, J. (1997). Branching rules for propositional satisfiability. In D. Du, J. Gu, & P. M. Pardalos (Eds.), *DIMACS series on discrete mathematics and theoretical computer science: Satisfiability problem: Theory and applications* (Vol. 35, pp. 351–364). Providence, RI: American Mathematical Society.

Whitehead, A. N., & Russell, B. (1950). *Principia mathematica* (second ed.). London: Cambridge University Press. (Original work published 1912)

Wolfram, S. (1999). *The mathematica book* (4th ed.). New York: Cambridge University Press.

Wolsey, L. A. (1976). Facets and strong valid inequalities for integer programs. *Operations Research*, *24*, 367–372.

Wolsey, L. A. (1998). *Integer programming.* New York: John Wiley & Sons.

Zane, F. (1998). *Circuits, CNFs, and satisfiability (boolean circuits, circuit complexity).* Unpublished doctoral dissertation, University of California, San Diego, San Diego, CA.

Zemel, E. (1978). Lifting the facets of zero-one polytopes. *Mathematical Programming*, *15*, 268–277.

Zhang, H. (1997). SATO: An efficient propositional theorem prover. In W. McCune (Ed.), *Lecture notes in artificial intelligence 1249: Automated deduction – cade-14: 14th international conference on automated deduction* (Vol. 1249, pp. 272–275). Townsville, North Queensland, Australia: Springer Verlag.