DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 35, 1997

Algorithms for the Satisfiability (SAT) Problem: A Survey

Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah

ABSTRACT. The satisfiability (SAT) problem is a *core* problem in mathematical logic and computing theory. In practice, SAT is fundamental in solving many problems in automated reasoning, computer-aided design, computer-aided manufacturing, machine vision, database, robotics, integrated circuit design, computer architecture design, and computer network design. Traditional methods treat SAT as a discrete, constrained decision problem. In recent years, many optimization methods, parallel algorithms, and practical techniques have been developed for solving SAT. In this survey, we present a general framework (an algorithm space) that integrates existing SAT algorithms into a unified perspective. We describe sequential and parallel SAT algorithms including variable splitting, resolution, local search, global optimization, mathematical programming, and practical SAT algorithms. We give performance evaluation of some existing SAT algorithms. Finally, we provide a set of practical applications of the satisfiability problems.

1. Introduction

An instance of the satisfiability (SAT) problem is a Boolean formula that has three components [**101, 188**]:

- A set of n variables: x_1, x_2, ..., x_n.
- A set of literals. A literal is a variable ($Q = x$) or a negation of a variable ($Q = \bar{x}$).
- A set of m distinct clauses: C_1, C_2, ..., C_m. Each clause consists of only literals combined by just logical *or* ($\vee$) connectives.

The *goal* of the satisfiability problem is to determine whether there exists an assignment of truth values to variables that makes the following Conjunctive Normal

1991 *Mathematics Subject Classification.* Primary 03B05, 03F20, 90A05, 68Q25, 68T15, 68Q42, 90C27, 90B40, 68T01, 68Q15; Secondary 68Q22, 68P10.

Key words and phrases. Combinatorial optimization, conjunctive normal form, constraint satisfaction problem, decision theory, logic inference, nonlinear programming, parallel and distributed processing, propositional logic, satisfiability problem, search, theorem proving.

This research was supported in part by 1987-1988 and 1988-1989 ACM/IEEE Academic Scholarship Awards, and is presently supported in part by NSERC Research Grant OGP0046423, NSERC Strategic Grants MEF0045793 and STR0167029, NSF Grant CCR-9203941, ONR Grant N00014-94-1-0382, NSF Grants MIP 92-18715 and MIP 96-32316, 1996 DIMACS Workshop on the satisfiability (SAT) problems, and DIMACS visitor program.

Form (*CNF*) formula satisfiable:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m, \tag{1.1}$$

where $\wedge$ is a logical *and* connective.

The SAT problem is a *core* of a large family of computationally intractable NP-complete problems [**101, 188**]. Such NP-complete problems have been identified as central to a number of areas in computing theory and engineering. Since SAT is NP-complete, it is unlikely that any SAT algorithm has a fast worst-case time behavior. However, clever algorithms can rapidly solve many SAT formulas of practical interest. There has been great interest in designing efficient algorithms to solve most SAT formulas.

In practice, SAT is fundamental in solving many problems in automated reasoning, computer-aided design, computer-aided manufacturing, machine vision, database, robotics, integrated circuit design automation, computer architecture design, and computer network design (see Section 14). Therefore, methods to solve SAT formulas play an important role in the development of efficient computing systems.

Traditional methods treat a SAT formula as a discrete, constrained decision problem. In recent years, many optimization methods, parallel algorithms, and practical techniques have been developed. In this survey, we present a general framework (an algorithm space) that integrates existing SAT algorithms into a unified perspective. We describe sequential and parallel SAT algorithms and compare the performance of major SAT algorithms including variable setting, resolution, local search, global optimization, mathematical programming, and practical SAT algorithms. At the end of this survey, we give a collection of practical applications of the satisfiability problem.

This paper is organized as follows.

1. Introduction
2. Constraint Satisfaction Problems
3. Preliminaries
4. An Algorithm-Space Perspective of SAT Algorithms
5. SAT Input Models
6. Splitting and Resolution
7. Local Search
8. Global Optimization
9. Integer Programming Method
10. Special Subclasses of SAT
11. Advanced Techniques
12. Probabilistic and Average-Case Analysis
13. Performance and Experiments
14. Applications
15. Future Work
16. Conclusions

In the next section, we describe the constraint satisfaction problem (CSP) and its close relationship to the SAT problem. Section 3 gives preliminaries for the paper. In Section 4, we give a general framework (an algorithm space) that puts existing SAT algorithms into a unified perspective. This is followed by a brief overview of the basic SAT algorithm classes and a discussion of the general performance evaluation approaches for SAT algorithms. In Section 5, some SAT problem-instance models are given. Section 6 describes the variable setting and resolution procedures

for solving SAT formulas. Local search algorithms, global optimization techniques and integer programming approaches for solving SAT formulas are discussed, respectively, in Sections 7, 8, and 9. Section 10 discusses special subclasses of the SAT problem. Some advanced techniques for solving SAT formulas are described in Section 11. Section 12 gives probabilistic and average-case analysis of the SAT problem.

Experimental results and performance comparisons of several major SAT algorithms are given in Section 13. Presently for hard random 3-SAT problem instances, a complete SAT algorithm could solve a SAT problem with a few hundred variables. An incomplete SAT algorithm such as *WSAT* can solve SAT problem instances with 2,000 variables on an SGI Challenge with a 70 MHz MIPS R4400 processor [**475, 474**]. The randomized local search algorithm, e.g., *SAT1.5*, can solve various SAT problem instances with over 10,000 variables on a SUN SPARC 20 workstation comfortably [**209, 212, 219**]. Most practical SAT solvers used in industrial applications are problem specific. We collected some real experimental results in Section 13. Section 14 summarizes some applications of the SAT problem. Future work for SAT research is discussed in Section 15. Finally, Section 16 concludes the paper.

2. Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is to determine whether a set of constraints over discrete variables can be satisfied. Each constraint must have a form that is easy to evaluate, so any difficulty in solving such a problem comes from the interaction between the constraints and the need to find a setting for the variables that simultaneously satisfies all the constraints [**433**]. In a SAT formula, each constraint is expressed as a clause, making SAT a special case of the constraint satisfaction problem (see Figure 1). Due to this close relationship, any CSP algorithm can be transformed into a SAT algorithm, and this can usually be done in a way that maintains the efficiency of the algorithm.

A discrete CSP model consists of the following three components [**206, 228**]:

- n *variables*: $\{x_1, x_2, \ldots, x_n\}$. An assignment is a tuple of n values assigned to the n variables.
- n *domains*: $\{D_1, D_2, \ldots, D_n\}$. Domain D_i contains d possible *values* (also called *labels*) that x_i may be instantiated, *i.e.*, $D_i = \{l_{i,1}, l_{i,2}, \ldots, l_{i,d}\}$.
- A subset of $D_1 \times D_2 \times \ldots \times D_n$ is a set of *constraints*. A set of *order-l constraints* ($l \leq n$) imposed on a subset of variables $\{x_{i_1}, x_{i_2}, \ldots, x_{i_l}\} \subseteq \{x_1, x_2, \ldots, x_n\}$ is denoted as

$$C_{i_1,i_2,\ldots,i_l} \subseteq D_{i_1} \times D_{i_2} \times \ldots \times D_{i_l}.$$

An order-l constraint indicates the compatibility (*i.e.*, consistency/inconsistency or conflicting measure) among l variables for a given variable assignment. The variables *conflict* if their values do not satisfy the constraint. In practice, two frequently used constraints are *unary constraints* imposed on a single variable ($C_i \subseteq D_i$) and *binary constraints* imposed on a pair of variables ($C_{i,j} \subseteq D_i \times D_j$).

Solving a CSP entails minimizing local inconsistency and finding a consistent value assignment (*i.e.*, a consistent labeling) to the variables subject to the given constraints.

Constraint satisfaction problems are extremely common. Most NP-complete problems are initially stated as constraint satisfaction problems. Indeed, the proof

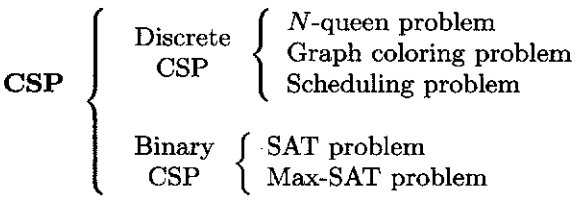


FIGURE 1. Some examples of the constraint satisfaction problem (CSP). SAT problem a special case of CSP, i.e., a CSP with binary values.

that a problem is NP-complete implies an efficient way to transform the problem into a constraint satisfaction problem. For a few special forms of the constraint satisfaction problem there exist algorithms that solve such formulas in polynomial worst-case time. When no polynomial-time algorithm is known for a particular form of constraint satisfaction problem, it is common practice to solve such formulas with a search algorithm.

Problems that are commonly formulated as constraint satisfaction or satisfiability problems for the purposes of benchmarking include graph coloring and the n-queens problems. In the case of the n-queens problem, although analytical solutions for this problem exist [**2, 10, 30**], they provide a restricted subset of solutions. In practical applications, one must use *a search algorithm* to find a *general* solution to the CSP or SAT problems.

3. Preliminaries

To simplify our discussion, throughout this paper, let:

- $\mathcal{F}$ be a *CNF* Boolean formula,
- m be the number of clauses in $\mathcal{F}$,
- n be the number of variables in $\mathcal{F}$,
- C_i be the ith clause,
- $|C_i|$ be the number of literals in clause C_i,
- $Q_{i,j}$ be the jth literal in the ith clause, and
- l be the average number of literals: $\frac{\sum_{i=1}^{m} |C_i|}{m}$,

where $i = 1, ..., m$ and $j = 1, ..., n$.

On Boolean space $\{0,1\}^n$, let:

- $F(\mathbf{x})$ be a function from $\{0,1\}^n$ to integer N,
- x_j be the jth variable,
- $\mathbf{x}$ be a vector of n variables,
- $C_i(\mathbf{x})$ be the ith clause function, and
- $Q_{i,j}(\mathbf{x})$ be the jth literal function of the ith clause function,

where $i = 1, ..., m$ and $j = 1, ..., n$.

On real space E^n, let:

- $N(\mathbf{x})$ be a real function from $\{0,1\}^n$ to E,
- $f(\mathbf{y})$ be a real function from E^n to E,
- y_j be the jth variable,
- $\mathbf{y}$ be a vector of n variables,

- $c_i(\mathbf{y})$ be the ith clause function, and
- $q_{i,j}(\mathbf{y})$ be the jth literal function of the ith clause function,

where $i = 1, ..., m$ and $j = 1, ..., n$.

On real space E^n, also let:

- w_j be the jth integer variable, and
- $\mathbf{w}$ be a vector of n integer variables,

where $i = 1, ..., m$ and $j = 1, ..., n$.

Following [**359**], a real-valued function f defined on a subset of E^n is said to be *continuous* at $\mathbf{y}$ if $f(\mathbf{y}_k) \to f(\mathbf{y})$. A set of real-valued functions $f_1, f_2, ..., f_m$ on E^n form a vector function $\mathbf{f} = (f_1, f_2, ..., f_m)$ whose ith component is f_i. It is *continuous* if each of its component functions is continuous.

If f has second partial derivatives which are continuous on this set, we define the *Hessian* of f at $\mathbf{y}$ to be the $n \times n$ matrix denoted by

$$\mathbf{H}(\mathbf{y}) = \nabla^2 f(\mathbf{y}) = \left[\frac{\partial^2 f(\mathbf{y})}{\partial y_i \partial y_j}\right]. \tag{3.1}$$

We call $\mathbf{y} \in E^n$ with $f(\mathbf{y}) = 0$ a solution of f, denoted as y^*.

Two aspects of iterative optimization algorithms are their global convergence and local convergence rates [**359**]. *Global convergence* concerns, starting from an initial point, whether the sequence of points will converge to the final solution point. *Local convergence rate* is the rate at which the generated sequence of points converge to the solution.

4. An Algorithm-Space Perspective of SAT Algorithms

In this section, we first describe various formulations of SAT, then give an algorithm-space perspective that provides some insights into developing efficient algorithms for solving SAT. Following this, we give a brief overview of the basic sequential and parallel SAT algorithms, and discuss various categories of algorithms and performance evaluation methods.

4.1. Formulations of SAT. SAT problem can be expressed by Conjunctive Normal Form (*CNF*) formulas (e.g., $(x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$) or Disjunctive Normal Form (*DNF*) formulas (e.g., $(\bar{x}_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$). Instances of SAT can be formulated based on discrete or continuous variables [**538, 540**].

Discrete Formulations. These can be classified as unconstrained versus constrained.

(a) *Discrete Constrained Feasibility Formulations.* The goal is to satisfy all constraints. One possible formulation is the *CNF* formulas given by (1.1). A second formulation is the *DNF* formulas [**207**] discussed in Section 7.10.

(b) *Discrete Unconstrained Formulations.* A common formulation for *CNF* formulas exists [**209, 212, 472**]. The goal is to minimize $N(\mathbf{x})$, the number of unsatisfied clauses, under the interpretation that numeric variable $x_i = 1$ ($x_i = 0$) if Boolean variable $x_i = true$ ($x_i = false$), respectively. That is,

$$\min_{\mathbf{x} \in \{0,1\}^n} N(\mathbf{x}) = \sum_{i=1}^{m} C_i(\mathbf{x}) \tag{4.1}$$

where

$$C_i(\mathbf{x}) = \prod_{j=1}^{n} Q_{i,j}(x_j) \tag{4.2}$$

$$Q_{i,j}(x_j) = \begin{cases} 1 - x_j & \text{if } x_j \text{ in } C_i \\ x_j & \text{if } \bar{x}_j \text{ in } C_i \\ 1 & \text{otherwise} \end{cases} \tag{4.3}$$

In this case, $N(\mathbf{x}) = 0$ when all the clauses are satisfied.

A similar formulation for *DNF* formulas exists (See Section 7.10, [**235**], [**207**], [**213**]). Under the interpretation that numeric variable $x_i = 1$ ($x_i = 0$) if Boolean variable $x_i = true$ ($x_i = false$), respectively, the goal is to solve

$$\min_{\mathbf{x}\in\{0,1\}^n} F(\mathbf{x}) = \sum_{i=1}^{m} C_i(\mathbf{x}), \tag{4.4}$$

where

$$C_i(\mathbf{x}) = 1 - \prod_{j=1}^{n} Q_{i,j}(x_j) \tag{4.5}$$

$$Q_{i,j}(x_j) = \begin{cases} x_j & \text{if } x_j \text{ in } C_i \\ 1 - x_j & \text{if } \bar{x}_j \text{ in } C_i \\ 1 & \text{otherwise} \end{cases} \tag{4.6}$$

All the clauses are satisfied when $F(\mathbf{x}) = 0$.

Alternatively, *DNF* formulas can be solved as follows:

$$\max_{\mathbf{x}\in\{0,1\}^n} F(\mathbf{x}) = \sum_{i=1}^{m} C_i(\mathbf{x}), \tag{4.7}$$

where

$$C_i(\mathbf{x}) = \prod_{j=1}^{n} Q_{i,j}(x_j), \tag{4.8}$$

and $Q_{i,j}(x_j)$ is given by (4.6).

Usually, the question of falsifiability for a *DNF* formula is more interesting than the question of satisfiability. This can be solved as follows:

$$\min_{\mathbf{x}\in\{0,1\}^n} F(\mathbf{x}) = \sum_{i=1}^{m} C_i(\mathbf{x}), \tag{4.9}$$

where $C_i(\mathbf{x})$ is given by (4.8). A formula is falsifiable if $F(\mathbf{x}) = 0$ for some $\mathbf{x}$.

(c) *Discrete Constrained Formulations.* There are various forms of this formulation. One approach is to formulate SAT formulas as instances of the 0-1 integer linear programming (ILP) problem.

Another approach is to minimize the objective function $N(\mathbf{x})$, the number of unsatisfiable clauses, subject to a set of constraints, as follows [**538, 540**]:

$$\begin{aligned} &\min_{\mathbf{x}\in\{0,1\}^n} \quad N(\mathbf{x}) = \sum_{i=1}^{m} C_i(\mathbf{x}) \\ &\text{subject to} \quad C_i(\mathbf{x}) = 0 \quad \forall i \in \{1, 2, \ldots, m\}. \end{aligned} \tag{4.10}$$

A formulation based on *DNF* can be defined similarly.

This formulation uses additional constraints to guide the search. The violated constraints provide another mechanism to bring the search out of a local minimum. This formulation is used in a Lagrange multiplier-based method to solve the SAT problem (see Section 8.7 and [**538, 540**]).

Continuous Formulations. In formulating a discrete instance of SAT in continuous space, we transform discrete variables in the original formula into continuous variables in such a way that solutions to the continuous problem are binary solutions to the original formula. Such a transformation is potentially beneficial because an objective in continuous space may "smooth out" some infeasible solutions, leading to a smaller number of local minima explored. In the following, we show two such formulations.

(a) *Continuous Unconstrained Formulations.* There are many possible formulations in this category. A simple formulation, *UniSAT* (Universal SAT Problem Models) [**207, 209, 211, 210**], suggests:

$$\min_{\mathbf{y}\in E^n} f(\mathbf{y}) = \sum_{i=1}^{m} c_i(\mathbf{y}), \tag{4.11}$$

where

$$c_i(\mathbf{y}) = \prod_{j=1}^{n} q_{i,j}(y_j) \tag{4.12}$$

$$q_{i,j}(y_j) = \begin{cases} |y_j - T| & \text{if } x_j \text{ in } C_i \\ |y_j + F| & \text{if } \bar{x}_j \text{ in } C_i \\ 1 & \text{otherwise} \end{cases} \tag{4.13}$$

where T and F are positive constants.

Two special formulations to (4.13) exist. In the *UniSAT5* model [**211, 213**]

$$q_{i,j}(y_j) = \begin{cases} |y_j - 1| & \text{if } x_j \text{ is in } C_i \\ |y_j + 1| & \text{if } \bar{x}_j \text{ is in } C_i \\ 1 & \text{otherwise} \end{cases} \tag{4.14}$$

and in the *UniSAT7* model [**211, 213**]:

$$q_{i,j}(y_j) = \begin{cases} (y_j - 1)^2 & \text{if } x_j \text{ is in } C_i \\ (y_j + 1)^2 & \text{if } \bar{x}_j \text{ is in } C_i \\ 1 & \text{otherwise} \end{cases} \tag{4.15}$$

Values of $\mathbf{y}$ that make $f(\mathbf{y}) = 0$ are solutions to the original formula in (1.1). *UniSAT5* can be solved with efficient, discrete, greedy local search algorithms (Section 8 and [**213**]). *UniSAT7* requires computationally expensive continuous optimization algorithms, rendering them applicable to only small formulas (Section 8 and [**213, 218**]).

(b) *Continuous Constrained Formulations.* This generally involves a heuristic objective function that measures the quality of the solution obtained (such as the number of clauses satisfied). One formulation similar to (4.11) is as follows.

$$\begin{aligned} &\min_{\mathbf{y}\in E^n} && f(\mathbf{y}) = \sum_{i=1}^{m} c_i(\mathbf{y}) \\ &\text{subject to} && c_i(\mathbf{y}) = 0 \quad \forall i \in \{1, 2, \ldots, m\} \end{aligned} \tag{4.16}$$

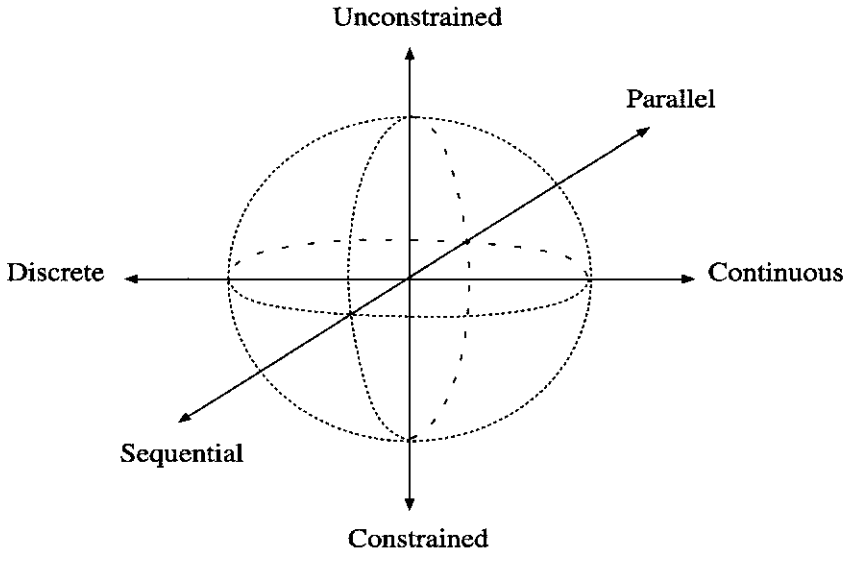


FIGURE 2. The algorithm space is a unified framework for discrete search algorithms and continuous optimization algorithms. The octants represent eight basic classes of algorithms.

where $c_i(\mathbf{y})$ is defined in (4.12).

The key in this approach lies in the transformation. When it does not smooth out local minima in the discrete space or when solution density is low, continuous methods are much more computationally expensive to apply than discrete methods.

Since (4.16) is a continuous constrained optimization problem with a nonlinear objective function and nonlinear constraints, we can apply existing Lagrange-multiplier methods to solve it. Our experience is that a Lagrangian transformation does not reduce the number of local minima, and continuous Lagrangian methods are at least an order-of-magnitude more expensive to apply than the corresponding discrete algorithms [**80**].

4.2. The Algorithm Space. Discrete search algorithms relate to continuous optimization methods in operations research. Many discrete search problems can be solved with numerical algorithms in the real space. A unified framework for search and optimization would shed light on developing efficient algorithms for a search problem. Figure 2 shows a typical *algorithm space* that unifies a variety of search and optimization algorithms in terms of variable domain, constraint used, and parallelism in the algorithms [**213**].

Satisfiability is expressed with discrete variables, but some algorithms do their calculations with continuous variables. This leads to the discrete-continuous axis in the space. Satisfiability has a set of constraints that must be satisfied exactly, but some procedures (e.g., local search) consider changes in variable values in clauses that do not satisfy the constraints (typically, these algorithms assign some cost to non-satisfying constraints and then look for the least-cost solution). This defines the vertical axis in Figure 2 showing constraint characteristics in the algorithm

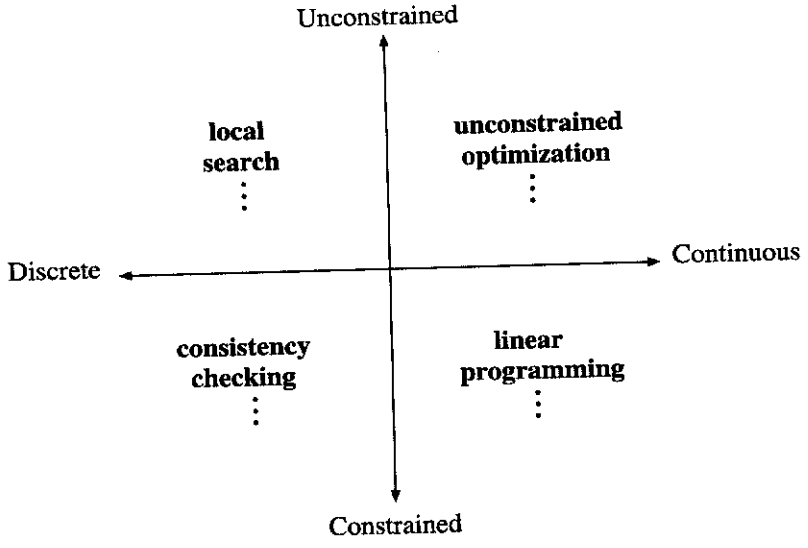


FIGURE 3. A 2-dimensional cross section of the algorithm space cut at the sequential side. It indicates a unified framework for some discrete search algorithms and continuous optimization techniques for solving SAT.

space. Most SAT algorithms are sequential, while some have been implemented in parallel. A third axis indicating parallelism in the algorithms is added in the algorithm space. Following the three axes, the algorithm space is divided into eight octants, representing the four sequential algorithm classes, *i.e.*, discrete constrained algorithms, discrete unconstrained algorithms, continuous constrained algorithms, and continuous unconstrained algorithms, and four parallel algorithm classes, *i.e.*, parallel discrete constrained algorithms, parallel discrete unconstrained algorithms, parallel continuous constrained algorithms, and parallel continuous unconstrained algorithms.

Figure 3 gives some typical examples for the four sequential classes of SAT algorithms in the space In the discrete search space (left half of Figure 3), variables, values, constraints, and the objective functions are defined with discrete values. If one handles a discrete search problem with consistency checking or constraint resolution, the approach belongs to the class of *discrete constrained methods* [**228, 361, 384, 542**]. Alternatively, one can formulate the constraints into an objective function and minimize the objective function without looking at any problem constraints. Algorithms in this category are usually called the *discrete, unconstrained methods* such as local search procedure. [**209, 212, 403, 487, 491**].

In the continuous search space (right half of Figure 3), variables, values, constraints, and objective functions are defined quantitatively with real values. If one solves a continuous optimization problem with explicit constraints, one uses *continuous constrained methods*, such as constrained minimization, primal methods, and cutting plane methods [**359**]. If the problem constraints are incorporated into an objective function, then the problem is transformed into an unconstrained one. The

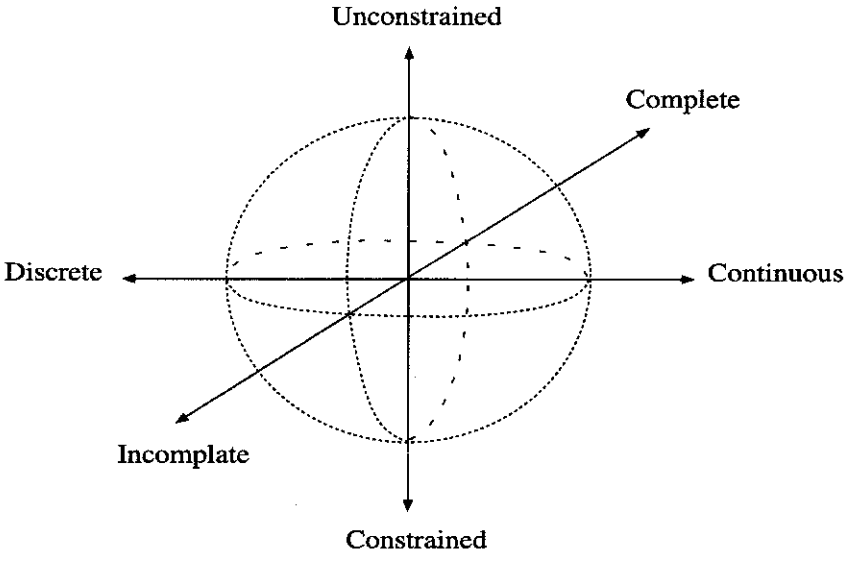


FIGURE 4. An algorithm space incorporating algorithm completeness for solving SAT. Each octant represents one class of SAT algorithms.

latter can be solved by the *continuous unconstrained methods*, such as the descent methods, conjugate direction methods, and Newton methods [**213, 214, 359**].

From an operations research point of view, most discrete search algorithms have continuous optimization versions, and most constrained search methods have unconstrained counterparts. For instance, discrete consistency algorithms are constrained algorithms. If we formulate the amount of "inconsistency" into an objective function, a local search method can often be used to solve an input efficiently. Furthermore, local search works in discrete search space. By extending a search problem into a real search space, constrained and unconstrained global optimization algorithms can be developed to solve SAT [**35, 214, 213, 258, 285, 284**].

The algorithm space provides a unified and global perspective on the development of search and optimization algorithms for solving SAT. In general, for a given instance of a search problem if one can find an algorithm in one octant, then one could possibly find some closely related algorithms in other octants. In the left two quadrants in Figure 3, for example, once we had consistency algorithms and local search algorithms for solving SAT, it would be natural to think about unconstrained optimization algorithms for solving SAT in the right two quadrants — something must be put there to meet the natural symmetry. This was the original incentive to develop unconstrained optimization algorithms for solving SAT [**213**].

There are other ways of looking at a variety of SAT algorithms. A different algorithm space for SAT that incorporates algorithm completeness was given in [**213**] (see Figure 4).

4.3. Basic SAT Algorithm Classes. Following the algorithm space, a number of major SAT algorithm classes can be identified. They are given in Figure 5 in chronological order. Most existing SAT algorithms can be grouped into these

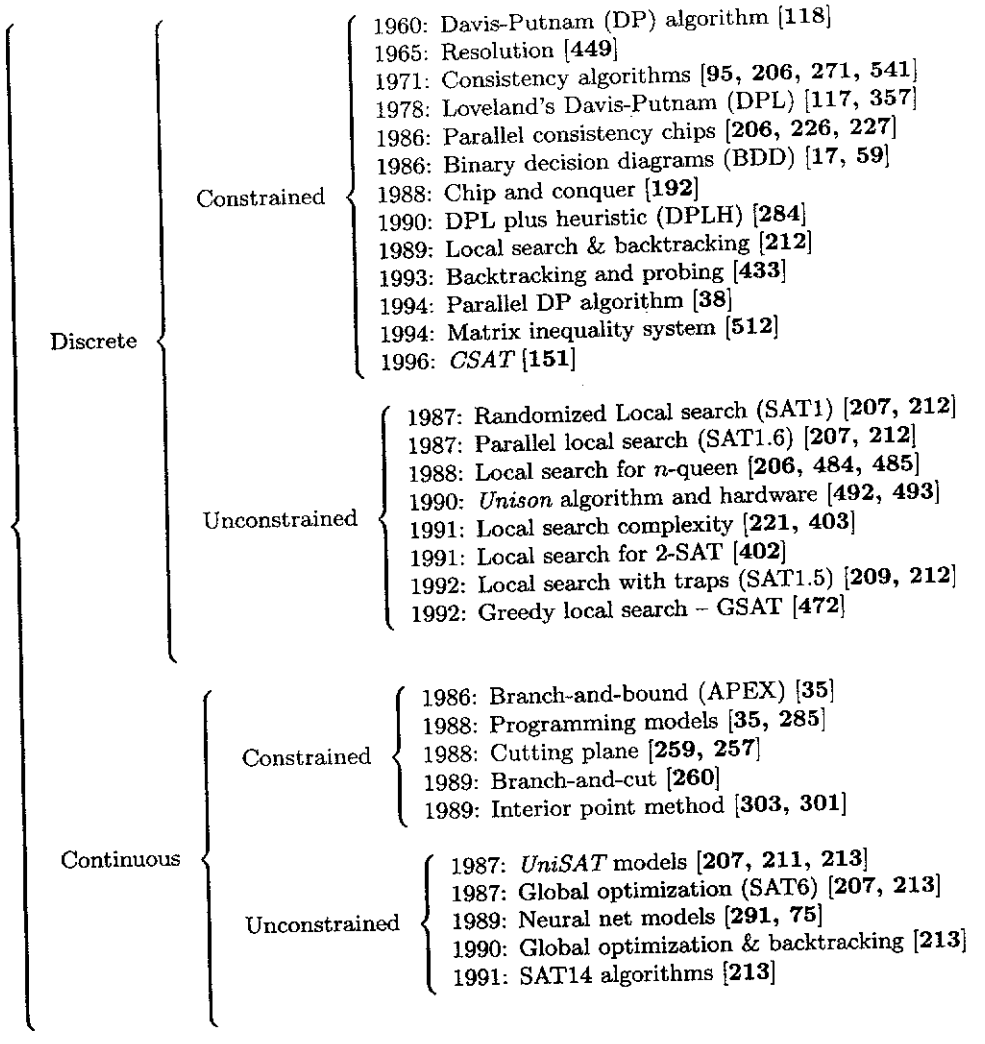


FIGURE 5. Some typical algorithms for the SAT problem.

categories.

- *Discrete, constrained algorithms.* Algorithms in this category treat a SAT formula as an instance of a constrained decision problem, applying discrete search and inference procedures to determine a solution. One straightforward way to solve an instance of SAT is to enumerate all possible truth assignments and check to see if one satisfies the formula. Many improved techniques, such as consistency algorithms [**228, 361**], backtracking algorithms [**34, 53, 64, 328, 425**], term-rewriting [**130, 268**], production system [**482**], multi-valued logic [**478**], Binary Decision Diagrams [**59, 17**], chip

and conquer [**192**], resolution and regular resolution [**195, 357, 397, 449, 514, 525, 549**], independent set algorithm [**281**], and matrix inequality system [**512**], have been proposed.

Many of the discrete constrained algorithms eliminate one variable at a time. This can be done either by making repeated use of resolution, as was done in the original version of the Davis-Putnam (DP) procedure [**118**], or by assigning some variable each possible value and generating a sub-formula for each value, as was done in Loveland's modification to the DP procedure [**117, 357**]. Resolution generates only one new formula, but in the worst case the number of clauses in that new formula will be proportional to the square of the number of clauses in the original formula. Assigning values to a variable (often called searching) generates two new formulas. For random formulas, resolution methods are fast when the number of clauses is small compared to the number of values [**166, 86**], while search methods are fast except when the number of clauses is such that the expected number of solutions is near one [**433**]. The two approaches can be combined, using resolution on some variables and search on others.

Other specific algorithms using these principles include simplified DP algorithms [**181, 203, 430**], and a simplified DP algorithm with strict ordering of variables [**269**]. The DP algorithm improved in certain aspects over Gilmore's proof method [**197**]. Analyses of SAT algorithms often concentrates on algorithms that are simple because it is difficult to do a correct analysis of the best algorithms. Under those conditions where simple algorithms are fast, related practical algorithms are also fast. (It is difficult to tell whether a practical algorithm is slow under conditions that make the corresponding simplified algorithm slow.)

A number of special SAT problems, such as 2-satisfiability and Horn clauses, are *solvable* in polynomial time [**5, 101, 397**]. There are several linear time algorithms [**18, 155**] and polynomial time algorithms [**402, 462**] existing.

- *Discrete, unconstrained algorithms.* In this approach, the number of unsatisfiable *CNF* (or satisfiable *DNF*) clauses is formulated as the value of the objective function, transforming the SAT formula into a discrete, unconstrained minimization problem to the objective function. Local search is a major class of discrete, unconstrained search methods [**209, 212, 228, 221, 403, 472**]. It can be used to solve the transformed formula (see Section 7).
- *Constrained programming algorithms.* Methods in this class were developed based on the fact that *CNF* or *DNF* formulas can be transformed to instances of Integer Programming, and possibly solved using Linear Programming relaxations [**35, 258, 259, 285, 303, 301, 401, 548**]. Many approaches, including branch-and-bound [**35**], cutting-plane [**259, 257**], branch-and-cut [**260**], interior-point [**303, 301**], and improved interior-point [**479**], have been proposed to solve the integer program representing the inference problem. Researchers found integer programming methods faster than resolution for certain classes of problems, although these methods do not possess a robust convergence property and often fail to solve hard instances of satisfiability [**35, 258, 259, 285, 303, 301**].

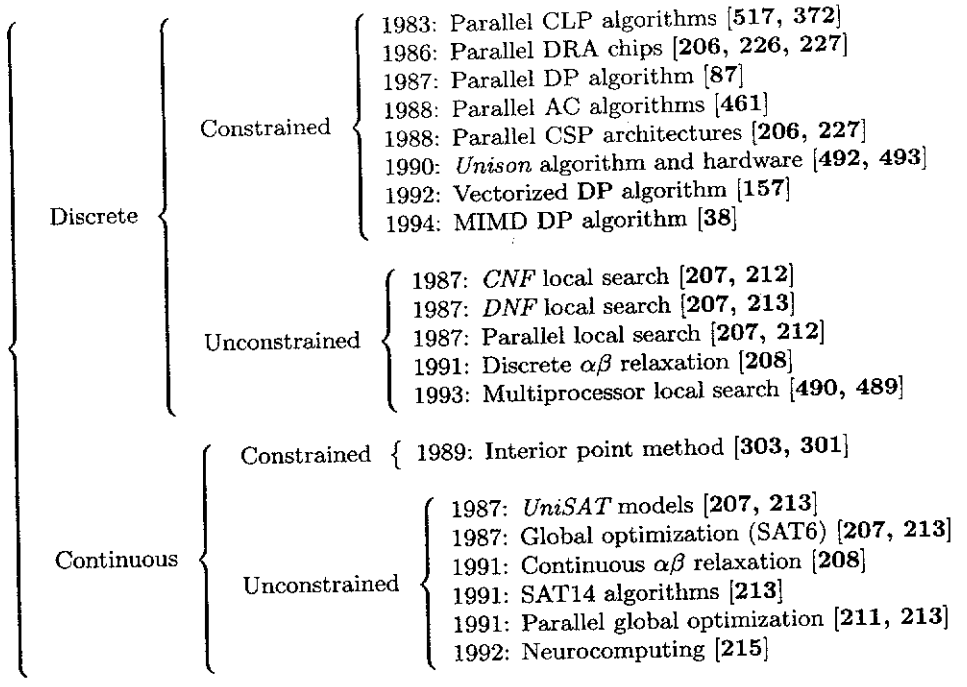


FIGURE 6. Some parallel SAT/CSP algorithms.

- *Unconstrained, global optimization algorithms.* Special models have been formulated to transform a discrete formula on Boolean space $\{0,1\}^n$ (a decision problem) into an unconstrained *UniSAT* problem on real space E^n (an unconstrained global optimization problem). The transformed formulas can be solved by many existing global optimization methods [**207, 209, 211, 213, 228**] (see Section 8).

4.4. Parallel SAT Algorithms. In practice, most sequential SAT algorithms can be mapped onto parallel computer systems, resulting in parallel SAT algorithms [**215**]. A speedup greater than the number of processors sometimes occurs because of correlations among variable settings that lead to solutions [**386, 340**]. Accordingly, as given in Figure 6, there are *four* classes of parallel algorithms for solving SAT.

- *Parallel, discrete, constrained algorithms.* Many discrete, constrained SAT and CSP algorithms have been implemented in parallel algorithms or put on special-purpose, hardware VLSI architectures. These include parallel consistent labeling algorithms [**517, 372**], parallel discrete relaxation (DRA) chips [**226, 206, 227**], parallel arc consistency (PAC) algorithms [**461**], parallel constrained search architectures [**206, 227**], parallel *Unison* algorithms [**492**], parallel *Unison* architectures [**493**], parallel DP algorithms [**38, 87, 157**], and parallel logical programming languages [**99, 353, 532, 533, 534**].

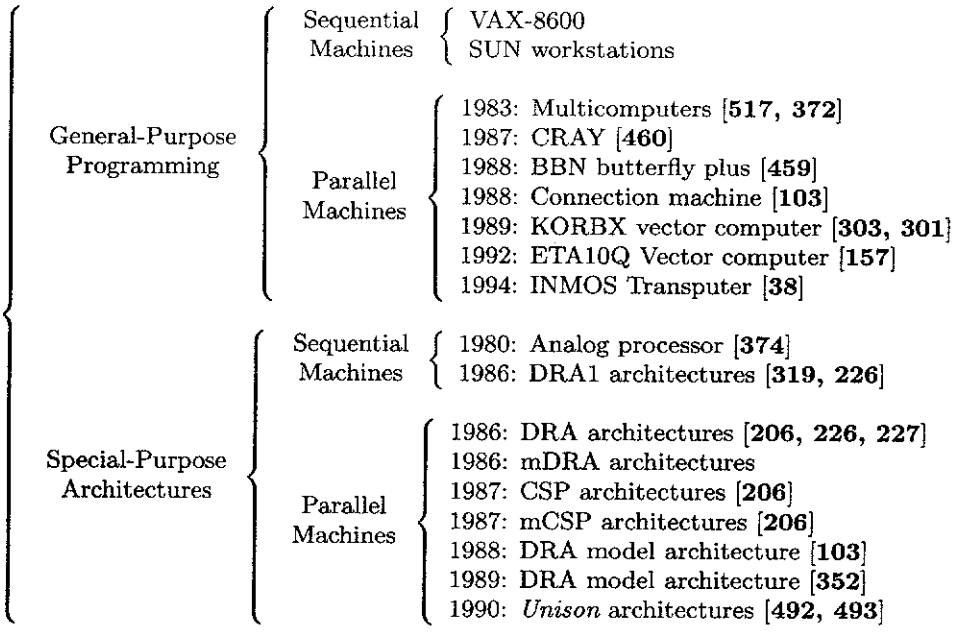


FIGURE 7. Computer architectures used for running SAT/CSP algorithms.

- *Parallel, discrete, unconstrained algorithms.* A number of discrete local optimization algorithms were implemented on parallel computing machines. These include *CNF* local search [**207, 212**], *DNF* local search [**207, 213**], parallel local search [**207, 212**], and multiprocessor local search [**490, 489**]. A new $\alpha\beta$ relaxation technique was developed in a parallel and distributed environment [**208**].
- *Parallel, constrained programming algorithms.* Kamath *et al.* implemented an interior point zero-one integer programming algorithm on a *KORBX(R)* parallel/vector computer [**303, 301**].
- *Parallel, unconstrained, global optimization algorithms.* Several of these algorithms have been implemented: *UniSAT* models [**207, 213**], parallel, continuous $\alpha\beta$ relaxation [**208**], and parallel global optimization algorithms [**211, 213**].

Computer architectures affect the data structures, implementation details, and thus the performance of SAT algorithms. A variety of computer systems have been used for running SAT algorithms (Figure 7). Most early studies of CSP/SAT algorithms were performed on sequential computers. Recent work has been concentrated on parallel programming on multiprocessors. McCall *et al.* [**372, 517**] simulated an 8-processor architecture with various system, topology and performance criteria for the forward checking CSP algorithm. Samal implemented several parallel AC algorithms on a CRAY computer [**460**] and an 18-node BBN Butterfly Plus MIMD, shared-memory, homogeneous parallel processor [**459**]. Cooper and Swain implemented parallel AC algorithms on a Connection Machine [**103**]. Kamath and

Karmarkar *et al.* implemented an interior point zero-one integer programming algorithm for SAT on a *KORBX(R)* parallel/vector computer [**303, 301**]. Recently, Fang and Chen implemented a vectorized DP algorithm on an *ETA10Q* vector computer [**157**]. Speckenmeyer and Böhm have experimented with the parallelization of variants of the Davis-Putnam-Loveland (DPL) procedure on a message based MIMD Transputer system built with 320 (INMOS T800/4MB) processors [**38**]. In their implementation, for some small k, each of 2^k processors solves a formula arising at depth k of a DPL search tree, and computation ceases as soon as one processor reports that its formula is satisfiable. Speckenmeyer noticed that the time to completion was usually less than $N/2^k$ where N is the time taken by the serial version [**38**].

Research works continue by building special-purpose VLSI architectures to speed up SAT/CSP computations. For an n-variable and m-value instance of CSP, Wang and Gu [**543, 544**] gave an $O(n^2d^2)$ time parallel DRA2 algorithm and an SIMD DRA2 architecture. Furthermore, Gu and Wang [**226**] gave an $O(n^2d)$ time parallel DRA3 algorithm and a dynamic DRA3 architecture for solving general DRA problems. Later, Gu and Wang [**206, 227**] developed an $O(nd)$ time, massively parallel DRA5 algorithm and a parallel DRA5 VLSI architecture. For problems of practical interest, parallel DRA algorithms running on special-purpose VLSI architectures offer many orders of magnitude in performance improvement over sequential algorithms.

Recently, Sosič, Gu, and Johnson have developed a number of parallel algorithms and architectures for differential, non-clausal inference of SAT formulas [**492, 493**].

An extreme example of parallel processing is to compute using chemistry with DNA molecules. This would appear to lead a factor of about 10^{23} degrees of parallelism with a slow down of perhaps 10^{10} in the time for computation steps, but this approach has not been investigated in enough detail to determine its practical limitations [**3, 355**]. This SAT evaluation approach is both parallel and random — if it says you have a solution then definitely you do, if it says you do not then probably you do not.

4.5. Algorithm Categories. Some SAT algorithms are *complete* (they definitely determine whether an input has a solution or does not have one) [**59, 118, 117, 357, 449**], while others are *incomplete* (they sometimes determine whether or not the input has a solution, but in other cases they cannot find one) [**212, 213, 301, 402**].

Most incomplete algorithms find one solution (or perhaps several solutions) in *favorable cases*, but give up or do not terminate in other cases. In such cases one does not know whether the input has no solution or the algorithm did not search hard enough. Some incomplete algorithms can verify that a formula has no solution but can not find one if at least one solution exists. Such is the case for incomplete algorithms that check for patterns that imply unsatisfiability. In the strict sense of the word *algorithm*, incomplete algorithms are not algorithms at all, but such procedures are of particular interest for inputs that are so difficult that a complete algorithm cannot solve them in reasonable time.

Complete algorithms can perform one of the following actions: (1) determine whether or not a solution exists, (2) give the variable settings for one solution, (3) find all solutions or an optimal solution, (4) prove that there is no solution.

Algorithms of the first type would be of theoretical interest only were it not for the fact that any such algorithm can be modified, with little loss of efficiency, to give an algorithm of the second type. Algorithms of the third type are needed when there is some measure of the solution quality, and the optimal solution is sought or when the overall problem has constraints in addition to those of the SAT instance. The algorithms are essential to many important practical applications that are NP-hard in nature. Recently, Major et. al. used SAT to precede a program to calculate chemical interaction energies to predict RNA folding [**362**]. Gu and Puri developed an efficient complete SAT algorithm for asynchronous computer circuit design, aiming at producing the minimal circuit structure [**216, 438**]. Incomplete algorithms cannot optimize solution quality, playing little role in solving practical optimization problems.

Requiring a program to produce each solution in explicit form ensures that the worst-case time will be exponential whether or not P = NP (because some inputs have an exponential number of solutions). An alternative is to give the solutions in some compressed form. For example, some algorithms implicitly list all solutions by giving *cylinders of solutions*, *i.e.*, the settings of some variables with the understanding that the remaining variables are *don't cares* which can have any value. For some formulas, using this approach to represent all solutions is much more compact than an explicit representation [**64, 376**]. Binary Decision Diagrams (BDD) are a more sophisticated and compact way to represent the set of all solutions [**59, 17**]. Some instances of SAT, however, have a structure such that it is faster to generate the solution to various subsets of the constraints (depending on a subset of the variables) and then test whether those various solution sets have anything in common rather than try to solve the entire formula at once. This type of SAT algorithm shows greater efficiency improvements for certain practical engineering design problems [**439**].

The techniques used in complete SAT algorithms can usually be adapted to provide *exact solutions* to optimization problems. The techniques used in incomplete SAT algorithms can usually be adapted to provide *approximate solutions* to optimizations problems. They normally lead to algorithms that produce low (but not necessarily the lowest) value of the function.

For random sets of formulas, the probability that a particular formula has at least one solution is perhaps the most important parameter for determining how difficult the set will be for a particular algorithm. The best known algorithms have difficulty when the probability is near 0.5, but are fast when the probability is close to 0 or 1. We use formulas generated from the 3-SAT model as an example. Figure 8 shows, for 50 variables, the real execution results of the DP algorithm for 100 to 500 clauses. The computing time used by a program for the DP algorithm ([**118, 117, 357**]) is shown for the 3-SAT model (solid line) and the average 3-SAT model (dotted line) [**207, 212, 383, 109**]. Random formulas generated in the left region are usually satisfiable, and the procedure is fast. Random formulas in the right region are usually unsatisfiable, and the procedure is fast. For random formulas in the middle, many are satisfiable and many are unsatisfiable; the procedure is slow. Because the DP algorithm is a complete algorithm, it is able to verify satisfiability and unsatisfiability. So it gives results for random formulas in all three regions.

The results of the DP algorithm may not hold for a different SAT algorithm. A local search may often find a solution for a satisfiable *CNF* much more quickly than the DP algorithm but does not always verify satisfiability and cannot prove

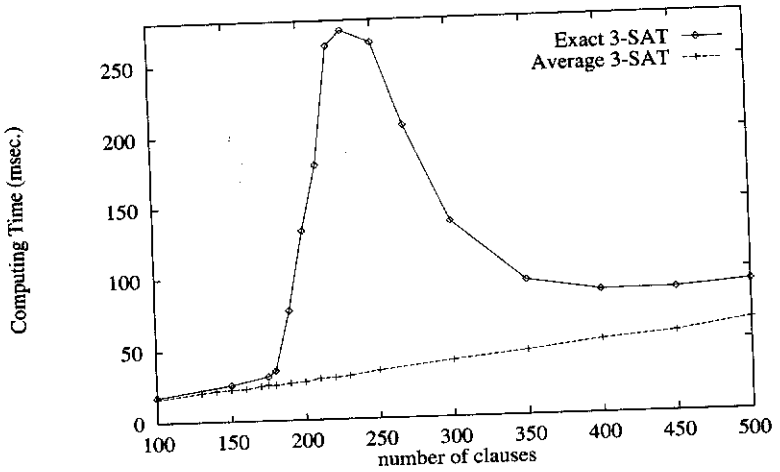


FIGURE 8. Computing time for the exact and the average 3-SAT models (with 50 variables) on a SUN SPARC 1 workstation. The horizontal axis is measured by m or m/n.

unsatisfiability. In particular, it gives no answer if a *CNF* formula is not satisfiable. Thus, for most formulas in the peak region and nearly all formulas in the right region, a local search algorithm will not terminate in a reasonable amount of time.

4.6. Performance Evaluation. The performance of an algorithm can be determined *experimentally* or *analytically*. It is feasible to do experimental studies with typical or random formulas, but not with worst-case formulas (there are too many formulas of a given size to experimentally determine which one leads to the worst-case time). It is feasible to do analytical studies with random or worst-case formulas but not with typical formulas (typical sets of formulas seldom have a mathematical structure suitable for analysis).

Experimental studies are sometimes inconclusive because they consider a relatively small number of input possibilities. Such restrictions are often forced because the space of likely input formulas, and even the size of such formulas, is so large. Analytical studies are intended to determine performance over broad families of inputs where each family typically represents a class of formulas of a particular size. However, such studies have the drawback that only the simplest of algorithms can be analyzed. To compensate for this, several features of a complex algorithm can be removed, leaving a rather simple, more analyzable one. The simplified algorithm usually contains one or two simple techniques, such as the unit-clause-rule, or the pure-literal-rule. An analytical result on the simplified algorithm provides a bound on the performance of the complex algorithm, and this bound is sometimes sufficient to understand the behavior of the complex algorithm. Such an approach has the following side benefit: analytical studies can suggest which simple techniques should be included in practical algorithms. In fact, most of the 6 prize winners of the 1991 SAT contest were associated with analytical studies of SAT algorithms

[69, 70]. The two top winners were associated with both experimental and analytical studies of SAT algorithms. The analytical studies of SAT algorithms involve the following.

1. Worst-Case Studies.

Unless P = NP, all SAT algorithms have a worst-case time that is superpolynomial in the input size [101]. A number of studies have concentrated on the worst-case analysis of variable setting algorithms for solving SAT [192, 385, 320].

2. Probabilistic Studies.

Since the typical performance of many satisfiability algorithms is much better than any proven worst-case results, there is considerable interest in evaluating the probabilistic performance of these algorithms. Such studies use some model for generating random formulas and then calculate the performance of algorithms on these formulas. The two most widely used measures of performance are average time and probabilistic time.

Average time is a weighted average of the time (or some related measure, such as the number of nodes) to solve a given sample of formulas. An algorithm must solve each formula for the average to be defined. In *probabilistic time* studies, an algorithm is given a deadline (usually specified as a polynomial in the length of input formulas), and one studies the fraction of formulas that are solved within the deadline. Probabilistic time studies can be performed on algorithms which give up on some fraction of the formulas so long as that fraction is less than the goal fraction.

For incomplete algorithms, the average time is not defined so only the fraction of inputs solved can be studied. One can also use various hybrid measures, such as the average time used to solve the easiest 90 percent of the inputs.

The literature contains a number of studies of the average time and probabilistic time performance of certain SAT algorithms [53, 168, 417, 167, 201, 202, 203, 269, 424, 432, 427]. Despite the worst-case complexity of SAT, algorithms and heuristics with polynomial average time complexities have been reported [82, 83, 165, 425, 429, 430, 431, 552]. This subject is treated in more detail in Section 12.

3. Number of Solutions.

Some researchers investigated the number of solutions of random SAT formulas. Extending Iwama's work [281] Dubois gave a combinatorial formula computing the number of solutions of any set of clauses [148]. Dubois and Carlier also studied the mathematical expectation of the number of solutions for a probabilistic model [149].

During the past two decades, many performance studies were performed through sampling techniques [313, 424, 501],[1] experimental simulations [54, 191, 241],

[1]Knuth [313] first showed how to measure the size of a backtrack tree by repeatedly following random paths from the root. Purdom [424] gave a modified version of Knuth's algorithm which greatly increases the efficiency of Knuth's method by occasionally following more than one path from a node. Stone and Stone [501] presented a variant of the algorithms of Knuth and Purdom for estimating the size of the unvisited portion from the statistics of the visited portion.

analytical studies [**53, 176, 177, 432, 425, 429, 500**], as well as the combined effort of the above approaches [**191, 266, 313, 424, 500**].

5. SAT Input Models

In this section, we describe several basic SAT input models and their characteristics.

5.1. Random Input Models. The running time of a SAT algorithm depends on the type of input being solved. The following SAT input models are often used to generate a variety of input types.

- **Hardest formulas.** Generate that formula that is the most difficult for the algorithm being measured. This approach is often used for analytical studies. There are too many possible formulas to use this approach in experimental studies. Bugrara and Brown [**65**] reported the effects these minor variations have on the average time needed by the simple backtracking algorithm.

  Experimental studies sometimes include results for the hardest formulas from the set of formulas tested, but such results are quite different from what the results would be if the entire set of possible formulas had been tested.

Most analytical studies use the following two basic models to generate random *CNF* formulas. Each model has several variations depending on whether identical clauses are permitted, whether a variable and its negation can occur in a clause, etc.

- **The l-SAT model.** In the l-SAT model, a randomly generated *CNF* formula consists of m independently generated random clauses. Each clause is chosen uniformly from the set of all possible clauses of exactly l literals that can be composed from a variable set $X = \{x_1, \ldots, x_n\}$ such that no two literals are equal or complementary. The number of possible clauses is $2^l \binom{n}{l}$. This model is sometimes called the *fixed-clause-length* model. Similar models were used in [**53, 82, 83, 168, 417, 212, 213, 383, 376, 429**].
- **The average l-SAT model.** In the average l-SAT model, a randomly generated *CNF* formula consists of m independently generated random clauses. In each clause, each of n variables occurs positively with probability $p(1-p)$, negatively with probability $p(1-p)$, both positively and negatively with probability p^2, not at all with probability $(1-p)^2$, where p can be a function of m and n. The average number of literals in a clause is $l = 2pn$. This model is also called the *random-clause-length* model. This model and variations were used in [**165, 166, 203, 209, 212, 213, 259, 260, 303, 301, 479**].

Most papers use just one model, but the performance of simple backtracking has been considered under a number of related models [**65**].

5.2. Hardness. Various SAT algorithms differ greatly in the amount of time they need to solve particular inputs. For example, Iwama's algorithm [**281**] is fast for random formulas with lots of solutions and slow for random formulas with few solutions, while simple backtracking [**431**] is fast on formulas with few solutions and slow on formulas with many solutions. Therefore, the *hard-and-easy distributions* of SAT formulas depend not only on the inherent property of the SAT input models but also on the algorithms used to solve the formulas. Any particular SAT formula is easy for some algorithm (for example a table lookup algorithm with that formula

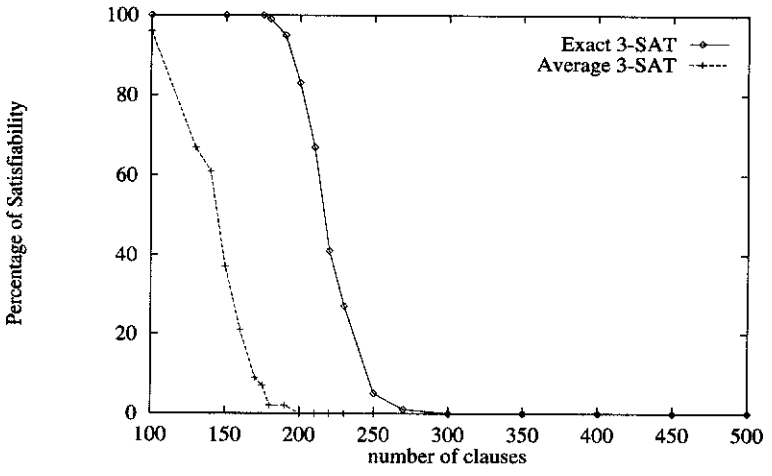


FIGURE 9. Percentage of satisfiability for formulas with 50 variables generated by the 3-SAT and the average 3-SAT input models, respectively. The horizontal axis is measured by m or m/n.

in its lookup table). Thus, hardness is a *property* of large sets of formulas rather than individual formulas.

For sets of formulas generated by random models with parameters, the probability of finding a solution varies with the parameter settings. Those sets generated with parameters set in regions where solutions are going from unlikely to common are particularly difficult for all algorithms that have been studied (see Figure 9).

For random l-SAT formulas fewer literals and larger number of clauses reduce the possibility of making all clauses jointly satisfiable. Therefore the computing time for random l-SAT formulas increases, up to a point, when m/n increases or the number of literals l ($l > 3$) in each clause decreases (Figure 8). Inspection of Figure 8 reveals a "hump" of difficulty for l-SAT formulas where 50% of the sample space is satisfiable, but a "flat" increase in difficulty for random l-SAT formulas in a correspondingly similar region of the parameter space.

5.3. Comparison of Random Input Models. The structural properties of random formulas generated by the two input models given above can be quite different and this can have a significant impact on the performance of a complete SAT algorithm. This significance is felt especially in the region of the parameter space for which random formulas are nearly equally likely to be satisfiable or unsatisfiable. Figure 8 shows, for 50 variables, the actual computing time of a complete SAT ($SAT14.11$) algorithm[2] for random formulas generated from the 3-SAT model and the average 3-SAT model [**207, 212, 214, 213**]. Figure 9 shows the percent of random formulas that are satisfiable as a function of formula size for both models. For a complete algorithm, the problem instances generated from the average 3-SAT model is much easier than those generated from the 3-SAT model. It takes

[2] $SAT14.11$ is a backtracking algorithm combined with coordinate descent in the real space [**213**].

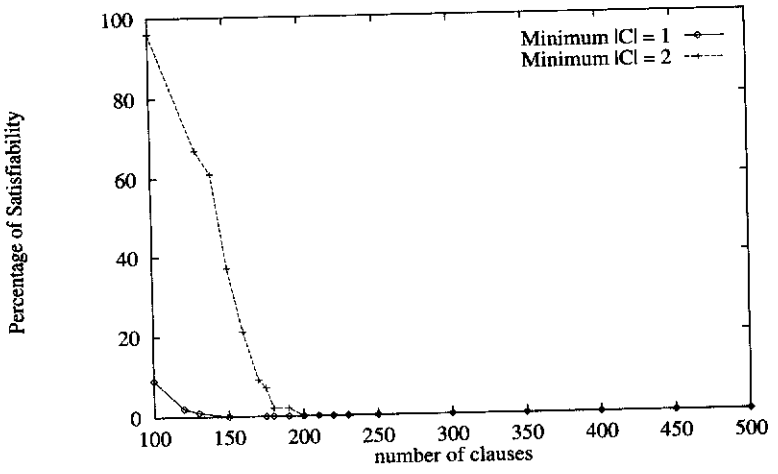


FIGURE 10. Percentage of satisfiability for two average 3-SAT problem models with $|C|_{min} = 1$ and $|C|_{min} = 2$ (50 variables), respectively. Problem instances generated with smaller length of the shortest clauses have much lower percentage of satisfiability.

a complete algorithm much less computing time to solve formulas generated from the average 3-SAT model.

For an incomplete algorithm such as local search, however, the situation is different. In Figure 9, for the same number of clauses (i.e., the same m/n values), problem instances generated from the average 3-SAT model have much lower percentage of satisfiability (compared to those generated from the 3-SAT model). The sat-and-unsat boundary of the average 3-SAT model is shifted to the left and is drawn by smaller m/n values than those for the 3-SAT problem model. For the same m/n values, more problem instances generated from the average 3-SAT model are unsatisfiable, making it harder for a local search algorithm to handle the average 3-SAT problem model. Experimental results confirmed that it took a local search algorithm (*SAT1* for example) much longer time to solve problem instances generated from the average l-SAT models [**209**].

Many factors can affect the property of the random models significantly. For the same average 3-SAT problem model even a slight variation to the length of the shortest clause in a *CNF* formula would significantly shift the sat-and-unsat boundary. In Figure 10, the solid curve was generated from an average 3-SAT model. The length of the shortest clause in the model was 1. The dotted curve was generated from the same average 3-SAT model but the length of the shortest clause was set to 2. Clearly, shorter clauses enforce tighter constraints and generate much harder random instances for the same model.

Incomplete algorithms that fail on unsatisfiable inputs can be effective only in the half-planes $m/n < 2^l/l$ for the l-SAT model and $pn > \ln(n)$ for the average l-SAT model, where the probability that a random formula is satisfiable is high (see Section 12). Incomplete algorithms that fail on satisfiable inputs can be effective only in regions complementary to those above.

Experience with the best complete algorithms has caused some to conclude the following:

1. Average l-SAT formulas are easy for the best algorithms;
2. l-SAT formulas are difficult even for the best algorithms; and
3. Formulas generated by both models are of similar difficulty when the average clause length is large.

Obviously, there are some conflicts in these beliefs.

5.4. Practical Input Models. Random input models such as those discussed above are suitable for analytical studies of SAT algorithms because they generate formulas which possess a symmetry that can be exploited for analysis. Actual formulas often have a different structure. Therefore, structured problem instances and practical SAT applications are essential to evaluate the performance of SAT algorithms. Examples of these are the following:

- **Regular SAT models.** Models derived from problems such as graph coloring and n-queens, are used to assess the performance of SAT algorithms [**211, 213**].
- **Practical applications problems.** Models derived from practical application domains, such as integrated circuit design, mobile communications, computer architecture and network design, computer-aided manufacturing, and real-time scheduling, have a variety of special characteristics (see Section 14).

Some experiments strongly suggest that there is little correlation between the performance of a SAT algorithm tested through random input models and the performance of the same algorithm tested through practical input models. Local search is faster for some random inputs but can be slower than a complete SAT algorithm for problems raised from practical applications. The boundary phenomenon discussed in random models is an artifact of some probabilistic models. It has not yet been observed in practical input models.

Practical applications are ultimately the *most important*, although it is difficult for people outside the area of application to understand how important or difficult a particular application problem is. It is also difficult to develop a general theory on the speed of SAT algorithms on applications. Much research is, therefore, done on the more regular source of problems in the hope of better understanding the speed that SAT algorithms will have when applied to a wide range of practical applications.

6. Splitting and Resolution

Recursive replacement of a formula by one or more other formulas, the solution of which implies the solution of the original formula, is an effective paradigm for solving *CNF* formulas. Recursion continues until one or more primitive formulas have been generated and solved to determine the satisfiability of the original. One way to achieve this is through splitting.

In splitting, a variable v is selected from a formula, and the formula is replaced by one sub-formula for each of two possible truth assignments to v. Each sub-formula has all the clauses of the original except those satisfied by the assignment to v and otherwise all the literals of the original formula except those falsified by the assignment. Neither sub-formula contains v, and the original formula has a

satisfying truth assignment if and only if either sub-formula has one. Splitting insures that a search for a solution terminates with a result.

Another effective paradigm is based on resolution [**449**]. In resolution, a variable v is selected and a resolvent (see below) obtained using v is added to the original formula. The process is repeated to exhaustion or until an empty clause is generated. The original formula is not satisfiable if and only if an empty clause is a resolvent. Although there is only one new formula on each step, the total number of steps (or resolvents) can be extremely large compared to the number of clauses in the original formula. Many algorithms that use resolution form all possible resolvents using a particular variable at one time. When this is done, the original clauses that contain the variable and its negation may be dropped. An algorithm may use both splitting and resolution.

Early examples of these approaches are the two forms of the Davis Putnam procedure. The original DP procedure used resolution [**118**] while the revised version, *i.e.*, the *Davis-Putnam-Loveland* (DPL) *procedure*, used splitting [**117, 357**]. Combining splitting with depth-first search in the DPL procedure avoids memory explosion that occurs on many inputs when they are solved by the original DP procedure.

Most recursive SAT algorithms use the following primitive conditions to stop the recursion:

1. formulas with an empty clause have no solution.
2. formulas with no clauses have a solution.
3. formulas with no variables (*i.e.*, all variables have been assigned values) are trivial.

The following subsections present various SAT algorithms, organized by the basic approach that each algorithm takes. Some of these algorithms are much simpler than you would want to use in practice but are of interest because it has been possible to analyze their running time for random formulas.

6.1. Resolution. Given two clauses $C_1 = (v \vee x_1 \vee ... \vee x_{l_1})$ and $C_2 = (\bar{v} \vee y_1 \vee ... \vee y_{l_2})$, where all x_i and y_j are distinct, the resolvent of C_1 and C_2 is the clause $(x_1 \vee ... \vee x_{l_1} \vee y_1 \vee ... \vee y_{l_2})$, that is, the disjunction of C_1 and C_2 without v or $\bar{v}$. The resolvent is a logical consequence of the logical *and* of the pair of clauses. Resolution is the process of repeatedly generating resolvents from original clauses and previously generated resolvents until either the null clause is derived or until no more resolvents can be created [**449**]. In the former case (a refutation) the formula is unsatisfiable and in the latter case it is satisfiable.

For some formulas the order in which clauses are resolved can have a big effect on how much effort is needed to solve it. The worst-case associated with the best possible order (the order is selected after the formula is given) has received considerable study [**181, 514, 232, 522**]. These studies all used formulas that have no solution, but where this is not obvious to the resolution algorithm. Eventually a much stronger result was shown: nearly all random l-SAT formulas need exponential time when the ratio of clauses to variables is above a constant (whose value depends on l) [**86**]. The constant is such that nearly all of the formulas in this set have no solution.

A number of restrictions and at least one extension to resolution have been proposed and applied to *CNF* formulas. Restrictions aim to shorten the amount of time needed to compute a resolution derivation by limiting the number of possible

resolvents to choose from at each resolution step. The extension aims to provide shorter derivations than possible for resolution alone by adding equivalences which offer more clauses on which to resolve. A nice treatment of these refinements can be found in [**63**], Chapter 4. We mention here a few of these.

Set of Support [**551**]. Split a given formula into two sets of clauses $\mathcal{F}_1$ and $\mathcal{F}_s$ such that $\mathcal{F}_1$ is satisfiable. Permit only resolutions involving one clause either in $\mathcal{F}_s$ or an appropriate previous resolvent. Set $\mathcal{F}_s$ is called the support set. This restriction can be useful if a large portion of the given formula is easily determined to be satisfiable.

P- and N-Resolution. If one of the two clauses being resolved has all positive literals (resp. negative literals), then the resolution step can be called a P-resolution (resp. N-resolution) step. In P-resolution (resp. N-resolution) only P-resolution (resp. N-resolution) steps are used. Clearly there is great potential gain in this restriction due to the usually low number of possible resolvents to consider at each step. However, it has been shown that some formulas solved in polynomial time with general resolution require exponential time with N-resolution.

Linear Resolution. We have linear resolution if every resolution step except the first involves the most recently generated resolvent (the other clause can be a previous resolvent or a clause in the given formula). Depending on the choice of initial clause and previous resolvents it is possible not to complete a refutation.

Regular Resolution[**514**]. In every path of a resolution tree no variable is eliminated more than once.

Davis-Putnam Resolution. Once all the resolvents with respect to a particular variable have been formed, the clauses of the original formula containing that variable can be dropped. Doing this does not change the satisfiability of the given formula, but it does change the set of solutions to the extent that the value of that variable is no longer relevant. When dropping clauses, it is natural to first form all the resolvents for one variable, then all the resolvents for a second variable, and so on. When doing resolution in this way, it is easy to find one satisfying assignment if the formula is satisfiable. At the next to last step the formula has just one variable, so each value can be tested to see which one satisfies the formula (perhaps both will). Pick a satisfying value and plug it into the formula for the next step, converting it into a one variable formula. Solve that formula and proceed in this manner until an assignment for all variables is found.

Extended Resolution [**514**]. For any pair of variables a, b in a given formula $\mathcal{F}$, create a variable z not in $\mathcal{F}$ and append the following expression to $\mathcal{F}$: $(z \vee a) \wedge (z \vee b) \wedge (\bar{z} \vee \bar{a} \vee \bar{b})$. Judicious use of such extensions can result in polynomial size refutations for problems that have no polynomial size refutations without extension.

The following strategies help reduce the time to compute a resolution derivation.

Subsumption. If the literals in one clause are a subset of those in another clause, then the smaller clause is said to *subsume* the larger one. Any assignment of values to variables that satisfies the smaller clause also satisfies the larger one, so the larger one can be dropped without changing the set of solutions. Subsumption is of particular importance in resolution algorithms because resolution tends to produce large clauses.

Pure Literals. A literal is *pure* if all its occurrences are either all positive or all negative. No resolvents can be generated by resolving on a pure literal, but all clauses containing a pure literal can be removed without loss. An important improvement to the basic resolution algorithm is to first remove clauses containing pure literals (before resolving on non-pure literals) [**118**].

Although resolution can be applied to SAT, the main reason for interest in resolution is that it can be applied to the more difficult problem of solving sentences of first order predicate logic. There is a vast literature on that subject. Bibel has a good book on the topic [**32**].

6.2. Backtracking. Backtracking algorithms are based on splitting. During each iteration, the procedure selects a variable and generates two sub-formulas by assigning the two values, *true* and *false*, to the selected variable. In each sub-formula, those clauses containing the literal which is *true* for the variable assignment are erased from the formula, and those clauses which contain the literal which is *false* have that literal removed. Backtrack algorithms differ in the way they select which variable to set at each iteration. The unit clause rule, the pure literal rule, and the smallest clause rule, are three most common ones. We state each algorithm informally.

The flow of control in splitting-based algorithms is often represented by a search tree. The root of the tree corresponds to the initial formula. The internal nodes of the tree correspond to sub-formulas that cannot be solved directly, whereas the leaf nodes correspond to sub-formulas that can be solved directly. The nodes are connected with arcs that can be labeled with variable assignments.

Simple Backtracking [53]. If the formula has an empty clause (a clause which always has value *false*) then exit and report that the formula has no solution. If the formula has no variables, then exit and report that the formula has a solution. (The current assignment of values to variables is a solution to the original formula.) Otherwise, select the first variable that does not yet have a value. Generate two sub-formulas by assigning each possible value to the selected variable. Solve the sub-formulas recursively. Report a solution if any sub-formula has a solution, otherwise report no solution.

Unit Clause Backtracking [425]. This algorithm is the same as simple backtracking except for how variables are selected. If some clause contains only one of the unset variables then select that variable and assign it a value that satisfies the clause containing it; otherwise, select the first unset variable.

In practice, this improved variable selection often results in much faster backtracking [**34**].

Clause Order Backtracking [64]. This algorithm is the same as simple backtracking except for how variables are selected. If this setting does not solve the formula, then select the first clause that can evaluate to both *true* and *false* depending on the setting of the unset variables. Select variables from this clause until its value is determined.

By setting only those variables that affect the value of clauses, this algorithm sometimes avoids the need to assign values to all the variables. The algorithm as stated finds all the solutions, but in a compressed form. The solutions come in cylinders, where some variables have the value "don't care." Thus, a single solution with unset variables represents the set of solutions obtained by making each possible assignment to the unset variables.

Probe Order Backtracking [433]. This algorithm is the same as simple backtracking except for how clauses are selected. Temporarily set all the unset variables to some predetermined value. Select the first clause that evaluates to *false* with this setting. Return previously unset variables back to unset and continue as in clause order backtracking.

For practical formulas one should consider adding the following five refinements to probe order backtracking: stop the search as soon as one solution is found, carefully choose the probing sequence instead of just setting all variables to a fixed value [**348, 487, 491**], probe with several sequences at one time [**69, 70**], carefully select which variable to set [**69, 70**], use resolution when it does not increase the input size [**166**]. The sixth best prize winning entry in the 1992 SAT competition used an improvement on probe order backtracking [**70**].

Franco [**165**] noticed that a random assignment solves a nonzero fraction of the formulas in the average l-SAT model when pn is large compared to $\ln m$. Simple uses of that idea does not lead to good average time [**433**], but combining the idea with clause order backtracking leads to probe order backtracking, which is fast when pn is above $\ln m$. Probe order backtracking appears to have some similarities to one method that humans use in problem solving in that it focuses the algorithm's attention onto aspects of the problem that are causing difficulty, *i.e.*, setting variables that are causing certain clauses to evaluate to *false*. For the same reason it is somewhat similar to some of the incomplete searching algorithms discussed in Section 7.

Shortest Clause Backtracking. This algorithm is the same as clause order backtracking except for the clause selected. In this case, select the shortest clause.

The corresponding idea for constraint satisfaction is to first set a variable in the most constraining relation. This idea is quite important in practice [**34**].

Jeroslow-Wang [284]. A backtrack search can sometimes be terminated early by checking whether the remaining clauses can be solved by a Linear Programming relaxation (see Sections 9.2 and 9.3). An implementation of this idea can be expensive. Jeroslow and Wang have proposed a simpler and effective technique that is similar in spirit. The idea is, before splitting, to apply a procedure that iteratively chooses the variable and value which, in some sense, maximizes the chance of satisfying the remaining clauses. The procedure does not backtrack and is, therefore, reasonably fast. Assignments determined by the procedure are temporarily added

to the current partial truth assignment. If the procedure succeeds in eliminating all clauses then the search is terminated and the given formula is satisfiable. Otherwise, the procedure fails, control is passed to the split, temporary assignments are undone, and backtracking resumes.

The choice of variable and value at each iteration maximizes the weight $w(\mathcal{S}_{i,j})$ where, for a subset of clauses $\mathcal{S}$, $w(\mathcal{S}) = \sum_{C \in \mathcal{S}} 2^{-|C|}$, and for $i \in \{0,1\}$, $1 \leq j \leq n$, $\mathcal{S}_{i,j}$ is the subset of remaining clauses containing variable v_j as a positive literal if $i = 0$ and as a negative literal if $i = 1$. The length of clause C, denoted $|C|$ above, is the number of literals that are not falsified by the current partial assignment and the sum is over clauses that are not satisfied by the current partial assignment. The weight given above may be compared to that given by Johnson in [**286**] (see also, *Other Non-Backtracking Heuristics* below).

6.3. Backtracking and Resolution. Some algorithms have adapted ideas inspired by resolution to splitting algorithms. For example, from the resolution view point, pure literals are interesting in that they lead to a single sub-formula that is no more complex than the original formula, while from the perspective of splitting, pure literals lead to two sub-formulas, but the solutions to the sub-formula where the literal has the value *false* are a subset of the one where the literal has the value *true*. Therefore, the original formula has a solution if and only if the formula associated with the *true* literal does.

The Pure Literal Rule Algorithm [201]. Select the first variable that does not have a value. (If all variables have values, then the current setting is a solution if it satisfies all the clauses.) If some value of the selected variable results in all clauses that depend on that variable having the value *true*, then generate one sub-formula by assigning the selected variable the value that makes its literals *true*. Otherwise, generate a sub-formula for both values of the selected variable. Solve the one or two sub-formulas.

6.4. Clause Area. A clause with l distinct literals leads to the fraction $1/2^l$ of the possible variable settings not being solutions. One can think of the clause as blocking out area $1/2^l$ on the Venn diagram for the formula. Iwama showed that combining this idea with inclusion-exclusion and careful programming leads to an algorithm which runs in polynomial average time when $p > \sqrt{(\ln m)/n}$ [**281**]. If the sum of the area of all clauses is less than 1, then some variable setting leads to a solution. This idea works particularly well with shortest-clause backtracking since that algorithm tends to eliminate short clauses. See [**170**] for a probabilistic analysis of this idea. No average-time analysis has been done.

6.5. Improved Techniques for Backtracking. This section considers some refinements that can be added to the basic backtracking and resolution techniques. Several of these are similar to techniques that have already been discussed.

Branch Merging. This is complementary to preclusion. Backtracking is frequently used on problems such as the n-queens problem where there is a known symmetry group for the set of solutions. In such cases many search trees possess equivalent branches which can be merged to reduce search effort [**34, 547**]. The use of the symmetry group can greatly speed up finding the solutions. See [**72, 73**] for examples from the field of group theory. Brown, Finklestein, and Purdom [**51, 52**]

gave additional problems that arise in making the backtracking techniques work with a backtracking algorithm which needs to set variables in different orders on different branches of the search tree.

Search Rearrangement. This is also known as *most-constrained search* or *nonlexicographic ordering search.* When faced with several choices of extending a partial solution, it is more efficient to choose the one that offers the fewest alternatives [**34**]. That is, nodes with fewer successors should be generated early in the search tree, and nodes with more successors should be considered later. The vertical (variable) ordering and horizontal (value) ordering are special cases of search rearrangement [**54, 176, 425, 429, 432, 501**]. The rule used to determine which variable to select next is often called the branching rule. Many researchers are actively investigating the selection of branching variables in the DP procedures. Hooker studied the branching rule and its effect with respect to particular problem instances [**256**]. Böhm and Speckenmeyer experimented with branching effect with a parallel DP procedure implemented on an MIMD machine [**38**]. Boros, Hammer, and Kogan developed branching rules that aim at the fastest achievement of q-Horn structures [**41**]. Several particular forms of search rearrangement were discussed in Section 6.2.

From 2-SAT to General SAT. In many practical applications, the constraints in the problems are coded as 2-SAT formulas. In SAT problem formulation, very frequently in practical applications, many of the constraints will be coded as 2-SAT clauses.

An important heuristic to SAT problem solving is to first solve 2-SAT clauses with fast polynomial time algorithms. This fast operation can significantly reduce the search space. The truth assignment to the rest of the variables can be handled with a DP procedure. This idea has been used in SAT solver *Stamm* [**69, 70**], Gallo and Pretolani's 2-SAT relaxation [**69, 70, 422**], and Larrabee's algorithm [**328, 477**]. Similar ideas to solving 2-SAT clauses were developed. Eisele's SAT solver uses a weighted number of occurrences whereas occurrences in 2-SAT clauses count more than other occurrences [**69, 70**]. Dörre further added a limited amount of forward checking to quickly determine 2-SAT formulas in the Eisele-Dörre SAT solver [**69, 70**]. In the SAT contest [**69, 70**] the winning programs with 2-SAT solvers were slightly slower than those without.

Similar techniques were developed that use Horn-SAT relaxation in satisfiability testing [**108, 183**]. In Crawford's Tableau [**108**], Horn clauses are separated from non Horn clauses. Based on the DPL procedure, Tableau applies in priority the unit clause rule and if necessary branches on a variable selected in the non Horn clauses using three successive heuristics.

Backmarking and Backjump. When a failure is observed or detected, the algorithm simply records the source of failure and jumps back to the source of failure while skipping many irrelevant levels on the search tree [**189, 191**]. The more effective one's search rearrangement is, the less need there is for backjumping. Good search orders tend to be associated with the source of failure being one level back.

Backtracking with Lookahead. A lookahead processor is a preprocessing filter that prunes the search space by inconsistency checking [**226, 227, 361, 390**]. Backtracking with lookahead processing is performed by interplaying a depth-first tree traversal and a lookahead tree pruning processor that deletes nodes on the search tree whose value assignments are *inconsistent* with those of the partial search path. Techniques in this class include partial lookahead, full lookahead [**226, 227, 241**], forward checking [**241, 241**], network-based heuristics [**361, 128**], and discrete relaxation [**226, 227, 312, 451**].

Backtracking for Proving Non-Existence. Dubois, Andre, Boufkhad, and Carlier have recently proposed a complete SAT algorithm, *CSAT* [**151**]. The *CSAT* was developed for the proof of the non-existence of a solution. The algorithm uses a simple branching rule and a local processing at the nodes of search trees (to detect further search path consistency and make search decision). It performed efficiently on some DIMACS benchmarks.

Intelligent Backtracking. This is performed directly to the variable that causes the failure, reducing the effect of thrashing behavior. Methods in this category include dependency-directed backtracking [**496, 145**], revised dependency-directed backtracking [**414**], simple intelligent backtracking [**178**], and a number of simplifications [**56, 119, 120, 121, 123, 125, 190, 241, 452**].

Freeman [**175**] recently present an intelligent backtracking algorithm, *POSIT*, for PrOpositional SatIstiability Testbed. In this algorithm he used Mom's heuristic, detecting failed literals, and minimizing constant factors to speed up backtracking search.

Some effort was devoted to the development of backtracking-oriented programming languages, special-purpose computer architectures, and parallel processing techniques:

Macro Expansion. In some applications of backtracking that require relatively little storage, this method can be used to decrease the running time of the program by increasing its storage requirements. The idea is to use macros in assembly language in such a way that some work is done at assembly time instead of many times at run time. This increases the speed at which nodes are processed in the tree [**34**].

Backtrack Programming. Much work has focused on developing a new programming language for backtracking search. This includes the sequential[3] Prolog programming language [**94, 499**], Prolog with intelligent backtracking scheme [**58, 323, 413**], and logic programming [**248**].

Special-Purpose Architectures. Special-purpose hardware machines were built to prune search space [**226, 227, 374**], perform backtracking search, and do AI computations [**531, 532, 534**].

[3]There is no backtracking mechanism in parallel Prolog programming languages.

Parallel Processing. Many parallel processing techniques have been developed to speed up search computation [**23, 99, 531, 343, 535, 344, 342, 250, 353, 517, 372, 461, 532, 550**].

Branch and bound. Also known as ordered depth-first search. Select a variable. For each possible value of the variable generate a sub-formula and compute some quick to compute upper bound on the quality of the solution of the sub-formula. Solve recursively all sub-formulas except those that have a cost above that of the best solution that has been found so far. Branch and bound is recognized as a generalization of many heuristic search procedures such as A^*, AO^*, SSS^*, B^*, alpha-beta, and dynamic programming algorithm [**6, 343, 556, 539, 537, 536, 328, 333, 341, 410, 266, 556, 555, 549, 554**].

6.6. Some Remarks on Complexity. The worst-case time for all known SAT algorithms is exponential in the first power of the input size. The naive algorithm that tries every variable setting requires time 2^n for n variable formulas. For l-SAT, the best known bound on worst-case complexity has been worked down from 1.618^n [**385**] to slightly below 1.5^n obtained by Schiermeyer [**464, 465**]. Other work on the topic is given in [**192**].

As with other NP-complete problems there are no exponential lower bound results for SAT. However, it has been proven that all resolution algorithms need time that is exponential in the first power of the input size [**232, 86, 522**]. No such lower bound analyses have been done on splitting-based algorithms.

For a comprehensive treatment of the complexity of propositional proofs, see a recent survey by Urquhart [**525**].

7. Local Search

Local search is a major class of discrete, unconstrained optimization procedures that can be applied to a discrete search space. Such procedures can be used to solve SAT by introducing an objective function that counts the number of unsatisfiable (*CNF*) or satisfiable (*DNF*) clauses and solving to minimize the value of this function [**207, 209, 212, 221, 403, 472**].

In this section, we summarize the basic framework, including a search space model, four essential components, and present ideas used in the early development of local search algorithms for the SAT problem. We then describe randomized local search, randomized local search with trap handling, and greedy local search in detail.

7.1. Framework. *Local search*, or *local optimization*, is one of the primitive forms of continuous optimization applied to a discrete search space. It was one of the early techniques proposed to cope with the overwhelming computational intractability of NP-hard combinatorial optimization problems.

There have been two major periods for the development of local search. Early greedy local search method was able to solve small size, unconstrained problems such as TSP [**351, 401**]. During the middle and late eighties, more powerful techniques (see Section 7.3) for randomized local search were developed. These randomized local search algorithm can handle large size, constrained problems such as CSP and SAT problems efficiently [**206, 209, 488, 491**].

Given a minimization (maximization) problem with objective function f and feasible region R, a typical local search procedure requires that, with each solution point $\mathbf{x}_k \in R$, there is a predefined *neighborhood* $N(\mathbf{x}_k) \subset R$. Given a current solution point $\mathbf{x}_k \in R$, the set $N(\mathbf{x}_k)$ is searched for a point $\mathbf{x}_{k+1}$ with $f(\mathbf{x}_{k+1}) < f(\mathbf{x}_k)$ ($f(\mathbf{x}_{k+1}) > f(\mathbf{x}_k)$). If such a point exists, it becomes the new current solution point, and the process is iterated. Otherwise, $\mathbf{x}_k$ is retained as a *local optimum* with respect to $N(\mathbf{x}_k)$. Then, a set of feasible solution points is generated, and each of them is "locally" improved within its neighborhood. To apply local search to a particular problem, one needs only to specify the neighborhood and the procedure for obtaining a feasible starting solution.

Local search can be efficient for two reasons. First, at the beginning of search, a full assignment is assigned to all the variables in the search space. Search efforts are focused on a single path in the search space. Second, local search refines for improvement within its local neighborhood using a testing for improvement and, if there is any improvement, takes an action for improvement. Since the objective function has a polynomial number of input numbers, both *testing* and *action* can be done efficiently. Little effort is needed to generate the next solution point. A major weakness of local search is that the algorithm has a tendency to get stuck at a locally optimum configuration, i.e., a local minimum.

Greedy local search pursues only paths where every step leads to an improvement, but this leads to a procedure that becomes stuck much more often than the randomized local search. Greedy local search procedure gets stuck in flat places as well as at local minima.

Many search techniques, such as statistical optimization [**74, 467**], simulated annealing [**308**], stochastic evolution [**457**], and conflict minimization [**206, 380, 485, 491**], are either local search or variations of local search. For most search problems encountered, in terms of computing time and memory space, local search often achieves many orders of magnitude of performance improvement over conventional techniques such as Branch-and-Bound [**209, 212, 436, 487, 491**].

7.2. A Three-Level Search Space Model. A large number of real experimental data suggest that there are several typical local minimum structures (see Figure 11). A *valley* and a *basin* are ideal cases that one can find a global minimum quickly. Local search and the related heuristics can handle a *terrace* and a *plateau* without much difficulty. The most difficult situation is a *trap* where a group of local minima is confined in a "well." The search process walks around the set of local minima periodically and cannot get away without special mechanism. In general there may be many traps in a search problem. The characteristics of a trap are closely related to the search algorithm, the objective function used, and the search space structure.

Further observations suggest that a search space may be roughly divided into several different levels, depending on the problem structures. A three-level search space structure was proposed during the development of the $SAT1.5$ algorithm (see Section 7.7) [**209, 219**]. An informal example of the model is given in Figure 12. In the model, a search space is roughly viewed in three levels: top level, middle level, and bottom level. The *top level* is the upper open portion of the search space with smoothing edges. Most optimization algorithms can descend quickly in the top level and thus perform quite well. The *middle level* is the middle portion of the search space where there are relatively "big mountain peaks." During the

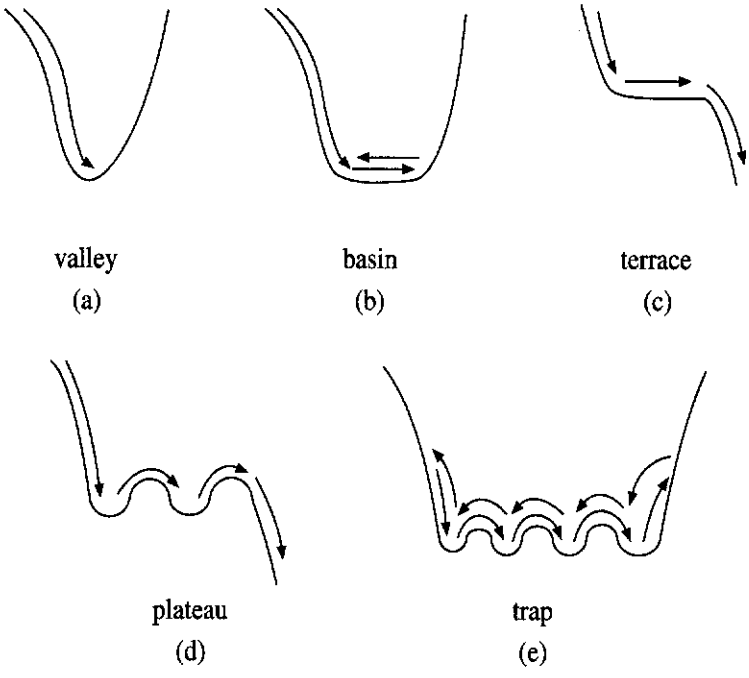


FIGURE 11. There are a number of local minimum structures. A trap is a "well" of local minima and is difficult to deal with.

descent, the search process may encounter problems and it may have to use some tunneling and random heuristics (see Section 7.3) to proceed. The *bottom level* is the bottom portion of the valleys (particular the lowest valley) where there are many traps. When local search falls into a trap it may become locked into a loop of local minima. Most algorithms do not succeed in this stage and have difficulty continuing.

For the SAT problem, with high probability, a greedy local search will fall into a trap much more easily. In this case some variables are updated very quickly. The related clauses oscillate between the *sat* and *unsat* states. The search is limited to these states. Without any help, there is little chance of getting out to explore other states.

The above observations suggest to use multiphase search to handle the NP-hard problems [**209, 223, 219**]. That is we may use an *open search* in the top level, a *peak search* for searching "coarse" peak structures in the middle level, and a *trap search* for tracking "fine" rugged trap surface structures in the valleys.

The major heuristics used in local search are discussed in the next subsection.

7.3. Four Components in Local Search. A number of efficient local search algorithms for the SAT problem have been developed since 1987. Previous experience indicated that the greedy local search strategy *alone* can not be adapted to perform well on SAT formulas. Past lessons showed that the following **four components** are crucial to the development of an efficient local search algorithm for

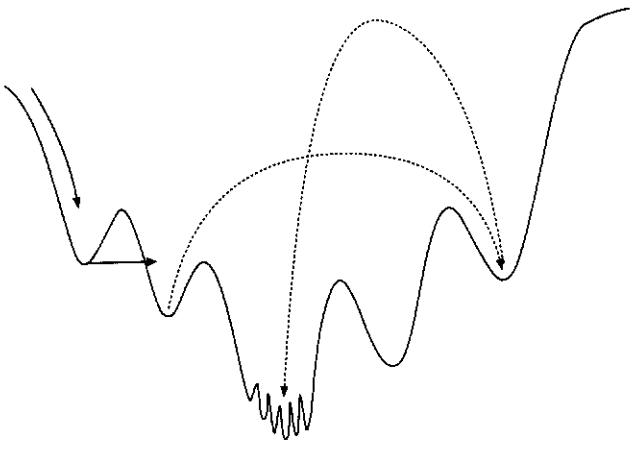


FIGURE 12. An informal example of the three-level search space model. A search process would go through an *open search* in the upper open portion of the search space, a *peak search* in the middle portion of the search space, and a *trap search* in the valley portion of the search space.

the satisfiability and NP-hard problems. They are: (1) the min-conflict heuristics, (2) the best-neighbor heuristics, (3) the random value/variable selection heuristics, and (4) the trap handling heuristics.

1. The Min-Conflicts Heuristics.

Different forms of min-conflict heuristics were proposed during 1985 and 1987 for solving the SAT and CSP problems [**206**].[4] The min-conflict heuristics aim at performing local conflict minimization in Boolean, discrete, and real spaces [**454**]: [5] Min-conflict heuristics are important to handle constraints in a constrained optimization problem.

Min-Conflict Heuristic (Boolean Space) [206]. Multiple values to be assigned to a variable are represented by a vector of Boolean labels. Each Boolean label, either "1" or "0," indicates the variable's instantiation to a specific value. Two labels are *conflicting* if their values do not satisfy the given constraint. The *conflicts* (due to an assignment) are formulated as a set of objective functions. The objective functions are minimized by changing values assigned to the labels.

Min-Conflict Heuristic (Discrete Space) [206]. Interrelated objects are chosen as variables. Two variables are *conflicting* if their values do not satisfy the

[4] In the early days *min-conflict* was variously called inconsistency removing, inconsistency resolution, conflict resolution, enforce local consistency, and local conflict minimization [**206**]. Later, Minton shortened these words into a concise term: min-conflicts.

[5] The min-conflict heuristics also work in real space (see examples in [**210, 211, 213**] and Section 8).

given constraint. The *number of conflicts* (due to an assignment) is formulated in an objective function. The objective function is iteratively minimized by changing values assigned to the variables.

Min-Conflict for SAT [207, 221, 209, 210, 212]. Using *inconsistency* as objective [**206**], the objective function for the SAT problem gives the number of unsatisfied clauses. A *CNF* is true *if and only if* the objective function takes the global minimum value 0 on the corresponding solution point.

This objective function is the basis of the design of the *SAT1*, *SAT2*, *SAT3*, and *GSAT* algorithms [**209, 210, 221, 212, 472**].

Performance. The min-conflict heuristics have been applied to solve the CSP and SAT problems since 1985 [**206, 207, 485, 484, 221, 209**]. They showed significant performance improvements when compared to traditional backtracking search algorithms. The effectiveness of min-conflicts heuristic was further observed by Russel and Norvig [**454**], Kumar [**321, 322**], Johnson [**295**], Minton *et al.* [**380**], and Selman *et al.* [**472**].

Early local search algorithm can only handle small size path-finding problem such as TSP. By introducing the min-conflicts heuristics, local search can now handle difficult constrained problems.

2. The Best-Neighbor Heuristics.

Local search proceeds by taking *any* feasible solution point that reduces the objective function. Among many neighboring feasible solution points, local search does not take into account its neighbors' relative performance with respect to the objective function.

Best-Neighbor Heuristic [209, 212, 213]. A *greedy* algorithm selects the *best neighbor* that yields the minimum value to the objective function and takes this *best neighbor direction* as the descent direction of the objective function.

In a real search space, continuous optimization algorithms can find the best neighbor feasible solution efficiently. A number of local and global optimization algorithms have been developed to solve the SAT problem [**209, 212, 213**]. The first version of the *GSAT* algorithm was proposed as a *greedy* local search algorithm [**472**].

Performance. A greedy local search *alone* may become stuck at local minima much more often and therefore may not be efficient in practice. Therefore, the best neighbor heuristic should be used in conjunction with random value/variable selection and trap handling heuristics described next.

3. The Random Value/Variable Heuristics.

Random value assignment and random variable selection techniques are fundamental to the design of an effective local search algorithm for NP-hard problems [**228, 224**].

Random Flip Heuristic [**206, 209, 221, 210, 212**]: Randomly flip the truth values of $1 \leq k \leq n$ variables in the SAT formula.

This simple heuristic was first implemented in several *SAT1* algorithms as *local handler*(s) in 1987. It has been proven to be effective in improving the performance of greedy local search algorithms [**209, 221, 210, 212**].

During 1988 to 1990 a similar heuristic, **random swap**, was used to develop local search algorithms for the CSP (e.g., n-queen) problems. It showed significant performance improvement for solving large-size n-queen problems [**206, 486, 487, 488, 491**].

Random Value (Assignment) Heuristics [**206, 209, 221, 212, 351, 401, 487, 488, 491**]. These include: randomly select a value that generates the minimum number of conflicts; randomly select a value if there is a symmetry (i.e., more than one value producing the same performance); and randomly select a value for conflict minimization when local minima are encountered.

A simple random value assignment heuristic, random disturbance, was early used in solving the TSP problem.

Random Variable (Selection) Heuristics [**206, 209, 221, 212**]. There are two important heuristics:

1. **Any Variable Heuristic:** select any variable randomly.
2. **Bad Variable Heuristic:** select a variable from the set of conflicting variables randomly.

The random variable selection heuristic is one of the most important heuristics in the design of local search algorithms for NP-hard problems. It was first used in the local search solution for the SAT problem [**207**] and then used for the local search solution for the CSP (e.g., n-queen) problems [**487**].

Conflicting variables in the SAT problem contribute to the unsatisfied clauses. Accordingly we have:

Bad Variable Heuristic for the SAT problem [**209, 212, 487, 488, 491, 486, 484**]: randomly select a variable in the unsatisfied clauses for conflict minimization.

The bad variable heuristic was first implemented to solve the large size n-queen problems during 1988 to 1990 [**487, 488, 491, 486, 484**] and was implemented in the *SAT2* algorithm in 1990 [**209, 212**]. The bad variable heuristic was independently developed by Papadimitriou for the 2-SAT problem in 1991 [**402**] and was used in the *WSAT* algorithm by Selman *et al.* in 1994 [**475**].

Partial/Pre- Random Variable Selection Heuristics [**206, 209, 221, 212**]. Partial variable random selection makes use of partial or alternating variable selection techniques. Variants of partial random selection include partial and alternating selection of conflicting and non-conflicting variables, a combination of partial deterministic and partial random variable selection, partial interleaved selection of the different search phases, and partial random selection with meta-heuristic control. The simplest selection strategies include: select a variable deterministically (randomly) and select another variable randomly for conflict minimization; select a

variable deterministically (randomly) from the set of conflicting variables and select another variable randomly for conflict minimization; select a variable deterministically and select another variable randomly from the set of conflicting variables for conflict minimization; during certain periods of search, select a variable deterministically (randomly) and select another variable randomly for conflict minimization; during certain periods of search, select a variable deterministically (randomly) from the set of conflicting variables and select another variable randomly for conflict minimization; during certain periods of search, select a variable deterministically and select another variable randomly from the set of conflicting variables for conflict minimization.

Partial Random Variable Selection Heuristics for the SAT problem: a variable may be selected from the unsatisfied clauses in a random, partially alternating, partially periodic, or partially interleaving order.

The partial and pre- random variable selection heuristics were implemented in the *SAT3* algorithm in 1990 [**209, 212**] and were used to solve the large size n-queen problems around 1990 [**487, 488, 491, 486, 484**]. A similar heuristic to the partial random variable selection, **random walk**, was developed by Selman, Kautz, and Cohen independently in 1994 [**475**].

Performance. Random and partial variable selection heuristics were introduced in the design of $SAT1$, $QS2$, $QS3$, and $QS4$ algorithms [**207, 221, 209, 212, 485, 487, 488, 491**]. They can overcome the weakness of the greedy local search algorithms. Compared to greedy local search, they can offer many orders of magnitude of performance improvements in terms of computing time, solving hard and large satisfiability problems and CSP problems with multi-million variables in seconds [**209, 212, 487, 491**]. They were used in the design of $SAT1.5$, $SAT2$, and $SAT3$ algorithms [**209, 212**].

Selman *et al.* have recently developed and applied a number of random variable selection heuristics to improve the performance of the greedy *GSAT* algorithm [**475**].

4. The Trap Handling Heuristics.

The search is a process of combating local minima. When the search process is approaching the final search stage, trap handling heuristics are needed to cope with local minima and traps (see Sections 7.2 and 7.7).

Tunneling Heuristic [**221, 212, 545**]: Change the value of a variable if it does not change the value of the objective function.

Tunneling Heuristic for the SAT Problem [**221, 212, 472**]: Flip the truth value of a variable if it does not change the value of the objective function (see Section 7.6).

Local Tracking Heuristics [**209, 219**]. Local tracking heuristics are used to track and break local loops (a periodic occurrence of a set of local minima). Several frequently used heuristics include: track local loop(s) when falling into a trap; give

low priority to flip to variables in a local minimum loop; give high priority to flip to variables that lead to a *new* descending direction; lock and release trapping variables periodically, adaptively, or statistically; move gently in a trap to handle fine local structures; move strongly in a trap to handle coarse local structures; jump out of a trap if walking inside it sufficiently long.

Multiphase Search Heuristics [**207, 209, 488, 491, 212, 436, 545, 219**]. Multiphase heuristics are a part of multispace search heuristics [**222, 228, 224**]. They have been developed to adapt to the different phases of a search process: perform a poor initial search and then a serious local search for conflict minimization; perform a good initial search and then a serious local search for conflict minimization; perform a good initial search, then a rough local search, and a serious local search for conflict minimization; perform an initial search, and then a rough local search and a serious local search alternatively for conflict minimization; perform a rough initial search, then a coarse local search, and finally, a fine local search for conflict minimization.

Multispace Search Heuristics [**222, 228, 224**]. Structural multispace operations have been developed that empower a search process with an information flux which is derived from a sequence of stepwise structural transformations (see Section 11.4). These include multispace scrambling, extradimension transition, search space smoothing, multiphase search, local to global passage, tabu search, and perturbations. They can disturb the environment of forming local minima and facilitate efficient local search when there are many local minima.

Performance. Trap handling heuristics have significantly improved the search efficiency of the *SAT1.5* algorithm [**209, 212, 219**] (see Section 7.7). Multiphase and multispace search heuristics have been applied to a variety of practical applications and found to be effective [**222, 209, 212, 224, 219, 436, 545, 488, 491**].

7.4. Boolean Local Relaxation. Boolean local relaxation may be viewed as a deterministic local search. It was an early inconsistency relaxation technique developed for solving the constraint satisfaction and satisfiability problems. For a variable having m values, m Boolean labels are used to indicate the variables' instantiation to the particular Boolean values. The conflicts produced by an assignment are coded in a set of Boolean objective functions (one for each label). The objective function for the ith variable and kth label, $f_{i,k}$, is defined as [**206**]:[6]

$$f_{i,q} = \prod_{j=1}^{n} \sum_{p=1}^{m} l_{i,q} \wedge C_{i,j}(q,p) \wedge l_{j,p}, \tag{7.1}$$

where $C_{i,j}(q,p)$ is a constraint between labels $l_{i,q}$ and $l_{j,p}$. Note that the right hand side of Eq. (7.1) is a *CNF* formula with *extended literals*.

The Boolean relaxation is a local conflict minimization process (Figure 13) [**206**]. During each iteration, the algorithm checks each variable for every label and iteratively minimizes the objective functions by flipping bits (truth values) assigned to the labels: If the objective function does not change, keep it; If the objective function can be reduced, keep the best (i.e., update the label), and then report

[6] In [**206**] the objective function was defined for label $l_{i,k}$ directly.

```
procedure DRA()
boolean inconsistency;
begin
  inconsistency := TRUE;
  k := 0;
  while inconsistency = TRUE do
  begin
    inconsistency := FALSE;
    for variable i := 1 to n
      for label q := 1 to m
      begin
        f_{i,q}^{k+1} := evaluate_objective_function(l, C);
        /* local conflict minimization */
        if f_{i,q}^{k+1} = f_{i,q}^{k} then continue;
        if f_{i,q}^{k+1} < f_{i,q}^{k} then update label value;
          inconsistency := TRUE;
      end;
    k := k + 1;
  end;
end;
```

FIGURE 13. **DRA:** A local relaxation algorithm.

the *inconsistency* status. The iteration will terminate once the *inconsistency* signal turns off.

The Boolean local relaxation algorithm was suitable to VLSI implementation. During 1985 to 1988, several parallel algorithms and architectures, such as DRA2, DRA3, and DRA5 were implemented to speed up CSP/SAT computations [**206, 226, 544, 227**]. Furthermore they were combined with backtracking search for CSP/SAT applications [**206**].

Because of its iterative local conflict minimization and its direct applications to SAT/CSP, Boolean local search made itself a *predecessor* of several early local search algorithms for CSP and SAT problems.

7.5. Constraint Satisfaction, Simulated Annealing, and Complexity Study. Early work on constraint satisfaction, simulated annealing, and complexity theory contributed significantly to the original development of local search algorithms for the SAT problem. Four notable early developments are: (1) the *SAT1* algorithms, (2) the n-queen models and algorithms for scheduling applications, (3) a simulated annealing algorithm, and (4) a 2-SAT algorithm.

1. The *SAT1* Algorithms.

Objective functions in DRA algorithms were defined for Boolean labels. During the late eighties, Gu [**206**] observed that if the *conflicts* of all the Boolean objective functions were formulated in *one* objective function, then the global minimum of the objective function would correspond to a conflict-free solution of the given

CSP problem. Accordingly, the iterative local minimization procedure used in the Boolean relaxation would become a local search procedure to minimize the objective function. This idea led directly to the early design of *SAT1* algorithms where the objective function was defined as the number of unsatisfied clauses over *all* the variables [**206, 207**]. Thus, the global minimum of the objective function corresponds to the solution of the SAT problem. Following this, Gu developed a number of randomized local search algorithms for the SAT problem. Furthermore efficient heuristics (Section 7.3) were developed to improve the performance of the local search algorithms. Due to the important industrial applications at that time, the effectiveness of the *SAT1* algorithms was tested through two CSP benchmarks, i.e., the SAT problem and the n-queen problems.

An important industrial application for SAT is VLSI engineering. The *SAT1* algorithm family, the first local search algorithm for SAT, was developed for theoretical study and VLSI applications. During the late eighties, there was little progress in the theoretical analysis of the *SAT1* algorithms. The *SAT1* algorithm was applied to VLSI circuit testing and synthesis. All these problems can be formulated as instances of the SAT and MAX-SAT problems. The *SAT1* algorithm was found to be efficient for many VLSI design problems.

Another important application area for the SAT problem is industrial scheduling. During the late eighties, IBM and NASA were working on a number of important scheduling projects. These applications involved solving large size scheduling and task assignment problems under critical timing, spatial, and resource constraints. The scheduling and task assignment problems are well-known as the satisfiability problem. SAT formulas characterize these problems precisely. Significant local search solutions to the scheduling and task assignment problems were derived from the *SAT1* algorithm family. Due to its abstract *CNF* formulation, however, the SAT problem was not able to provide a descriptive geometric model that was able to demonstrate the scheduling and task assignment operations expressively.

2. N-Queen Scheduling Models and the QS Algorithms.

The n-queen problem is a benchmark for constraint satisfaction problem. During the middle and late eighties, Gu worked on various n-queen problem models for combinatorial optimization [**206, 228**]. He found that, by a remarkable coincidence, the n-queen model represents a significant model for scheduling and task assignment problems.

The underlying structure of the n-queen problem, represented by a complete constraint graph, gives a relational model with fully specified constraints among the multiple objects [**206**]. Variations on the dimension, the objects' relative positions, and the weights on the constraints led to a **hyper-queen** problem model which consists of several simple and basic models:

- **n–queen** problem: the *base* model. N queens are indistinguishable and the constraints among queens are specified by the binary values (i.e., 1 or 0).
- **w–queen** problem: the *weighted* n-queen model. N queens are distinguishable (each is associated with a cost) and the constraints among queens are specified by some weights.

- **3d–queen** problem: queens are to be placed in a 3-dimensional ($l \times m \times n$) rectangular cuboid. A special case, **nm–queen**, is to place queens on an n by m rectangle.
- **q+–queen** problem: more than one queens are allowed to be placed on the same row or the same column.

Based on the n-queen, the *hyper*-queen problem can model the objects/tasks, the performance criteria, the timing, spatial, and resource constraints for a wide range of scheduling and task assignment problems. This made the n-queen problem a general model for many industrial scheduling and task assignment problems. By a remarkable coincidence, the models of several difficult scheduling projects at that time were either the n-queen or the *hyper*-queen problems [**292, 501**]. All of them required efficient solutions to the n-queen or *hyper*-queen problems.

Scheduling problems modeled by various *hyper*-queen models have specific performance criteria and are known to be NP-hard. When scheduling computational tasks onto multiprocessors, for example, one can use a *hyper*-queen model where there are $q+$ weighted queens to be placed on a t by p rectangle. Let t denote the execution time, p the number of processors, q_i the execution time of the ith task, and c_{ij} the communication time from the ith task to the jth task, the goal is to place the *task queens* onto the t by p board and minimize the longest execution path, following the given topological constraints.

The *hyper*-queen models freed the original n-queen problem from its puzzle game background. Many practical applications of the n-queen and *hyper*-queen models to real world problems have been found. These include [**214, 491**] task scheduling, real-time system, task assignment, computer resource management, VLSI circuit design, air traffic control, communication system design, and so on.

Polynomial time, analytical solutions for the n-queen problem exist but they cannot solve the general search problems and have no use in practice [**2, 10, 30, 156, 252, 445**]. Following local conflict minimization [**206, 226**], a $QS1$ algorithm was developed during late 1987 and was implemented during early 1988. It was the first local search algorithm developed for the n-queen problem [**206, 484, 485, 486**]. Three improved local search algorithms for the n-queen problem were developed during 1988 to 1990 [**321, 322, 295, 454**]. $QS2$ is a near linear-time local search algorithm with an efficient random variable selection strategy [**487**]. $QS3$ is a near linear-time local search algorithm with efficient pre- and random variable selection and assignment [**487**]. $QS4$ is a linear time local search algorithm with efficient partial and random variable selection and assignment techniques [**488, 491**]. Compared to the first local search algorithm [**206**], partial and random variable selection/assignment heuristics have significantly improved search efficiency by orders of magnitude. $QS4$, for example, was able to solve 3,000,000 queens in a few seconds.

Three years after releasing the $QS1$ algorithm, Minton *et al.* independently reported a similar local search algorithm for the n-queen problem [**379, 380**]. A major difference between Minton's algorithms and Sosic and Gu's algorithms was that Minton's first algorithm was a one dimensional local search without using random heuristics.

Early local search solutions for scheduling applications were developed during the late eighties. Since then more than one hundred industrial companies worldwide have developed software systems for various scheduling applications.

3. A Simulated Annealing Algorithm for *Max-SAT*.

Motivated by the method of simulated annealing, Hansen and Jaumard [**239**] proposed a steepest ascent, mildest descent algorithm for the maximum satisfiability (Max-SAT) problem. In this approach, Hansen and Jaumard focused on *a* local change and defined an objective function based on *a* switching variable and its related clauses. The objective function maximizes local compensation for each variable which can be used for solving the Max-SAT problem. The objective function can not be used for the SAT problem unless another objective function whose global minimum corresponds to a solution of the SAT problem is given. Furthermore, Hansen and Jaumard used local optima checking to handle the local optimum and found it by providing additional guidance to the search direction.

4. Theoretical Study for *SAT1* and *2-SAT*.

During the early nineties, researchers started to work on the theoretical analysis of local search algorithms for CSP and SAT problems. In 1991 two theoretical studies that focused on the SAT problem were reported. Gu and Gu took three algorithms (i.e., $SAT1.1$, $SAT1.2$, and $SAT1.3$) from the *SAT1* algorithm family and made average time complexity study for the SAT problem [**221**].

During the study of the complexity of a certain natural generalization of SAT, Papadimitriou gave a randomized algorithm for the 2-SAT problem [**402**]. Furthermore Papadimitriou showed that such a randomized algorithm finds assignments for 2-SAT instances in $O(n^2)$ steps with probability approaching one, where n is the number of variables. With further extensions [**404**], in theory, the algorithm can be applied to solve the random 3-SAT problems.

Early on, local search method for the large size n-queen scheduling problem attracted great attention in the AI area. This was due to the close relationship between CSP and SAT: the SAT problem is a special case of CSP. The n-queen problem, on the other hand, is a typical benchmark problem in CSP. If one can find an efficient (non-analytical) search algorithm for the n-queen problem, then the algorithm can be directly translated to an efficient algorithm for the SAT problem.

Analytical solutions exist for the n-queen problem with n greater than or equal to 4 [**10, 156, 252, 445**]. They consist of a restricted subset of solutions [**10**]. If one formulates the n-queen problem as a CSP, backtracking can be used to search for any general solution. In practice, backtracking search is too slow to solve the n-queen problem for n larger than 96 [**501**]. Thus local search algorithms for solving large size n-queen problems become a breakthrough point in this direction. Following recent work for solving large scale n-queen problems, Selman, Levesque and Mitchell reported empirical results of *GSAT*, a greedy local search algorithm for solving SAT [**472**]. Selman [**471**] has recently acknowledged that local search solutions to large-size n-queen problems was "the original impetus" to the development of the *GSAT* algorithm [**472**].

7.6. Randomized Local Search. In this section, we describe the basic structure and major components of the randomized local search algorithms for the SAT problem.

```
procedure SAT1.0 ()
begin
    /* initialization */
    get_a_SAT_instance();
    x_0 := select_a_random_initial_point();
    F(x_0) := evaluate_objective_function(x_0);

    /* search */
    k := 0;
    while F(x_k) ≠ 0 do
    begin
        for each variable i := 1 to n do
            /* if flip(x_i,x̄_i) does not increase F */
            if test_flip(x_i,x̄_i) then
            begin
                x_{k+1} := perform_flip(x_i,x̄_i);
                F(x_{k+1}) := evaluate_objective_function(x_{k+1});
            end;
        /* random flips */
        if local then local_handler();
        k := k + 1;
    end;
end;
```

FIGURE 14. **SAT1.0:** a randomized local search procedure for the SAT problem [**209, 212, 221**]. Random flips are introduced (1) to disorder the sequence with which the variables are selected for local optimization, and (2) to perturb local search with randomized downhill, tunneling, or uphill moves [**221, 212**].

Model. Most discrete local search procedures were developed based on a discrete, unconstrained optimization model, the *SAT1 model* [**207, 209, 212, 221**]. In this model, the truth values assigned to the variables are defined as:

$$x_i = \begin{cases} 1 & \text{if the variable has value } \textit{true} \\ -1 & \text{if the variable has value } \textit{false} \end{cases} \tag{7.2}$$

The objective function, $F(\mathbf{x})$, in the *SAT1* model counts the number of unsatisfied clauses as its objective value. A *CNF* is true *if and only if* $F(\mathbf{x})$ takes the global minimum value 0 on the corresponding $\mathbf{x}$.

Basic Local Search. The *SAT1.0* algorithm for the SAT problem is shown in Figure 14. It consists of an initialization stage and a search stage. At the beginning of search, a SAT formula is generated. An initial random solution is chosen. The number of unsatisfiable clauses is computed and is assigned as the value of the objective function. During each iteration, function *test_swap*() performs a test to see if the objective function would increase. If *test_flip*() returns *true*, a flip operation is performed by procedure *perform_flip*(). Then function *evaluate_objective_function*() updates the objective function.

```
procedure Local_handler ()
begin
    random select some variable x_t's;
    x_{k+1} := perform_flip(x_t's,x̄_t's);
    F(x_{k+1}) := evaluate_objective_function(x_{k+1});
end;
```

FIGURE 15. A simple local handler used in the *SAT1* algorithms [**209, 212, 221**]. Random flips or a new random solution were applied to the algorithm if (1) $F \neq 0$ (*SAT1.0* algorithm), (2) $F > 0$ (*SAT1.1* algorithm), (3) $F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k)$ (*SAT1.2* algorithm), and (4) $F > 0$ and $F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k)$ (*SAT1.3* algorithm) [**209, 212, 221**].

The procedure terminates when the objective function is reduced to zero, i.e., a solution to the given SAT instance is found. In practice, before the objective function reduces to zero, the procedure may become stuck at local minima. In the *SAT1.0* algorithm [**209, 221**], a simple local handler performing random flips was used (Figure 15). This combined the greedy local descent (reducing objective function) with the random uphill moves (increasing objective function), improving *SAT1*'s convergence performance effectively. In the *SAT1* algorithm family, one or more local handlers were implemented [**221, 209, 212**]. If the algorithms have difficulty to proceed, the algorithms will call the local handlers and use other heuristics (see Section 7.3) to improve algorithms' convergence performance.

The random flips used in the *SAT1* algorithms make the *order* of selecting which variable for local examination (i.e., the *for* loop) trivial [**221, 209, 212**]. One can essentially select any variable randomly for examination during any phase of the local search.

Random Flips (Noise). If the local search procedure becomes stuck at a local minimum, further progress may be achieved by using a noise perturbation to change its location in the search space. The effectiveness with which local minima are handled significantly affects the performance of a local search algorithm. Researchers have proposed a number of techniques such as *jumping*, *climbing*, *annealing*, and *indexing* to handle local minima [**222**]. In simulated annealing, a search process occasionally moves up rather than down in the search space, with large uphill moves being less likely than small ones. The probability of large uphill moves is gradually reduced as the search progresses.

A variety of local handlers have been designed for use in the local search algorithms [**221, 212**]. *SAT*1.0 [**207, 221, 209, 212**] uses a local handler that may randomly negate the truth values of one or up to n variables (a new solution point) (Figure 15). The basic idea is to generate *random exchanges* in some current solution points when the search is stuck at a local minimum. The search accepts a modified point as a new current solution *not only when the value of the objective function is better but also when it is worse* [**221, 207, 209, 212**] (Traditional local search such as *GSAT* used the greedy local descent and restart [**472**]). This simple

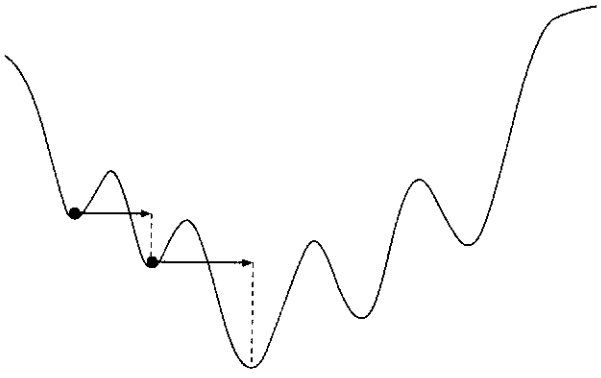


FIGURE 16. Tunneling

local handler has effectively improved the convergence performance of the $SAT1.0$ algorithm.

Tunneling Heuristic. A local handler and its activating condition(s) have significant effect on the performance (running time and average running time) of a local search algorithm for the SAT problem. The conditions for activating local handlers differ from algorithm to algorithm (see Figure 15). In $SAT1.1$ algorithm, the local handler is called if the objective function is not zero (an aggressive strategy) [**221, 212**]. In $SAT1.2$ algorithm, the local handler is called if the objective function does not increase [**221, 212**]. In $SAT1.3$ algorithm, the local handler is called if the objective function does not increase or the objective function is greater than zero for some iterations [**221, 212**]. In the last two algorithms, the condition *"objective function does not increase"* means that the objective value is either reduced (*local descent*) or remained unchanged (*tunneling heuristic*).

Instead of making a random swing in the vertical direction in the search space, whenever a local minimum is encountered, one can *tunnel* through the rugged terrain structure in a horizontal direction, moving from one local basin to another local basin in an attempt to locate a better locally optimal solution. A tunnel (see Figure 16) can be thought of as a short-cut passing through a mountain separating points of equal elevation. Whenever a local minimum is encountered, a tunnel is made through a mountain to a neighboring basin as long as this does not change/increase the objective function. Tunneling can be used to search a region with local minima effectively. The behavior of local search with tunneling illustrates the fact that seemingly innocuous changes in an optimization routine can have a surprisingly large effect on its performance. When tunneling was first implemented in the $SAT1$ algorithm in the late eighties. it was proven to be effective in solving some SAT problems.

Parallel Local Search. Several parallel algorithms and VLSI architectures have been developed to accelerate CSP and the SAT problems [**226, 227, 212, 492**]. Depending on implementations, there are several ways of grouping variables or clauses together in parallel so they can be evaluated simultaneously. In the $SAT1$ algorithms, the most frequently used part of computation is the function

evaluate_objective_function(). It takes O(ml) time to update the objective function. The execution of *evaluate_objective_function* can be done in a simple bit-parallel manner in O(m) time on a sequential computer.

A computer word has 32 or 64 bits (such as the *DEC Alpha* machine). The number of literals in a clause of most practical *CNF* formulas is much less than 32. In a local search algorithm, therefore, one can pack all the literals in a clause into the bits of a computer word and then evaluate all the literals in one clause in parallel. For m clauses, instead of O(ml), it will take procedure *evaluate_objective_function* O(m) time to evaluate and update the objective function. Occasionally, a clause may have more than 32 literals, they can be packed in several computer words and all of them can be evaluated simultaneously. This general bit-parallel evaluation method was implemented in the *SAT*1.7 algorithm [**207, 212**].

Complete Local Search. Local search algorithms are incomplete, *i.e.*, they can find some solutions for certain *CNF* formulas and give no answer if the *CNF* formula is not satisfiable. To overcome this problem, researchers developed complete local search algorithms to test satisfiability as well as unsatisfiability. The basic idea in the *SAT1.11* and *SAT1.13* algorithms [**207, 212**] was to combine local search with a systematic search procedure, keeping local search's efficiency while maintaining search completeness by the systematic search method [**207, 212**]. If at a node of the search tree a solution point is found unsatisfiable, then the algorithm backtracks and continues searching until a solution is found or unsatisfiability is proven.

The *SAT1.11* and *SAT1.13* algorithms were two early experiments of complete local search algorithms [**207, 212**]. Probe order backtracking is a simplified version of complete local search [**433, 434**]. Recently Crawford studied a complete local search algorithm [**110**]. He used weights assigned to clauses to help choose branch variables. Variables occurring in heavily weighted clauses were given precedence.

7.7. Randomized Local Search with Trap Handling. Based on early observation of trap phenomenon and the development of a three-level search space model (Section 7.2), Gu *et al.* developed a *SAT1.5* algorithm with trap handling ability [**209, 219**]. The *SAT1.5* algorithm can monitor and break local minimum loops and can handle multiple traps during the search. The current version of the *SAT1.5* algorithm contains advanced data structures and complicated trap detection/handling methods [**209, 212, 219**]. For the sake of simplicity, Figure 17 gives a brief outline of the algorithm.

The *SAT1.5* starts with an initial random solution and a set of limiting parameters. *Max_Time*, for example, specifies the maximum number of times allowed to restart a new search. The number of unsatisfiable clauses is computed and is assigned as the value of the objective function. The first *while* loop is limited by the *Max_Time*. Procedure *complete_flip*() flips all the variables that can reduce the value of the objective function. *Evaluate_objective_function*() updates the objective function.

The second *while* loop is a randomized local search with trap tracking and handling. Trap detection facilities are installed several places in the *while* loop to record trap statistics. They are essential to figure out trap "height," "width," and other parameters for subsequent decision making. A trap may contain a global minimum solution and it must be searched with reasonable effort. Leaving a trap too early

```
procedure SAT1.5 ()
begin
    /* initialization */
    get_a_SAT_instance();
    x_0 := select_a_random_initial_point();
    F(x_0) := evaluate_objective_function(x_0);

    /* search */
    k := 0; Restart_Times := 0;
    while F > 0 and Restart_Times < Max_Time do
    begin
        /* Open Search: flip all variables that reduce F */
        x_{k+1} := complete_flip(x_k);
        F := evaluate_objective_function(x_{k+1});

        /* parameters for trap tracking */
        Clean_trap_records(); Trapping_Times := 0;
        /* Peak Search: randomized local search */
        while F > 0 and Trapping_Times < Max_Trapping_Times do
        begin
            /* randomly select one var for randomized local search */
            x_i := select_one_var_to_flip(x_{k+1});
            x_{k+1} := randomized_local_search(x_i, ¬x_i);
            F := evaluate_objective_function(x_{k+1});
            /* Trap Search */
            if a trap is detected then
            begin
                Trapping_Times + +;
                /* random flip vars in conflicting clauses */
                x_{k+1} := strong_flip(x_{k+1});
                F := evaluate_objective_function(x_{k+1});
                /* random flip a few percent (pct) of variables */
                x_{k+1} := gentle_flip(x_{k+1}, pct);
                F := evaluate_objective_function(x_{k+1});
                /* random flip a small set of variables */
                x_{k+1} := weak_flip(x_{k+1}, set);
                F := evaluate_objective_function(x_{k+1});
                /* initialization for a new trap */
                Clean_trap_records();
            end;
        end;
        if F > 0 then
            x_{k+1} := restart_a_new_random_point();
            Restart_Times + +;
    k := k + 1;
    end;
end;
```

FIGURE 17. **SAT1.5:** a randomized local search procedure with trap handling.

or too late could result in either losing solutions or wasting computing time. The time to jump out of a trap is determined by parameter *Max_Trapping_Times*.

When the search algorithm jumps out of a trap, there are several alternatives to pursue. One is to start a new search. In the *while* loop, several randomized local search procedures deploying random value and random variable heuristics (see Section 7.3) are grouped together with partial random selection heuristics. They together select a variable for randomized local search (the objective function F may increase during the search).

If a trap is detected, a number of strategies can be used to conduct a trap search [**212, 219**]. In one approach proposed by Gu *et al.*, a sequence of random flip operations is performed (see Figure 17). The intensity of the flip operations evolves from strong to weak, tailored to the "coarse" as well as "fine" structures in a trap. That is, a variable is flipped in each *unsat clause* to force it to value *true* (procedure *strong_flip*()), followed by a random flip of a few percent of the variables (procedure *gentle_flip*()), and finally, a random flip of a small number of variables (procedure *weak_flip*()). Additional facilities for hill climbing, tabu search, and variable locking/unlocking were developed. The *SAT1.5* algorithm can walk on the rugged surface of a trap adaptively.

The real execution performance of the *SAT1.5* algorithm (Section 13.2) suggests that it is presently one of the fastest local search algorithms for the SAT problem.

7.8. Greedy Local Search. Traditional local search proceeds by taking a feasible solution point that reduces the value of the objective function. Among many neighboring solution points, local search does not evaluate its neighbors' relative performance with respect to the objective function. A *greedy* algorithm selects the *best neighbor* that yields the minimum value of the objective function and takes this *best neighbor direction* as the descent direction of the objective function. In a real search space, continuous optimization algorithms can find the best neighbor solution efficiently. Unconstrained local and global optimization algorithms have been developed for solving the SAT problem (see [**209, 213**] and Section 8).

In the discrete search space, a greedy local search algorithm searches for the best neighbor solution. This requires that during each iteration the algorithm examine all the possible moves and select one with maximum descent. Greedy local search is a special case of the coordinate descent in the real space.

Selman *et al.* proposed a greedy local search procedure, i.e., *GSAT*, for the SAT problem [**472**]. During each search step, the algorithm evaluates all the moves and selects the best move that gives the *greatest* decrease in the total number of unsatisfied clauses. If the algorithm becomes stuck at a local minimum, *GSAT* uses *side-walk* (a form of tunneling heuristic) to move aside. In *GSAT* procedure, two parameters, *MAX-TRIES* and *MAX-FLIPS*, were used to control the algorithm's maximum running state.

Late eighties VLSI researchers experimented with a large number of practical SAT formulas with the greedy local search and found that greedy local search became stuck at local minima much more easily. Accordingly, Gu proposed a method of combining local descent with random, multiphase search, and trap handling heuristics (see Section 7.3 and Section 7.7). These ideas were used in the subsequent *SAT1* algorithm family design [**221, 207, 209, 212**].

Recently Selman *et al.* used the bad variable heuristic and the partial random variable selection heuristics (Section 7.3) in their *random walk* heuristic [**475, 474**].

```
procedure GSAT ()
begin
   for i := 1 to MAX-TRIES
      T := a randomly generated truth assignment
         for j := 1 to MAX-FLIPS
            if T satisfies α then return T
            p := a propositional variable such that a change
                 in its truth assignment gives the largest
                 increase in the total number of clauses
                 of α that are satisfied by T
            T := T with the truth assignment of p reversed
         end for
   end for
   return "no satisfying assignment found"
end
```

FIGURE 18. **GSAT:** a Greedy local search procedure for the **SAT** problem [**472**]. MAX-FLIPS, MAX-TRIES are constants, and α is a set of clauses. During each search step, *GSAT* takes the best neighbor that gives the maximum descent to the objective function.

They found that these random heuristics (such as random flips, selecting a variable in unsat clause, and partial random variable selection) improved the performance of the *GSAT* algorithm significantly [**475**].

7.9. Tabu Local Search. Mazure, Sais, and Gregoire proposed a tabu search algorithm, *TSAT*, for satisfiability problem [**370**]. The basic idea behind the *TSAT* is to avoid using randomness in local search algorithm design. *TSAT* makes a systematic use of a tabu list of variables in order to avoid recurrent flips and thus escape from local minima. The tabu list is updated each time a flip is made. *TSAT* keeps a fixed length-chronologically-ordered FIFO list of flipped variables and prevents any of the variables in the list from being flipped again during a given amount of time.

In this study, Mazure *et al.* found that the optimal length of the tabu list is crucial to the algorithm's performance. They showed that, for random 3SAT instances, the optimal length of the tabu list $L(n)$ for *TSAT* is [**370**]:

$$L(n) = 0.01875n + 2.8125. \tag{7.3}$$

Furthermore, they noted that a slight departure from the optimal length leads to a corresponding graceful degradation of the performance of *TSAT*. A more important distance from this optimal length leads to a dramatic performance degradation.

7.10. Local Search for *DNF* formulas. Using the well-known DeMorgan laws, we can obtain an unconstrained optimization model, the *SAT4 model*, for *DNF* formulas [**207, 213**]: With *SAT4*, a *CNF* formula

$$(x_1 + \bar{x}_2)\ (\bar{x}_1 + x_2 + \bar{x}_4)\ (x_2 + \bar{x}_3)$$

can be transformed into a *DNF* formula:

$$\bar{x}_1 x_2 + x_1 \bar{x}_2 x_4 + \bar{x}_2 x_3.$$

For the transformed formula, the objective is to determine whether there exists an assignment where all clauses are falsified. That is, to solve (4.9).

A number of local search algorithms were developed for *DNF* formulas. Except for different definition and evaluation schemes in the objective function, they have similar structures as in *CNF* local search algorithms. In *SAT*1.4 [**207**], one of the early *DNF* local search algorithms, the objective function is defined as the number of satisfiable *DNF* terms. Our goal here is to reduce the objective function to zero. Experimental results indicate that *DNF* local search algorithms are faster than *CNF* local search algorithms.

7.11. A Historical Note. Early work in constraint satisfaction, simulated annealing, and complexity study contributed to the development of local search algorithms for the SAT problem (see Sections 7.3, 7.4, and 7.5, and Figure 19). A

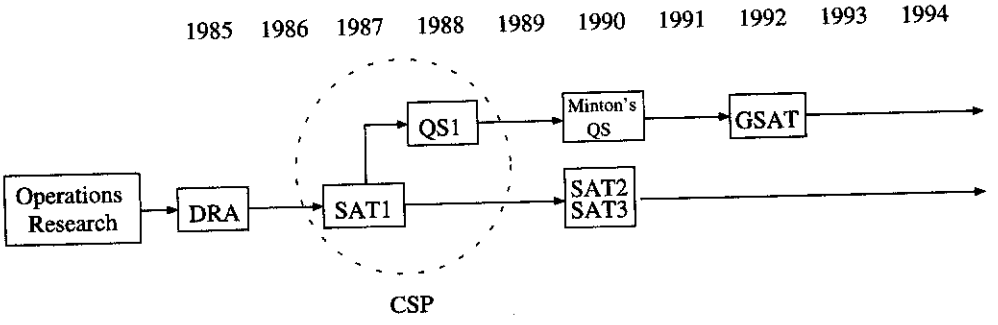


FIGURE 19. Early development of local search algorithms for SAT problem. There were two major approaches: randomized local search (*SAT1*) and greedy local search (*GSAT*). The *SAT1* was the first local search algorithm developed for the VLSI engineering and scheduling applications. The *GSAT* algorithm was derived from the early local search algorithms for the n-queen problem.

special event was the n-queen debate in ACM *SIGART Bulletin* during 1990 and 1992.

Early the *SAT1* algorithms were applied to solve VLSI circuit design problems. In addition, Gu and Sosič implemented the same local search method for the n-queen scheduling problems. Later they published two short papers in *SIGART Bulletin* [**486, 488**]. By accident, these two papers triggered a debate. Major discussions centered around two questions raised by the *SIGART* readers.

First, Jack Mostow mentioned that Steve Minton at the same time published "a hill-climbing algorithm very similar to Gu's" for the n-queen problem at AAAI'90. He was interested to know the original source of the local search algorithm for the n-queen problem. Lewis Johnson reviewed the original local search results for the n-queen problem [**206**] and found that [**295**]: "It is now clear that the n-queens problem is a solved problem; in fact, it has been solved for many years."

The second question was about local search. In *SIGART Bulletin*, Vol. 2, No. 2, 1991, Bo Bernhardsson showed that the analytical solutions for n-queen problem was published in 1969. In Jun Gu's reply entitled "*On a General Framework for Large-Scale Constrained Optimization,*" he explained that the analytical solutions

to n-queen problem only offer a restricted set of solutions which cannot solve a general search problem; the local search algorithm for n-queen can be used to solve general constraint satisfaction problems. This discussion continued in a number of *SIGART* issues [**296**]. In August 1991, M. Valtorta showed more analytical solutions to the n-queen problem and the Tower of Hanoi problem [**526**]. Many readers agreed that the analytical solutions are restricted but some also believed that local search can only solve problems like the n-queen. Satisfiability problem is the core of many NP-complete problems. So, finally, Gu wrote a short article "*Efficient Local Search for Very Large-Scale Satisfiability Problem*" [**209**] and discussed the *SAT1* algorithms as a general example of local search.

During the two-year period, many researchers including Jack Mostow, Steve Minton, Bart Selman, and Dennis Kibler participated in various discussions.

8. Global Optimization

Local search proceeds by taking *any* solution point that decreases the value of the objective function as the next solution point. Since there may be many neighboring solution points and a local search does not take into account its neighbors' relative performance to the objective function, a local search may get stuck in a local minimum or a basin. To escape from such local minima, global search strategies need to be developed. One such strategy is the tunneling heuristic discussed in Section 7.6. Another strategy is to select the best neighboring point that yields the minimum value to the objective function. When there is no neighboring point that lead to decrease in the objective function, a direction is picked to minimize the increase in the objective function.

Global optimization is concerned with the characterization and computation of global minima and maxima of unconstrained nonlinear functions and constrained nonlinear problems [**162, 163, 267, 405**]. Global optimization problems belong to the class of NP-hard problems.

The concept of optimization is well rooted as a principle underlying the analysis of many complex decision problems. When one deals with a complex decision problem, involving the selection of values to a number of interrelated variables, one should focus on a single objective (or a few objectives) designed to qualify performance and measure the quality of the decision. The *core* of the optimization process is to minimize (or maximize) an objective function subject to constraints imposed upon values of decision variables in an instance.

Most optimization algorithms are designed as an iterative refinement process. Typically, in seeking a vector that solves an optimization problem, a search algorithm selects an initial vector $\mathbf{y}_0$ and generates an improved vector $\mathbf{y}_1$. The process is repeated to find a better solution $\mathbf{y}_2$. Continuing in this fashion, a sequence of ever-improving points $\mathbf{y}_0$, $\mathbf{y}_1$, ..., $\mathbf{y}_k$, ..., is found that approaches a solution point $\mathbf{y}^*$. When it is not possible to find neighboring points to improve, strategies are applied to help escape from local minima.

There are three aspects in designing global search strategies to solve SAT:

- *Problem formulations and transformations.* As discussed in Section 4.1, there are alternative formulations of an instance of SAT, and global search strategies may need to be tailored to the formulation used. In Section 8.1, we present the *UniSAT* model that transforms a SAT formula represented as an instance of a discrete constrained decision problem in Boolean $\{0, 1\}$ space

into a continuous optimization problem [**207, 211, 213**]. In Section 8.7, we present strategies based on discrete Lagrange multipliers to transform a SAT formula into an instance of a discrete constrained optimization problem [**538, 540**]. Other more general transformations are presented in Section 11.

- *Strategies to select a direction to move.* Since a search trajectory lacks global information in a search space, strategies to select a direction to move are either steepest descent or hill climbing. A steepest-descent approach chooses the direction with the maximum gradient. A hill-climbing approach, on the other hand, chooses the first point in the neighborhood of the current point that reduces the objective function. For large formulas, hill-climbing methods are much faster than steepest descent because they descend in the first direction, rather than the best direction, that leads to improvement.
- *Strategies to help escape from local minima.* Many possible strategies have been studied in the past. These include local handlers that use a combination of restarts, backtracking and random swaps (see Section 7.3 and [**207, 221, 209, 227, 210**]), Morris' "break-out" strategy [**388**], Wah and Shang's Discrete Lagrangian Method (DLM) [**538, 540**], Glover and Hansen's tabu list [**199, 239**], stochastic methods such as simulated annealing (SA) [**308, 74**], and genetic algorithms (GA) [**253, 377**]. In Section 8.8, we examine the effects of some of these strategies.

8.1. *UniSAT*: **Universal SAT Input Models.** In *UniSAT* models, we extend discrete search space $\mathbf{x} \in \{0,1\}^n$ into real space $\mathbf{y} \in E^n$, so that each solution point and the objective function can be characterized quantitatively. Furthermore, by encoding the solution of a SAT formula into the objective function, a direct correspondence between the solutions of the SAT formula and the global minimum points of the objective function can be established. Subsequently, the SAT formula is transformed into an instance of an unconstrained global optimization problem on E^n.

In *UniSAT* models, using the universal DeMorgan laws, all Boolean $\vee$ and $\wedge$ connectives in *CNF* formulas are transformed into $\times$ and $+$ of ordinary addition and multiplication operations, respectively. The *true* value of the *CNF* formula is converted to the 0 value of the objective function. Given a *CNF* formula $\mathcal{F}$ from $\{0,1\}^n$ to $\{0,1\}$ with m clauses $C_1, \ldots, C_m$, we define a real function $f(\mathbf{y})$ from E^n to E that transforms the SAT into an unconstrained global optimization problem:

$$\min_{\mathbf{y} \in \mathbf{E}^n} f(\mathbf{y}) \tag{8.1}$$

where

$$f(\mathbf{y}) = \sum_{i=1}^{m} c_i(\mathbf{y}). \tag{8.2}$$

A clause function $c_i(\mathbf{y})$ is a product of n *literal functions* $q_{i,j}(y_j)$ $(1 \leq j \leq n)$:

$$c_i = \prod_{j=1}^{n} q_{i,j}(y_j). \tag{8.3}$$

In the *UniSAT5* model [**207, 209, 213**]

$$q_{i,j}(y_j) = \begin{cases} |y_j - 1| & \text{if literal } x_j \text{ is in clause } C_i \\ |y_j + 1| & \text{if literal } \bar{x}_j \text{ is in clause } C_i \\ 1 & \text{if neither } x_j \text{ nor } \bar{x}_j \text{ is in } C_i \end{cases} \tag{8.4}$$

and in the *UniSAT7* model [**207, 211, 209, 213**]:

$$q_{i,j}(y_j) = \begin{cases} (y_j - 1)^2 & \text{if } x_j \text{ is in clause } C_i \\ (y_j + 1)^2 & \text{if } \bar{x}_j \text{ is in clause } C_i \\ 1 & \text{if neither } x_j \text{ nor } \bar{x}_j \text{ is in } C_i \end{cases} \tag{8.5}$$

The correspondence between $\mathbf{x}$ and $\mathbf{y}$ is defined as follows (for $1 \leq i \leq n$):

$$x_i = \begin{cases} 1 & \text{if } y_i = 1 \\ 0 & \text{if } y_i = -1 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Clearly, $\mathcal{F}$ has value *true* iff $f(\mathbf{y}) = 0$ on the corresponding $\mathbf{y} \in \{-1, 1\}^n$.

The *UniSAT5 model* on real space is a direct extension of the discrete *SAT4 model* on Boolean space. A model similar to *UniSAT5* was proposed independently in the neural network area [**291**]. A significant difference between the neural network model and *UniSAT5* is their efficiency and practical applicability. The neural network model can only be handled by traditional nonlinear programming methods that are extremely slow [**291**], whereas *UniSAT5* can be easily solved in conjunction with the local search approach by simple discrete accounting techniques [**207, 213**].

The *UniSAT* models transform SAT from a discrete, constrained decision problem into an unconstrained global optimization problem [**207, 211, 209, 213**]. A good property of the transformation is that *UniSAT* models establish a correspondence between the global minimum points of the objective function and the solutions of the original SAT formula. A *CNF* $\mathcal{F}$ has value *true if and only if* $f(\mathbf{y})$ takes the global minimum value 0 on the corresponding solution $\mathbf{y}^*$.

Following the above formulation, with the *UniSAT5* and *UniSAT7* models, a *CNF* $\mathcal{F}$

$$(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$$

is translated into

$$f(\mathbf{y}) = |y_1 - 1||y_2 + 1| + |y_1 + 1||y_1 - 1||y_3 - 1|$$

and

$$f(\mathbf{y}) = (y_1 - 1)^2(y_2 + 1)^2 + (y_1 + 1)^2(y_1 - 1)^2(y_3 - 1)^2,$$

respectively.

The solution of the SAT formula corresponds to a set of global minimum points of the objective function. Finding a *true* value of $\mathcal{F}$ is equivalent to finding a *false* value, *i.e.*, 0, of f(y).

The translation of SAT formulas into nonlinear programs is quite different from the integer programming approach described in the next section. In the integer programming approach, one views a SAT formula as an instance of the 0/1 Integer Programming problem and tries solving its Linear Programming relaxation [**35, 258, 259, 285, 303, 301, 548**]. If the solution is non-integer, one rounds off the values to the nearest integers and checks whether the solution corresponds to a solution of the original formula. If the rounded off values do not correspond to a solution, one computes another solution of the linear programming problem.

```
procedure SAT6.0 ()
begin
    /* initialization */
    obtain_a_SAT_instance();
    y0 := select_an_initial_solution();
    f(y0) := evaluate_object_function(y0);

    /* search */
    k := 0;
    while not(solution_testing()) do
        for some y_{i(k)}s ∈ yk
        begin
            /* minimizer */
            if test_min(f(y_{i(k)}s)) then
            begin
                y_{k+1} := perform_min(f(y_{i(k)}s));
                f(y_{k+1}) := evaluate_object_function();
            end
            if close_to_solution() then x := approximate(y_{k+1});
        end;
        /* local handler */
        if local then local_handler();
        k := k + 1;
    end;
    end;
```

FIGURE 20. **SAT6.0:** A general global optimization algorithm for the satisfiability problem.

8.2. A Global Optimization Algorithm for solving SAT. Many families of unconstrained global optimization algorithms for the *UniSAT* problem have been developed [**207, 211, 213**]. *SAT6.0*, a basic global optimization algorithm, is shown in Figure 20. To start, procedure *obtain_a_SAT_instance*() initializes a (given or generated) SAT instance. An objective function, f, is formulated according to a given *UniSAT* model. The SAT formula thus becomes a minimization problem to the objective function. To begin, procedure *select_an_initial_solution*() selects an initial starting point $\mathbf{y}_0 \in E^n$. The corresponding value of the objective function, $f(\mathbf{y}_0)$, is evaluated by function *evaluate_object_function*().

The optimization process is an iterative minimization to the objective function. Function *test_min*() tests if the value of the objective function can be minimized. If this is true, the minimization operation is performed by procedure *perform_min*(), followed by *evaluate_object_function*() that updates the value of the objective function. Procedures *test_min*(), *perform_min*(), and *evaluate_object_function*() are usually performed together without distinction. Depending on the global optimization strategy, the objective function can be minimized in one or up to n dimensions. Methods capable of optimizing f in one dimension include line search, coordinate descent, and coordinate Newton's methods. Methods that optimize f in more than one dimensions include the steepest descent methods, multi-dimensional Newton's methods, and many others.

As the iterative improvement progresses, a global minimum point may be approached gradually. The *closeness* between the present solution point and the global minimum solution point can be tested by solution-point testing or objective-value testing. Procedure *close_to_solution*() performs closeness testing. If the present solution point is sufficiently close to a global minimum point, procedure *approximate*() performs the *round-off* operation that converts a solution point $\mathbf{y}$ in real space E^n to a solution point $\mathbf{x}$ in Boolean space $\{0,1\}^n$ which may be a solution of the original SAT formula. Procedure *solution_testing*() takes the solution generated from procedure *approximate*() and substitutes it into the given *CNF* formula to verify its correctness.

In practice, the search process could be stuck at a locally optimum point. To improve the convergence performance of the algorithm, one or more local handlers may be added. One effective local handler in $SAT6$ is to negate the truth values of up to n variables.

Any existing unconstrained global optimization methods can be used to solve the *UniSAT* problems (see textbooks and literature). So far many global optimization algorithms have been developed [**207, 211, 213**]. These include the basic algorithms, steepest descent methods, modified steepest descent methods, Newton's methods, quasi-Newton methods, descent methods, cutting-plane methods, conjugate direction methods, ellipsoid methods, homotopy methods, and linear programming methods. In each algorithm family, different approaches and heuristics can be used to design objective functions, select initial points, scramble the search space, formulate higher-order local handlers, deflect descent directions, utilize parallelism, and implement hardware architectures to speed up computations.

8.3. A Discrete Global Optimization Algorithm. Although nonlinear problems are intrinsically more difficult to solve, an unconstrained optimization problem is conceptually simple and easy to handle. Many powerful solution techniques have been developed to solve unconstrained optimization problems, which are based primarily upon calculus and simple accounting, rather than upon algebra and pivoting, as in the Simplex method. Based on a coordinate descent method [**359**], Gu has recently given a simple algorithm, the $SAT14.5$ algorithm [**213, 214**], for the *UniSAT5* problem (see Figure 21). The kernel of $SAT14.5$ is a discrete *minimizer* that minimizes objective function f by the discrete coordinate descent method.

Given a function f on E^n, the $SAT14.5$ algorithm initially chooses a vector $\mathbf{y}$ from E^n and then minimize function f with respect to variables y_j $(1 \leq j \leq n)$ in *minimizer* until $f < 1$. Since each variable y_j appears in one clause function c_i at most once, function $f(\mathbf{y})$ can be expressed as

$$f(\mathbf{y}) = a_j|y_j - 1| + b_j|y_j + 1| + d_j$$

for $(1 \leq j \leq n)$, where a_j, b_j, and d_j are *local gain factors* that are independent of y_j. They can be computed in $O(ln)$ time. Therefore, $f(\mathbf{y})$ takes its minimum value with respect to y_j at point either $y_j = 1$ or $y_j = -1$. Thus, the *minimizer* optimizes function f as follows: if $a_j \geq b_j$ then set y_j equal to 1; otherwise set y_j equal to -1.

In practice, before $f < 1$, the algorithm could be stuck at a local minimum point. To overcome this problem, a simple local handler is added. The local handler simply generates a new initial vector $\mathbf{y}$ to start an independent search. In

```
Procedure SAT14.5 ()
begin
   /* initialization*/
   y := initial_vector();
   local := search := 0; limit := bn log n;

   /* search */
   while (f(y)≥ 1 and local ≤ limit) do
   begin
      old_f := f(y); search := search + 1;
      /* minimizer */
      for i := 1 to n do
         minimize f(y) with respect to y_i;
      /* local handler */
      if (f(y)=old_f or (search > b' log n and f(y)≥ 1)) then
      begin
         y := initial_vector();
         search := 0; local := local + 1;
      end;
   end;
   if f(y) < 1 then y* := round_off(y) else y* := enumerate();
end;
```

FIGURE 21. **SAT14.5:** A global optimization algorithm for the *UniSAT5* problem.

the *SAT14.5* algorithm, if the objective function f can no longer be reduced *or* after $b' \log n$ (b' is a constant, see [**213, 214**]) iterations of the *while* loop f is still at least one, then the *local-handler* is called.

8.4. A Continuous Global Optimization Algorithm. Based on a continuous coordinate descent method [**359**], Gu, Huang and Du have recently developed the $SAT14.7$ algorithm for solving *UniSAT7* problems on E^n [**213**]. For the objective function described in the *UniSAT7* input model, if only one variable, e.g., x_i, is selected for optimization, then

$$F(x_i) = a_i(x_i - 1)^2 + b_i(x_i + 1)^2 + c_i \tag{8.6}$$

where a_i, b_i, and c_i are constants that can be computed in $O(ml)$ time. Here, $F(x_i)$ can be minimized at:

$$x_i = \frac{a_i - b_i}{a_i + b_i}. \tag{8.7}$$

8.5. Complete Global Optimization Algorithms. The $SAT14.5$, $SAT14.6$, and $SAT14.7$ algorithms are *incomplete* algorithms. In order to achieve high computing efficiency and to make them complete algorithms, we combine in $SAT14.11$ to $SAT14.20$ global optimization algorithms with backtracking/resolution procedures [**207, 213**]. Therefore, these algorithms are able to verify satisfiability as well as unsatisfiability. Figure 22 gives a typical backtracking global optimization algorithm.

For small and medium size problems, backtracking is able to verify unsatisfiability quickly for certain classes of formulas but is slow when it comes to verifying

```
Procedure SAT14.11 ()
begin
    /* initialization */
    get_a_SAT_instance();
    x0 := select_an_initial_point();
    f := evaluate_object_function(x0);

    /* backtracking with global optimization */
    x* := backtracking(x0);
end;

Procedure backtracking(xi)
begin
    /* global optimization assigns v to xi */
    v := global_optimization();
    xi := v;
    Vi := Vi - {v};
    /* append variable xi to the partial path */
    path[xi] := i;

    if path broken then backtracking;
    if solution found then return x*;
    else backtracking(next xi);
end;
```

FIGURE 22. **SAT14.11:** a complete global optimization algorithm with backtracking.

satisfiability, as all possible resolutions need to be tried out before concluding that the inference relation holds or that the input formula is satisfiable. From our experience, a *combined* global optimization algorithm with backtracking/resolution procedures would perform well for certain classes of satisfiable and unsatisfiable formulas.

Recently some researchers investigated the number of solutions of SAT formulas. Extending Iwama's work [**281**], Dubois gave a combinatorial formula computing the number of solutions of a set of any clauses [**148**]. He and Carlier also studied the mathematical expectation of the number of solutions for a probabilistic model [**149**]. For an incomplete SAT algorithm, the number of solutions can have a strong effect on its computing efficiency. For a complete SAT algorithm, however, the number of search levels plays a crucial role. In $SAT14.11$ to $SAT14.20$ algorithms, the number of solutions is an important strategy to interplay global optimization and backtracking/resolution procedures [**212, 213**].

8.6. Continuous Lagrangian-Based Constrained Optimization Algorithms. In previous subsections, we have discussed unconstrained (discrete or continuous) formulations of SAT problems based on optimizing a single unconstrained objective function. To avoid getting trapped in local minima, algorithms for solving these problems must have strategies to escape from local minima. Some of these strategies, such as random restarts and tunneling, move the search to a new starting point and start over. In the process of doing so, vital information obtained during the descent to the current local minimum may be lost. Other strategies

may rely on an internal or an external force to bring the search trajectory out of a local minimum. Although they work well for continuous problems, they may have difficulty in dealing with SAT problems whose objective values are integers.

One way to bring a search out of a local minimum is to formulate a SAT problem as a constrained optimization problem as shown in (4.10) and (4.16). By using the force provided by the violated constraints, the search trajectory can be brought out of a local minimum. One way to implement this idea is compute the sum of the constraints weighted by penalties and to update the penalties continuously during the search. The difficulties with this approach lies in the choice of the proper penalties. A more systematic approach is to use a Lagrangian formulation. In this and the next subsections, we show two Lagrangian formulations of SAT problems, one in the continuous space and the other in the discrete space.

As indicated in (4.16), a SAT problem can first be transformed into a continuous constrained optimization problem.

$$\begin{aligned} &\min_{\mathbf{y}\in E^n} \quad F(\mathbf{y}) = \sum_{i=1}^{m} c_i(\mathbf{y}) \\ &\text{subject to} \quad c_i(\mathbf{y}) = 0 \quad \forall i \in \{1, 2, \ldots, m\} \end{aligned} \tag{8.8}$$

where $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, and $c_i(\mathbf{y})$ is defined in (4.12) and (4.13) and repeated as follows.

$$\begin{aligned} c_i(\mathbf{y}) &= \prod_{j=1}^{n} q_{i,j}(y_j) \\ q_{i,j}(y_j) &= \begin{cases} (1-y_j)^2 & \text{if } x_j \text{ in } C_i \\ y_j^2 & \text{if } \bar{x}_j \text{ in } C_i \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

Here, $F(\mathbf{y})$ is a scalar differentiable function that takes the norm of its argument so that $F(\mathbf{y}) = 0$ iff $c_i(\mathbf{y}) = 0$ for all i.

There are three advantages in reformulating the original discrete unconstrained problem into a continuous constrained problem. First, a continuous objective function may smooth out local minima in the discrete space, allowing global/local search methods to bypass these local minima in the continuous space. Second, a continuous objective value can indicate how close the constraints are being satisfied, hence providing additional guidance in leading to a satisfiable assignment. Third, when the search is stuck in a local minimum and some of the constraints are violated, the violated constraints can provide a force to lead the search out of the local minimum. This is more effective than restarting from a new starting point, as local information observed during the search can be preserved.

Active research in the past two decades has produced a variety of methods for finding global solutions to nonconvex nonlinear optimization problems [**508, 267, 163, 240, 405, 377**]. In general, transformational and non-transformational methods are two approaches in solving these problems.

Non-transformational approaches include discarding methods, back-to-feasible-region methods, and enumerative methods. Discarding methods [**277, 377**] drop solutions once they were found to be infeasible, and back-to-feasible-region methods [**299**] attempt to maintain feasibility by reflecting moves from boundaries if such moves went off the current feasible region. Both of these methods have been combined with global search and do not involve transformation to relax constraints.

Last, enumerative methods [**267**] are generally too expensive to apply except for problems with linear objectives and constraints, and for bilinear programming problems [**26**].

Transformational approaches, on the other hand, convert a problem into another form before solving it. Well known methods include penalty, barrier, and Lagrange-multiplier methods [**359**]. *Penalty methods* incorporate constraints into part of the objective function and require tuning penalty coefficients either before or during the search. *Barrier methods* are similar except that barriers are set up to avoid solutions from going out of feasible regions. Both methods have difficulties when they start from an infeasible region and when feasible solutions are hard to find. However, they can be combined with other methods to improve their solution quality.

In *Lagrangian methods*, Lagrange variables are introduced to gradually resolve constraints through iterative updates. They are exact methods that optimize the objective using Lagrange multipliers to meet the Kuhn-Tucker conditions [**359**]. Eq. (8.8) can be reformulated using Lagrange multipliers into the following unconstrained problem.

$$L(\mathbf{y},\lambda) = F(\mathbf{y}) + \lambda^T \mathbf{c}(\mathbf{y}) \qquad \text{(Lagrangian function)} \tag{8.9}$$

$$\mathcal{L}(\mathbf{y},\lambda) = F(\mathbf{y}) + ||\mathbf{c}(\mathbf{y})||_2^2 + \lambda^T \mathbf{c}(\mathbf{y}) \qquad \text{(Augmented Lagrangian function)} \tag{8.10}$$

where $\mathbf{c} = (c_1(y), c_2(y), \ldots, c_m(y))$, and λ^T is the transpose of the set of Lagrange multipliers. The augmented Lagrangian formulation is often preferred because it provides better numerical stability.

According to classical optimization theory [**359**], all the extrema of (8.10), whether local or global, are roots of the following sets of equations.

$$\nabla_{\mathbf{y}} \mathcal{L}(\mathbf{y},\lambda) = 0 \text{ and } \nabla_{\lambda} \mathcal{L}(\mathbf{y},\lambda) = 0 \tag{8.11}$$

These conditions are necessary to guarantee the (local) optimality to the solution of (8.8).

Search methods for solving (8.10) can be classified into local and global algorithms. Local minimization algorithms, such as gradient-descent and Newton's methods, find local minima efficiently and work best in uni-modal problems. Global methods, in contrast, employ heuristic strategies to look for global minima and do not stop after finding a local minimum [**406, 508, 359**]. Note that gradients and Hessians can be used in both local and global methods [**508**].

Local search methods can be used to solve (8.11) by forming a Lagrangian dynamic system that includes a set of dynamic equations to seek equilibrium points along a gradient path. These equilibrium points are called *saddle-points* of (8.11), which correspond to the constrained minima of (8.8). The Lagrangian dynamic system of equations are as follows.

$$\frac{d\mathbf{y}(t)}{dt} = -\nabla_{\mathbf{y}} \mathcal{L}(\mathbf{y}(t),\lambda(t)) \quad \text{and} \quad \frac{d\lambda(t)}{dt} = \nabla_{\lambda} \mathcal{L}(\mathbf{y}(t),\lambda(t)) \tag{8.12}$$

Optimal solutions to the continuous formulation are governed by the Saddle Point Theorem which states that $\mathbf{y}^*$ is a local minimum to the original problem defined in (8.8) if and only if there exists λ^* such that $(\mathbf{y}^*, \lambda^*)$ constitutes a saddle point of the associated Lagrangian function $F(\mathbf{y},\lambda)$. Here, a *saddle-point* $(\mathbf{y}^*, \lambda^*)$ of Lagrangian function $F(\mathbf{y},\lambda)$ is defined as one that satisfies the following condition.

$$F(\mathbf{y}^*,\lambda) \leq F(\mathbf{y}^*,\lambda^*) \leq F(\mathbf{y},\lambda^*) \tag{8.13}$$

for all $(\mathbf{y}^*, \lambda)$ and all $(\mathbf{y}, \lambda^*)$ sufficiently close to $(\mathbf{y}^*, \lambda^*)$.

There are four advantages in using a Lagrangian formulation to solve constrained optimization problems.

- Saddle points of (8.11) can be found by local gradient descent/ascent methods defined in (8.12). The first equation in (8.12) has a minus sign that optimizes the original variables along a descending path, whereas the second equation optimizes along an ascending path. Alternatively, (8.12) can be considered as a global search algorithm that has a local-search component based on a descent algorithm in the original variable space. When the search reaches a local minimum, the search is brought out of the local minimum using the weights imposed by its Lagrange multipliers. This mechanism allows the search to continue in its present trajectory without any breaks.
- Lagrangian search is similar to penalty-based methods in the sense that the Lagrange variables are increased like penalties when constraints are violated. However, it is more general than penalty-based methods because the increase of a Lagrange variable is self-adjusting and is governed by the amount that the corresponding constraint is violated.
- The search modeled by (8.12) can be started from any starting point and will continue until a saddle point is found.
- Since assignments of $\mathbf{y}$ where the constraints in (8.8) are satisfied are also assignments that minimize the objective, saddle points of (8.11) found by solving (8.12) correspond to satisfiable assignments to the original SAT problem.

It is important to note out that a Lagrangian search modeled by (8.12) is incomplete: if it does not find a solution in a finite amount of time, it does not prove whether the original SAT problem is satisfiable or not. Hence, infinite time will be required to prove unsatisfiability.

Unfortunately, continuous gradient-based local search methods for solving (8.12) are very time consuming. Our experience [**81**] indicates that continuous descent methods are several orders of magnitude more complex than discrete descent methods. For instance, it takes over one hour of CPU time on a Sun SS10 workstation to solve a problem with 200 variables and 60 constraints. Consequently, continuous formulations are not promising in solving large SAT problems. In the next subsection, we extend continuous Lagrangian methods to discrete Lagrangian methods. Surprisingly, discrete methods work much better and can solve some benchmark problems that cannot be solved by other local/global search algorithms.

8.7. Discrete Lagrangian-Based Constrained Optimization Algorithms. To overcome the computational complexity of continuous Lagrangian methods while preserving their benefits, we show in this subsection a discrete constrained formulation of a SAT problem and its solution using a discrete Lagrangian method. The discrete Lagrangian method is extended from the theory of continuous Lagrangian methods.

Recall (4.10) in Section 4.1 the following discrete constrained formulation of a SAT problem.

$$\begin{aligned} &\min_{\mathbf{y}\in\{0,1\}^n} && N(\mathbf{y}) = \sum_{i=1}^{m} U_i(\mathbf{y}) \\ &\text{subject to} && U_i(\mathbf{y}) = 0 \quad \forall i \in \{1, 2, \ldots, m\}. \end{aligned} \tag{8.14}$$

1. Set initial x randomly by a fixed random seed
2. Set initial λ to be zero
3. **while** x is not a solution, *i.e.*, $N(x) > 0$
4. update x: $x \longleftarrow x - \Delta_x L(x, \lambda)$
5. **if** condition for updating λ is satisfied **then**
6. update λ: $\lambda \longleftarrow \lambda + c \times U(x)$
7. **end if**
8. **end while**

FIGURE 23. Generic discrete Lagrangian algorithm $\mathcal{A}$ for solving SAT problems.

Without going into all the details [**538**], the continuous Lagrangian method can be extended to work on discrete problems. The *discrete Lagrangian function* for (8.14) is defined as follows.

$$L(\mathbf{y}, \lambda) = N(\mathbf{y}) + \lambda^T U(\mathbf{y}) \tag{8.15}$$

where $\mathbf{y} \in \{0,1\}^n$, $U(\mathbf{y}) = (U_1(\mathbf{y}), \ldots, U_m(\mathbf{y})) \in \{0,1\}^m$, and λ^T is the transpose of $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_m)$ that denotes the Lagrange multipliers. (Note that λ_i can be continuous variables.)

In a definition similar to that in (8.13), a saddle point $(\mathbf{y}^*, \lambda^*)$ of $L(\mathbf{y}, \lambda)$ in (8.15) is defined as one that satisfies the following condition.

$$L(\mathbf{y}^*, \lambda) \leq L(\mathbf{y}^*, \lambda^*) \leq L(\mathbf{y}, \lambda^*) \tag{8.16}$$

for all λ sufficiently close to λ^* and for all $\mathbf{y}$ whose Hamming distance between $\mathbf{y}^*$ and $\mathbf{y}$ is 1.

Similar to (8.12), the *Discrete Lagrangian Method* (DLM) for solving SAT problems can be defined as a set of difference equations,

$$\mathbf{y}^{k+1} = \mathbf{y}^k - \Delta_{\mathbf{y}} L(\mathbf{y}^k, \lambda^k) \tag{8.17}$$

$$\lambda^{k+1} = \lambda^k + U(\mathbf{y}^k), \tag{8.18}$$

where $\Delta_{\mathbf{y}} L(\mathbf{y}, \lambda)$ is the *discrete gradient operator* with respect to $\mathbf{y}$ such that $\Delta_{\mathbf{y}} L(\mathbf{y}, \lambda) = (\delta_1, \delta_2, \cdots, \delta_n) \in \{-1, 0, 1\}^n$, $\sum_{i=1}^n |\delta_i| = 1$, and $(\mathbf{y} - \Delta_{\mathbf{y}} L(\mathbf{y}, \lambda)) \in \{0,1\}^n$. Informally, $\Delta_{\mathbf{y}}$ represents all the neighboring points of $\mathbf{y}$.

8.8. An Implementation of a Basic Discrete Lagrangian Algorithm. Figure 23 shows the pseudo code of $\mathcal{A}$, a generic discrete Lagrangian algorithm implementing (8.17) and (8.18). It performs descents in the original variable space of $\mathbf{y}$ and ascents in the Lagrange-multiplier space of λ. In discrete space, $\Delta_{\mathbf{y}} L(\mathbf{y}, \lambda)$ is used in place of the gradient function in continuous space. We call one *iteration* as one pass through the while loop.

In the following, we describe some of the considerations in implementing DLM $\mathcal{A}$.

(a) *Initial Points and Restarts* (Lines 1-2). DLM is started from either the origin or from a random initial point generated by calling *drand48()* using a fixed random seed 101. Further, λ is always set to zero. The fixed initial points allow the results to be reproducible easily.

(b) *Descent and Ascent Strategies* (Line 4). There are two ways to calculate $\Delta_{\mathbf{y}} L(\mathbf{y}, \lambda)$: greedy and hill-climbing, each involving a search in the range of Hamming distance one from the current $\mathbf{y}$ (assignments with one variable flipped from the current assignment $\mathbf{y}$).

In a *greedy strategy*, the assignment leading to the maximum decrease in the Lagrangian-function value is selected to update the current assignment. Therefore, all assignments in the vicinity need to be searched every time, leading to computation complexity of $O(m)$, where m is the number of variables in the SAT problem. In *hill-climbing*, the first assignment leading to a decrease in the Lagrangian-function value is selected to update the current assignment. Depending on the order of search and the number of assignments that can be improved, hill-climbing strategies are generally less computationally expensive than greedy strategies.

A comparison of the two strategies show that hill-climbing is orders of magnitude faster and results in solutions of comparable quality.

(c) *Conditions for updating* λ (Line 5). The frequency in which λ is updated affects the performance of a search. The considerations here are different from those of continuous problems. In a discrete problem, descents based on discrete gradients usually make small changes in $L(\mathbf{y}, \lambda)$ in each update of $\mathbf{y}$ because only one variable changes. Hence, λ should not be updated in each iteration of the search to avoid biasing the search in the Lagrange-multiplier space of λ over the original variable space of $\mathbf{y}$.

Experimental results show that λ should be updated only when $\Delta_x L(x, \lambda) = 0$. At this point, a local minimum in the original variable space is reached, and the search can only escape from it by updating λ. This strategy amounts to pure descents in the original $\mathbf{y}$ variable space, while holding λ constant, until a local minimum is reached.

Note that this strategy is similar to Morris' "break out" strategy [**388**] and Selman and Kautz's *GSAT* [**473, 474**] that applies adaptive penalties to escape from local minima. One problem that is overlooked in these strategies is the growth of penalty terms. In solving a difficult SAT problem, penalty terms may grow to become very large as the search progresses, causing large swings in the objective function and delaying convergence of the search. Solutions to this issue are discussed next.

(d) *Amount of update of* λ (Line 6). A parameter c controls the magnitude of changes in λ. In general, c can be a vector of real numbers, allowing non-uniform updates of λ across different dimensions and possibly across time. For simplicity, $c = 1$ has been found to work well for most of the benchmarks tested. However, for some larger and more difficult problems, a smaller c can result in shorter search time.

The update rule in Line 6 results in nondecreasing λ. This is true because $U(x)$ is either 0 or 1: when a clause is not satisfied, its corresponding λ is increased; and when a clause is satisfied, its corresponding λ is not changed. In contrast, in applying Lagrangian methods to solve continuous problems with equality constraints, the Lagrange multiplier λ_i of constraint $g_i(x) = 0$ increases when $g_i(x) > 0$ and decreases when $g(x) < 0$.

When there is no mechanism to reduce the Lagrange multipliers, they can grow without bound, causing large swings in the Lagrangian-function value and making the search terrain more rugged. Although this strategy does not worsen the search time for most of the benchmark problems tested, λ values can become very large as

time goes on for a few difficult problems requiring millions of iterations. When this happens, the search has difficulty in identifying an appropriate direction to move.

This situation is illustrated in the first two graphs of Figure 24 that show the behavior of DLM when it was applied to solve one of the more difficult DIMACS SAT benchmark problems. Here, the search is stuck in a sub-optimal basin in the space of the objective function where the number of unsatisfied clauses fluctuates around 20. Since the search terrain modeled by L becomes more rugged as the Lagrange multipliers increase, the search will have difficulty to escape from this region.

To overcome this problem, λ should be reduced periodically. For instance, in the last two graphs of Figure 24, λ was scaled down by a factor 1.5 every 10,000 iterations. This strategy, when combined with other strategies to be discussed next, restricts the grown of Lagrange multipliers, leading to the solution of some of the more difficult benchmark problems.

(e) *Plateaus in the Search Space.* In binary problems like SAT, a search may find a very small subset of variables that can lead to no degradation in the objective function. Flipping variables in this small subset successively may likely lead to a cycle in the search space. To avoid such an undesirable situation, variables that have been flipped in the recent past can be stored in a tabu list [**199, 239**] and will not be flipped until they are out of the list.

Further, for large SAT problems formulated as discrete optimization problems, the search may encounter large and flat, but suboptimal, basins. Here, gradients in all directions are the same and the search may wander forever. The discrete gradient operator $\Delta_{\mathbf{y}} L(\mathbf{y}, \lambda)$ may have difficulties in basins/plateaus because it only examines adjacent points of $L(y, \lambda)$ that differ in one dimension. Hence, it may not be able to distinguish a plateau from a local minimum.

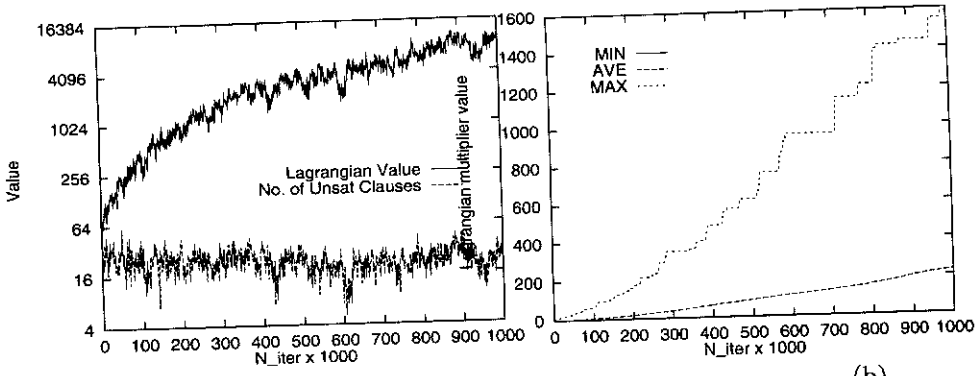
One way to escape is to allow uphill moves. For instance, in *GSAT*'s random walk strategy [**475**], uphill walks are allowed based on probability p. However, the chance of getting a sequence of uphill moves to get out a deep basin is small since each walk is independent.

There are two effective strategies that allow a plateau to be searched.

(a) *Flat-move strategy.* We need to determine the time to change λ when the search reaches a plateau. As indicated earlier, updating λ when the search is in a plateau changes the surface of the plateau and may make it more difficult for the search to find a local minimum somewhere inside the plateau. To avoid this situation, a strategy called *flat move* [**538**] can be employed. This allows the search to continue for some time in the plateau without changing λ, so that the search can traverse states with the same Lagrangian-function value. How long should flat moves be allowed is heuristic and possibly problem dependent. Note that this strategy is similar to Selman's "sideway-move" strategy [**474**].

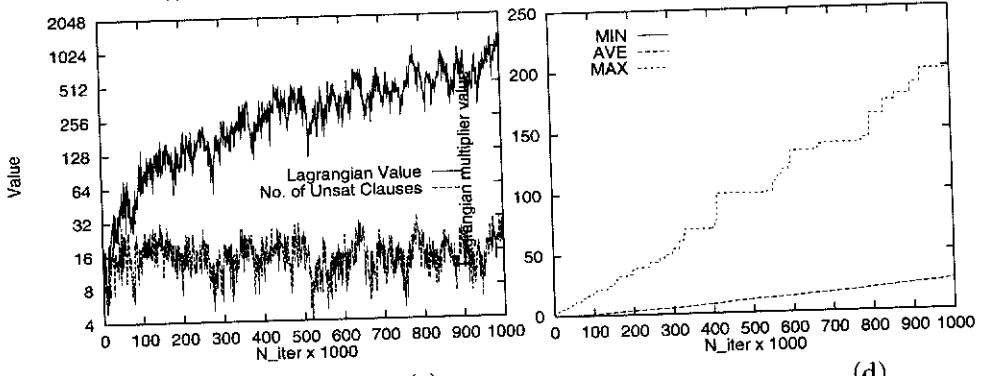
(b) *Tabu list.* This search strategy aims to avoid revisiting the same set of states in a plateau. In general, it is impractical to remember every state the search visits in a plateau due to the large storage and computational overheads. A tabu list [**199, 239**] can be kept to maintain the set of variables flipped in the recent past and to avoid flipping a variable if it is in the tabu list.

The last four graphs of Figure 24 illustrate the performance of DLM when the search maintains a tabu list of size 30, when it is allowed to stay in a basin



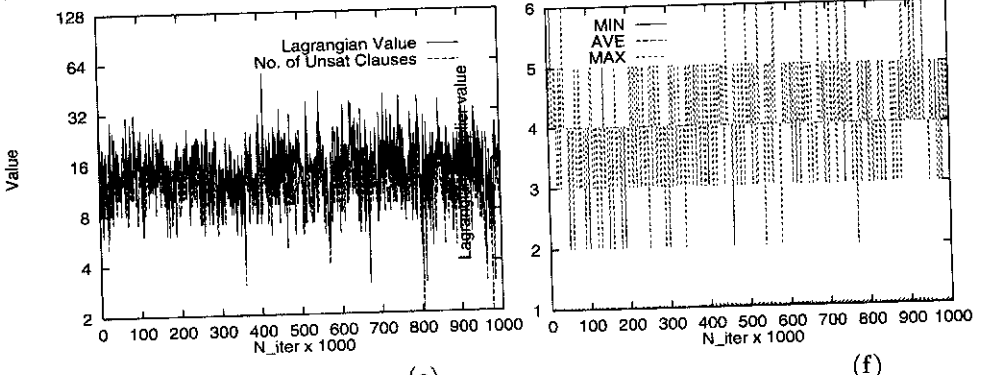
(a) (b)

Without tabu list, flat moves and periodic scaling of λ



(c) (d)

With tabu list of size 30, flat moves of limit 50, but without periodic scaling of λ



(e) (f)

With tabu list of 30, flat moves of limit 50, and periodic scaling of λ by a factor of 1.5 every 10,000 iterations

FIGURE 24. Execution profiles of "*g125-17*," one of the difficult DIMACS benchmark problem. Figures (a), (c), and (e) plot the Lagrangian-function values and the number of unsatisfied clauses versus the number of iterations. Figures (b), (d), and (f) plot the minimum, average and maximum values of Lagrange multipliers.

within 50 moves (flat-move limit), and when all Lagrange multipliers are periodically scaled down. These graphs show significant reduction in the growth of Lagrangian-function values and Lagrange multipliers.

By using these strategies, DLM can solve successfully many of the hard problems in the DIMACS benchmark suite [**538**]. These results are presented in Section 13.

8.9. Convergence Property and Average Time Complexity. Gu, Gu and Du [**218**] have analyzed the convergence ratios of three basic methods: the steepest descent method, Newton's method, and the coordinate descent method for objective function f defined in the *UniSAT7* input model. They prove that, subject to certain conditions [**359**], the steepest descent method has a linear convergence ratio $[(A-a)/(A+a)]^2 < 1$, Newton's method has a convergence ratio of order two, and the coordinate descent method has a convergence ratio $\left(1 - \frac{a}{A(n-1)}\right) < 1$, where $A \geq a > 0$ are the largest and smallest eigenvalues of the Hessian matrix $\mathbf{H(y)}$, respectively.

From these convergence properties, Gu, Gu, and Du roughly estimate that, subject to certain conditions [**359**], the *UniSAT7* problem can be solved in $O(\log(n+m))$ iterations by the steepest descent method and can be solved in $O(m\log(n+m))$ iterations by the coordinate descent method, on the assumption that the algorithm is not stuck at a local minimum point.

Gu and Gu have made some preliminary analysis of the typical time complexity of some global optimization SAT algorithms [**214**]. It shows that, the *SAT*14.5 algorithm, with probability at least $1-e^{-n}$, finds a solution within $k = O(n(\log n)^2)$ iterations of the *while* loop for a randomly generated satisfiable *CNF* formula with $l \geq 3$ and $m/n \leq \alpha 2^l/l$, where $\alpha < l$ is a constant. From this and the fact that the run time of procedure *enumerate*() is $O(2^n)$, the typical time complexity of the *SAT*14.5 algorithm is

$$(1-e^{-n})O(n(\log n)^2(lmn)) + e^{-n}O(2^n) = O(ln(n\log n)^2).$$

Clearly, the *SAT*14.5 algorithm can give an answer to an unsatisfiable *CNF* formula in $O(2^n)$ time.

9. Integer Programming Method

In this section, we first give an integer program (IP) formulation of SAT. Then we describe some traditional techniques of using the integer programming approach to solve SAT.

9.1. An Integer Programming Formulation for SAT. In order to represent SAT inputs in the framework of mathematical programming, we identify logic value *true* with integer 1 and *false* with -1. Similar in *UniSAT* models (Section 8.1), all Boolean $\vee$ and $\wedge$ connectives are transformed into $+$ and $\times$ of ordinary addition and multiplication operations, respectively. Using a standard transformation, the ith clause C_i is transformed into a linear inequality [**303, 301**]:

$$\sum_{j=1}^{n} q_{i,j}(w_j) \geq 2 - |C_i| \tag{9.1}$$

where

$$q_{i,j}(w_j) = \begin{cases} w & \text{if literal } x_j \text{ is in clause } C_i \\ -w & \text{if literal } \bar{x}_j \text{ is in clause } C_i \\ 0 & \text{if neither } x_j \text{ nor } \bar{x}_j \text{ is in } C_i \end{cases} \tag{9.2}$$

where w_j is the jth integer variable.

To restrict $w_j = \pm 1$, $j = 1, 2, ...n$, requires that extra constraints be added to insure that each w_j be in the closed interval $[-1, 1]$, *i.e.*, $-1 \leq w_j \leq 1$ for $j = 1, 2, ...n$.

Following the above formulation, for example, a *CNF* $\mathcal{F}$

$$(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3)$$

is translated into

$$w_1 - w_2 \geq 0$$

$$-w_1 + w_2 + w_3 \geq -1$$

$$w_2 + w_3 \geq 0$$

so an integer programming formulation is obtained for SAT as: finding $w_j = \pm 1$, such that

$$\begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} \geq \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \tag{9.3}$$

or

$$\begin{pmatrix} -1 & 1 & 0 \\ 1 & -1 & -1 \\ 0 & -1 & -1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{9.4}$$

While the Simplex method is effective for solving linear programs (LP), there is no single technique that is fast for solving integer programs. Therefore, approaches developed try to solve the integer program as an integer linear program (ILP).[7] If the solution is non-integer, one rounds off the values to the nearest integers and checks whether this corresponds to a solution of the original input. If the rounded off values do not correspond to a solution, adds a new constraint and computes a solution of the modified linear program. So far most methods developed to solve the integer programs for SAT indirectly work on the corresponding integer linear programs.

Researchers have observed that the optimal integer-programming solution is usually not obtained by *rounding* the linear-programming solution although this is possible in certain cases (see Section 10). The closest point to the optimal linear-program may not even be feasible. In some cases, the nearest feasible integer point to the linear-program solution is far removed from the optimal integer point. Thus, when using an integer linear program to solve the integer program for SAT, it is not sufficient simply to round linear-programming solutions.

In the following sections, we describe existing integer programming methods to solve SAT.

[7] An integer linear program (ILP) is a linear program further constrained by integrality restrictions.

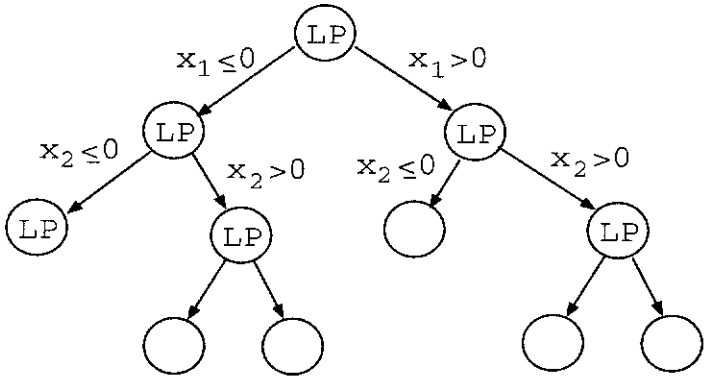


FIGURE 25. An example of a branch-and-bound tree.

9.2. Linear Program Relaxation. A basic method to solve an integer program is the linear program relaxation. In this approach, the LP relaxation is achieved by replacing $x_i \in \{0,1\}$ with $0 \leq x_i \leq 1$. The LP relaxation can be solved efficiently with some sophisticated implementations of Dantzig's Simplex method, such as MINOS [**389**], or some variations of Karmarkar's interior point method [**305**].

Hooker early reported that by solving a linear programming of SAT, one frequently produces an integer solution [**259**]. Kamath *et al.* used MINOS 5.1 to solve linear programming relaxation [**303, 301**]. They tried some small SAT inputs and found that the Simplex method failed to find integral solutions to the linear programming relaxations in majority of instances tested.

9.3. Branch and Bound Method. Branch-and-bound is essentially a strategy of "divide and conquer." It is a straightforward and the most successful way to solve the integer programming problem. The idea is to systematically partition the linear-programming feasible region into manageable subdivisions and make assessments of the integer-programming problem based on these subdivisions. When moving from a region to one of its subdivisions, we add one constraint that is not satisfied by the optimal linear-programming solution over the parent region. So the linear programs corresponding to the subdivisions can be solved efficiently. In general, there are a number of ways to divide the feasible region, and as a consequence there are a number of branch-and-bound algorithms.

We show the basic procedures of branch-and-bound with a simple example shown in Figure 25. The method starts with the fractional solution given by its corresponding LP relaxation. Then a variable of fractional solution is selected. For example, let x_1 be a variable, and set $x_1 \leq 0$ as an additional constant; *i.e.*, branch on x_1 with constraint $x_1 \leq 0$. Resolve the LP relaxation with this augmented constraint. If it still produces a non-integer solution, branch on another non-integer variable, say x_2, first with constraint $x_2 \leq 0$, and resolve the LP with extra constraint $x_1 \leq 0$ and $x_2 \leq 0$. This process continues until solving the augmented LP yields an integer solution, *i.e.*, an incumbent solution, so there is no need to branch

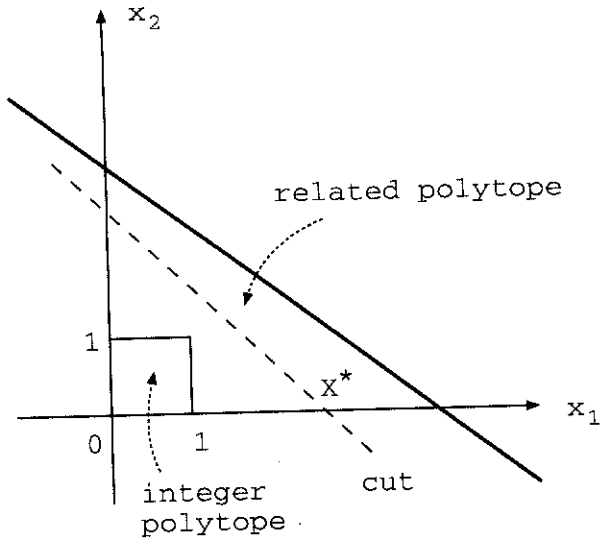


FIGURE 26. An illustration of the cutting-plane method.

further at that node. Since we do not know this to be optimal, a backtracking procedure is required to search with extra constraints $x_1 \leq 0$ and $x_2 \geq 0$ and resolve the augmented LP and continue the process until an integer solution is obtained.

The above process produces a binary tree as shown in Figure 25. In this way, we implicitly exhaust all possibilities and conclude with an optimal solution. Note that each time we obtain an incumbent solution we get a new upper bound on the minimum value of the objective function. It at the same node the LP yields an objective function with value that exceeds the best upper bound obtained so far, then we can *fathom* that node, since any solution obtained at its successors can only be worse.

9.4. Cutting-Plane Method. Unlike partitioning the feasible region into subdivisions, as in branch-and-bound approaches, the cutting-plane algorithm solves integer programs by modifying linear-programming solutions until an integer solution is obtained. It works with a single linear program, which it refines by adding new constraints. The new constraints successively reduce the feasible region until an integer optimal solution is found.

The idea of the cutting plane method can be illustrated from a simple geometric interpretation (Figure 26). The feasible region for the integer program, *i.e.*, an integer polytope, consists of those integer lattice points satisfying all constraints. A *cut* is an inequality satisfied by all the feasible solutions of the integer program. A cutting plane is a hyperplane defined by that inequality and it conflicts with the solution X^* of the linear-programming relaxation. The cutting plane passes between X^* and the integer polytope and cuts off a part of the relaxed polytope containing the optimal linear-programming solution X^* without excluding any feasible integer points. After the cut, the resulting linear program is solved again. If the optimal

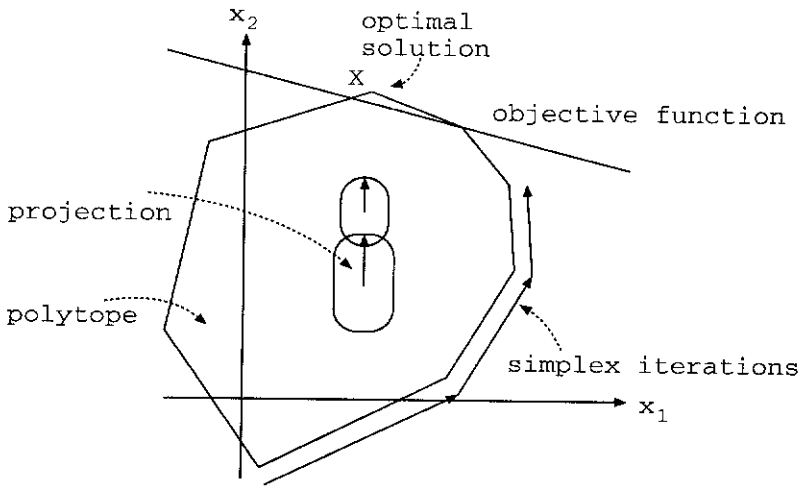


FIGURE 27. The basic ideas of the Simplex method and Karmarkar's method.

values for the decision variables in the linear program are all integer, they are optimal; otherwise, a new cut is derived from the new optimal linear-programming tableau and appended to the constraints.

In practice, the branch-and-bound procedures almost always outperform the cutting-plane algorithm. Nevertheless, the algorithm has been important to the evolution of integer programming. Historically, it was the first algorithm developed for integer programming that could be proven to converge in a finite number of steps. In addition, even though the algorithm generally is considered very inefficient, it has provided insights into integer programming that have led to other, more efficient algorithms.

9.5. Interior Point Method. The most important advance in linear programming solution techniques was recently introduced by Karmarkar [**305**]. As shown in Figure 27, compared to the Simplex method which jumps from a corner point to another corner point of the LP polytope until the optimal solution is found, Karmarkar's algorithm constructs an ellipsoid inside the polytope and uses nonlinear transformations to project better solution guesses in the interior of the polytope. Unlike the Simplex method which approaches the optimal solution indeed by step-by-step searching and has an exponential worst-case complexity, Karmarkar's algorithm has been proven to be a polynomial time algorithm.

To apply Karmarkar's algorithm on integer programming, first the 0/1 integer program is transformed to a ± 1 integer program. Then the potential function x^2 is used, and obviously the optimal integer solution to the original IP problem is at the point that the potential function achieves a maximum. However, using Karmarkar's algorithm on integer programming may get stuck at a local minimum, *i.e.*, it does not guarantee to find the optimal solution by projection. Therefore, it is an incomplete algorithm.

9.6. Improved Interior Point Method. It is expected that a sequence of interior points

$$w^{k+1} = w^k + \alpha \triangle w^* \tag{9.5}$$

is generated such that the potential function in Karmarkar's algorithm is minimized. It is crucial to determine the descent direction $\triangle w^*$ of the potential function around w^k and the step size α.

In the original Karmarkar's algorithm, the step size α is assumed with (0,1]. They used $\alpha = 0.5$ in their experiments to solve SAT inputs. If the potential function is well represented by the quadratic approximation around the given point, then if we move along the Newton direction and have the appropriate values for certain parameters, we will reach the minimum; otherwise, recall that the step size is chosen so that it reaches a minimum of the objective function on that line of the given descent direction. So there is no reason to restrict α within (0,1].

This suggests the necessity to use line search to choose optimal step size. Following this idea, Shi, Vannelli, and Vlach have recently given an improved interior point algorithm [**479**]. In their algorithm, the step size α is determined by a golden-section search [**359**]. Experiments show significant improvements on Karmarkar's algorithm.

10. Special Subclasses of SAT

Certain subclasses of SAT that are known to be solved in polynomial time have been identified and explored. There are at least three reasons for discussing such subclasses in this section. First, a given formula can be preprocessed and examined to determine whether it is a member of a polynomial-time solvable subclass of SAT. If so, a special, fast algorithm can be brought to bear on the formula. Second, a portion of a given formula may a member of such a subclass and its solution may make solving the given formula easier. Third, study of such subclasses reveals, in part, the nature of "easy" SAT formulas. On the other hand, as reported in Section 12, studies of random formulas indicate that these known classes contain only a small fraction of the formulas that can be solved rapidly.

Below, we consider some of the more notable polynomial-time subclasses. When we say *apply unit resolution* we mean apply the unit clause rule to exhaustion.

10.1. 2-SAT. A *CNF* formula containing clauses of one or two literals only is solved in linear time by applying unit resolution [**18, 155**].

10.2. Horn and Extended Horn Formulas. A *CNF* formula is Horn if every clause in it has at most one positive literal. This class is widely studied, in part because of its close association with Logic Programming. Horn formulas can be solved in linear time using unit resolution [**144, 279, 469**].

The class of extended Horn formulas was introduced by Chandru and Hooker [**79**] who were looking for conditions under which a Linear Programming relaxation could be used to find solutions to propositional formulas. A theorem of Chandrasekaran [**84**] characterizes sets of linear inequalities for which 0-1 solutions can always be found (if one exists) by rounding a real solution obtained using an LP relaxation. Extended Horn formulas can be expressed as linear inequalities that belong to this family of 0-1 problems. The following graph-theoretic characterization, taken from [**504**], is simpler than the LP characterization.

Let C be a clause constructed from a variable set V, and let R be a rooted directed tree with root s (*i.e.*, a directed tree with all edges directed away from s) and with edges uniquely labeled with variables in V. Then C is *extended Horn w.r.t. R* if the positive literals of C label a (possibly empty) dipath P of R, and the set of negative literals in C label an edge-disjoint union of dipaths $Q_1, Q_2, ..., Q_t$ of R with exactly one of the following conditions satisfied:

1. $Q_1, Q_2, ..., Q_t$ start at the root s.
2. $Q_1, Q_2, ..., Q_{t-1}$, (say), start at the root s, and Q_t and P start at a vertex $q \neq s$ (if P is empty, Q_t can start from any vertex).

A clause is *simple extended Horn w.r.t. R* if it is extended Horn w.r.t. R and only Condition 1 above is satisfied. A *CNF* formula is (*simple*) *extended Horn w.r.t. R* if each of its clauses is (simple) extended Horn w.r.t. R. A formula is (*simple*) *extended Horn* if it is (simple) extended Horn w.r.t. some such rooted directed tree R.

One tree R for a given Horn formula is a star (one root and all leaves with an edge for each variable in the formula). Hence, the class of extended Horn formulas is a generalization of the class of Horn formulas.

Chandru and Hooker [**79**] showed that unit resolution alone can determine whether or not a given extended Horn formula is satisfiable. A satisfying truth assignment for a satisfiable formula may be found by applying unit resolution, setting values of unassigned variables to $1/2$ when no unit clauses remain, and rounding the result by a matrix multiplication [**79**]. This algorithm cannot, however, be reliably applied unless it is known that a given formula is extended Horn. Unfortunately, the problem of recognizing extended Horn formulas is not known to be solved in polynomial time.

10.3. Formulas from Balanced $(0, \pm 1)$ Matrices. The class of formulas from balanced $(0, \pm 1)$ matrices, which we call balanced formulas here, has been studied by several researchers (see [**100**] for a detailed account of balanced matrices and a description of balanced formulas). The motivation for this class is the question, for SAT, when do Linear Programming relaxations have integer solutions?

Express a *CNF* formula of m clauses and n variables as an $m \times n$ $(0, \pm 1)$-matrix M where the rows are indexed on the clauses, the columns are indexed on the variables, and a cell $M(i, j)$ has the value $+1$ if clause i has variable j as an unnegated literal, the value -1 if clause i has variable j as a negated literal, and the value 0 if clause i does not have variable j as a negated or unnegated literal. A *CNF* formula is a *balanced formula* if in every submatrix of M with exactly two nonzero entries per row and per column, the sum of the entries is a multiple of four [**510**].

Let a *CNF* formula be cast, in standard fashion, as a linear programming problem of the form $\{x : Mx \geq 1 - n(M), 0 \leq x \leq 1\}$ where $n(M)$ is a column vector whose components are the number of negated literals in clauses at the rows corresponding to those components. If M is balanced, then for every submatrix A of M, the solution to $\{x : Ax \geq 1 - n(A), 0 \leq x \leq 1\}$ is integral [**100**]. From this it follows that balanced formulas may be solved in polynomial time using linear programming.

Balanced formulas have the property that, if every clause contains more than one literal, then for every variable v there are two satisfying truth assignments: one

with v set to *true* and one with v set to *false*. Thus, the following is a simple linear-time algorithm for finding solutions to known balanced formulas [**100**]. Apply unit resolution to the given formula. If a clause is falsified, the formula is unsatisfiable. Otherwise, repeat the following as long as possible: choose a variable and set its value to *true*, then apply unit resolution. If a clause becomes falsified, then the formula is unsatisfiable, otherwise all clauses have been satisfied by the assignment resulting from the variable choices and unit resolution.

Unlike extended Horn formulas, balanced formulas are known to be recognized in polynomial time [**100**].

10.4. Single-Lookahead Unit Resolution. This class was developed as a generalization of other classes including Horn, extended Horn, simple extended Horn, and balanced formulas [**466**]. It is peculiar in that it is defined based on an algorithm rather than on properties of formulas. The algorithm, called SLUR, selects variables sequentially and arbitrarily and considers both possible values for each selected variable. If, after a value is assigned to a variable, unit resolution does not result in a clause that is falsified, the assignment is made permanent and variable selection continues. If all clauses are satisfied after a value is assigned to a variable (and unit resolution is applied), the algorithm returns a satisfying assignment. If unit resolution, applied to the given formula or to both sub-formulas created from assigning values to the selected variable on the first iteration, results in a clause that is falsified, the algorithm reports that the formula is unsatisfiable. If unit resolution results in falsified clauses as a consequence of both assignments of values to a selected variable on any iteration except the first, the algorithm reports that it has given up.

A formula is in the class SLUR if, for all possible sequences of selected variables, SLUR does not give up on that formula. SLUR takes linear time with the modification, due to Truemper [**513**], that unit resolution be applied simultaneously to both branches of a selected variable, abandoning one branch if the other finishes first without falsifying a clause. Note that due to the definition of this class, the question of class recognition is avoided.

All Horn, extended Horn, and balanced formulas are in the class SLUR. Thus, an important outcome of the results on SLUR is the observation that no special preprocessing or testing is needed for a variety of special subclasses of SAT when using a reasonable variant of the DPL algorithm.

A limitation of all the classes above is that they do not represent many interesting unsatisfiable formulas. There are several possible extensions to SLUR which improve the situation. One is to add a 2-SAT solver to the unit resolution step. This extension is at least able to handle all 2-SAT formulas which is something SLUR cannot do. This extension can be elegantly incorporated into SLUR due to an observation of Truemper: "Whenever SLUR completes a sequence of unit resolutions, and if at that time the remaining clauses are nothing but a subset of the original clauses (which they would have to be if all clauses have at most two literals), then effectively the SLUR algorithm can start all over. That is, if fixing of a variable to both values leads to an empty clause, then the formula has been proved to be unsatisfiable. Thus, one need not augment SLUR by the 2-SAT algorithm, because the 2-SAT algorithm (at least one version of it) does exactly what the modified SLUR does." Another extension of SLUR is to allow a polynomial number of backtracks, giving up if at least one branch of the DPL tree does not

terminate at a leaf where a clause is falsified. Thus, unsatisfiable formulas with short DPL trees can be solved. However, such formulas are uncommon.

10.5. q-Horn Formulas. This class of propositional formulas was developed by Boros, Crama, Hammer, Saks, and Sun in [**44**] and [**43**]. We choose to characterize the class of q-Horn formulas as a special case of monotone decomposition of matrices [**511, 513**]. As in the case of balanced $(0, \pm 1)$ matrices, express a *CNF* formula of m clauses and n variables as an $m \times n$ $(0, \pm 1)$-matrix M where the rows are indexed on the clauses, the columns are indexed on the variables, and a cell $M(i, j)$ has the value $+1$ if clause i has variable j as an unnegated literal, the value -1 if clause i has variable j as a negated literal, and the value 0 if clause i does not have variable j as a negated or unnegated literal. In the monotone decomposition of M, columns are scaled by -1 and the rows and columns are partitioned into submatrices as follows:

$$\left(\begin{array}{c|c} A^1 & E \\ \hline D & A^2 \end{array}\right)$$

where the submatrix A^1 has at most one $+1$ entry per row, the submatrix D contains only -1 or 0 entries, the submatrix A^2 has no restrictions other than the three values of -1, $+1$, and 0 for each entry, and the submatrix E has only 0 entries. If the monotone decomposition of M is such that A^2 has no more than two nonzero entries per row, then the formula represented by M is *q-Horn*.

A recent result by Truemper [**513**] can be used to find a monotone decomposition for a matrix associated with a q-Horn formula in linear time. Once a q-Horn formula is in its decomposed form it can be solved in linear time as follows. Treat submatrix A^1 as a Horn formula and solve it in linear time using a method such as in [**144, 279, 469**] which returns a minimum, unique truth assignment for the formula with respect to *true*. If the Horn formula is unsatisfiable then the q-Horn formula is unsatisfiable. Otherwise, the returned assignment satisfies A^1 and some or all rows of D. The set of *true* variables in every truth assignment satisfying A^1 contains the set of variables *true* in the returned minimum, unique truth assignment. Therefore, since elements of D are either 0 or -1, no truth assignment satisfying A^1 can satisfy any rows of D that are not satisfied by the minimum, unique truth assignment. Hence, the only way A^1 and D both can be satisfied is if A^2, minus the rows collinear with those of D that are satisfied by the minimum, unique truth assignment, can be satisfied. Since A^2 represents a 2-SAT formula, any subset is also 2-SAT and can be solved in linear time. If the answer is unsatisfiable then the q-Horn formula is unsatisfiable; if the answer is satisfiable then such a satisfying assignment plus the minimum, unique truth assignment returned earlier are a solution to the q-Horn formula.

The developers of the class q-Horn also offer a linear-time solution to formulas in this class. The main result of [**43**] is that a q-Horn formula can be recognized in linear time. See [**42**] for a linear-time algorithm for solving q-Horn formulas.

Formulas in the class q-Horn are thought to be close to what might be regarded as the largest easily definable class of polynomially solvable propositional formulas because of a result due to Boros, Crama, Hammer, and Saks [**44**]. Let $\{v_1, v_2, ..., v_n\}$ be a set of Boolean variables, and P_k and N_k, $P_k \cap N_k = \emptyset$ be subsets of $\{1, 2, ..., n\}$ such that the kth clause in a *CNF* formula is given by $\vee_{i \in P_k} v_i \vee_{i \in N_k} \bar{v}_i$. Construct

the following system of inequalities:

$$\sum_{i \in P_k} \alpha_i + \sum_{i \in N_k} (1 - \alpha_i) \leq Z, \ (k = 1, 2, ..., m), \text{ and}$$

$$0 \leq \alpha_i \leq 1, \ (i = 1, 2, ..., n).$$

where $Z \in R^+$. If all these constraints are satisfied with $Z \leq 1$ then the formula is q-Horn. On the other hand, the class of formulas such that the minimum Z required to satisfy these constraints is at least $1 + 1/n^\epsilon$, for any fixed $\epsilon < 1$, is NP-complete. For more information on the subject of q-Horn formulas will appear in [**513**].

10.6. Renamable Formulas. Suppose clauses of a *CNF* formula $\mathcal{F}$ are constructed from a set V of variables and let $V' \subset V$. Define $switch(\mathcal{F}, V')$ to be the formula obtained as follows: for every $v \in V'$, reverse all unnegated occurrences of v in $\mathcal{F}$ to negated occurrences and all negated occurrences of v to unnegated occurrences. For a given formula $\mathcal{F}$, if there exists a $V' \subset V$ such that $switch(\mathcal{F}, V')$ is Horn, extended Horn, etc., then the formula is said to be renamable Horn, extended Horn, etc., respectively.

The algorithms given above work even if a given formula is renamable to a formula in the class for which they apply. Additional classic references to Horn renamability are [**339**] and [**19**].

It is interesting to note that there exist formulas in the class of SLUR formulas that are not members of either renamable extended Horn formulas or balanced formulas [**466**].

10.7. Formula Hierarchies. Some sets of clauses not falling into one of the polynomially solvable classes defined above may be reduced to "equivalent" formulas that are members of at least one of these classes. If such reductions are efficient, these sets can be solved in polynomial time. Such reductions can take place in stages where each stage represents a class of polynomially solved formulas, and lower stages represent classes of perhaps lower time complexity than classes represented by higher stages. The lowest stage is a polynomially solved base class, such as one of the classes above.

An example of such a hierarchy is found in [**182**]. The base class, at stage 0, is Horn. Consider a stage 1 formula that is not Horn. By definition of the hierarchy, there is a variable v which, if set to *true*, leaves a set of non-satisfied clauses and non-falsified literals that is Horn. If this Horn formula is found to be satisfiable, we can conclude the original formula is. Otherwise, setting v to *false* leaves a set of clauses that is a stage 1 formula (empty formulas are considered to belong to every stage). Thus, the above process can be repeated (on stage 1 formulas) to exhaustion. Since it takes linear time to solve Horn formulas and in the worst-case a linear number of Horn systems must be considered, the process for solving formulas at stage 1 has quadratic complexity. The above concept can be expanded to higher stages to form a hierarchy: at stage i, when setting v to *true*, a sub-formula is at stage $i - 1$, and when setting v to *false*, a sub-formula is at stage i. Thus, solutions to stage i formulas are carried out recursively leading to a time complexity that is bounded by m^i. An alternative way to solve formulas at stage i in the hierarchy is to use i-resolution (resolution is not applied unless at least one clause has at most i literals) [**62**].

The only remaining question is to determine whether a given formula is a stage i formula. This can be done with a bottom-up approach described in [**182**].

For other information on such Hierarchies see, for example, [**113, 184**].

10.8. Pure Implication Formulas. Pure implication formulas are defined recursively as follows:

1. A variable is a pure implication formula.
2. If $\mathcal{F}_1$ and $\mathcal{F}_2$ are pure implication formulas then $(\mathcal{F}_1 \rightarrow \mathcal{F}_2)$ is a pure implication formula.

Eliminating parentheses on right to left associativity, a pure implication formula can be written $\mathcal{F}_1 \rightarrow \mathcal{F}_2 \rightarrow \ldots \rightarrow \mathcal{F}_p \rightarrow z$ where z is a variable. We call the z variable of a formula the *right-end* variable.

The satisfiability problem is trivial for a pure implication formula but the problem of falsifiability is NP-complete even if all variables except the right-end variable occur at most twice in the formula. Furthermore, the complexity of determining falsifiability seems to increase at least exponentially with the number of occurrences of the right-end variable [**249**]; this yields a hierarchy of classes starting from linear-time solvability and going through NP-completeness. This is possibly due to the fact that the expressive power of pure implication formulas at the lower levels of the hierarchy is extremely limited. Despite this lack of expressibility, it seems that the lower levels of the hierarchy are incomparable with other special polynomial-time-solvable classes such as 2-SAT and SLUR. To make this more concrete, define a class of *CNF* formulas related to pure implication formulas and call it PICNF(k).

A formula in the class PICNF(k) consists only of the following kinds of clause groups:

1: $(\bar{x}_{\pi_1} \vee \bar{x}_{\pi_2} \vee x_{\pi_3}) \wedge (x_{\pi_1} \vee x_{\pi_2}) \wedge (x_{\pi_1} \vee \bar{x}_{\pi_3})$
2: $(\bar{x}_{\pi_1} \vee \bar{x}_{\pi_2} \vee \bar{x}_{\pi_3}) \wedge (x_{\pi_1} \vee x_{\pi_2}) \wedge (x_{\pi_1} \vee x_{\pi_3})$
3: (x_{π_1})

where the number of type 2 groups is fixed at k and each variable occurs at most twice in a PICNF(k) formula. The falsifiability question for a given pure implication formula with right-end variable occurring at most k times is identical to the satisfiability question for a formula in class PICNF(k). If all but one totally negated clauses are removed from such a formula, a complete set of at most n partial truth assignments, each of which can be extended to satisfying truth assignments, can be constructed in linear time. Doing this for each totally negated clause results in k such sets of partial truth assignments. Multiplying these sets to find consistent assignments spanning all k sets can determine whether the given formula is satisfiable. This can be accomplished in $O(n^k)$ time, matching the complexity of falsifiability of pure implication formulas. A recent result [**172**] shows this can be reduced to $O(k^k n^2)$ time. We remark that the problem of determining satisfiability for formulas of the union of the classes PICNF(k), for all k, is NP-complete.

The class PICNF(k), k fixed, is incomparable to other polynomially solved classes discussed above. For example, there are SLUR *CNF* formulas that are not represented as PICNF(k) formulas and vice versa (particularly many unsatisfiable PICNF(k) formulas are not SLUR *CNF* formulas). Also, although it is easy to construct a PICNF(k) formula that is renamable Horn (and therefore q-Horn), even

the PICNF(1) set $(\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (x_1 \vee x_3) \wedge (x_1 \vee x_4)$ is not q-Horn.

PICNF(k) is interesting because, for k fixed, it contains formulas that are not in other polynomial-time solvable classes and the severe lack of expressibility of PICNF(k) formulas may be exploited to assist complexity investigations of class hierarchies. In particular, why should the hierarchies discussed above have $O(|F|^k)$ complexity when a complexity of $O(2^k|F|)$, say, is not inconsistent with any developed theory? PICNF(k) may be useful in answering this question.

10.9. Non-linear Formulations. An optimization problem with 0-1 variables can be reduced to a constrained nonlinear 0-1 program. Such programs are expressed as follows:

$$\max_{\mathbf{x}\in\{0,1\}^n} F(\mathbf{x}) = \sum_{k=1}^{p} c_k T_k \tag{10.1}$$

subject to

$$g_i(\mathbf{x}) = \sum_{k=1}^{p_i} a_{ik} T_{ik} < b_i;\ i = 1, 2, \ldots, m$$

where

$$T_k = \prod_{j\in N_k} x_j;\ N_k \subseteq N = \{1, 2, \ldots, n\},\ k = 1, 2, \ldots, p \tag{10.2}$$

and

$$T_{ik} = \prod_{j\in N_{ik}} x_j;\ N_{ik} \subseteq N,\ k = 1, 2, \ldots, p_i,\ i = 1, 2, \ldots, m$$

Problems in propositional logic, originating, for example, in graph theory, can be expressed this way by associating 0 to *false*, 1 to *true*, $1 - a$ to $\bar{a}$, and $a \cdot b$ to $a \wedge b$.

Several methods for solving the above formulation have been proposed [**235, 237**]. In some restricted cases these methods have polynomial time complexity. Thus, the rich literature on this subject can be carried over to the domain of propositional satisfiability to provide low complexity algorithms for SAT under corresponding conditions.

A notable example involves functions for which the co-occurrence graph is a partial k-tree [**107**]. The *DNF* formulation expressed by equations (4.7)-(4.8) is in the form of equations (10.1) and (10.2). Let $F(\mathbf{x})$ be such a *DNF* function. The co-occurrence graph of $F(\mathbf{x})$ has a vertex set corresponding to the variables $\{x_1, x_2, \ldots, x_n\}$ with an edge between x_i and x_j $(i \neq j)$ if these variables occur simultaneously in at least one product term of $F(\mathbf{x})$. A simple, undirected graph G is a k-tree if there is an ordering $\{x_{\pi_1}, x_{\pi_2}, \ldots, x_{\pi_n}\}$ of all its vertices such that, for all $j = 1, 2, \ldots, n-k$, in the subgraph G_j induced by vertices $\{x_{\pi_j}, x_{\pi_{j+1}}, \ldots, x_{\pi_n}\}$ the vertex x_{π_j} has degree k and its neighbors induce a complete subgraph of G_j. A partial k-tree is any graph obtained by deleting edges from a k-tree. If the co-occurrence graph of $F(\mathbf{x})$ is a partial k-tree, then $F(\mathbf{x})$ can be solved in linear time [**107**]. Since the maximization problem for *DNF* formulas is the same as the minimization problem for *CNF* formulas (by using $1-x$ for literal x and x for literal $\bar{x}$), *CNF* formulas can be solved in linear time if their corresponding co-occurrence graph is a partial k-tree.

Another example is a linear time algorithm for determining whether a 2-SAT formula has exactly one solution, that is, uniquely solvable. The question of determining unique solvability is a tough one in general and it is even hard to determine whether linear time algorithms exist for special subclasses of SAT[8]. However, one is presented for 2-SAT in [**236**] using the framework of pseudo-boolean functions (that is, of the form (10.1) and (10.2)). Finally, we mention the result of [**106**] where a polynomial time algorithm for producing a parametric representation of all solutions to a 2-SAT formula is presented.

10.10. Nested and Extended Nested Satisfiability. The complexity of nested satisfiability has been studied in [**314**]. That study was inspired by Lichtenstein's theorem of planar satisfiability [**347**]. Index all variables in a *CNF* formula. A clause C_1 straddles a clause C_2 if the index of a literal of C_2 is strictly between two indices of literals of C_1. Two clauses overlap if they straddle each other. A formula is nested if no two clauses overlap. The problem of determining satisfiability for nested formulas, the clauses ordered so that clause C_i does not straddle clause C_j when $i < j$, can be solved in linear time [**314**].

An extension to nested satisfiability has been proposed in [**238**]. We prefer to skip the details and just mention that this extension can be recognized and solved in linear time. For details, the reader is referred to [**238**].

11. Advanced Techniques

In this section, we describe a number of advanced optimization techniques for satisfiability testing. They have been used in practical engineering applications and have proven to be effective for certain classes of SAT.

11.1. General Boolean Representations. In practice, many problems in integrated circuit design, such as logic verification, test pattern generation, asynchronous circuit design, logic optimization, sequential machine reduction, and symbolic simulation, can be expressed as Boolean satisfiability problems with arbitrary Boolean functions. Each representation has corresponding algorithms for satisfiability testing. A Boolean representation affects the performance of Boolean manipulation methods accordingly. Thus, efficient representation and manipulation of Boolean functions is crucial to many practical applications. Many different representations have been proposed for manipulating Boolean functions. However, many Boolean functions derived from practical circuit design problems suffer from an exponential size in their representations, making satisfiability testing infeasible.

Most SAT algorithms work on *conjunctive normal form (CNF)* formulas, *i.e.*, input formulas must be expressed as a product of sums of literals. The *CNF* formula is a canonical formula used in most analytical studies but is not an efficient representation in practical application problems. Many real engineering design problems use non-clausal representations rather than the *CNF* formula. Algorithms in this category may be regarded as *non-clausal* inference algorithms for satisfiability testing [**215**]. Compared to *CNF* formulas, a non-clausal, general Boolean representation is much more compact and efficient, although the transformation of an arbitrary non-clausal expression into *CNF* can be done in polynomial time by introducing

[8]An almost linear algorithm for unique Horn-SAT has been obtained by Berman et al. [**29**] and improved into a linear time algorithm by a slight modification due to Pretolani [**421**] (Minoux developed a quadratic time algorithm in [**378**])

new variables. This will result in clause-form representation of substantially larger sizes [**192, 415**]. While this is not critical in complexity theory, it will have serious impact in solving practical application problems.

In practice, a SAT algorithm can be made much more efficient if it works directly on problems represented in a compact number of general Boolean formulas rather than a large collection of *CNF* clauses. For a non-clausal SAT algorithm, the evaluation of arbitrarily large, complex Boolean functions is a key to its efficiency [**228**].

The next two subsections describe a sequential and a parallel Boolean representation and manipulation methods.

11.2. Binary Decision Diagram (BDD). Ordered Binary Decision Diagrams (OBDDs) [**59, 60**] is an efficient representation and manipulation method for arbitrary Boolean functions. This representation is defined by imposing restrictions on the Binary-Decision-Diagram (BDD) representation introduced by Lee [**334**] and Akers [**9**], such that the resulting form is canonical. The OBDD representation and its manipulation method are an extremely powerful technique in various practical applications. It is particularly useful with formulas where one needs to consider every solution, such as cases where one must search for optimal solutions. Although the OBDD representation of a function may have size exponential in the number of variables, many useful functions have more compact representations in practice.

A BDD gives a graphical representation of Boolean functions. It is a directed acyclic graph with two types of leaf nodes, **0** and **1**. Each non-leaf node is labeled with a Boolean variable v and has two out-going edges labeled 0 (the left edge) and 1 (the right edge). A BDD can be utilized to determine the output value of the function by examining the input values. Every path in a BDD is unique, *i.e.*, no path contains nodes with the same variables. This means that if we arbitrarily trace out a path from the function node to the leaf node 1, then we have automatically found a value assignment to function variables for which function will be 1 regardless of the values of the other variables.

Given a simple example Boolean function $F = (a + b) \cdot (a + c)$, the BDD of function F can be constructed to determine its binary value, given the binary values of variables a, b, and c. At the root node of BDD, we begin at the value of variable a. If $a = 1$, then $F = 1$ and we are finished. If $a = 0$, we look at b. If $b = 0$, then $F = 0$ and again we are finished. Otherwise, we look at c, its value will be the value of F. The complete BDD for function F is shown in Figure 28, where all the paths from the root function node F to the leaf node 1 are highlighted. Each highlighted path yields a satisfiable assignment. For F, the satisfiable assignments are $a = 1$, $b = -$, $c = -$ and $a = 0$, $b = 1$, $c = 1$, where '$-$' denotes a don't care assignment.

It is well known that the BDD size for a given function depends on the variable order chosen for the function (e.g., {a,b,c} in Figure 28). Since the early introduction of BDDs, several extensions have been proposed to reduce BDD sizes in practical applications. In an ordered BDD [**59, 60**], the input variables are ordered, and every path from the root node to the leaf node visits the input variables in an ascending order. In practice, a simple topological based ordering heuristic [**363**] yields small size BDDs for practical Boolean instances. A reduced ordered BDD is an ordered BDD where each node represents a unique logic function. Bryant showed that the reduced ordered BDD of a Boolean function is well-defined and is

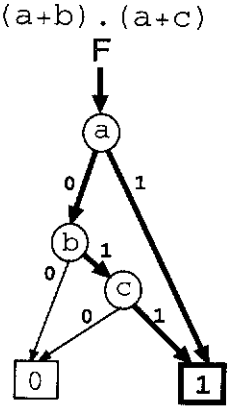


FIGURE 28. A simple BDD example for $F = (a + b) \cdot (a + c)$.

a canonical representation of the function; *i.e.*, two functions are equivalent if their reduced ordered BDDs are isomorphic [**59, 60**].

The DBDD is efficient to search for optimal solutions for arbitrarily complicated Boolean expressions. In VLSI circuit design, many practical problems require the enumeration of *all* possible assignments for a given Boolean formula. The best assignment that yields the minimum cost (e.g., minimal circuit structure, minimum chip area, and maximum circuit speed) is then selected from these possible assignments. Since most algorithms for satisfiability testing are designed for finding one truth assignment, they are impractical for selecting an optimal assignment. BDDs are very useful in such situations, since a simple and incremental enumeration of all possible paths from the root node to the leaf node **1** yields all the truth assignments. Thus, once the BDD for a Boolean function has been constructed, it is straightforward to enumerate all assignments or find an optimal solution.

The BDD method can effectively handle small and medium size formulas. For larger size formulas, a partitioning into a set of smaller sub-formulas before applying the BDD algorithms has been suggested. This approach works well for asynchronous computer circuit design problems [**216, 438**].

11.3. The *Unison* Algorithms. Based on total differential of a Boolean function, the *Unison* algorithm is capable of evaluating arbitrarily large, complex Boolean functions [**215, 493, 492**]. The *Unison* algorithm is built with a network of multiple *universal Boolean elements* (UBEs). The topology of the *Unison* network specifies the structure of Boolean functions. By dynamically reconfiguring the UBE's functionality, *Unison* is adaptable to evaluate general Boolean functions representing the SAT/CSP problems.

The *total differential*, dF, of a Boolean function F represents the difference in the function value due to the difference in input values. For a Boolean function $F(x, y)$ of two variables, x and y, the total differential is calculated from differences in input, dx and dy, as:

$$dF = F_x dx \oplus F_y dy \oplus F_{xy} dx\, dy, \tag{11.1}$$

where $\oplus$ is the *Exclusive-OR* operation [**506**]. Let $F(x, y)$ be a Boolean function of two dependent variables x and y; *i.e.*, $x = G(x_1, x_2)$ and $y = H(y_1, y_2)$. Following

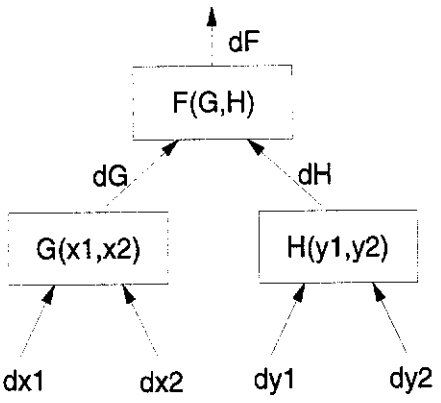


FIGURE 29. The relation between total differentials.

(11.1), the total differential dF is:

(11.2)
$$dF(G(\vec{x}), H(\vec{y})) = F_x\, dG(x_1, x_2) \oplus F_y\, dH(y_1, y_2) \oplus F_{xy}\, dG(x_1, x_2)\, dH(y_1, y_2).$$

It can be observed from (11.2) that the value of dF depends on total differentials dG and dH, rather than the function values $G(x_1, x_2)$ and $H(y_1, y_2)$. By recursively applying (11.2) to dG, dH, and their dependent variables, the total differential dF can be evaluated based on only total differentials of the independent variables (see Figure 29).

The *Unison* algorithm works in two phases: initialization and evaluation. The initialization phase computes partial derivatives that determine the function to be evaluated in the evaluation phase. The partial derivatives are constant during the evaluation phase. The evaluation phase reads input values and computes the final results. The calculation is performed in a bottom–up fashion, starting from the independent variables.

A computer word is used to evaluate one Boolean operation, so the code to calculate dF would produce only one result. With one computer word, however, the computer is able to perform many bitwise *AND* and bitwise *Exclusive-OR* operations in one instruction. In *Unison* algorithm's implementation, we take advantage of this machine feature to increase execution speed and to reduce memory space. In one of our implementations on the *NeXT* and *SUN* workstations [**492**], the *Unison* algorithm uses 32 bits of a computer word to pack 32 Boolean operations. If the ith bit in each operand is initialized to represent the ith Boolean operation, then the ith bit of dF will have the result of the ith Boolean operation. Each of the 32 bitwise operations is independent of the others. And the *Unison* algorithm simultaneously evaluates 32 Boolean operations in one machine instruction. The parallel implementation of the *Unison* algorithm is straightforward which can be implemented in any programming language that supports *bitwise* Boolean operations. Data structures and implementation details of the *Unison* algorithm are discussed in [**492**].

The *Unison* architecture is built with a network of multiple universal Boolean elements (UBEs). The connection topology of the *Unison* network specifies the structure of the Boolean function evaluated by *Unison*. The structure of Boolean functions specifies the connectivity between Boolean expressions of two variables. Each UBE accomplishes a 2-variable, simple Boolean function in *Unison*. The outputs of two UBEs can be used as inputs to another UBE. This enables the construction of a network of UBEs capable of evaluating arbitrarily large, complex Boolean functions. By dynamically reconfiguring the UBE's functionality, *Unison* is adaptable to the evaluation of different Boolean functions representing SAT/CSP problems. In *Unison* architectures, there is essentially no limit on the number of bits one would like to implement. One can put as many UBE's on a chip as possible as long as the hardware resource permits. The detailed implementations of the *Unison* architecture, e.g., its network structure, UBE structures, and two CMOS hardware implementations, are described in detail in [**493, 492**].

Combined with parallel evaluation, partial evaluation, and incremental evaluation techniques, *Unison* can be incorporated into a variety of search and optimization algorithms for satisfiability testing. It is especially important in real-time applications where hardware processing with different Boolean functions is required. It provides an efficient approach for fast non-clausal processing of SAT inputs.

11.4. Multispace Search. Many search and optimization methods have been developed in combinatorial optimization, operations research, artificial intelligence, neural networks, genetic algorithms, and evolution programming. An optimization algorithm seeks a value assignment to variables such that all the constraints are satisfied and the performance objective is optimized. The algorithm operates by changing values to the variables in the value space. Because value changing does not affect the formula structure and the search space, it is difficult for a value search algorithm to handle the pathological behavior of local minima.

Multispace search is a new optimization approach developed in recent years [**222, 228, 224**]. The idea of multispace search was derived from principles of non-equilibrium thermodynamic evolution that structural changes are more fundamental than quantitative changes, and that evolution depends on the growth of new structure in biological system rather than just information transmission. A search process resembles the evolution process, and structural operations are important to improve the performance of traditional value search methods [**222, 228, 224**].

In multispace search, any active component related to the given input structure can be manipulated, and thus, be formulated as an independent search space. For a given optimization problem, for its variables, values, constraints, objective functions, and key parameters (that affect the input structure), we define the variable space, the value space (*i.e.*, the traditional search space), the constraint space, the objective function space, the parameter space, and other search spaces, respectively. The totality of all the search spaces constitutes a *multispace*.

The basic idea of multispace search is simple. Instead of being restricted in the value space, the multispace is taken as the search space. In the multispace, components other than value can be manipulated and optimized as well. During the search, a multispace search algorithm not only alters values in the value space; as shown in Figure 30, it also walks across the variable space and other active spaces, changes dynamically the input structure in terms of variables, parameters, and

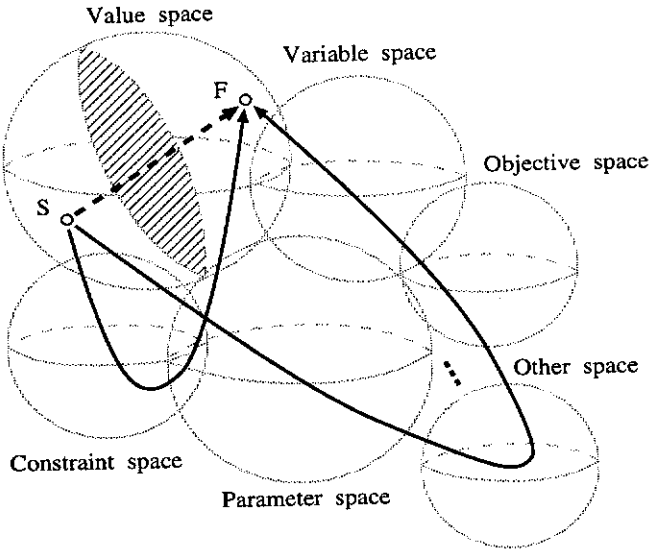


FIGURE 30. In the value space, a traditional search process (dashed line) cannot pass a "wall" of high cost search states (hatched region). It fails to reach the final solution state, F. A multispace search process (solid lines) scrambles across different search spaces. It could bypass this "wall" through the other search spaces.

other components, and *constructs* systematically a sequence of structured, intermediate instances. Each intermediate instance is solved by an optimization algorithm, and the solution found is used as the initial solution to the next intermediate instance. By interplaying value optimization with structured operations, multispace search incrementally constructs the final solution to the search instance through a sequence of structured intermediate instances. Only at the *last* moment of the search, the reconstructed instance structure approaches the original instance structure, and thus the final value assignment represents the solution of the given search input.

Multispace search algorithm combines traditional optimization algorithms with structural multispace operations. In each search step, multispace search performs two fundamental operations: a traditional value search and the structural reconfiguration of the intermediate instance during each individual search phase. According to the active event in the scrambling schedule [**222, 223**], the search process enters a specified search space and performs structural operations to the intermediate instance structures, followed by a traditional value search that optimizes the constructed intermediate instance. The resulting intermediate solution is then used as the initial instance to the next phase of multispace search.

The major structural operations in multispace search [**222, 223**] include multispace scrambling [**223, 228**], extradimension transition (e.g., air bridge, real dimension, and extra dimension) [**212, 214, 213**], search space smoothing [**225**], multiphase search [**488, 491, 212, 436, 545, 208, 219**], local to global passage

[208, 208], tabu search [199], and perturbations (e.g., jumping, tunneling, climbing, and annealing) [210, 211, 212, 213, 308].

In the next two subsections we describe two preprocessing methods for satisfiability testing in multispace search: partitioning input size and partitioning variable domain.

11.5. Partitioning to Reduce Input Size. Due to excessive computing time, a large size NP-hard problem is difficult to solve. Partitioning a large input into a set of smaller sub-instances may permit efficient solution of the input. There are two partitioning methods, each consisting of a partitioning, a conquer, and an integration procedure. For constructive partitioning (e.g., divide and conquer), partitioning, conquer, and integration procedures are well defined and easy to implement. For destructive partitioning, it is difficult to design the partitioning and integration procedures.

We give an industrial case study that requires a SAT solver. The SAT solver uses an efficient input size partitioning as a preprocessing step. This problem arises in asynchronous circuit design. Asynchronous circuits are indispensable in many low power and high performance digital computer systems. Due to their important applications in mobile, portable, and military communication systems, there has been great interest in the automated design and synthesis of asynchronous circuits [89, 330, 350, 528]. The design of asynchronous control and interface circuits, however, has proven to be an extremely complex and error-prone task. The core problem in asynchronous circuit synthesis can be formulated as an instance of SAT to satisfy the complete state coding (CSC) constraints, *i.e.*, the SAT-Circuit problem [529].

In this practical application problem, an optimal solution with minimal circuit layout area is sought. An incomplete SAT solver such as local search, unfortunately, does not guarantee an optimal solution, and therefore, is not applicable. Previous researchers used efficient resolution and branch-and-bound procedures to handle the SAT-Circuit problem. For most asynchronous circuit design problems, unfortunately, they were not able to find an optimal solution and, for difficult asynchronous circuit design problems, they could not locate even one solution.

Gu and Puri have recently developed a partitioning technique for satisfiability testing and applied it to asynchronous circuit design [223, 216, 438]. The partitioning preprocessor, at the beginning, decomposes a large size SAT formula that represent the given asynchronous circuit design into a number of smaller, disjoint SAT formulas. Each small size SAT formula can be solved efficiently. Eventually, the results of these sub-formulas are integrated together and contribute to the solution of the original formula. This preprocessor avoids the problem of solving very large SAT formulas and guarantees to finding one best solution in practice. This partitioning preprocessing is destructive since, during the search, extra variables are introduced to resolve the critical CSC constraints. Furthermore, they built a complete, incremental SAT solver based on binary decision diagrams (BDD). Their system is able to find an optimal solution to the asynchronous circuit design problem efficiently.

11.6. Partitioning Variable Domains. A variable domain contains values to be assigned to variables. The size of a variable domain, along with the number of variables, determine the computational complexity of an optimization algorithm. From a theoretical point of view, even a small reduction in the variable domain

would result in significant improvements in computing efficiency. It is, however, difficult to make use of variable-domain reduction techniques in solving optimization problems. Recently, Wang and Rushforth have studied mobile cellular network structures and developed a novel variable-domain reduction technique for channel assignment in these networks [**545, 546**].

The rapid growth of mobile cellular communication services has created a direct conflict to the limited frequency spectrum. Channel assignment is an important technique to the efficient utilization of frequency resource for mobile cellular communications. Among several channel assignment problems, the fixed channel assignment (FCA) is essential to the design and operation of cellular radio networks. An FCA algorithm assigns frequency channels to calls such that the frequency separation constraints are satisfied and the total bandwidth required by the system is minimized. By encoding the constraints into clauses, the problem becomes an instance of SAT. For a given cellular communication system, there are numerous ways to assign a channel to a call request. An optimal channel assignment decision can significantly improve the cellular system capacity without requiring extra cost. For a fixed mobile cellular system, the capacity of the cellular system is mainly determined by the performance of the channel assignment algorithms.

Wang and Rushforth's channel assignment algorithm was developed based on the structure of cellular frequency reuse patterns. Using their variable domain partitioning technique, they partition a mobile cellular network with larger variable domain into two networks: a minimum network with a fixed and small variable domain (due to the known frequency reuse patterns) and a difference network with an even smaller variable domain [**545, 546**]. Channels are assigned separately to the minimum network and to the difference network, and the superposition of these two assignments constitutes an assignment to the original network.

Because this variable domain partitioning approach decomposes an instance of a channel assignment problem with a large number of assignments into two separate channel assignment sub-instances with considerably smaller numbers of assignments, it dramatically reduces the computational complexity and thus the computing efficiency for solving given inputs, in addition to the significantly improved solution results. This novel partitioning technique can be applied to solve the channel assignment problem with any existing channel assignment algorithms. During numerous channel assignment experiments, this algorithm outperformed all available algorithms for solving the practical channel assignment problem benchmarks. Experimental evidence suggests that this partitioning approach is both efficient and effective.

11.7. Parallel SAT Algorithms and Architectures. Many parallel SAT and CSP algorithms have been developed. In a recent survey [**215**], the following parallel algorithms for solving SAT were discussed:

1. 1987: Parallel DP algorithms
2. 1986: Parallel discrete relaxation chips
3. 1987: Parallel backtracking architecture
4. 1987: Parallel local search algorithm
5. 1989: Parallel interior point method
6. 1990: Parallel, differential, non-clausal inference
7. 1991: Parallel $\alpha\beta$ relaxation
8. 1991: Parallel global optimization

9. 1992: Neural network approach
10. 1993: Multiprocessor local search

Some of ideas of these techniques are described in this paper.

For the following two reasons, algorithms running on loosely-coupled, multiprocessor parallel computers offer limited performance improvements for solving SAT. First, in the worst case, a SAT algorithm may suffer from the exponential growth in computing time. In order to solve a SAT formulas effectively, we will need a computer that has much larger speedup than what is available today. This computer will require the integration of at least a few million processors in a tightly-coupled manner. This is infeasible in the current computer system integration technology.

Second, as the processor gets much faster, the communication overhead among processors in a parallel machine becomes a bottleneck, which may often take 70 % to even 90 % of the total computing time [**293**]. Ideally one would expect the speedup on a parallel computer to increase linearly with increasing number of processors. Due to serious off-processor communication delays, after certain saturation point, adding processors does not increase speedup on a loosely-coupled parallel machine. Processor communication delay also makes process creation, process synchronization, and remote memory access very expensive in a loosely-coupled multiprocessor system. For this reason, the speedup on a multiprocessor is normally less than the number of processors used. With simple SAT algorithms, however, speedup is sometimes greater than the number of processors [**386**]. Variable settings similar to those that are already known not to lead to a solution are also unlikely to lead to a solution. The obvious methods of parallelizing simple SAT algorithms break up the tendency to search similar settings at about the same time.

From our experience, tightly coupled parallel computing, which effectively reduces off-processor communication delays, is a key to the parallel processing of SAT formulas [**215**]. In order to use a tightly-coupled parallel architecture for SAT computation, one must map a computing structure to the input structure and must reduce the total number of sequential computing steps through a large number of symmetrical interactions among simple processors [**206, 215**]. Several different approaches, e.g., special-purpose parallel VLSI architectures [**226, 227**], bit-parallel programming on sequential machines [**207, 212, 493, 492**], and tight programming on parallel computer systems, are promising alternatives in this direction. These approaches are capable of providing a tight mapping between a formula structure and a computing structure, resulting in faster computation. The computational power of these approaches are orders of magnitude greater than standard sequential algorithms running on uniprocessor machines or parallel algorithms running on loosely coupled multiprocessors.

Parallel processing does not change the worst-case complexity of a SAT algorithm unless one has an exponential number of processors. Parallel processing, however, does delay the effect of exponential growth of computing time, allowing one to solve larger size instance of SAT.

11.8. The *Multi-SAT* Algorithm. The problem structures of real world practical applications vary significantly, making it difficult to develop an efficient SAT algorithm to solve a wider range of the practical application problems. Many efficient algorithms have been developed for the SAT problem. They each can solve a class of problem instances efficiently. Backtracking algorithms can handle some small size, hard problem instances, providing complete solutions. Local search could

handle fairly large-size satisfiable problem instances quickly. BDD SAT solver is able to solve practical problem instances with performance criteria. Lagrangian-base global search method can provide solutions to a wide range of SAT problem instances. Problem size and domain partitioning techniques can further enhance the existing SAT algorithms, so they can solve much larger size practical problem instances. If we combine the *niches* of several efficient algorithms together, they may handle a much wider range of SAT problem instances efficiently.

Another school of concern for the *Multi-SAT* algorithm comes from the existing challenge for SAT algorithm's design and testing. A good local search algorithm consists of several basic components. These components are sensitive to algorithm parameter setting, algorithm running environment, input size, problem structure, and initial starting points. We will select from various min-conflicts heuristics, random value assignment heuristics, random variable selection heuristics, partial random variable selection heuristics, multiphase search heuristics, and multispace search heuristics. Combined with hundreds of problem instances and initial starting points, the process for the design, implementation, and experimentation of the algorithm is very time-consuming. A *Multi-SAT* algorithm can relieve the load of this task, facilitating quick design and testing of the algorithm.

Two algorithm integration approaches (hybrid algorithm and algorithm clustering) have been proposed for algorithm integration. In the *hybrid algorithm* approach, algorithms in different classes are integrated in a single algorithm. The algorithm would make use of different algorithmic niches according to some decision procedures. Early examples of this approach include combining local search with backtracking [**212**] and combining global optimization with backtracking [**206, 227, 213**]. The effectiveness of a hybrid algorithm may be limited due to the overheads of decision making and algorithmic context switching.

In the *algorithm clustering* approach ("Future Work" in [**213**]), algorithms in different classes are optimized individually to achieve the best performance. Each algorithm is executed on a computer. A cluster of computers is used to execute several algorithms selected from different classes. The individual results of the algorithms' executions are integrated together, producing the final result. The algorithm clustering approach does not suffer from any performance degradation due to direct algorithm integration. Computer hardware prices continue to decrease, a cluster of computers can be built in a cost-effective way (a PC can now be purchased with around $1,000). The only requirement for clustering computation is a multi-tasking integration software.

In our first implementation of the *Multi-SAT* algorithm, we select several efficient SAT algorithms. These include, for examples, *DPL* and *CSAT* from backtracking algorithm, *SAT1*, *GSAT*, and *SAT3* from local search algorithm, *BDD* SAT solver from binary decision diagram algorithm, and *DLM* from Lagrangian-base global search method. Combining problem size and domain partitioning techniques, they together support an effective satisfiability testing for problem instances with uncertain structures, using "many stones" to shoot "one bird." By changing parameters and initial starting points, *Multi-SAT* can track an algorithm structure, allow a detailed study of the entire problem spectrum, and provide a cost-effective tool kit for practical satisfiability testing.

A basic software system for the *Multi-SAT*, *Clustor*, is shown in Figure 31. We have an *algorithm tool kit* collecting candidate algorithms from different algorithm classes, a *problem instance database* for user to select the problem instances,

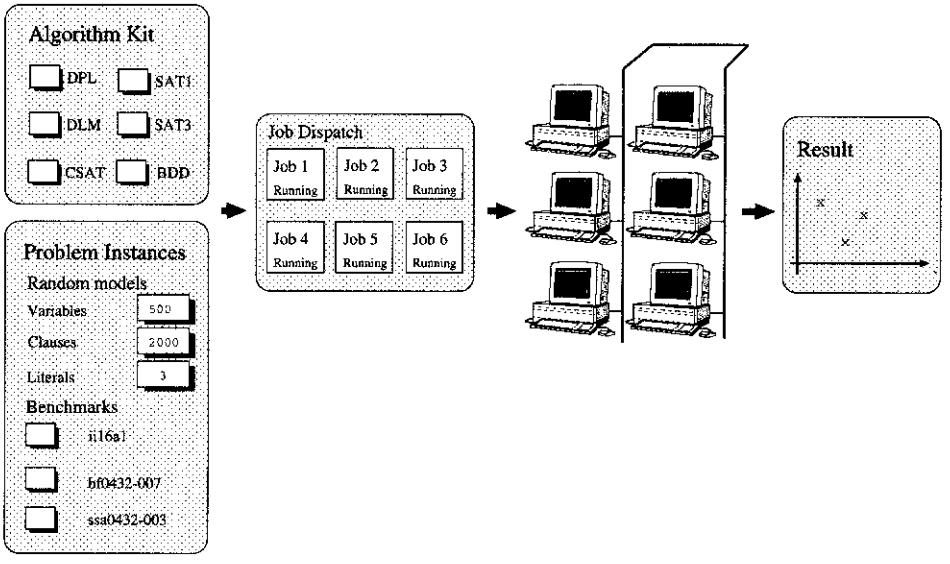


FIGURE 31. A software system, *Clustor*, for *Multi-SAT* algorithm.

a distributed system software, *job dispatcher*, for remote job execution control and execution result collection, and a network of computers executing the selected algorithms. The software system can be run on a PC platform or a UNIX platform under an interactive graphical interface.

12. Probabilistic and Average-Case Analysis

Probabilistic and average-case analysis can give useful insight into the question of what SAT algorithms might be effective under certain circumstances. Sometimes, one or more structural properties shared by each of a collection of formulas may be exploited to solve such formulas efficiently; or structural properties might force a class of algorithms to require super-polynomial time. Such properties may be identified and then, using probabilistic analysis, one may hope to argue that these properties are so common for a particular class of formulas that the performance of an algorithm or class of algorithms can be predicted for most of the formulas in the class.

The main drawbacks of this approach are: 1) some distribution of input formulas must be assumed and chosen distributions may not represent reality very well; 2) results are usually sensitive to the choice of distribution, unlike results obtained using randomized algorithms; 3) the state of analytical tools is such that distributions yielding to analysis are typically symmetric with independent components; 4) few algorithms have yielded to analysis. Despite these drawbacks, probabilistic results can be a useful supplement to worst-case results (which can be overly pessimistic, especially for NP-complete problems) in understanding algorithmic behavior.

This section reviews some notable probabilistic and average-case results for certain SAT algorithms. The results we present are based mainly on two distributions, the l-SAT distribution and average l-SAT distribution (see Section 5), partly because these have been the most widely used. Both are distributions over *CNF*

formulas. Since the character of results is different for both distributions, we devote one subsection to each.

12.1. Average l-SAT Model. The parameters of this distribution are the number of clauses m, the number of variables n from which clauses are constructed, and the probability $p(n,m)$ that an unnegated variable or negated variable appears in a given clause (see Section 5.1 for more details). Since variables are placed in clauses independently, it is possible that null clauses, clauses with complementary literals (tautological clauses), or unit clauses exist in a random formula. This does not mimick reality very well. However, the mathematics associated with average-time analyses for average l-SAT models is usually tractable. It would be straightforward but tedious to modify average l-SAT calculations to account for no clauses of length 0 or 1, but such results are unknown to us. In addition, tautological clauses exist with high probability only over part of the parameter space.

Results presented below are asymptotic (that is, they apply when $n, m \to \infty$).

Satisfiable and Unsatisfiable Formulas. The following results highlight those regions of the parameter space where random formulas are unsatisfiable or satisfiable with high probability. It is easy to see that the average number of literals in a clause is $2pn$ and the average number of times a variable appears in a random formula is $2pm$. If $pn > \ln(m)$, a random truth assignment satisfies a random formula in probability [**165**], and if $pn = c\ln(m) : 1 > c \geq 1/2$, and $\lim_{n,m\to\infty} m^{1-c}/n^{1-\epsilon} < \infty$, $1 > \epsilon > 0$, a random formula is satisfiable in probability [**167**]. If $pn < \ln(m)/2$, a random formula contains an empty clause, and therefore is unsatisfiable, in probability. Thus, the only region of the parameter space where random formulas may be difficult, in a probabilistic sense, is defined by $pn = c\ln(m) : 1 > c \geq 1/2$, $\lim_{n,m\to\infty} m^{1-c}/n^{1-\epsilon} = \infty$, $1 > \epsilon > 0$.

Polynomial-Time Solvable Classes. Many of the special polynomial-time solvable classes discussed in Section 10 can be identified with regions in the parameter space as well. Here we give some examples taken mainly from [**171**].

If $pn < 1/\sqrt{m^{1+\epsilon}} : \epsilon > 0$, a random formula is a Horn formula in probability. That is, all the non-empty clauses are Horn clauses. If $pn < \sqrt{n^{1-\epsilon}/m} : 1 > \epsilon > 0$, $\lim_{n,m\to\infty} m/n \leq 1$, a random formula is extended Horn in probability. If $pn > \sqrt{n^{1+\epsilon}/m} : \epsilon > 0$, a random formula is not extended Horn in probability. This implies, when $2pn \to \infty$ (no empty clauses, in probability), the parameter subspace where random formulas are usually extended Horn is sharply defined. Surprisingly, simple extended Horn formulas are abundant in a relatively small subspace of the parameter space. If $pn < 1/\sqrt{m^{1+\epsilon}} : \epsilon > 0$, a random formula is a simple extended Horn formula in probability but if $pn > 1/\sqrt{m^{1-\epsilon}} : \epsilon > 0$, a random formula is not simple extended Horn, in probability.

Random formulas are balanced, in probability, only if $pn < \sqrt{n^{1-\epsilon}/m} : \epsilon > 0$. Thus, when $\lim_{n,m\to\infty} m/n \leq 1$, balanced formulas are generated in abundance over a region of the parameter space that is no larger than the subspace over which random formulas are extended Horn in probability. The same statement is believed to hold when $\lim_{n,m\to\infty} m/n > 1$.

Weakening SLUR so that it always chooses to expand the *true* path of a selected variable, if possible, the parameter subspace where random formulas can be solved

by SLUR in probability is at least as large as given by the three regions 1) $p < 1$ and $pn > 3\ln(m)$: $\lim_{n,m\to\infty} m/n > 1$; 2) $p < 1$: $\lim_{n,m\to\infty} m/n < 1$; 3) $pn < \ln(m)/2$. This is because no clauses containing either all negated (pure negative clause) or all unnegated variables (pure positive clause) are in a random formula, in probability, in region 1) (see *Exploitable Properties* below); random formulas are extended Horn, or have no pure positive or negative clauses, in probability, in region 2); and random formulas contain empty clauses, in probability, in region 3).

In summary, the SLUR class, modified as above, dominates nearly the entire parameter space; balanced and extended Horn formulas are frequently generated only when either the average number of occurrences of a variable in a formula tends to 0 or random formulas tend to have a large number of empty clauses; Horn formulas and simple extended Horn formulas are commonly generated over a small portion of the parameter space.

Exploitable Properties. If a random formula is in one of the special, polynomial-time solvable subclasses of SAT discussed earlier, it can be dealt with efficiently. The same is true if a random formula has one or more of certain other exploitable properties. Three of these are described here (taken from [**171**]).

A clause is pure if it contains only negated variables or only unnegated variables. Call a formula that has no pure clauses a non-P-formula. A satisfying truth assignment for any non-P-formula can be obtained in linear time. If $pn > (1+\epsilon)\ln(m)$: $\epsilon > 0$, a random formula is a non-P-formula, in probability.

A clause is a tautology if, for some variable v, both v and $\bar{v}$ are in the clause. Such clauses may be removed from a formula without affecting the Boolean function it expresses. If enough tautological clauses exist in a formula, it is relatively easy to solve. If $p^2n > (1+\epsilon)\ln(m) : \epsilon > 0$, all clauses of a random formula are tautological, in probability.

If all m clauses of a formula contain more than $\log_2(m)$ literals then the formula must be satisfiable. A random formula has this property, in probability, when $pn > 1.55\log_2(m)$.

Average-Case Results. Although the above results show that random formulas are efficiently solved, in probability, over nearly all of the parameter space of the average l-SAT model, they do not imply that polynomial-average-time algorithms exist over a significant portion of the parameter space. For example, if, out of a set of n^{100} formulas, $n^{100} - 1$ formulas can be solved by algorithm A in $O(n)$ time but one formula requires 2^n time using A, then the set is solved by A in polynomial-time, in probability, but the average complexity of A over the set is exponential in n (assuming all formulas are equally likely). Thus, A would get "stuck" on the above set of formulas even though it almost always finds a solution to a random formula in linear time. This consideration has motivated the average-case analysis of algorithms under the average l-SAT model. The results say that exploiting some of the above properties individually is not enough to insure polynomial-time average complexity but, by exploiting certain properties collectively, nearly the entire parameter space is covered by a collection of algorithms with polynomial-average-time complexity. Here we give two examples.

Determining unsatisfiability from the existence of an empty clause in a given formula alone is not strong enough to give polynomial-average-time if $pn < \ln(m)/2$

since the probability that an empty clause exists in a random formula does not tend to 1 fast enough. However, the empty clause check can be combined with other methods to achieve polynomial-average-time complexity. For example, preprocess a given formula by making all unit resolutions and all resolutions involving variables that occur in the formula exactly twice; use backtracking, with the empty clause check, to find solutions to the processed formula. Polynomial-average-time is achieved when either

1. $pn < (\epsilon - 1 - \delta)\ln(m)/(2\epsilon)$: $m = n^{\epsilon}$, $\epsilon > 1$, and, δ is such that $\epsilon - 1 - \delta > 0$; or
2. $2.64(1 - e^{-2\beta pn}(1 + 2\beta pn)) < \beta e^{-2pn}$: $m = \beta n$, β a constant; or
3. $pn < (1 - \epsilon - \delta)\ln(m)/\epsilon$: $m = n^{\epsilon}$, $1 > \epsilon > 2/3$, and, δ is such that $1 - \epsilon - \delta > 0$; or
4. $pn \leq (\ln(m)/4)^{1/3} n^{2/3-\epsilon}$: $m = n^{\epsilon}$, $2/3 \geq \epsilon > 0$ [**166**].

This subspace includes nearly all of the half plane $pn < \ln(m)/2$; that is, the region for which empty clauses exist, in probability.

We remark that the above algorithm finds *all* solutions to a given formula.

A variant of the DPL algorithm, called probe-order backtracking, that works well for the half plane $pn > \ln(m)$ exploits the preponderance of non-P-formulas that results from generating formulas in that region [**433**]. Given formula $\mathcal{F}$, if an empty clause exists in $\mathcal{F}$, output "unsatisfiable." If there is no clause in $\mathcal{F}$ containing only unnegated variables, output "satisfiable." Otherwise, select a clause in $\mathcal{F}$ containing only unnegated variables $\{v_1, v_2, ..., v_k\}$. For $i = 1, ..., k$, set v_i to *true*, set $v_1, v_2, ..., v_{i-1}$ to *false*, and recursively apply probe-order backtracking. Output "satisfiable" if and only if at least one of these invocations has output "satisfiable." Probe order backtracking runs in polynomial-average-time when $pn > \ln(m)$.

Other interesting results are found in [**168, 417, 169, 203, 281, 431, 430, 425**].

Average Number of Solutions. In the average l-SAT model, the average number of solutions per formula is approximately

$$\exp[n \ln 2 + m \ln(1 - e^{-pn})].$$

Thus, when m/n and pn are such that the exponent is negative, formulas have very few solutions, but when the exponent is positive formulas have many solutions, on average. When m/n is below $\ln(2)$, the average number of sub-formulas generated by simple backtracking is about the same as the average number of solutions. When m/n is above $\ln(2)$, small values of pn still lead to few sub-formulas and large values lead to a huge number of sub-formulas, but there is an intermediate range of values where the average number of solutions is near zero while the average number of nodes is an exponential function of n [**431**].

The average-time analysis for backtracking is done for a version of the algorithm that finds all solutions. When one wants just one solution, there is no need for the algorithm to solve the second sub-formula in those cases where the first sub-formula has a solution. So far no analysis has shown just how much time can be saved

by stopping early. Since stopping early can have an effect only on formulas that have solutions, the analysis in [**431**] shows that there is a considerable range of pn values where simple backtracking takes exponential average time whether or not the algorithm stops at the first solution.

Additional Commentary. An average l-SAT analysis of unit clause backtracking [**425**] shows that the conditions under which it is fast or slow are similar to the conditions under which simple backtracking is fast or slow. Again, the analyzed version of the algorithm finds all solutions. However, for moderate values of m/n there is a range of pn values where simple backtracking takes exponential time but unit clause backtracking takes polynomial time. For small values of m/n unit clause backtracking has no significant advantage because the number of solutions controls the running time, and for large values of m/n it has little advantage because interesting formulas occur with large pn values, and so unit clauses are rare.

The average l-SAT analysis of probe order backtracking shows that, in addition to being fast under conditions where simple backtracking is fast, it is fast under various other conditions. It is fast when pn is below 1. When m/n is small, the typical l-SAT formula does not use most of the variables. Thus, most formulas with one solution have an exponential number of solutions (one for each setting of the unused variables). Simple backtracking takes no advantage of variables that do not appear in the formula, but clause order backtracking does. Clause order backtracking is also fast when pn^2 is large compared to $\ln m + \ln n$. When p is this large, setting just a few variables (to a random setting) tends to satisfy all of the clauses. Clause order backtracking notices this while simple backtracking does not.

No average l-SAT analysis has been done for shortest clause backtracking. (See [**376**] for a partial analysis of the l-SAT case.) It clearly has all the advantages of unit clause backtracking and it should be much faster when pn is large, but it is hard to know just how much faster. The first four prize winning entries in the 1992 SAT competition all used shortest clause backtracking [**70**] with refinements to decide which of the various variables from shortest clause to select. (The fifth prize winning entry used a form of hypergraph searching.)

The pure literal rule algorithm is one of the first to have its average time computed [**66, 201, 203, 430**]. It has the essence of the pure literal rule from the DP procedure [**118**]; by removing most of the good features, an analyzable algorithm is obtained. Although one would never use this algorithm in practice (other simple algorithms are much better) it rapidly solves a wide class of formulas in polynomial average time, but does not find all solutions. It played an important role in the early history of average-time analysis of SAT algorithms because its analysis is so simple, and the cases where it is fast are so different from those of simple backtracking (the other simple to analyze SAT algorithm).

The almost pure literal algorithm [**426**] extends this idea by noting that when there are few occurrences of a literal, then assigning a value that makes that literal *false* leads to a sub-formula that is almost a subset of the sub-formula obtained by setting the literal to *true*. Thus, if a formula contains one clause that has the only occurrence of a literal, any solutions to the *false* sub-formula that are not also solutions to the *true* sub-formula have *false* values for all remaining literals in the special clause.

Another case where resolution does not increase the input size is when a variable has one positive and one negative occurrence. Franco used this idea plus the pure literal rule to develop an algorithm that is fast for small m so long as p is not too large [**166**]. Using this algorithm for small p and probe order backtracking for large p leads to an algorithm that is fast for $m < n^{1/3}$ times logarithm factors. More clever algorithms based on the same ideas combined with better analyses will probably lead to an algorithm that is fast when m is smaller than a constant times n.

12.2. The l-SAT Model. The parameters of this distribution are the number of clauses m, the number of variables n from which clauses are constructed, and the number of variables l in each clause. Clauses are constructed independently. A clause is uniformly given by a set of l distinct variables that are negated independently with probability 1/2. Thus, it is not possible that null clauses or clauses with complementary literals exist in a random formula.

The probabilistic analysis of SAT algorithms using the l-SAT model often seems to be more difficult than using the average l-SAT model. Some of this difference is associated with the structure of sub-formulas generated as a result of assigning a value to a variable on an iteration of a particular algorithm. If such sub-formulas are distributed according to the same model as the original formula, the analysis can proceed easily. In the case of the l-SAT model, statistical dependence between clauses after an iteration often prevents this. A notable exception, however, is in the analysis of variants of the unit clause rule.

Another reason for the relative success of analysis under the average l-SAT model is many algorithms that are unworkable under the l-SAT model are effective under the average l-SAT model. A notable example is the probe order backtracking algorithm of the previous section. Under the average l-SAT model, if $pn > \ln(m)$, purely negative or positive clauses are rare so probe order backtracking works well. However, in the case of the l-SAT model, negative clauses and positive clauses make up a fixed percentage of input clauses, so probe order backtracking is ineffective in this case.

In what follows, when we refer to l-SAT, we assume $l \geq 3$ unless specifically stated (as in, for example, 2-SAT).

Satisfiable and Unsatisfiable Formulas. It is easy to show that random l-SAT formulas are unsatisfiable, in probability, if $m/n > -1/\log_2(1-2^{-l}) \approx 2^l$ [**168, 417**]. It has also been shown that a random 2-SAT formula is satisfiable, in probability, if $m/n < 1$ [**86, 200**]. This implies that random l-SAT formulas are satisfiable, in probability, if $m/n < 1$. The gap between 1 and $-1/\log_2(1-2^{-l})$ has intrigued a number of researchers. The question is whether there is some function $f(l)$ such that, for large n, m, if $m/n < f(l)$ then random l-SAT formulas are satisfiable, in probability, and if $m/n > f(l)$ then random l-SAT formulas are unsatisfiable, in probability. Several results have shaved some of the gap from above and below but the question is still open for $l > 2$. For the 2-SAT case, $f(2) = 1$ [**86, 200**]. For the 3-SAT case, from above, it is known that random l-SAT formulas are unsatisfiable, in probability, if $m/n > 4.758$ [**304**]. This has been recently improved to 4.64 [**150, 311**]. From below, for $l > 2$, it is known that random l-SAT formulas are satisfiable, in probability, if $m/n < max\{2^l/(4l), 1\}$ [**91**]. This result comes with an algorithm for SAT (explained in *Algorithms* below) that finds

a solution in polynomial time, almost always. For 3-SAT, this has been improved to $m/n < 3.003$ [**179**] (algorithm explained in *Algorithms* below).

Polynomial-Time Solvable Classes. l-SAT formulas that are members of certain polynomial-time solvable classes are not generated frequently enough, for interesting ratios m/n, to assist in determining satisfiability. This is unlike the situation for the average l-SAT model. We illustrate with a few examples.

The probability that a clause is Horn is $(l+1)/2^l$. Therefore, the probability that a random l-SAT formula is Horn is $((l+1)/2^l)^n$ which tends to 0 for any fixed l. A formula is hidden Horn if there is a set of variables (a *switch set*) whose literals can all be reversed to yield a Horn formula. Regardless of switch set, there are only $l+1$ out of 2^l ways (negation patterns) that a random clause can become Horn. Therefore, the expected number of successful switch sets is $2^n((l+1)/2^l)^m$ which tends to 0 if $m/n > 1/(l - \log_2(l+1))$. Thus, random l-SAT formulas are not hidden Horn, in probability, if $m/n > 1/(l - \log_2(l+1))$.

Associated with a q-Horn formula (see Section 10.5) is a partition C_1, C_2 of clauses, and a partition V_1, V_2 of variables such that no clause in C_1 has a variable in V_2 and, for each clause in C_2, there are at least one and at most two variables taken from V_2. The probability that a particular pairwise partition has this property can be computed. Multiplying by the number of pairwise partitions gives the expected number of such partitions which is an upper bound on the probability that one exists. We find that no such partitions exist with $|V_1| < (|V_1| + |V_2|)/2$, in probability, if $m/n > 1/\log_2(2^{l+1}/(l^2 - l + 2)) = 1/(l - \log_2(l^2 - l + 2) + 1)$. Coupled with the above hidden Horn result on l-SAT formulas, we have the remarkable result that random l-SAT formulas are not q-Horn, in probability, if $m/n > 2/(l - \log_2(l+1))$. This bound can be reduced considerably, however the point we make is that, for large enough l, even the following simple algorithm is more effective, in some probabilistic sense, on random l-SAT formulas than looking for q-Horn formulas: randomly remove all but 2 literals from every clause; solve the resulting 2-SAT formula; if it's satisfiable, return a satisfying truth assignment, otherwise give up.

Algorithms. We mention the two best positive results to date and one negative result. The first algorithm, called SC for Short Clause, iteratively selects a variable and assigns it a value until either a solution is found or it gives up because it has reached a dead end. Such an assignment may satisfy some clauses and falsify some literals. There is no backtracking in SC. Variables are selected as follows: if there is a clause with one non-falsified literal, choose the variable and value that satisfies that clause; otherwise, if there is a clause with two non-falsified literals, choose one of the variables and value that satisfies that clause; otherwise, choose the variable arbitrarily. This algorithm is a restricted version of GUC [**83**] (Generalized Unit Clause) that always chooses a variable and value that satisfies a clause with the fewest number of non-falsified literals. The analysis of SC is given in [**91**]. The result is that SC does not give up, in probability, if $m/n < 2^l/(4l)$.

By adding a limited amount of backtracking to GUC, Frieze and Suen get an algorithm for 3-SAT, called GUCB, that finds a satisfying assignment, in probability, when $m/n < 3.003$ [**179**]. Backtracking is managed as follows. Consider the sequence of variable selections and assignment up to a given iteration h in

the execution of GUCB. Let this sequence be represented as a list of variable-value pairs $\{(x_{\pi_1}, v_1), (x_{\pi_2}, v_2), ..., (x_{\pi_h}, v_h)\}$. Suppose, for $p \geq 1$, $v_p = false$, $v_{p+1} = ... = v_h = true$, and two clauses contain one non-falsified literal but no truth assignment will satisfy both. Then set $v_{p+1} = v_{p+2} = ... = v_h = false$, update all clauses accordingly (satisfied clauses and falsified literals) and continue iteratively selecting variables and assigning values.

Finally, we mention the important result in [**86**] that resolution proofs must be exponentially large, in probability, for random unsatisfiable l-SAT formulas generated with m/n fixed. Thus, for $m/n > \log_2(1 - 2^{-l})$ (fixed), resolution requires exponential time, in probability. This, of course, implies that DPL trees are also exponential in size for $m/n > \log_2(1 - 2^{-l})$.

Other Non-Backtracking Heuristics The algorithms SC and GUC mentioned above repeatedly choose a variable and a value until either a satisfying assignment is found or a clause becomes falsified in which case the algorithm gives up. The heuristic used to select the variable and value is strongly associated with how often the algorithm succeeds. A reasonable heuristic is to make the choice that maximizes the number of assignments satisfying the formula that remains after the selected value is assigned to the selected variable. Alternatively, the selected variable and value might maximize the expected number of satisfying assignments. This expectation can be approximated as follows. Suppose a formula has m_i clauses of i literals for all $1 \leq i$ and n distinct variables. If all clauses are statistically independent and all clauses of i literals are equally likely, the average number of satisfying assignments is $2^n(1/2)^{m_1}(3/4)^{m_2}(7/8)^{m_3}...$. Thus, we may choose a variable and value that maximize this number. Equivalently, the choice may be made to maximize the log of this number or $n + m_1 \log(1/2) + m_2 \log(3/4) - m_3 \log(7/8)...$ which is approximately $n + m_1(1/2) + m_2(1/4) + m_3(1/8)...$. Removing n, which is unimportant, leaves Johnson's heuristic described in [**286**]. Although this heuristic has not been analyzed on the l-SAT model, experiments have shown it to be quite effective when used in conjunction with unit resolution.

13. Performance Evaluation

The most important measure of a SAT algorithm's performance remains its practical problem-solving ability. For inputs requiring only one solution, both complete algorithms and incomplete algorithms are applicable. For inputs requiring all solutions or an optimal solution, only complete algorithms will work. The past two decades have seen the proliferation of different algorithms for solving SAT: resolution, local search, global optimization, BDD SAT solver, and multispace search, among others. Previous experience indicates that these techniques complement rather than exclude each other by being effective for particular instances of SAT.

In this section, we summarize the experimental performance of several typical SAT algorithms on some random instances, DIMACS benchmarks, structured instances, and practical industrial benchmarks. A fuller version of SAT algorithms' benchmarking results will appear in a forthcoming paper, "Algorithms for the Satisfiability (SAT) Problem: Benchmarking," by the same authors.

13.1. Experiments on Random Formulas. In this section, we give experimental results for the following SAT algorithms in solving random l-SAT formulas and random average l-SAT formulas:

1. *SAT1.3*: a sequential *CNF* local search algorithm [**207, 221, 209, 212**].
2. *SAT1.7*: a parallel *CNF* local search algorithm [**207, 212**].
3. *SAT1.13*: a complete *CNF* local search algorithm [**207, 212**].
4. *SAT1.4*: a sequential *DNF* local search algorithm [**207**].
5. *SAT1.8*: a parallel *DNF* local search algorithm [**207**].
6. *SAT1.18*: a complete *DNF* local search algorithm [**207**].
7. *SAT14.6*: an optimized, discrete global optimization algorithm [**207, 209, 213**].
8. *SAT14.16*: a complete global optimization algorithm [**207, 209, 213**].
9. *SAT14.7*: a continuous, global optimization [**207, 209, 218**].
10. *SAT14.11*: a complete, continuous global optimization [**207, 209, 211**].
11. *DPL*: a Davis-Putnam algorithm in Loveland form [**117**].
12. *GSAT*: a sequential, greedy local search algorithm [**472**].
13. *IP*: a parallel interior point zero-one integer programming algorithm [**303, 301**].

Real Execution Times. In Table 1 we give real execution times of some local search and global optimization algorithms for solving l-SAT instances. All the results were run on a SUN SPARC 2 workstation. The number of clauses (m), the number of variables (n), and the number of literals per clause (l), are given in the first three columns. Symbol "G/L" in Column 4 stands for the number of times that all the algorithms hit *global/local* minimum points. From these results we can observe that, in terms of global convergence and local convergent rate, these local search and global optimization algorithms exhibit desirable convergent properties and fast computing speed for instances in the table.

Among optimization algorithms, the parallel *CNF* local search ($SAT1.7$) algorithm was much faster than the sequential local search ($SAT1.3$) algorithm. The $SAT1.7$ algorithm had comparable computing performance with the *DNF* parallel local search ($SAT1.8$) algorithm. Discrete global optimization ($SAT14.6$) algorithm was slightly slower than parallel local search algorithms. Complete local search ($SAT1.13$) algorithm and complete global optimization ($SAT14.16$) algorithm, due to a systematic bookkeeping, were slightly slower than parallel local search but significantly faster than the sequential local search algorithm.

As discussed in [**221, 212**], beyond a certain range of hardness, for example, for $m = 8500, n = 1000$, and $l = 4$, the computing time of these optimization algorithms started to increase.

The experimental results shown in Table 1 were collected from early reports in [**207, 212, 213**]. The present local and global optimization algorithms are much more faster than their previous versions [**209, 212, 219**].

Performance Comparison with the DP Algorithm. The execution results of the DPL algorithm and some optimization algorithms for solving l-SAT instances are given in Table 2. We executed each algorithm ten times and report the average execution times. Because DPL was slow for large size instances, we set a maximum execution time of $120 \times m/n$ seconds as the time limit of its execution. Symbol "S/F" in Column 4 stands for DPL's *success/failure* in giving an answer within such a time limit. For DPL, the average execution time does not include the maximum execution time limit if some of the ten executions were successful; the

TABLE 1. Real execution performance averaged over ten runs of some **local and global optimization algorithms** on a SUN SPARC 2 Workstation. Time Units: seconds. Symbol "G/L" stands for the number of times that all the algorithms hit *global/local* minimum points.

Problems			Execution Time						
m	n	l	G/L	SAT1.3	SAT1.7	SAT1.13	SAT1.8	SAT14.6	SAT14.16
100	100	3	10/0	0.003	0.001	0.001	0.002	0.002	0.003
200	100	3	10/0	0.007	0.004	0.010	0.004	0.005	0.008
300	100	3	10/0	0.035	0.008	0.015	0.004	0.014	0.012
400	100	3	10/0	0.464	0.027	0.145	0.030	0.040	0.215
1000	1000	3	10/0	0.036	0.030	0.048	0.029	0.033	0.051
1500	1000	3	10/0	0.087	0.055	0.078	0.049	0.058	0.081
2000	1000	3	10/0	0.192	0.084	0.113	0.080	0.093	0.115
2500	1000	3	10/0	0.371	0.124	0.158	0.114	0.133	0.180
3000	1000	3	10/0	0.872	0.179	0.310	0.164	0.241	0.359
3500	1000	3	10/0	6.878	0.636	1.008	0.588	0.919	1.357
1000	1000	4	10/0	0.026	0.022	0.045	0.027	0.029	0.040
2000	1000	4	10/0	0.094	0.061	0.103	0.061	0.057	0.092
3000	1000	4	10/0	0.239	0.094	0.160	0.091	0.109	0.166
4000	1000	4	10/0	0.483	0.144	0.230	0.135	0.162	0.234
5000	1000	4	10/0	1.004	0.227	0.338	0.210	0.267	0.330
6000	1000	4	10/0	2.410	0.383	0.465	0.359	0.388	0.478
7000	1000	4	10/0	5.999	0.865	0.729	0.852	0.756	0.840
8000	1000	4	10/0	36.17	1.896	2.088	1.821	2.595	2.641
8500	1000	4	10/0	140.3	10.79	7.974	10.51	12.79	12.12
10000	1000	5	10/0	2.899	0.451	0.610	0.393	0.464	0.567
11000	1000	5	10/0	3.799	0.489	0.800	0.426	0.580	0.750
12000	1000	5	10/0	6.729	0.593	0.839	0.505	0.649	0.844
13000	1000	5	10/0	9.541	0.761	1.154	0.681	0.978	1.064
14000	1000	5	10/0	21.41	1.107	1.308	0.969	1.282	1.652
15000	1000	5	10/0	60.80	1.671	2.207	1.429	2.047	2.166
10000	400	6	10/0	12.58	0.497	0.625	0.463	0.514	0.771
10000	500	6	10/0	4.353	0.377	0.640	0.342	0.345	0.553
10000	600	6	10/0	2.571	0.328	0.439	0.280	0.331	0.534
10000	700	6	10/0	1.989	0.284	0.550	0.248	0.289	0.491
10000	800	6	10/0	1.776	0.277	0.494	0.256	0.287	0.452
10000	900	6	10/0	1.305	0.289	0.523	0.248	0.278	0.476
10000	1000	6	10/0	1.140	0.264	0.488	0.227	0.269	0.473
20000	1000	7	10/0	3.238	0.500	1.124	0.421	0.496	1.004
30000	2000	7	10/0	4.110	0.882	1.733	0.722	0.910	1.460
40000	3000	7	10/0	5.557	1.289	2.382	1.114	1.250	2.196
50000	4000	7	10/0	6.793	1.666	3.036	1.386	1.632	2.730
60000	5000	7	10/0	7.942	1.260	3.719	1.833	1.971	3.402
10000	1000	10	10/0	0.143	0.050	0.377	0.034	0.048	0.312
20000	2000	10	10/0	0.408	0.124	0.821	0.090	0.099	0.664
30000	3000	10	10/0	0.726	0.258	1.311	0.197	0.179	1.076
40000	4000	10	10/0	0.963	0.305	1.826	0.241	0.328	1.511
50000	5000	10	10/0	1.262	0.441	2.372	0.357	0.395	1.887

average execution time was taken as the maximum execution time limit only if all ten executions failed. Symbol "G/L" in Column 6 stands for the number of times that all the remaining SAT optimization algorithms hit the *global/local* minimum points.

TABLE 2. Performance comparison averaged over ten runs between a **DPL** and some optimization algorithms on a SUN SPARC 2 workstation for solving 3-SAT problem instances. Time units: seconds. Symbol "S/F" in Column 4 stands for DPL's *success/failure* to give an answer within a time limit of $120 \times m/n$ seconds, whereas symbol "G/L" stands for the number of times that all the remaining SAT optimization algorithms hit *global/local* minimum points.

Problems			Execution Time							
m	n	l	S/F	DPL	G/L	SAT1.7	SAT1.13	SAT1.8	SAT14.6	SAT14.16
500	500	3	10/0	2.159	10/0	0.013	0.021	0.011	0.013	0.027
750	500	3	10/0	2.916	10/0	0.015	0.030	0.013	0.021	0.033
1000	500	3	10/0	3.657	10/0	0.035	0.048	0.032	0.031	0.047
1250	500	3	9/1	5.797	10/0	0.049	0.064	0.044	0.067	0.072
1500	500	3	6/4	9.147	10/0	0.108	0.117	0.089	0.115	0.108
1000	500	4	10/0	4.684	10/0	0.026	0.040	0.024	0.024	0.038
1500	500	4	10/0	7.960	10/0	0.040	0.075	0.043	0.042	0.066
2000	500	4	8/2	10.27	10/0	0.066	0.105	0.062	0.069	0.088
2500	500	4	2/8	15.96	10/0	0.074	0.130	0.085	0.109	0.152
3000	500	4	1/9	46.33	10/0	0.118	0.201	0.115	0.153	0.234
3000	500	5	10/0	16.90	10/0	0.094	0.137	0.082	0.074	0.139
4000	500	5	5/5	28.39	10/0	0.119	0.204	0.103	0.144	0.188
5000	500	5	0/10	>1200	10/0	0.180	0.253	0.170	0.196	0.288
6000	500	5	0/10	>1440	10/0	0.313	0.370	0.267	0.313	0.471
7000	500	5	0/10	>1680	10/0	0.591	0.575	0.478	0.604	0.623
10000	1000	10	10/0	101.8	10/0	0.047	0.382	0.038	0.049	0.315
12000	1000	10	10/0	124.3	10/0	0.073	0.458	0.053	0.052	0.382
14000	1000	10	10/0	145.2	10/0	0.077	0.562	0.058	0.063	0.430
16000	1000	10	10/0	167.1	10/0	0.098	0.596	0.073	0.078	0.517
18000	1000	10	10/0	188.6	10/0	0.136	0.716	0.102	0.095	0.583

From numerous algorithm executions, we observe that, for random l-SAT instances listed in Table 2, DPL was slower than the rest of the SAT optimization algorithms. As the input size increases, the number of failures, F, increased quickly. For some slightly large inputs, such as $m = 5000, n = 500$, and $l = 5$, all ten algorithm executions failed after a reasonably long time limit. Due to its $O(m^{O(n/l)})$ average run-time complexity, even for some fairly easy instances, such as $m = 10000, n = 1000$, and $l = 10$, DPL took an excessive amount of time to find a solution. In comparison, local search and global optimization algorithms were successful for all ten executions. They were able to find a solution to the given instances efficiently.

Table 2 suggests that DPL may not be a suitable candidate for large size random l-SAT instances. This observation should not be generalized to other application cases. In many other applications, as observed by others [**69, 70, 126**], DPL performed very well.

Performance Comparison with *GSAT*. Table 3 compares the performance between some local search and global optimization algorithms running on a SUN SPARC 2 workstation and *GSAT* [**472**] running on a MIPS computer with comparable computing power [**471**]. Since *GSAT* is essentially a version of sequential local search (*i.e.*, *SAT*1) algorithm, for solving 3-SAT instances generated from the same input model used in [**383**], local search and global optimization algorithms performed approximately tens to hundreds times faster than *GSAT*. Among them, parallel *DNF* local search (*SAT*1.8) algorithm and complete global optimization (*SAT*14.16) were the best.

TABLE 3. Performance comparison between some optimization algorithms running on a SUN SPARC 2 workstation and the **GSAT algorithm** running on a MIPS computer with comparable computing power for the 3-SAT problem instances. Time units: seconds.

Problems			Execution Time					
m	n	l	GSAT	SAT1.7	SAT1.13	SAT1.8	SAT14.6	SAT14.16
215	50	3	0.400	0.019	0.100	0.006	0.020	0.030
301	70	3	0.900	0.054	0.010	0.020	0.020	0.020
430	100	3	6.000	0.336	0.040	0.420	0.050	0.370
516	120	3	14.00	0.596	0.810	1.136	0.410	0.040
602	140	3	14.00	0.260	8.060	0.750	1.990	0.170
645	150	3	45.00	0.102	0.190	0.120	0.040	0.870
860	200	3	168.0	1.776	0.970	0.070	6.710	0.490
1062	250	3	246.0	3.106	20.71	0.070	12.43	0.090
1275	300	3	720.0	8.822	19.66	3.750	19.14	4.510

TABLE 4. Performance comparison between some optimization algorithms running on a SUN SPARC 2 workstation and **an interior point zero-one integer programming algorithm** running on a *KORBX(R)* parallel/vector computer for solving average 3-SAT problem instances. Time units: seconds. Symbol "S/F" stands for the number of times that IP hits the *global/local* minimum points, whereas symbol "G/L" stands for the number of times that the remaining SAT algorithms hit the *global/local* minimum points.

Problems			Execution Time							
m	n	l	S/F	IP	G/L	SAT1.7	SAT1.13	SAT1.8	SAT14.6	SAT14.16
100	50	5	52/0	0.7	10/0	0.001	0.004	0.001	0.001	0.004
200	100	5	70/0	1.1	10/0	0.006	0.010	0.006	0.005	0.007
400	200	7	69/0	3.5	10/0	0.007	0.014	0.007	0.007	0.018
800	400	10	31/0	5.6	10/0	0.009	0.034	0.009	0.003	0.030
800	400	7	20/0	7.8	10/0	0.014	0.032	0.014	0.009	0.026
1000	500	10	49/0	7.4	10/0	0.012	0.037	0.012	0.006	0.039
2000	1000	10	10/0	18.5	10/0	0.032	0.091	0.032	0.019	0.083
2000	1000	7	50/0	21.5	10/0	0.056	0.099	0.056	0.055	0.055
2000	1000	3	49/1	50.4	10/0	2.657	0.162	2.657	3.917	27.19
4000	1000	4	1/1	1085.4	10/0	10.63	11.07	10.63	6.826	9.555
4000	1000	10	10/0	25.1	10/0	0.055	0.189	0.055	0.044	0.163
8000	1000	10	10/0	38.0	10/0	0.219	0.456	0.219	0.254	0.353
16000	1000	10	10/0	66.4	10/0	0.603	1.042	0.603	0.625	1.052
32000	1000	10	10/0	232.4	10/0	1.701	2.720	1.701	1.611	2.434

Performance Comparison with Interior Point Zero-One Integer Programming Algorithm. Recently, Kamath *et al.* used an interior point zero-one integer programming algorithm to solve SAT [**303, 301**]. They implemented their algorithm in FORTRAN and C languages and ran the algorithm on a *KORBX(R)* parallel/vector computer with instances generated from the average 3-SAT input model. The *KORBX(R)* parallel computer operates in scalar mode at approximately 1 MFlops and at 32 MFlops with full vector concurrent mode. Their execution results are given in Columns 4 and 5 of Table 4.

We ran local search and global optimization algorithms for the same instances (listed in [**303, 301**]) on a SUN SPARC 2 workstation. The results are given in

TABLE 5. **WSAT** (*GSAT* with random walk)'s real execution performance for hard random 3-SAT problem instances on an SGI Challenge with a 70 MHz MIPS R4400 processor. Time unit: seconds [**475**].

Problems		*GSAT*			Walk		
n	m	time	flips	R	time	flips	R
100	430	.4	7554	8.3	.2	2385	1.0
200	860	22	284693	143	4	27654	1.0
400	1700	122	2.6×10^6	67	7	59744	1.1
600	2550	1471	30×10^6	500	35	241651	1.0
800	3400	*	*	*	286	1.8×10^6	1.1
1000	4250	*	*	*	1095	5.8×10^6	1.2
2000	8480	*	*	*	3255	23×10^6	1.1

TABLE 6. **WSAT** (*GSAT* with random walk)'s real execution performance for hard random 3-SAT problem instances on a PC. Time unit: seconds [**370**].

n	m	inst.	time	flips	solved	ratio
100	430	500	0.18	2803	88%	31,85
200	860	500	1.99	18626	73%	255,85
400	1700	500	15.03	204670	100%	2046,70
600	2550	500	19.59	250464	62%	4013,85
800	3400	500	140.61	1809986	67%	26854,39
1000	4250	500	369.88	4633763	57%	81009,84
2000	8240	50	3147.26	26542387	16%	1658899,19

Table 4. Apparently, as compared to the interior point zero-one integer programming algorithm running on a parallel computer, in addition to improved global convergence, local search and global optimization algorithms were much simpler and achieved several orders of magnitude of performance improvements in terms of computing time.

13.2. Experiments on Hard Random Formulas. We compare the performance of two local search algorithms and a tabu search algorithm for the hard random 3-SAT problem instances generated from the **mwff** generator. All three programs were written in *C*. Table 5 give the real execution performance of the *WSAT* (*GSAT* with random walk) on an SGI Challenge with a 70 MHz MIPS R4400 processor [**475, 474**].

Tables 6 and 7 show the experimental results of *WSAT* and *TSAT* (Tabu search for SAT) programs written in *C* under Linux 1.1.59 for PC [**370**]. On a same machine, Mazure, Sais, and Gregoire compared the *GSAT* with *TSAT* and found that *TSAT* was more efficient in most cases. In addition, *TSAT* was able to solve

TABLE 7. **TSAT**'s real execution performance for hard random 3-SAT problem instances on a PC. Time unit: seconds [**370**].

n	m	inst.	time	flips	solved	ratio
100	430	500	0.11	1633	93%	17,60
200	860	500	0.73	9678	74%	130,78
400	1700	500	11.51	145710	100%	1457,10
600	2550	500	13.92	167236	65%	2580,80
800	3400	500	99.45	1143444	71%	16150,34
1000	4250	500	292.10	3232463	62%	51802,29
2000	8240	50	3269.15	29415465	40%	735386,63

more problem instances compared to the *GSAT*. The testing for *TSAT* for large size example with $n = 2000$ and $m = 8240$, however, was terminated at $n/m = 4.12$, before entering into the hard region of the random 3-SAT instances.

The performance of the *SAT1.5* algorithm [**209, 219**] (Section 7.7) is shown in Table 8. For hard problem instances in the transition region [**383**], *SAT1.5* can solve large-size SAT problem instances efficiently. It took *WSAT* on average 3,255 seconds to solve the $n = 2,000$ and $m = 8,480$ instances on an SGI Challenge with a 70 MHz MIPS R4400 processor. On a SUN SPARC 20 workstation, the *SAT1.5* algorithm was able to solve the same problem instance in some 530 seconds on average [**219, 220**]. For hard, large size SAT problem instances with $n > 5,000$, *SAT1.5* algorithm was able to handle the problems comfortably.

13.3. Experiments on Structured Instances. We now take a look at the performance of SAT algorithms for some structured instances.

Instances Generated from the N-Queens Problem. To assess the performance of local search and global optimization algorithms with *non-binary* instances, we also tested SAT instances generated from instances of the n-queens problem. Figure 32 compares the performance between DP and some optimization algorithms. It also compares the performance between DP and *SAT*14.11 [**211**], a complete, continuous global optimization algorithm. Due to expensive floating point computations, the execution time of *SAT*14.11 is higher than those of other discrete local search and global optimization algorithms.

DIMACS Instances. For the same SAT formulas generated from instances of the Boolean inference problem [**302**], the performance of *SAT1.7* [**212**], a parallel local search algorithm, and a simple backtracking algorithm [**328**] is shown in Tables 9 and 10, respectively. An algorithm may be effective for only one type of input. The results suggest that it can be much more efficient if we use several different types of algorithms to handle the same inputs simultaneously.

In Table 11, we compare A_2 [**538**] with WSAT, GSAT, and Davis-Putnam's algorithm in solving the circuit diagnosis benchmark problems. We present average execution times and average number of iterations of A_2 as well as published average execution times of WSAT, GSAT and Davis-Putnam's method [**475**]. We did not attempt to reproduce the reported results of GSAT and WSAT, since the results

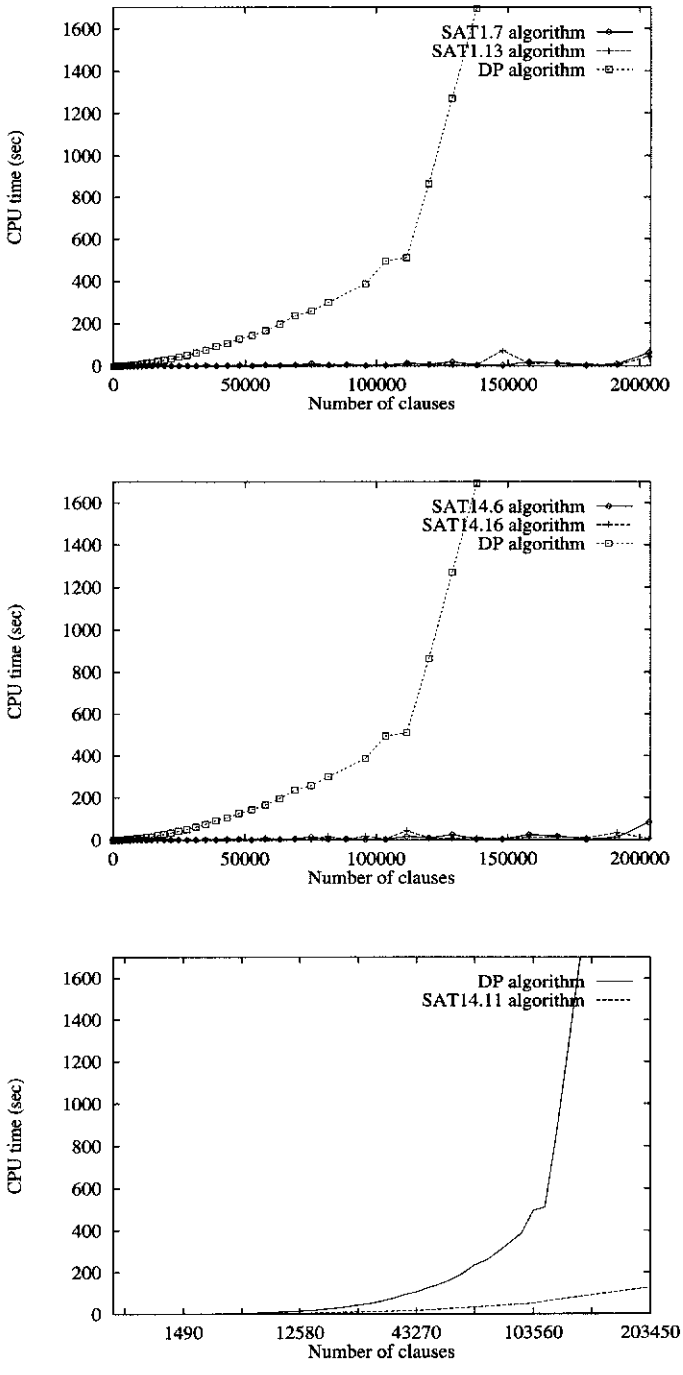


FIGURE 32. Comparison of DP with **SAT1.7**, **SAT1.13**, **SAT14.6**, **SAT14.16**, and **SAT14.11** for solving SAT instances generated from CSP instances

TABLE 8. Real execution performance of the **SAT1.5 algorithm** for hard random 3-SAT problem instances on a SUN SPARC 20 workstation. For each problem, 30 random instances were tested. The minimum (T_{min}), maximum (T_{max}), and average (T_{mean}) execution times were recorded. "S" indicates the number of success cases of finding solutions within the time limit (*T-limit*). Time unit: second.

n	m	m/n	S/30	T_{min}	T_{mean}	T_{max}	T-limit
1000	4230	4.2300	20/30	6.04	206.90	956.15	1000
1000	4240	4.2400	15/30	0.55	223.26	891.11	1000
1000	4250	4.2500	14/30	1.69	88.370	454.79	1000
1000	4260	4.2600	10/30	24.4	243.25	914.35	1000
2000	8460	4.2300	12/30	115.8	779.44	2069.9	3000
2000	8480	4.2400	14/30	17.64	530.32	1360.9	3000
2000	8500	4.2500	7/30	58.09	789.35	1677.6	3000
2000	8510	4.2550	9/30	59.08	840.33	2322.8	3000
2000	8460	4.2300	15/30	58.95	684.32	4508.2	5000
2000	8480	4.2400	15/30	15.31	1273.5	4057.9	5000
2000	8500	4.2500	15/30	112.7	1527.8	3644.5	5000
2000	8520	4.2600	9/30	123.2	1522.9	4338.5	5000
3000	12690	4.2300	12/30	430.6	1787.2	2876.4	5000
3000	12700	4.2333	18/30	122.5	2101.5	4479.4	5000
3000	12720	4.2400	11/30	270.6	1503.6	3840.7	5000
3000	12740	4.2467	12/30	229.1	2062.9	4807.4	5000
3000	12680	4.2267	15/30	356.1	2788.3	9510.2	10000
3000	12700	4.2333	11/30	503.3	3681.1	8247.9	10000
3000	12720	4.2400	15/30	30.09	2300.3	7002.3	10000
3000	12740	4.2467	8/30	563.3	2620.5	5330.9	10000
4000	16920	4.2300	11/30	739.83	4064.5	11498.2	12000
4000	16930	4.2325	10/30	1733.5	5472.0	10187.8	12000
4000	16940	4.2350	7/30	571.20	1948.9	4768.92	12000
4000	16960	4.2400	10/30	294.80	3709.0	9921.77	12000
5000	21150	4.2300	8/30	2024.7	3867.9	8134.81	9000
5000	21175	4.2350	6/30	1640.1	2982.7	4193.68	9000
5000	21200	4.2400	3/30	2935.8	4435.7	6357.65	9000
5000	21225	4.2450	4/30	3883.5	6025.6	10980.9	15000
10000	41000	4.1000	30/30	294.44	1315.6	3849.38	20000
10000	41800	4.1800	8/18	4294.5	8387.9	16654.8	20000
10000	42000	4.2000	4/30	963.53	5877.8	12020.3	20000
10000	42200	4.2200	2/30	9270.6	14241.9	19213.4	20000

may depend on initial conditions, such as the seeds of the random number generator and other program parameters. We ran A_2 on an SGI Challenge[9] so that our timing

[9]Based on a single-CPU 150-MHz SGI Challenge with MIPS R4400 at the University of Illinois National Center for Supercomputing Applications, we estimate empirically that it is 15.4%

TABLE 9. Performance of **SAT1.7** on a SUN SPARC 10 workstation. Time Units: seconds.

Problems			Ten Trials		Execution Time		
Name	m	n	Global	SAT	Min	Mean	Max
ii16a1.sat	1650	19368	10/10	YES	10.320	125.51	417.42
ii16b1.sat	1728	24792	10/10	YES	0.6100	6.1130	28.760
ii16c1.sat	1580	16467	10/10	YES	0.5400	1.8740	3.7500
ii16d1.sat	1230	15901	10/10	YES	0.4900	1.3810	2.8200
ii16e1.sat	1245	14766	10/10	YES	0.5300	0.9720	1.4800
ii16a2.sat	1602	23281		N/A			
ii16b2.sat	1076	16121	10/10	YES	1.9000	39.118	102.60
ii16c2.sat	924	13803	10/10	YES	0.3500	14.109	41.650
ii16d2.sat	836	12461	10/10	YES	0.3300	19.840	52.410
ii16e2.sat	532	7825	10/10	YES	0.5000	6.8830	21.980
ii32a1.sat	459	9212	10/10	YES	0.3600	3.5740	10.330
ii32b1.sat	228	1374	10/10	YES	0.1100	0.7390	1.6700
ii32b2.sat	261	2558	10/10	YES	0.1000	1.9040	4.4700
ii32b3.sat	348	5734	10/10	YES	1.6400	10.559	19.330
ii32b4.sat	381	9618	10/10	YES	0.5100	2.3060	4.7800
ii32c1.sat	225	1280	10/10	YES	0.0100	0.1150	0.4800
ii32c2.sat	249	2182	10/10	YES	0.0600	0.3980	0.9000
ii32c3.sat	279	3272	10/10	YES	0.6900	5.4900	16.850
ii32c4.sat	759	20862	10/10	YES	5.5200	361.80	1496.3
ii32d1.sat	332	2703	10/10	YES	0.2200	1.0680	3.1000
ii32d2.sat	404	5153	10/10	YES	0.2100	0.9140	2.1800
ii32d3.sat	824	19478	10/10	YES	1.7100	49.522	109.52
ii32e1.sat	222	1186	10/10	YES	0.0200	0.3260	1.0700
ii32e2.sat	267	2746	10/10	YES	0.0400	0.1130	0.3400
ii32e3.sat	330	5020	10/10	YES	0.4500	5.2700	13.910
ii32e4.sat	387	7106	10/10	YES	0.2700	10.734	46.750
ii32e5.sat	522	11636	10/10	YES	0.4900	23.424	84.470

TABLE 10. Performance of **a simple backtracking algorithm** on a SUN SPARC 10 workstation. Time Units: seconds.

Name	m	n	SAT	Time	Name	m	n	SAT	Time
ii16a1.sat	1650	19368	YES	1.285	ii16b1.sat	1728	24792	YES	1.490
ii16c1.sat	1580	16467	N/A	1.956	ii16d1.sat	1230	15901	YES	1.660
ii16e1.sat	1245	14766	N/A	2.125					
ii16a2.sat	1602	23281	YES	1.430	ii16b2.sat	1076	16121	YES	1.505
ii16c2.sat	924	13803	YES	2.016	ii16d2.sat	836	12461	YES	1.665
ii16e2.sat	532	7825	N/A	2.051					
ii32a1.sat	459	9212	YES	1.160	ii32b1.sat	228	1374	YES	1.035
ii32b2.sat	261	2558	YES	1.035	ii32b3.sat	348	5734	YES	1.240
ii32b4.sat	381	9618	YES	1.285					
ii32c1.sat	225	1280	YES	0.000	ii32c2.sat	249	2182	YES	1.325
ii32c3.sat	279	3272	YES	1.240	ii32c4.sat	759	20862	YES	1.695
ii32d1.sat	332	2703	YES	1.035	ii32d2.sat	404	5153	YES	1.525
ii32d3.sat	824	19478	YES	1.755					
ii32e1.sat	222	1186	YES	0.000	ii32e2.sat	267	2746	YES	1.035
ii32e3.sat	330	5020	YES	1.565	ii32e4.sat	387	7106	YES	1.615
ii32e5.sat	522	11636	YES	1.655					

TABLE 11. Comparison of $\mathbf{A_2}$'s execution times in seconds averaged over 10 runs with respect to published results on some of the circuit diagnosis problems in the DIMACS archive, including the best known results obtained by WSAT, GSAT, and Davis-Putnam's algorithm [**475**].

Problem Id	n	m	A_2			WSAT	GSAT	DP
			SS 10/51	SGI	# Iter.			
ssa7552-038	1501	3575	0.228	0.235	7970	2.3	129	7
ssa7552-158	1363	3034	0.088	0.102	2169	2	90	*
ssa7552-159	1363	3032	0.085	0.118	2154	0.8	14	*
ssa7552-160	1391	3126	0.097	0.113	3116	1.5	18	*

- A_2: Sun SparcStation 10/51 and a 150-MHz SGI Challenge with MIPS R4400;
- GSAT, WSAT and DP: SGI Challenge with a 70 MHz MIPS R4400.

TABLE 12. Comparison of $\mathbf{A_2}$'s execution times in seconds averaged over 10 runs with published results on circuit synthesis problems from the DIMACS archive, including the best known results obtained by GSAT, integer programming, and simulated annealing [**475**].

Problem Id.	n	m	A_2			GSAT	Integer Prog.	SA
			SS 10/51	SGI	# Iter.			
ii16a1	1650	19368	0.122	0.128	819	2	2039	12
ii16b1	1728	24792	0.265	0.310	1546	12	78	11
ii16c1	1580	16467	0.163	0.173	797	1	758	5
ii16d1	1230	15901	0.188	0.233	908	3	1547	4
ii16e1	1245	14766	0.297	0.302	861	1	2156	3

- A_2: Sun SparcStation 10/51 and a 150-MHz SGI Challenge with MIPS R4400;
- GSAT and SA: SGI Challenge with a 70 MHz MIPS R4400;
- Integer Programming: VAX 8700.

results can be compared to those of GSAT and WSAT. Our results show that A_2 is approximately one order of magnitude faster than WSAT.

In Table 12, we compare A_2 [**538**] with the published results of GSAT, integer programming and simulated annealing on the circuit synthesis problems [**475**]. Our results show that A_2 performs several times faster than GSAT.

In Table 13, we compare the performance of the three versions of DLM with some of the best known results of GSAT on circuit-synthesis, parity-learning, some artificially generated 3-SAT, and some of the hard graph coloring problems. The results on GSAT are from [**476**], which are better than other published results. Our results show that DLM is consistently faster than GSAT on the "*ii*" and "*par*" inputs, and that A_1 is an order-of-magnitude faster than GSAT on some "*aim*" inputs.

---

slower than a Sun SparcStation 10/51 for executing A_2 to solve SAT benchmark problems. However, we did not evaluate the speed difference between a 150-MHz SGI Challenge and a 70-MHz SGI Challenge on which GSAT and WSAT were run.

TABLE 13. Comparison of **DLM**'s execution times in seconds averaged over 10 runs with the best known results obtained by GSAT [**476**] on the circuit-synthesis, parity-learning, artificially generated 3-SAT instances, and graph coloring problems from the DIMACS archive.

Problem Identification	n	m	A_1		GSAT	
			SS 10/51	Success Ratio	Time	Success Ratio
aim-100-2_0-yes1-1	100	200	0.19	10/10	1.96	9/10
aim-100-2_0-yes1-2	100	200	0.65	10/10	1.6	10/10
aim-100-2_0-yes1-3	100	200	0.19	10/10	1.09	10/10
aim-100-2_0-yes1-4	100	200	0.10	10/10	1.54	10/10
			A_2		GSAT	
ii32b3	348	5734	0.31	10/10	0.6	10/10
ii32c3	279	3272	0.12	10/10	0.27	10/10
ii32d3	824	19478	1.05	10/10	2.24	10/10
ii32e3	330	5020	0.16	10/10	0.49	10/10
par8-2-c	68	270	0.06	10/10	1.33	10/10
par8-4-c	67	266	0.09	10/10	0.2	10/10
			A_3		GSAT	
g125.17	2125	66272	1390.32	10/10	264.07	7/10
g125.18	2250	70163	3.197	10/10	1.9	10/10
g250.15	3750	233965	2.798	10/10	4.41	10/10
g250.29	7250	454622	1219.56	9/10	1219.88	9/10

- A_1, A_2, A_3: Sun SparcStation 10/51
- GSAT: SGI Challenge (model unknown)

We are designing new strategies to improve A_3's [**538**] performance. Tables 14 shows some preliminary but promising results of A_3 on some of the more difficult but satisfiable DIMACS benchmark inputs.

13.4. Experiments on Practical Industrial Benchmarks. Performance of the *SAT-Circuit* Solver with Partitioning Preprocessing. We compare in Table 15 Gu and Puri's SAT solver (having a partitioning preprocessing) [**216**] with existing algorithms [**331, 529**] for solving industrial asynchronous circuit design benchmarks, including the HP and Philips benchmarks. In the table, N and m are the initial number of states and initial number of signals, respectively. Correspondingly, N^f and m^f are the final number of states and final number of signals. Symbol A indicates the 2-level implementation area.

The experimental results indicate that, as compared to the previous methods [**331, 529**], the SAT-Circuit solver with partitioning preprocessing achieves many orders of magnitude of performance improvement in terms of computing time, in addition to a reduced implementation area. For example, in a large circuit $mr0$, SAT-Circuit took 2.80 seconds to solve the problem and yielded a two-level implementation area with 41 literals.[10] In contrast, Lavagno *et al.*'s algorithm took 1,084.5 seconds and an area of 86 literals. For this example, Vanbekbergen *et*

[10] *Literal* here is a standard unit measuring layout area.

TABLE 14. Execution times in CPU seconds over 10 runs of $\mathbf{A}_3$ to solve some of the more difficult DIMACS benchmark problems.

Prob. Id.	Succ. Ratio	Sun SS 10/51 Seconds		
		Avg.	Min.	Max.
par8-1	10/10	4.780	0.133	14.383
par8-2	10/10	5.058	0.100	13.067
par8-3	10/10	9.903	0.350	21.150
par8-4	10/10	5.842	0.850	16.433
par8-5	10/10	14.628	1.167	34.900
par16-1	5/10	11172.8	4630.6	20489.1
par16-2	1/10	856.9	856.9	856.9
par16-3	1/10	20281.6	20281.6	20281.6
par16-4	3/10	3523.1	1015.0	7337.9
par16-5	1/10	13023.4	13023.4	13023.4
par16-1-c	10/10	398.1	11.7	1011.9
par16-2-c	10/10	1324.3	191.0	4232.3
par16-3-c	10/10	987.2	139.8	3705.2
par16-4-c	10/10	316.7	5.7	692.66
par16-5-c	10/10	1584.2	414.5	3313.2
hanoi4	1/10	476.5	476.5	476.5
f1000	10/10	126.8	4.4	280.7
f600	10/10	16.9	2.1	37.2
f2000	10/10	1808.6	174.3	8244.7

Program parameters				
Flat region limit = 50; λ reset interval = 10,000; operation: $\lambda = \lambda/1.5$.				
Problem group	par-16-[1-5]	test par problems	f	hanoi4
Tabu length	100	50	50	50
Increment of λ	1	1/2	1/16	1/2

al.'s algorithm could not yield a solution within 3,600 seconds and aborted due to backtracking limit. For another benchmark circuit *mmu*0, SAT-Circuit solved it in 0.87 seconds, as compared to a pre-aborted 406.3 seconds for Vanbekbergen *et al.*'s approach [**529**].

Performance of a BDD SAT Solver with Partitioning Preprocessing. The BDD SAT-Circuit solver was implemented in C language. In this case, Gu and Puri tested their BDD SAT-Circuit solver with its ability to find all solutions (therefore, an optimal solution) for a large number of industrial asynchronous circuit benchmarks including the HP and Philips benchmarks [**216, 438**]. They also compared the performance of their BDD SAT-Circuit solver with the well known Lavagno *et al.*'s [**331**] asynchronous circuit design technique. The results of these experiments are given in Table 16 and Table 17. Table 16 compares the execution time of the BDD SAT solver with the execution time of a simple backtracking SAT algorithm of [**328**]. The experimental results are given for SAT instances generated from Gu and Puri's SAT formula partitioning preprocessor [**216**]. Since the BDD SAT-Circuit solver yielded all the solutions, they normalized the execution time of the backtracking algorithm for all the truth assignment. The experimental results (Table 16) show that the BDD SAT solver outperforms the backtracking SAT technique for the practical SAT instances representing asynchronous circuit design.

TABLE 15. Experimental results comparing the **SAT-Circuit solver** (with SAT formula partitioning preprocessing), Vanbekbergen *et al.*'s algorithm, and Lavagno *et al.*'s algorithm, on practical circuit benchmarks on a SUN SPARC-2 workstation. Time unit: seconds.

Circuit Specifications			*Preprocessing* [216]				*Vanbekbergen et al.* [529]				*Lavagno et al.* [331]		
Name	N	m	N^f	m^f	A	CPU	N^f	m^f	A	CPU	m^f	A	CPU
mr0	302	11	469	14	41	2.80	backtrack limit >			3600	13	86	1084.5
mr1	190	8	373	12	55	1.73	backtrack limit >			872.9	10	53	237.5
mmu0	174	8	441	11	49	0.87	backtrack limit >			406.3	state error		
mmu1	82	8	131	10	50	0.37	backtrack limit >			101.3	10	37	47.8
sbuf-ram-write	58	10	93	12	59	0.36	90	12	74	5.21	12	35	54.6
vbe4a	58	6	106	8	37	0.19	116	8	40	0.25	8	41	5.50
nak-pa	56	9	59	10	25	0.20	58	10	32	0.08	10	41	20.8
pe-rcv-ifc-fc	46	8	50	9	48	0.24	53	9	50	0.13	9	62	14.3
ram-read-sbuf	36	10	44	11	28	0.15	53	11	44	0.06	11	23	65.2
alex-nonfc	24	6	31	7	26	0.05	28	7	22	0.03	non-free-choice		
sbuf-send-pkt2	21	6	26	7	20	0.04	27	7	29	0.04	7	14	8.6
sbuf-send-ctl	20	6	32	8	33	0.09	28	8	35	0.03	8	43	3.4
atod	20	6	26	7	15	0.02	24	7	16	0.01	7	19	2.9
pa	18	4	34	6	18	0.12	31	6	22	0.06	state error		
alloc-outbound	17	7	29	9	33	0.09	24	9	27	0.04	9	23	2.5
wrdata	16	4	20	5	17	0.03	19	5	18	0.01	5	21	0.9
fifo	16	4	23	5	15	0.03	20	5	17	0.02	5	15	0.7
sbuf-read-ctl	14	6	18	7	16	0.06	16	7	20	0.01	7	15	1.5
nousc	12	3	16	4	12	0.01	16	4	12	0.01	4	14	0.5
vbe-ex2	8	2	12	4	18	0.08	12	4	18	0.03	4	21	0.5
nousc-ser	8	3	10	4	9	0.02	10	4	9	0.01	4	11	0.4
sendr-done	7	3	10	4	8	0.02	10	4	8	0.01	4	6	0.4
vbe-ex1	5	2	8	3	7	0.01	8	3	7	0.01	3	7	0.3

TABLE 16. Experimental results comparing the **BDD SAT-Circuit solver** and a backtracking SAT algorithm, both with SAT formula partitioning preprocessing, on practical asynchronous circuit benchmarks on a SUN SPARC-2 workstation. Time unit: second.

STG Benchmark Name	BDD SAT Solver	Backtracking satisfiability testing	STG Benchmark Name	BDD SAT Solver	Backtracking satisfiability testing
Mr0	58.3	>3,600	Mmu1	28.1	>3,600
SbufRamWr	32.7	>3,600	Vbe4a	1.95	>3,600
NakPa	0.53	5.4	RamRdSbuf	0.25	76.8
AlexNonFc	0.37	0.96	SbufSndPkt2	0.37	88.06
SbufSndCtl	18.27	353.6	AtoD	0.15	11.88
Pa	0.05	4.50	WrData	0.14	0.24
Fifo	0.05	0.10	SbufRdCtl	0.09	0.10
NoUsc	0.09	0.16	VbeEx2	3.94	0.80
NoUscSer	0.06	0.07	SendrDone	0.05	0.16
VbeEx1	0.03	0.04			

TABLE 17. Comparison of Implementation area and design time of the BDD SAT-Circuit solver (with SAT formula partitioning preprocessing) and Lavagno *et al.*'s technique for practical asynchronous circuit benchmarks on a SUN SPARC-2 workstation. Time unit: second.

Benchmark Name	*Benchmark*		*BDD SAT Solver*			*Lavagno and Moon et al.* [331]		
	Initial no. of states	Initial no. of signals	Final no. of signal	Circuit Area (literals)	CPU time sec.	Final no. of signal	Circuit Area (literals)	CPU time sec.
Mr0	302	11	15	41	58.36	13	86	1084.5
Mmu1	82	8	10	38	28.16	10	37	47.8
SbufRamWr	58	10	12	47	32.79	12	35	54.6
Vbe4a	58	6	8	30	1.95	8	41	5.5
NakPa	56	9	10	25	0.53	10	41	20.8
RamRdSbuf	36	10	11	25	0.25	11	23	65.2
SbufSndPkt2	24	6	7	21	0.37	7	14	8.6
SbufSndCtl	21	6	7	17	0.37	8	43	3.4
AtoD	20	6	8	30	18.27	7	19	2.9
Pa	20	6	7	14	0.15	Internal State Error		
WrData	16	4	5	18	0.05	5	21	0.9
Fifo	16	4	5	15	0.14	5	15	0.7
SbufRdCtl	14	6	7	16	0.05	7	15	1.5
NoUsc	12	3	4	12	0.09	4	14	0.5
VbeEx2	12	3	4	12	0.09	4	21	0.5
NoUscSer	8	2	4	18	3.94	4	11	0.4
AlexNonFc	8	3	4	9	0.06	Non-Free-Choice STG		
SendrDone	7	3	4	8	0.05	4	6	0.4
VbeEx1	5	2	3	6	0.03	3	7	0.3

They also calculated the implementation area of the designed circuits. Table 17 compares their BDD SAT solver with the well known Lavagno *et al.*'s asynchronous circuit design technique [**331**]. The BDD SAT-Circuit solver yielded reduced circuit implementation area than Lavagno *et al.*'s algorithm for almost all the circuits in the benchmark set [**331**]. Lavagno *et al.*'s method yields a total area of 449 literals in 1298.5 seconds. In comparison, for the same benchmarks, the BDD SAT solver achieved an area of 379 literals in 145.7 seconds. In addition, Lavagno *et al.*'s method was unable to solve some benchmark circuits, such as *Pa* and *AlexNonFc*. These results show that, as compared to existing techniques, the BDD SAT solver is capable of achieving an average of 20% reduction in implementation area for all the benchmarks. According to critical industrial evaluations, this BDD SAT solver offers a practical solution for complex industrial asynchronous circuit design problems.

14. Applications

Practical application problems are the driving forces for SAT research. They provide the ultimate benchmarks to test SAT algorithms and techniques. An effective SAT algorithm in one application problem will shed light on solving problems in other application areas.

The SAT problem has *direct* applications in mathematical logic, artificial intelligence, VLSI engineering, and computing theory. It also has *indirect* applications

through other transferable problems, e.g., constraint satisfaction problems and constrained optimization problems [228]. Due to the *UniSAT* models, some application problems in the real space are related to SAT as well. In the following, we list some applications that can be formulated as solved as instances of SAT.

- *Mathematics:* finding n-ary relations such as transitive closure [67], detecting graph and subgraph isomorphisms [105, 373, 375, 399, 442, 519, 558], the graph coloring problem [57, 243, 369, 375], mathematical cryptology [408, 447], the automata homomorphism problem [198], finding spanning trees and Euler tours in a graph [396], solving the traveling salesman problem [287, 288, 332, 400], and logical arithmetic [93].
- *Computer science and artificial intelligence:* the constraint satisfaction problem [13, 191, 206, 361, 451], the n-queens problem [191, 242, 485], extended inference [22], logical programming [96, 98, 139, 329], abductive inference for synthesizing composite hypotheses [297], semantic information processing [22, 161, 397], puzzles and cryptoarithmetic [189, 242, 275, 367, 368, 394], truth maintenance [122, 124, 127, 138, 371], production system [278, 381, 382], the soma cube and instant insanity problem [191], theorem proving [269, 316, 427, 557], and neural network computing [13, 14, 129, 251, 354, 272].
- *Machine vision:* image matching problem [22, 88, 450, 553], line and edge labeling problems [76, 159, 515, 541, 561], stereopsis, scene analysis and semantics-based region growing [22, 76, 158, 159, 160, 515, 541], the shape and object matching problem [67, 115, 247], syntactic shape analysis [116, 246, 310], shape from shading problem [12, 50, 173, 262, 263, 265, 264, 274, 364, 412], and image restoration [193].
- *Robotics:* related vision problem [88, 273], packing problem [133], and trajectory and task planning problems [46, 152].
- *Computer-aided manufacturing:* task planning [393], design [391, 392], solid modeling, configuring task [174], design cellular manufacturing system, scheduling [164, 356], and 3-dimensional object recognition [230, 264].
- *Database systems:* operations on objects [518, 521], database consistency maintenance, query-answering and redundancy-checking, query optimization [78, 518], concurrency control [31, 154, 360], distributed database systems [185], truth and belief maintenance [122, 124, 127, 138, 371], the relational homomorphism problem [242, 518], and knowledge organization for recognition system [244].
- *Text processing:* optical character recognition [90, 387, 502], character constraint graph model [270], printed text recognition [21, 270], handwritten text recognition [483], automatic correction of errors in text [520].
- *Computer graphics:* construction of 2-dimensional pictures and 3-dimensional graphical objects from constraints, reasoning of the geometrical features of 3-dimensional objects [55, 180].
- *Integrated circuit design automation:* circuit modeling [75, 509], logic minimization [254], state assignment [529, 530], state minimization [204, 441], asynchronous circuit synthesis [216, 438, 437, 439], I/O encoding for sequential machines [458], power dissipation estimation [135], logic partitioning [85, 143, 327, 398, 456], circuit layout and placement [11, 36, 47, 97, 112, 134, 234, 494], scheduling and high-level synthesis [48, 325, 409],

pin assignment [**45, 455**], floorplanning [**418, 503**], interconnection analysis [**141, 142**], routing [**1, 71, 131, 132, 233, 335, 407, 444, 448, 470**], compaction [**68, 140, 245, 306, 326, 346, 463, 480, 527**], performance optimization [**300, 317, 366, 453, 503**], testing and test generation [**136, 280, 328, 146, 419**], and verification [**315, 505, 516**]. Please also see: Jun Gu, *Satisfiability Problems in VLSI Engineering* [**217**], 1996.

- *Computer architecture design:* instruction set optimization [**4, 114, 205, 283, 436, 440**], computer controller optimization [**27, 33, 307, 358, 435, 441**], arithmetic logic circuit design [**77**], compiler system optimization [**7, 345**], scheduling [**37, 137, 186, 187, 231, 338**], fault-tolerant computing [**24, 261, 20**], task partitioning and assignment [**39, 40, 111, 276, 481**], load balancing [**349, 395, 560**], real time systems [**255, 282, 318, 497, 498, 507**], data flow consistency analysis [**7**], data module assignment in memory system [**7**], and parallel and distributed processing [**443, 468**].
- *High-speed networking:* contact the authors.
- *Communications:* contact the authors.
- *Security:* contact the authors.

In other areas such as industrial (chemical, transportation, construction, nuclear) engineering, management, medical research, social sciences, there are numerous SAT /CSP applications.

15. Future Work

A number of future research directions for the satisfiability problem have been discussed recently. They are further emphasized in the 1996 DIMACS satisfiability workshop.

General Boolean Expressions and Evaluation. Many practical application problems are expressed as Boolean satisfiability problems by a compact set of general Boolean functions. Although the transformation of a general Boolean expression into *CNF* can be done in polynomial time, it will result in a substantially larger clause-form representation [**192, 415**]. While this may not be critical in complexity theory, it will have serious impact on the time to solve these problems. To this end, efficient representation and manipulation of general Boolean functions is crucial to solving practical application problems.

Theoretical Issues. Recent research on SAT has brought up some interesting theoretical problems, such as the average time complexity analysis [**25, 212, 229, 269, 365, 423**], determining satisfiable-unsatisfiable boundary [**109, 309, 383**], global convergence and local convergence rate [**214, 218**], and the structure and hardness of input models [**102, 170, 196**]. Some of the problems, e.g., the average time complexity analysis, are extremely difficult [**337**]. So far only some preliminary efforts based on simplified assumptions were given [**49, 221, 214, 218**].

One of the recent efforts to solve SAT formulas is to find subclasses for which the problem is solvable in polynomial time [**153, 184**]. Future work in this direction aims at building hierarchies of formulae classes, analyzing the properties of such hierarchies, and qualitative evaluation of the hierarchies.

SAT Algorithm Development. The development of new algorithms and improved techniques for satisfiability testing has been a long-term effort of the research community and the industry. From computation/efficiency point of view, specific data structures and implementation details of SAT algorithms are crucial. The algorithm space shows a number of asymmetrical and irregular places, implying further opportunity for new SAT algorithm development.

From an experimental point of view, it is difficult to find a super algorithm that performs well for a wide range of SAT instances. Existing SAT algorithms complement rather than exclude each other by being effective for particular problem instances. One of the future directions is to continue the development of the *Multi-SAT* algorithm, integrating different algorithms using a cluster of computers [**220**] (Section 11.8). Computer hardware and memory space are becoming increasingly inexpensive. If one can trade hardware for improved performance, it can show a promising approach (in fact, trading memory space for speed was a basic design philosophy behind the RISC computer architectures).

For important practical applications, there may be significant problem domain information. Efficient SAT algorithms may be developed by exploring input- and application-specific structures (Section 11.5). Specialized algorithms tailored to particular applications, on the other hand, do provide key insights to general satisfiability testing.

Practical Application Case Study. It has been recognized by SAT researchers that practical application problems are the driving forces for SAT research; they are the ultimate benchmarks to test SAT algorithms. This direction was further addressed by the NSF, the advisory committee, and the organizing committee of the 1996 DIMACS Satisfiability workshop [**147, 289, 290**]. There has been a strong relationship between theory, algorithms, and applications of SAT. A major step in the future is to bring together theorists, algorithmists, and practitioners working on SAT and on industrial applications involving SAT, enhancing the interaction between the three research groups. It would be beneficial to research community and to industry if we can apply theoretical and algorithmic results on SAT to practical problems, while taking these practical problems for further theoretical/algorithmic study. In addition to theoretical/algorithmic study, in the future, we will also further concentrate on significant industrial case studies of SAT, practical applications of SAT algorithms, and practical and industrial SAT benchmarks.

Parallel Algorithms and Architectures. Implementing an algorithm on VLSI hardware architectures is a common practice to speed up algorithm execution. Not only does it offer faster execution speed, certain sequential portions of the algorithm may be implemented in hardware architectures in parallel form. For SAT *per se*, it has certain granularity at the search tree level, clause level, and variable level that lend itself well to parallel processing. A number of parallel algorithms and architectures for solving SAT have been developed and have been found to perform well at different levels of granularity. Two basic approaches have been taken in this direction: implementing parallel SAT inference algorithms on special-purpose VLSI chips [**226, 227**], and implementing tightly-coupled, parallel SAT algorithms on existing sequential computer machines [**207, 212, 212, 493, 492**].

Algorithm Engineering Approach. Aho, Johnson, Karp, Kosaraju, McGeoch, Papadimitriou, and Pevzner have recently proposed an algorithm engineering approach for the experimental testing of algorithms [8]. They believe that "Within theoretical computer science algorithms are usually studied within highly simplified models of computation and evaluated by metrics such as their asymptotic worst-case running time or their competitive ratio. These metrics can be indicative of how algorithms are likely to perform in practice, but they are not sufficiently accurate to predict actual performance. The situation can be improved by using models that take into account more details of system architecture and factors such as data movement and interprocessor communication, but even then considerable experimentation and fine-tuning is typically required to get the most out of a theoretical idea. **Efforts must be made to ensure that promising algorithms discovered by the theory community are implemented, tested and refined to the point where they can be usefully applied in practice.**"

16. Conclusions

The SAT problem is at the *core* of the class of NP-complete problems and has many practical applications. In recent years, many optimization methods, parallel algorithms, and practical techniques have been developed for solving the SAT problem. The past two decades have seen the proliferation of many SAT algorithms: resolution, local search, global optimization, BDD SAT solver, and multispace search, among others. Existing methods complement rather than exclude each other by being effective for particular instances of SAT. In this survey, we present a general algorithm space that integrates existing SAT algorithms into a unified perspective. We describe several major classes of SAT algorithms with the emphasis on introducing recent advances in SAT algorithms. We gave performance evaluation of some existing SAT algorithms. This survey also provides a set of practical applications of SAT. The area of SAT research is a rich land of well-developed theory and methods. To apply theoretical/algorithmic results to practical problems seems the ultimate way to test and benchmark SAT algorithms. Not only will the end results of such an endeavor have a major scientific/industrial impact, but in the process it will push optimization technology to its limit.

Acknowledgments

The authors are grateful to many people for their help in preparing this article. Moshe Vardi suggested that this survey be written for the 1996 DIMACS satisfiability workshop. Bob Johnson, Steve Cook, David Johnson, and Christos Papadimitriou have provided valuable comments to this article. We thank for Ding-Zhu Du, Panos Pardalos, Mauricio Resende, Craig Rushforth, Alasdair Urquhart, Andy Sage, Vaughan Pratt, Ranan Banerji, Lewis Johnson, Endre Boros, Peter Hammer, R.C.T. Lee, David Plaisted, Jinchang Wang, R.E. Stearns, Sandeep Shukla, Daniel Rosenkrantz, Harry B. Hunt III, Ming Li, Xiaotie Deng, Frank Hsu, M.V. Marathe, Dennis Kibler, Vladimir Gurvich, Leonid Khachiyan, Katta Murty, Stephen Vavasis, David Yau, Scott Kirkpatrick, Fred Glover, Pierre Hansen, Allen

Van Gelder, John Schlipf, Randy Bryant, Bart Selman, David Mitchell, Henry Kautz, John Mitchell, Jieh Hsiang, Guojie Li, Wen Gao, Wei Li, Anthony Vannelli, Rina Dechter, James Crawford, Vipin Kumar, Qianping Gu, Wei Wang, Ruchir Puri, Bin Du, Peter Vanbekbergen, Ewald Speckenmeyer, Kazuo Iwama, Giorgio Gallo, Olivier Dubois, H. Kleine Büning, Oliver Kullmann, Ingo Schiermeyer, Henri Morel, Hans Maaren, Philip C. Jackson, Max Böhm, Peter Heusch, Theo Lettmann, D. Pretolani, Cosimo Spera, Ian Gent, Toby Wals, Roberto Battiti, Marco Protasi, Lefteris Kirousis, Arne Lokketangen, and Chen Guo Liang, for their insightful comments on various early versions of this survey.

References

[1] L. Abel. On the order of connections for automatic wire routing. *IEEE Trans. on Computers*, pages 1227–1233, Nov. 1972.

[2] B. Abramson and M. Yung. Divide and conquer under global constraints: A solution to the n-queens problem. *Journal of Parallel and Distributed Computing*, 6:649–662, 1989.

[3] Leonard M. Adleman Molecular Computation of Solutions to Combinatorial Problems *Science*, 266:1021–1024, 1994.

[4] T. Agerwala. Microprogram Optimization : A Survey. *IEEE Trans. on Computers*, C-25:962–973, Oct. 1976.

[5] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.

[6] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, 1985.

[7] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers*. Addison-Wesley, Reading, 1986.

[8] A.V. Aho, D.S. Johnson, R.M. Karp, S.R. Kosaraju, C.C. McGeoch, C.H. Papadimitriou, and P. Pevzner, *Theory of computing: Goals and directions*, March 15, 1996.

[9] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, Jun. 1978.

[10] W. Ahrens. *Mathematische Unterhaltungen und Spiele (in German)*. B.G. Teubner (Publishing Company), Leipzig, 1918-1921.

[11] S.B. Akers. On the use of the linear assignment algorithm in module placement. In *Proc. of the 18th ACM/IEEE Design Automation Conference*, pages 137–144, 1981.

[12] J. Aloimonos. Visual shape computation. *Proceedings of the IEEE*, 76:899–916, Aug. 1988.

[13] J. A. Anderson and G. E. Hinton. *Models of Information Processing in the Brain. In* G. E. Hinton and J. A. Anderson, editors, *Parallel Models of Associative Memory*, chapter 1, pages 9–48. Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1981.

[14] J.A. Anderson and E. Rosenfeld, editors. *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, 1988.

[15] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings 33rd IEEE Symposium on the Foundations of Computer Science*, pages 14–23, 1992.

[16] K. J. Arrow and L. Hurwicz. Gradient method for concave programming, I: Local results. In K. J. Arrow, L. Hurwica, and H. Uzawa, editors, *Studies in Linear and Nonlinear Programming*. Stanford University Press, Stanford, CA, 1958.

[17] P. Ashar, A. Ghosh, and S. Devadas. Boolean satisfiability and equivalence checking using general Binary Decision Diagrams. *Integration, the VLSI journal*, 13:1–16, 1992.

[18] B. Aspvall, M.F. Plass, and R.E. Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8(3):121–132, Mar. 1979.

[19] B. Aspvall. Recognizing disguised NR(1) instances of the satisfiability problem. *Journal of Algorithms* 1, pages 97–103, 1980.

[20] A. Bagchi, B. Servatius, and W. Shi. Fault tolerance in massively parallel computers. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[21] H. Baird. Anatomy of a versatile page reader. *Proceedings of the IEEE*, 80(7):1059–1065, Jul. 1992.

[22] D. H. Ballard and C. M. Brown. *Computer Vision*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[23] P. Banerjee, M.H. Jones, and J.S. Sargent. Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):91–106, Jan. 1990.

[24] A.E. Barbour. Solutions to the minimization problem of fault-tolerant logic circuits. *IEEE Trans. on Computers*, 41(4):429–443, Apr. 1992.

[25] S. Ben-Davis, B. Chor, O. Goldreich, and M. Luby. On the theory of average case complexity. *J. of Computer and Systems Sciences*, 44:193–219, 1992.

[26] A. Ben-Tal, G. Eiger, and V. Gershovitz. Global minimization by reducing the duality gap. *Mathematical Programming*, 63:193–212, 1994.

[27] R. G. Bennetts. An Improved Method for Prime C-Class Derivation in the State Reduction of Sequential Networks. *IEEE Trans. on Computers*, C-20:229–231, Feb. 1971.

[28] A. Beringer, G. Aschemann, H.H. Hoos, M. Metzger, and A. Wei. GSAT versus simulated annealing. In *Proceedings of ECAI'94*, pages 130–134, 1994.

[29] K. Berman, J. Franco, and J. Schlipf. Unique satisfiability for Horn sets can be solved in nearly linear time. *Proc. Seventh Advanced Research Institute in Discrete Applied Mathematics* (ARIDAM), New Brunswick, New Jersey, May, 1992. In *Discrete Applied Mathematics* 60:77–91, 1995.

[30] B. Bernhardsson. Explicit solutions to the n-queens problems for all n. *ACM SIGART Bulletin*, 2(2):7, Apr. 1991, ACM Press.

[31] P.A. Berstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publication, 1987.

[32] Wolfgang Bibel. Automated theorem proving. *Vieweg*, 1982.

[33] N. N. Biswas. State Minimization of Incompletely Specified Sequential Machines. *IEEE Trans. on Computers*, C-23:80–84, Jan. 1974.

[34] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Comm. of ACM*, 18(11):651–656, Nov. 1975.

[35] C.E. Blair, R.G. Jeroslow, and J.K. Lowe. Some results and experiments in programming techniques for propositional logic. *Computers and Operations Research*, 5:633–645, 1986.

[36] J.P. Blanks. Near-optimal placement using a quadratic objective function. In *Proc. of the 22th ACM/IEEE Design Automation Conference*, pages 609–615, 1985.

[37] J. Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proc. of the International Workshop on Modeling and Performance Evaluation of Computer Systems*, pages 57–65, 1976.

[38] M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver — efficient workload balancing. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994. To appear in Annals of Mathematics and Artificial Intelligence.

[39] S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, Jan 1988.

[40] F. Bonomi. On job assignment for a parallel system of processor sharing queues. *IEEE Trans. on Computers*, 39(7):858–869, July 1990.

[41] E. Boros, P.L. Hammer, and A. Kogan. Computational experiments with an exact SAT solver. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[42] E. Boros, Y. Crama, P. L. Hammer. Polynomial-time inference of all valid implications for Horn and related formulae. *Annals of Mathematics and Artificial Intelligence* **1**, pages 21–32, 1990.

[43] E. Boros, P. L. Hammer, and X. Sun. Recognition of q-Horn formulae in linear time. *Discrete Applied Mathematics* **55**, pages 1–13, 1994.

[44] E. Boros, Y. Crama, P. L. Hammer, and M. Saks. A complexity index for satisfiability problems. *SIAM Journal on Computing* **23**, pages 45–49, 1994.

[45] H.N. Brady. An approach to topological pin assignment. *IEEE Trans. on CAD*, Vol. 3, pp. 250–255, July 1984.

[46] M. Brady, J.M. Hollerbach, T.L. Johnson, T. Lozano-Perez, and M.T. Mason, editors. *Robot Motion: Planning and Control.* The MIT Press, Cambridge, 1982.

[47] M.A. Breuer. Min-cut placement. *J. of Design Automation and Fault Tolerant Computing*, 1:343–362, 1976.

[48] F. Brewer and D. Gajski. Chippe: A system for constraint driven behavioral synthesis. *IEEE Trans. on CAD*, 9(7):681–695, July 1990.

[49] A.Z. Broder, A.M. Frieze, and E. Upfal. On the satisfiability and maximum satisfiability of random 3-CNF formulas. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 322–330, 1993.

[50] M.J. Brooks and B.K.P. Horn. Shape and source from shading. In *Proc. of IJCAI'85*, pages 932–936, Aug. 1985.

[51] C.A. Brown, L.A. Finklestein, and P.W. Purdom. Backtrack Searching in the Presence of Symmetry. *6th International Conference on Algebraic Algorithms and Error Correcting Codes (AAECC)*, Lecture Notes in Computer Science, Vol. 357, pp. 99-110.

[52] C.A. Brown, L.A. Finklestein, and P.W. Purdom. Backtrack Searching in the Presence of Symmetry. *Nordic Journal of Computing*, to appear.

[53] C.A. Brown and P.W. Purdom. An average time analysis of backtracking. *SIAM J. on Computing*, 10(3):583–593, Aug. 1981.

[54] C.A. Brown and P.W. Purdom. An empirical comparison of backtracking algorithms. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-4(3):309–316, May 1982.

[55] B. Bruderlin. Constructing three dimensional geometric objects defined by constraints. In *Proceedings of 1986 ACM SIGGRAPH Workshop in Interactive Graphics*, 1986.

[56] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[57] M. Bruynooghe. Graph coloring and constraint satisfaction. Technical Report CW 44, Dept. of Computer Science, Katholieke Universiteit Leuven, Dec. 1985.

[58] M. Bruynooghe and L.M. Pereira. *Deduction Revision by Intelligent Backtracking*, pages 194–215. Ellis Horwood Limited, 1984.

[59] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

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

[61] R.E. Bryant. Binary Decision Diagrams Applied to SAT and Related Problems. Presented at 1996 DIMACS Workshop on Satisfiability Problem: Theory and Applications. March 11, 1996.

[62] H. Kleine Büning. On generalized Horn formulas and k resolution. *Theoretical Computer Science* **116**, pages 405–413, 1993.

[63] H. Kleine Büning and T. Lettmann. Aussagenlogik: Deduktion und Algorithmen. B.G. Teubner, Stuttgart, 1993. English version to appear in 1996.

[64] K. Bugrara and P. Purdom. Clause order backtracking. Technical Report 311, 1990.

[65] K.M. Bugrara and C.A. Brown. On the average case analysis of some satisfiability model problems. *Information Sciences*, 40:21–37, 1986.

[66] K.M. Bugrara, Y.F. Pan, and P. Purdom. Exponential average time for the pure literal rule. *SIAM J. on Computing*, 18:409–418, 1989.

[67] A. Bundy, editor. *Catalogue of Artificial Intelligence Tools*. Springer-Verlag, Berlin, 1984.

[68] J.L. Burns and A.R. Newton. Efficient constraint generation for hierarchical compaction. In *Proc. Int'l Conf. on Computer Design*, pages 197–200. IEEE Computer Society, Oct. 1987.

[69] M. Buro and H.K. Büning. Report on a SAT competition. Technical report, FB-17 — Mathematik/Informatik, Universitat Paderborn, Nov. 1992.

[70] M. Buro and H.K. Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49:143–151, Feb. 1993.

[71] M. Burstein and R. Pelavin. Hierarchical channel router. In *Proc. of the 20th ACM/IEEE Design Automation Conference*, pages 591–597, Jun. 1983.

[72] G. Butler. Computing in Permutation and Matrix Groups II: Backtrack Algorithm. *Math. Comp.* 39: 671–680, 1982.

[73] G. Butler and C.W.H. Lam. A General Backtracking Algorithm for the Isomorphism Problem of Combinatorial Objects. J. Symbolic Computation 1:363-381, 1985.

[74] V. Černy. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[75] S. Chakradhar, V. Agrawal, and M. Bushnell. Neural net and Boolean satisfiability model of logic circuits. *IEEE Design & Test of Computers*, pages 54–57, Oct. 1990.

[76] I. Chakravarty. A generalized line and junction labeling scheme with applications to scene analysis. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-1(2):202–205, Apr. 1979.

[77] P.K. Chan, M.D.F. Schlag, C.D. Thomborson, and V.G. Oklobdzija. Delay optimization of carry-skip adders and block carry-lookahead adders using multidimensional dynamic programming. *IEEE Trans. on Computers*, 41(8):920–930, Aug. 1992.

[78] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the 9th ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[79] V. Chandru and J. N. Hooker. Extended Horn sets in propositional logic. *J. ACM* **38**, pages 205–221, 1991.

[80] Y. Chang and B. W. Wah. Lagrangian techniques for solving a class of zero-one integer linear programs. In *Proc. Computer Software and Applications Conf.*, 1995.

[81] Y.-J. Chang and B. W. Wah. Lagrangian techniques for solving a class of zero-one integer linear programs. In *Proc. Computer Software and Applications Conference*, pages 156–161, Dallas, TX, August 1995. IEEE.

[82] M.T. Chao and J. Franco. Probabilistic analysis of two heuristics for the 3-satisfiability problem. *SIAM J. on Computing*, 15:1106–1118, 1986.

[83] M. T. Chao and J. Franco. Probabilistic analysis of a generalization of the unit-clause literal selection heuristics for the k satisfiable problem. *Information Sciences*, 51:289–314, 1990.

[84] R. Chandrasekaran. Integer programming problems for which a simple rounding type of algorithm works. In W. Pulleyblank, ed. *Progress in Combinatorial Optimization*. Academic Press Canada, Toronto, Ontario, Canada, pages 101-106, 1984.

[85] H.R. Charney and D.L. Plato. Efficient partitioning of components. In *Proc. of the 5th Annual Design Automation Workshop*, pages 16–21, 1968.

[86] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *J. of ACM*, 35:759–770, 1988.

[87] W.T. Chen and L.L. Liu. Parallel approach for theorem proving in propositional logic. *Inform. Sci.*, 41(1):61–76, 1987.

[88] R. T. Chin and C. R. Dyer. Model-based recognition in robot vision. *ACM Computing Surveys*, 18(1):67–108, Mar. 1986.

[89] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, June 1987.

[90] Y. Chu, editor. *Special Section on Chinese/Kanji Text and Data Processing. IEEE Computer*, volume 18, number 1. IEEE Computer Society Press, 1985.

[91] V. Chvátal and B. Reed. Mick gets some (the odds are on his side). In *Proceedings on the Foundations of Computer Science*, 1992.

[92] A. Cichocki and R. Unbehauen. Switched-capacitor artificial neural networks for nonlinear optimization with constraints. In *Proc. of 1990 IEEE Int'l Symposium on Circuits and Systems*, pages 2809–2812, 1990.

[93] J. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2), 1987.

[94] W. F. Clocksin and C. S. Mellish. *Programming in Prolog (2nd Edition)*. Springer-Verlag, Berlin, 1984.

[95] M.B. Clowes. On seeing things. *Artificial Intelligence*, 2:79–116, 1971.

[96] J. Cohen, editor. *Special Section on Logic Programming. Comm. of the ACM*, volume 35, number 3. 1992.

[97] J.P. Cohoon and W.D. Paris. Genetic placement. In *Digest of the Int'l Conf. on Computer-Aided Design*, pages 422–425, 1986.

[98] A. Colmerauer. Opening the Prolog III universe. *BYTE Magazine*, pages 177–182, Aug. 1987.

[99] J.S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, Boston, 1987.

[100] M. Conforti, G. Cornuéjols, A. Kapoor, K. Vušković, and M. R. Rao. Balanced Matrices. Mathematical Programming: State of the Art. J. R. Birge and K. G. Murty, eds. Braun-Brumfield, United States. Produced in association with the 15th Int'l Symposium on Mathematical Programming, University of Michigan, 1994.

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

[102] S.A. Cook. Find hard instances of the satisfiability problem. Presented at the 1996 DIMACS Workshop on Satisfiability Problem: Theory and Applications. March 11, 1996.

[103] P. R. Cooper and M. J. Swain. Parallelism and domain dependence in constraint satisfaction. Technical Report TR 255, Dept. of Computer Science, Univ. of Rochester, Dec. 1988.

[104] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.

[105] D.G. Corneil and D.G. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM J. on Computing*, 9(2):281–297, 1980.

[106] Y. Crama, P. L. Hammer, B. Jaumard, and B. Simeone. Product form parametric representation of the solutions to a quadratic boolean equation. *R.A.I.R.O. Recherche opérationnelle/Operations Research* 21:287–306, 1987.

[107] Y. Crama, P. Hansen, and B. Jaumard. The basic algorithm for pseudo-boolean programming revisited. *Discrete Applied Mathematics* 29:171–185, 1990.

[108] J.M. Crawford and L.D. Auton. Experimental results on the crossover point in satisfiability problem. In *Proc. of AAAI'93*, pages 21–27, Aug. 1993.

[109] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[110] J.M. Crawford. Solving satisfiability problems using a combination of systematic and local search. Submitted to the DIMACS Challenge II Workshop, 1994.

[111] Z. Cvetanovic. The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems. *IEEE Trans. on Computers*, C-36(4):421–432, Apr 1987.

[112] W. Dai and E.S. Kuh. Hierarchical floorplanning for building block layout. In *Digest of Int'l Conf. on Computer-Aided Design*, pages 454–457, 1986.

[113] M. Dalal, and D. W. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing letters* **44**, pages 173–180, 1992.

[114] S. R. Das, D. K. Banerji, and A. Chattopadhyay. On Control Memory Minimization in Microprogrammed Digital Computers. *IEEE Trans. on Computers*, C-22:845–848, Sept. 1973.

[115] L. S. Davis. Shape matching using relaxation techniques. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-1(1):60–72, Jan. 1979.

[116] L. S. Davis and T. C. Henderson. Hierarchical constraint processes for shape analysis. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-3(3):265–277, May 1981.

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

[118] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. of ACM*, 7:201–215, 1960.

[119] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[120] J. de Kleer. Problems with ATMS. *Artificial Intelligence*, 28:197–224, 1986.

[121] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of 11th IJCAI*, pages 290–296, 1989.

[122] J. de Kleer. Exploiting locality in a TMS. In *Proceedings of AAAI'90*, pages 264–271, 1990.

[123] R. Dechter. Learning while searching in constraint satisfaction problems. In *Proceedings of AAAI'86*, 1986.

[124] R. Dechter. A constraint-network approach to truth maintenance. Technical Report R-870009, Computer Science Dept., UCLA, Los Angeles, 1987.

[125] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41(3), 1990.

[126] R. Dechter. Directional resolution: The Davis-Putnam procedure revisited. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[127] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI'88*, 1988.

[128] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[129] J.S. Denker, editor. *Neural Networks for Computing*, volume 151 of *AIP (Snowbird, Utah) Conference Proceedings*. American Institute of Physics, New York, 1986.

[130] N. Dershowitz, J. Hsiang, N. Josephson, and D. Plaisted. Associative-commutative rewriting. In *Proceedings of IJCAI*, pages 940–944, 1983.

[131] D.N. Deutsch. A 'dogleg' channel router. In *Proc. of the 13th ACM/IEEE Design Automation Conference*, pages 425–433, Jun. 1976.

[132] D.N. Deutsch. Compacted channel routing. In *Digest Int'l Conf. on Computer-Aided Design*, pages 223–225, Nov. 1985.

[133] J. P. A. Deutsch. A short cut for certain combinational problems. In *British Joint Comput. Conference*, 1966.

[134] S. Devadas. Optimal layout via Boolean satisfiability. In *Proceedings of ICCAD'89*, pages 294–297, Nov. 1989.

[135] S. Devadas, K. Keutzer, and J. White. Estimation of power dissipation in CMOS combinational circuits using Boolean function manipulation. *IEEE Transactions on CAD*, 11(3):373–383, Mar. 1992.

[136] S. Devadas, H.T. Ma, A.R. Newton, and A.S. Vincentelli. A synthesis and optimization procedure for fully and easily testable sequential machines. *IEEE Trans. on CAD*, 8(10):1100–1107, Oct. 1989.

[137] S.K. Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, Feb. 1978.

[138] V. Dhar and A. Crocker. A problem-solver/TMS architecture for general constraint satisfaction problems. Technical report, Dept. of Information Systems, New York University, 1989.

[139] M. Dincbas, H. Simonis, and P.V. Hentenryck. Solving a cutting-stock problem in constraint logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, 1988.

[140] J. Doenhardt and T. Lengauer. Algorithm aspects of one-dimensional layout. *IEEE Trans. on CAD*, CAD-6(5):863–878, 1987.

[141] W.E. Donath. Placement and average interconnection lengths of computer logic. *IEEE Trans. on Circuits and Systems*, CAS-26(4):272–277, Apr. 1979.

[142] W.E. Donath. Wire length distribution for placements of computer logic. *IBM J. of Research and Development*, 25(3):152–155, May 1981.

[143] W.E. Donath and A.J. Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin 15*, pages 938–944, 1972.

[144] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming* **1**, pages 267–284, 1984.

[145] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[146] B. Du and J. Gu. Sequential circuit test generation by Boolean satisfiability. 1996, to appear.

[147] D.Z. Du, J. Gu, and P.M. Pardalos, editors. *The Satisfiability (SAT) Problem*. DIMACS Volume Series on Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1996.

[148] O. Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81:49–64, 1991.

[149] O. Dubois and J. Carlier. Probabilistic approach to the satisfiability problem. *Theoretical Computer Science*, 81:65–75, 1991.

[150] O. Dubois and Y. Boufkhad. Analysis of the space of solutions for random instances of the satisfiability problem. *Proceedings of the fourth International Symposium on Artificial Intelligence and Mathematics*, Ft. Lauderdale, Florida, p. 175, 1996.

[151] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. *DIMACS Series Volume: Clique, Graph Coloring, and Satisfiability — Second DIMACS Implementation Challenge*. Editors: D.S. Johnson and M.A. Trick, 26, American Mathematical Society, 1996.

[152] C. Eastman. Preliminary report on a system for general space planning. *Comm. of ACM*, 15(2):76–87, Feb. 1972.

[153] T. Eiter, P. Kilpelainen, and H. Mannila. Some remarks on renaming and satisfiability hierarchies. Summary Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[154] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notion of consistency and predicate look in a database system. *Comm. of ACM*, 19(11), Nov. 1976.

[155] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multi-commodity flow problems. *SIAM J. on Computing*, 5(4):691–703, 1976.

[156] B.-J. Falkowski and L. Schmitz. A note on the queens' problem. *Information Processing Letters*, Vol. 23:39–46, July 1986.

[157] M.Y. Fang and W.T. Chen. Vectorization of a generalized procedure for theorem proving in propositional logic on vector computers. *IEEE Trans. on Knowledge and Data Engineering*, 4(5):475–486, Oct. 1992.

[158] O. Faugeras and K. Price. Semantic description of aerial images using stochastic labeling. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-3(6):633–642, Nov. 1981.

[159] O. D. Faugeras, editor. *Fundamentals in Computer Vision*. Cambridge University Press, London, 1983.

[160] J. A. Feldman and Y. Yakimovsky. Decision theory and artificial intelligence: I. a semantics-based region analyzer. *Artificial Intelligence*, 5:349–371, 1974.

[161] J. D. Findler. *Associative Networks: Representation and Use of Knowledge by Computers.* Academic Press, New York, 1979.

[162] C.A. Floudas and P.M. Pardalos. *A Collection of Test Problems for Constrained Global Optimization Algorithms.* Springer Verlag, New York, 1990.

[163] C.A. Floudas and P.M. Pardalos, editors. *Recent Advances in Global Optimization.* Princeton University Press, New York, 1992.

[164] M.S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Pitman, London, 1987.

[165] J. Franco. On the probabilistic performance of algorithms for the satisfiability problem. *Information Processing Letters*, 23:103–106, 1986.

[166] J. Franco. Elimination of infrequent variables improves average case performance of satisfiability algorithms. *SIAM J. on Computing*, 20:1119–1127, 1991.

[167] J. Franco and Y.C. Ho. Probabilistic performance of heuristic for the satisfiability problem. *Discrete Applied Mathematics*, 22:35–51, 1988/89.

[168] J. Franco and M. Paull. Probabilistic analysis of the Davis-Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.

[169] J. Franco. 1993. On the occurrence of null clauses in random instances of satisfiability. *Discrete Applied Mathematics* **41**, pp. 203–209.

[170] J. Franco and R. Swaminathan. Toward a good algorithm for determining unsatisfiability of propositional formulas. 1996.

[171] J. Franco, and R. Swaminathan. 1996. Average case results for Satisfiability algorithms under the random-clause-width model. To appear in Annals of Mathematics and Artificial Intelligence.

[172] J. Franco, J. Goldsmith, J. Schlipf, E. Speckenmeyer, R. P. Swaminathan. 1996. An algorithm for the class of pure implicational formulas. Proceedings of the *Workshop on Satisfiability*, Siena, Italy.

[173] R.T. Frankot and R. Chellappa. A method for enforcing integrability in shape from shading algorithms. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-10(4):439–451, Jul. 1988.

[174] F. Frayman and S. Mittal. *Cossack: A Constraints-based Expert System for Configuration Tasks.* Computational Mechanics Publications, Nadel, 1987.

[175] J.W. Freeman. Improvements to the davis-putnam procedure for satisfiability. Jan. 1994.

[176] E. C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1):24–32, Jan. 1982.

[177] E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4):755–761, Oct. 1985.

[178] E.C. Freuder and M.J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of 9th IJCAI*, pages 1076–1078, 1985.

[179] A. M. Frieze, and S. Suen. 1993. Analysis of simple heuristics for random instances of 3-SAT.

[180] T. W. Fuqua. Constraint kernels: Constraints and dependencies in a geometric modeling system. Master's thesis, Dept. of Computer Science, Univ. of Utah, Aug. 1987.

[181] Z. Galil. On the complexity of regular resolution and the Davis-Putnam procedure. *Theoretical Computer Science*, pages 23–46, 1977.

[182] G. Gallo, and M. G. Scutella. Polynomially solvable satisfiability problems. *Information Processing Letters* **29**, pages 221–227, 1988.

[183] G. Gallo and G. Urbani. Algorithms for testing the satisfiability of propositional formulae. *J. of Logic Programming*, 7:45–61, 1989.

[184] G. Gallo and D. Pretolani. Hierarchies of polynomially solvable SAT problems. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[185] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems*, 8(2), Jun. 1983.

[186] M.R. Garey and D.S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. Comput.*, 4:397–411, 1975.

[187] M.R. Garey and D.S. Johnson. Two-processors scheduling with start-times and deadlines. *SIAM J. on Computing*, 6:416–426, 1977.

[188] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, San Francisco, 1979.

[189] J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings of 12th Annual Allerton Conf. Circuit System Theory*, 1974.

[190] J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of 9th IJCAI*, page 457, 1977.

[191] J. Gaschnig. *Performance Measurements and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, Dept. of Computer Science, May 1979.

[192] A.V. Gelder. A satisfiability tester for non-clausal propositional calculus. *Information and Computation*, 79(1):1–21, Oct. 1988.

[193] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, Nov. 1984.

[194] S. Geman and C.-R. Hwang. *Diffusions of Global optimization.* Brown University, 1984.

[195] M.R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers, Los Altos, California, 1987.

[196] I.P. Gent and T. Walsh. The hardest random SAT problems. In *Proceedings of KI'94*, 1994.

[197] P.C. Gilmore. A proof method for quantification theory. *IBM J. Res. Develop.*, 4:28–35, 1960.

[198] A. Ginzberg. *Algebraic Theory of Automata.* Academic Press, New York, 1968.

[199] F. Glover. Tabu search — Part I. *ORSA Journal on Computing*, 1(3):190–206, Summer 1989.

[200] A. Goerdt. 1992. A threshold for unsatisfiability. *Proceedings of 17th annual Symposium on Mathematical Foundations of Computer Science*, Prague, Czechoslovakia.

[201] A. Goldberg. Average case complexity of the satisfiability problem. In *Proc. Fourth Workshop on Automated Deduction*, pages 1–6, 1979.

[202] A. Goldberg. On the complexity of the satisfiability problem. Technical Report Courant Computer Science No. 16, New York University, 1979.

[203] A. Goldberg, P.W. Purdom, and C.A. Brown. Average time analysis of simplified Davis-Putnam procedures. *Information Processing Letters*, 15(2):72–75, 6 Sept. 1982 (Some printer errors in this paper were corrected in *Information Processing Letters*, 16:213, 1983).

[204] A. Grasselli and F. Luccio. A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks. *IEEE Trans. on Computers*, EC-14:350–359, June 1965.

[205] A. Grasselli and U. Montanari. On the Minimization of READ-ONLY Memories in Microprogrammed Digital Computers. *IEEE Trans. on Computers*, C-19:1111–1114, Nov. 1970.

[206] J. Gu. Parallel algorithms and architectures for very fast search. Ph.D thesis. Technical Report UUCS-TR-88-005, July 1988.

[207] J. Gu. How to solve Very Large-Scale Satisfiability problems. Technical Report UUCS-TR-88-032, 1988, and Technical Report UCECE-TR-90-002, 1990.

[208] J. Gu. An $\alpha\beta$-relaxation for global optimization. Technical Report UCECE-TR-91-003, Apr. 1991.

[209] J. Gu. Efficient local search for very large-scale satisfiability problem. *SIGART Bulletin*, 3(1):8–12, Jan. 1992, ACM Press.

[210] J. Gu. *On Optimizing a Search Problem.* In N. G. Bourbakis, editor, *Artificial Intelligence Methods and Applications*, Vol. 1, chapter 2, pages 63–105. World Scientific Publishers, New Jersey, Jan. 1992.

[211] J. Gu. The *UniSAT* problem models (appendix). *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(8):865, Aug. 1992.

[212] J. Gu. Local search for satisfiability (SAT) problem. *IEEE Trans. on Systems, Man, and Cybernetics*, 23(4):1108–1129, Jul. 1993, and 24(4):709, Apr. 1994.

[213] J. Gu. Global optimization for satisfiability (SAT) problem. *IEEE Trans. on Knowledge and Data Engineering*, 6(3):361–381, Jun. 1994, and 7(1):192, Feb. 1995.

[214] J. Gu. *Optimization Algorithms for the Satisfiability (SAT) Problem. In* Advances in Optimization and Approximation. *Ding-Zhu Du (ed)*, pages 72–154. Kluwer Academic Publishers, 1994.

[215] J. Gu. Parallel algorithms for satisfiability (SAT) problem. *DIMACS Volume Series on Discrete Mathematics and Theoretical Computer Science*, Vol. 22, pages 105–161, American Mathematical Society, 1995.

[216] J. Gu and R. Puri. Asynchronous circuit synthesis by Boolean satisfiability. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 14(8):961–973, Aug. 1995.

[217] J. Gu. Satisfiability Problems in VLSI Engineering. Presented at *Annual Conference of the Institute for Operations Research and Management Science*, Oct. 1995, and *DIMACS Workshop on Satisfiability Problem*, Mar. 1996. To appear in *Discrete Applied Mathematics*.

[218] J. Gu, Q.P. Gu, and D.-Z. Du. Convergence properties of optimization algorithms for the satisfiability (SAT) problem. *IEEE Trans. on Computers*, 45(2): 209–219, Feb. 1996.

[219] J. Gu and Lizhoudu. An efficient implementation of the SAT1.5 algorithm. Technical Report, USTC, Sept. 1995.

[220] J. Gu. The *Multi-SAT* algorithm. *Siena SAT Workshop* and *Workshop on Semidefinite and Interior-Point Methods*, May 1996. To appear in *Discrete Applied Mathematics*.

[221] J. Gu and Q.P. Gu. Average time complexities of several local search algorithms for the satisfiability (SAT) problem. Technical Report UCECE-TR-91-004, 1991. In *Lecture Notes in Computer Science*, Vol. 834, pp. 146-154, 1994 and to appear in *IEEE Trans. on Knowledge and Data Engineering*.

[222] J. Gu. Optimization by multispace search. Technical Report UCECE-TR-90-001, Jan. 1990.

[223] J. Gu. *Multispace Search: A New Optimization Approach (Summary)*. In *Lecture Notes in Computer Science, Vol. 834*, pages 252–260. 1994. Also see *Multispace Search for Satisfiability and NP-hard Problems*.

[224] J. Gu. *Optimization by multispace search*. Kluwer Academic Publishers, 1997.

[225] J. Gu and X. Huang. Local search with search space smoothing: A case study of the traveling salesman problem (TSP). Technical Report UCECE-TR-91-006, Aug. 1991. In *IEEE Trans. on Systems, Man, and Cybernetics*, 24(5):728–735, May 1994.

[226] J. Gu, W. Wang, and T.C. Henderson. A parallel architecture for discrete relaxation algorithm. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-9(6):816–831, Nov. 1987.

[227] J. Gu and W. Wang. A novel discrete relaxation architecture. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(8):857–865, Aug. 1992.

[228] J. Gu. *Constraint-Based Search*. Cambridge University Press, New York, 1997.

[229] Y. Gurevich. Average case completeness. *J. of Computer and Systems Sciences*, 42(3):346–398, 1991.

[230] A. Guzman. *Computer Recognition of Three-Dimensional Objects in a Visual Scene*. PhD thesis, MIT, 1968.

[231] S. Ha and E.A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Trans. on Computers*, 40(11):1225–1238, Nov. 1991.

[232] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[233] G.T. Hamachi and J.K. Ousterhout. A switchbox router with obstacle avoidance. In *Proc. of the 21st ACM/IEEE Design Automation Conference*, pages 173–179, Jun. 1984.

[234] M. Hanan, P.K. Wolff, and B.J. Agule. Some experimental results on placement techniques. *J. of Design Automation and Fault-Tolerant Computing*, 2:145–168, May 1978.

[235] P. L. Hammer and S. Rudeanu. Boolean Methods in Operations Research and Related Areas. Springer-Verlag, New York, 1968.

[236] P. Hansen and B. Jaumard. Uniquely solvable quadratic Boolean equations. *Discrete Applied Mathematics* 12:147–154, 1985.

[237] P. Hansen, B. Jaumard, and V. Mathon. Constrained Nonlinear 0-1 Programming. *ORSA Journal on Computing* 5:97–119, 1993.

[238] P. Hansen, B. Jaumard, and G. Plateau. An extension of nested satisfiability. Les Cahiers du GERAD, G-93-27, 1993.

[239] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.

[240] E. R. Hansen. *Global optimization using interval analysis*. M. Dekker, New York, 1992.

[241] R. M. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[242] R. M. Haralick and L. G. Shapiro. The consistent labeling problem: Part 1. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-1(2):173–184, Apr. 1979.

[243] F. Harary. *Graph Theory*. Addison-Wesley, Reading, 1969.

[244] W. S. Havens. A theory of schema labeling. *Computational Intelligence*, 1(3 & 4):127–139, 1985.

[245] T. Hedges, W. Dawson, and Y.E. Cho. Bitmap graph build algorithm for compaction. In *Digest Int'l Conf. on Computer-Aided Design*, pages 340–342, Sep. 1985.

[246] T. C. Henderson and L. S. Davis. Hierarchical models and analysis of shape. *Pattern Recognition*, 14(1-6):197–204, 1981.

[247] T. C. Henderson and A. Samal. Multi-constraint shape analysis. *Image and Vision Computing*, 4(2):84–96, May 1986.

[248] P. V. Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, Cambridge, 1989.

[249] P. Heusch. The Complexity of the Falsifiability Problem for Pure Implicational Formulas. *Proc. 20th Int'l Symposium on Mathematical Foundations of Computer Science (MFCS'95)*, J. Wiedermann, P. Hajek (Eds.), Prague, Czech Republic. *Lecture Notes in Computer Science* (LNCS **969**), Springer-Verlag, Berlin, pages 221–226, 1995.

[250] G.E. Hinton and J.A. Anderson, editors. *Parallel Models of Associative Memory*. Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1981.

[251] G.E. Hinton and T.J. Sejnowski. *Learning and Relearning in Boltzmann Machine. In* D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations*, volume 1, chapter 7, pages 282–317. The MIT Press, Cambridge, 1986.

[252] E. J. Hoffman, J. C. Loessi, and R. C. Moore. Constructions for the solution of the m queens problem. *Mathematics Magazine*, pages 66–72, 1969.

[253] J. H. Holland. *Adaption in Natural and Adaptive Systems*. University of Michigan Press, Ann Arbor, 1975.

[254] S.J. Hong and S. Muroga. Absolute minimization of completely specified switching functions. *IEEE Trans. on Computers*, 40(1):53–65, Jan. 1991.

[255] P. Hood and Grover. Designing real time systems in Ada. Technical Report 1123-1, SofTech, Inc., Waltham, Jan. 1986.

[256] J. Hooker and V. Vinay. An empirical study of branching rules for satisfiability. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[257] J.N. Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12:217–239, 1988.

[258] J.N. Hooker. A quantitative approach to logical inference. *Decision Support Systems*, 4:45–69, 1988.

[259] J.N. Hooker. Resolution vs. cutting plane solution of inference problems: Some computational experience. *Operations Research Letter*, 7(1):1–7, 1988.

[260] J.N. Hooker and C. Fedjki. Branch-and-cut solution of inference problems in propositional logic. Technical Report 77-88-89, GSIA, Carnegie Mellon University, Aug. 1989.

[261] A.L. Hopkins and et. al. FTMP - a highly reliable fault-tolerant multiprocessor for aircraft. In *Proceedings of the IEEE*, pages 1221–1239, Oct. 1978.

[262] B.K.P. Horn. *Obtaining Shape from Shading Information, in* The Psychology of Computer Vision, *P.H. Winston, editor*, pages 115–155. McGraw-Hill, New York, 1975.

[263] B.K.P. Horn. Understanding image intensity. *Artificial Intelligence*, 8:301–231, 1977.

[264] B.K.P. Horn and M. J. Brooks, editors. *Shape from Shading*. The MIT Press, Cambridge, 1989.

[265] B.K.P. Horn and M.J. Brooks. The variational approach to shape from shading. *Computer Vision, Graphics & Image Processing*, 33(2):174–208, 1986.

[266] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, 1978.

[267] R. Horst and H. Tuy. *Global Optimization: Deterministic Approaches*. Springer Verlag, Berlin, 1990.

[268] J. Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, pages 255–300, 1985.

[269] T.H. Hu, C.Y. Tang, and R.C.T. Lee. An average case analysis of a resolution principle algorithm in mechanical theorem proving. *Annals of Mathematics and Artificial Intelligence*, 6:235–252, 1992.

[270] X. Huang, J. Gu, and Y. Wu. A constrained approach to multi font character recognition. *IEEE Transactions on Pattern Analysis And Machine Intelligence*, 15(8):838–843, Aug. 1993.

[271] D.A. Huffman. *Impossible Objects as Nonsense Sentences. In* B. Meltzer and D. Michie, Eds., *Machine Intelligence*, pages 295–323. Edinburgh University Press, Edinburgh, Scotland, 1971.

[272] R. A. Hummel and S. W. Zucker. On the foundations of relaxation labeling processes. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-5(3):267–287, May 1983.

[273] K. Ikeuchi. Model-based interpretation of range imagery. In *Proc. of Image Understanding Workshop*, pages 321–339. DARPA, Feb. 1987.

[274] K. Ikeuchi and B.K.P. Horn. Numerical shape from shading and occluding boundaries. *Artificial Intelligence*, 17(1-3):141–184, 1981.

[275] R. Impagliazzo, L. Levin, and M. Luby. Pseudo-random number generation from one way function. In *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 12–14, 1989.

[276] B. Indurkhya, H. S. Stone, and L. Xi-Cheng. Optimal partitioning of randomly generated distributed programs. *IEEE Trans. on Software Engineering*, SE-12:483–495, Mar 1986.

[277] L. Ingber. *Adaptive Simulated Annealing (ASA)*. Lester Ingber Research, 1995.

[278] T. Ishida. Parallel rule firing in production systems. *IEEE Trans. on Knowledge and Data Engineering*, 3(1):11–17, Mar. 1991.

[279] A. Itai and J. Makowsky. On the complexity of Herbrand's theorem. Working paper 243, Department of Computer Science, Israel Institute of Technology, 1982.

[280] N. Itazaki and K. Kinoshita. Test pattern generation for circuits with tri-state modules by z-algorithm. *IEEE Trans. on CAD*, 8(12):1327–1334, Dec. 1989.

[281] K. Iwama. CNF satisfiability test by counting and polynomial average time. *SIAM J. on Computing*, pages 385–391, 1989.

[282] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. on Software Engineering*, SE-12(9):890–904, Sept. 1986.

[283] T. Jayasri and D. Basu. An Approach to Organizing Microinstructions Which Minimizes the Width of Control Store Words. *IEEE Trans. on Computers*, C-25:514–521, May 1976.

[284] R.E. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and AI*, 1:167–187, 1990.

[285] R.G. Jeroslow. Computation-oriented reductions of predicate to propositional logic. *Decision Support Systems*, 4:183–197, 1988.

[286] D.S. Johnson. Approximation Algorithms for Combinatorial Problems. *J. of Computer and Systems Sciences*, 9:256–278, 1974.

[287] D.S. Johnson. More approaches to the traveling salesman guide. *Nature*, 330:525, 1987.

[288] D.S. Johnson. *Local Optimization and the Traveling Salesman Problem. In* M.S. Paterson, editor, *Lecture Notes in Computer Science, Vol. 443: Automata, Languages and Programming*, pages 446–461. Springer-Verlag, Berlin, 1990.

[289] D.S. Johnson. Private Communications, 1993–1996.

[290] D.S. Johnson and M.A. Trick, editors. *Clique, Graph Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. DIMACS Series Vol. 26. American Mathematical Society, 1996.

[291] J.L. Johnson. A neural network approach to the 3-satisfiability problem. *J. of Parallel and Distributed Computing*, 6:435–449, 1989.

[292] M.D. Johnston. Scheduling with neural networks — the case of the Hubble Space Telescope. In *NASA Memo*, 1989.

[293] R. R. Johnson. Critical issues in computer architecture design. Private Communication, 1987-1994.

[294] R.W. Johnson and A.M. McLoughlin. Computer-Aided Discovery of a Fast Matrix-Multiplication Algorithm. *NRL Memorandum Report 3994* May 1979.

[295] W. Lewis Johnson. Letter from the editor. *SIGART Bulletin*, 2(2):1, April 1991, ACM Press.

[296] W. Lewis Johnson. Letter from the editor. *SIGART Bulletin*, 2(5):1, Oct. 1991, ACM Press.

[297] J. R. Josephson, B. Chandrasekaran, J. W. Smith Jr., and M. C. Tanner. A mechanism for forming composite explanatory hypotheses. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-17(3):445–454, May/June 1987.

[298] P.C. Jackson Jr. Heuristic search algorithms for the satisfiability problem. Submitted to the third IEEE TAI Conference, Jul. 1991.

[299] A. E. W. Jones and G. W. Forbes. An adaptive simulated annealing algorithm for global optimization over continuous variables. *Journal of Optimization Theory and Applications*, 6:1–37, 1995.

[300] Y.C. Ju, B. Rao, and R.A. Saleh. Consistency checking and optimization of macro-models. *IEEE Trans. on CAD*, 10(8):957–967, Aug. 1991.

[301] A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G.C. Resende. Computational experience with an interior point algorithm on the satisfiability problem. *Annals of Operations Research*, 25:43–58, 1990.

[302] A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G.C. Resende. A continuous approach to inductive inference. *Mathematical Programming*, 57:215–238, 1992.

[303] A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G.C. Resende. Computational experience with an interior point algorithm on the satisfiability problem. *Mathematical Sciences Research Center, AT&T Bell Laboratories*, Oct. 1989.

[304] A. Kamath, R. Motwani, K. Palem, and P. Spirakis. 1995. Tail Bounds for occupancy and the satisfiability threshold conjecture. *Random Structures and Algorithms* **7**, pp. 59–80.

[305] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, (4):373–395, 1984.

[306] G. Kedem and H. Watanabe. Graph optimization techniques for IC layout and compaction. *IEEE Trans. on CAD*, CAD-3:12–20, 1984.

[307] J. Kella. State Minimization of Incompletely Specified Sequential Machines. *IEEE Trans. on Computers*, C-19:342–348, April 1970.

[308] S. Kirkpatrick, C.D. Gelat, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[309] S. Kirkpatrick, G. Gyorgyi, N. Tishby, and L. Troyansky. The statistical mechanics of k-satisfaction. In *Proceedings of Neural Information Processing Systems*, Nov. 1993.

[310] L.M. Kirousis and C.H. Papadimitriou. The complexity of recognizing polyhedral scenes. *J. of Computer and System Sciences*, 37:14–38, 1988.

[311] L. M. Kirousis, E. Kranakis, and D. Krizanc. A better upper bound for the unsatisfiability threshold. In *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, 60, 1996.

[312] J. Kittler and J. Illingworth. Relaxation labeling algorithms — A review. *Image and Vision Computing*, pages 206–216, 1985.

[313] D. E. Knuth. Estimating the efficiency of backtracking programs. *Mathematics of Computation*, 29(129):121–136, Jan. 1975.

[314] D. Knuth. Nested satisfiability. *Acta Informatica* 28:1-6, 1990.

[315] K.L. Kodandapani and E.J. McGrath. A wirelist compare program for verifying VLSI layouts. *IEEE Design and Test of Computers*, 3(3):46–51, 1986.

[316] R. Kowalski. A proof procedure using connection graphs. *J. ACM*, 22(4):572–595, Oct. 1975.

[317] M.R. Kramer and J. van Leeuwen. *The Complexity of Wire Routing and Finding Minimum Area Layouts for Arbitrary VLSI Circuits*, volume 2, chapter VLSI Theory, pages 129–146. Jai Press Inc., Greenwich, CT, 1984.

[318] C.M. Krishna, K.G. Shin, and I.S. Bhandari. Processor tradeoffs in distributed real-time systems. *IEEE Trans. on Computers*, C-36(9):1030–1040, Sept. 1987.

[319] D. C-L. Ku. DRA1 chip implementation report. Project Report, Dept. of Computer Science, Univ. of Utah, Mar. 1986.

[320] Oliver Kullmann A systematic approach to 3-SAT-decision, yielding 3-SAT-decision in less than 1.5045^n steps. *Theoretical Computer Science*, to appear.

[321] V. Kumar. Algorithms for constraint satisfaction problems: A survey. Technical Report TR-91-28, Dept. of Computer Science, Univ. of Minnesota, 1991.

[322] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *The AI Magazine*, 13(1):32–44, 1992.

[323] V. Kumar and Y.-J Lin. A data-dependency based intelligent backtracking scheme for Prolog. *J. of Logic Programming*, 5(2), June 1988.

[324] J. D. Laderman. A noncommutative algorithm for multiplying 3×3 matrices using 23 multiplications. *Bull. Amer. Math. Soc.*, 82(1):126–128, 1976.

[325] E.D. Lagnese and D.E. Thomas. Architectural partitioning for system level synthesis of integrated circuits. *IEEE Trans. on CAD*, 10(7):847–860, July 1991.

[354] R. P. Lippmann. An introduction to computing with neural net. *IEEE ASSP Magazine*, 4(2):4–22, April 1987.
[355] Richard J. Lipton DNA Solution of Hard Computational Problems *Science*, 268:542-544.
[356] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, Carnegie-Mellon University, May 1985.
[357] D.W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, 1978.
[358] F. Luccio. Extending the Definition of Prime Compatibility Classes of States in Incompletely Specified Sequential Machine Reduction. *IEEE Trans. on Computers*, C-18:537–540, Jun. 1969.
[359] D.G. Luenberger. *Linear and Nonlinear Programming.* Addison-Wesley, Reading, 1984.
[360] N.A. Lynch. Multi-level atomicity — a new correctness criterion for database concurrency control. *ACM Trans. on Database Systems*, 8(4), Dec. 1983.
[361] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–119, 1977.
[362] F. Major, M. Turcotte, D. Gautheret, G. Lapalme, E. Fillion, and R. Cedergren. The combination of symbolic and numerical computation for three-dimensional modeling of RNA. *Sciences*, 253:1255–1260, 1991.
[363] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proc. of ACM/IEEE International Conference on CAD*, 1988.
[364] J. Malik and D. Maydan. Recovering three-dimensional shape from a single image of curved objects. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-11(6):555–566, Jun. 1989.
[365] J.A. Makowsky and A. Sharell. On average case complexity of SAT symmetric distribution. *J. of Logic and Computation*, 5(1):71–92, Feb. 1995.
[366] D. Marple and A.E. Gamal. Area-delay optimization of programmable logic arrays. Technical report, Stanford University, Sept. 1986.
[367] T.A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, Dec. 1982.
[368] T.A. Marsland and J. Schaeffer. *Computers, Chess, and Cognition.* Springer-Verlag, New York, 1990.
[369] D.W. Matula, W.G. Marble, and J.D. Isaacson. *Graph coloring algorithms. In* Graph Theory and Computing. *R.C. Read, editor*, pages 109–122. Academic Press, New York, 1972.
[370] B. Mazure, L. Sais, and E. Gregoire. Tabu search for SAT. In *Proceedings of CP'95 Workshop on Solving Really Hard Problems*, pages 127–130, 1995.
[371] D. McAllester. Truth maintenance. In *Proceedings of AAAI'90*, pages 1109–1116, 1990.
[372] J. T. McCall, J. G. Tront, F. G. Gray, R. M. Haralick, and W. M. McCormack. Parallel computer architectures and problem solving strategies for the consistent labeling problem. *IEEE Trans. On Computers*, C-34(11):973–980, Nov. 1985.
[373] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.
[374] C.R. McLean and C.R. Dyer. An analog relaxation processor. In *Proceedings of the 5th International Conference on Pattern Recognition*, pages 58–60, 1980.
[375] K. Mehlhorn. *Data Structures and Algorithms: Graph Algorithms and NP-Completeness.* Springer-Verlag, Berlin, 1984.
[376] H.-M. Méjean, H. Morel, and G. Reynaud. A variational method for analyzing unit clause search. *SIAM J. on Computing,*
[377] Z. Michalewicz. *Genetic Algorithms + Data Structure = Evolution Programs.* Springer-Verlag, 1994.
[378] M. Minoux, "The unique Horn-satisfiability problem and quadratic Boolean equations," *Annals of Mathematics and Artificial Intelligence* special issue on connections between combinatorics and logic, J. Franco, M. Dunn, W. Wheeler, (eds.) 1992.
[379] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of AAAI'90*, pages 17–24, Aug. 1990.
[380] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[326] G. Lakhani and R. Varadarajan. A wire-length minimization algorithm for circuit layout compaction. In *Proc. of ISCAS'87*, pages 276–279, May 1987.

[327] B.S. Landman and R.L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Trans. on Computers*, C-20:1469–1479, Dec. 1971.

[328] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. on Computer-Aided Design*, 11(1):4–15, Jan. 1992.

[329] C. Lassez. Constraint logic programming. *BYTE Magazine*, pages 171–176, Aug. 1987.

[330] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelii. Algorithms for Synthesis of Hazard-free Asynchronous Circuits. In *Proc. of 28th DAC*, pages 302–308, 1991.

[331] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the State Assignment Problem for Signal Transition Graphs. In *Proc. of 29th DAC*, pages 568–572, 1992.

[332] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons, New York, 1985.

[333] E.L. Lawler and D.E. Wood. Brance-and-bound methods: a survey. *Operations Research*, 14(4):699–719, Jul.–Aug. 1966.

[334] C. Y. Lee. Representation of Switching Circuits by Binary Decision Programs. *Bell Systems Technical Journal*, 38:985–999, July 1959.

[335] C. Lee. An algorithm for path connections and its applications. *IEEE Trans. on Electronic Computers*, VEC-10:346–365, Sept. 1961.

[336] E.A. Lee. Consistency in dataflow graphs. *IEEE Trans. on Parallel and Distributed Systems*, 2(2):223–235, Apr. 1991.

[337] R.C.T. Lee. Private Communications, 1992-1993.

[338] J.P. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *ACM Performance Evaluation Review*, 14(1), May 1986. Special Issue.

[339] H. R. Lewis. Renaming a set of clauses as a Horn set. *Journal of the Association for Computing Machinery* **25**, pages 134–135, 1978.

[340] G. J. Li and B. W. Wah. How to Cope with Anomalies in Parallel Approximate Branch-and-Bound Algorithms. in *Proc. National Conf. on Artificial Intelligence*, AAAI, pages 212-215, Aug. 1984.

[341] G.-J. Li. *Parallel Processing of Combinatorial Search Problems*. PhD thesis, School of Electrical Engineering, Purdue University, West Lafayette, Dec. 1985.

[342] G. J. Li and B. W. Wah. How good are parallel and ordered depth-first searches? In *Proc. Int'l Conf. on Parallel Processing*, pages 992–999, University Park, PA, August 1986. Pennsylvania State Univ. Press.

[343] G. J. Li and B. W. Wah. Computational efficiency of combinatorial OR-tree searches. *Trans. on Software Engineering*, 16(1):13–31, January 1990.

[344] G.-J. Li and B. W. Wah. Parallel iterative refining A*: An efficient search scheme for solving combinatorial optimization problems. In *Proc. Int'l Conf. on Parallel Processing*, pages 608–615, University Park, PA, August 1991. Pennsylvania State Univ. Press.

[345] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[346] Y.-Z. Liao and C.K. Wong. An algorithm to compact a VLSI symbolic layout with mixed constraints. *IEEE Trans. on CAD*, CAD-2(2):62–69, 1983.

[347] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing* 11:329–343, 1982.

[348] K.J. Lieberherr and E. Specker. Complexity of partial satisfaction. *J. of ACM*, 28:411–421, 1981.

[349] F.C.H. Lin and R.M. Keller. The gradient model load balancing method. *IEEE Trans. on Software Engineering*, SE-13(1):32–38, Jan. 1987.

[350] K. J. Lin and C. S. Lin. Automatic Synthesis of Asynchronous Circuits. In *Proc. of 28th DAC*, pages 296–301, 1991.

[351] S. Lin. Computer solutions of the traveling salesman problem. *Bell Sys. Tech. Journal*, 44(10):2245–2269, Dec. 1965.

[352] W. M. Lin and V. K. P. Kumar. Parallel architectures for discrete relaxation algorithm. AAAI Workshop on Parallel Algorithms for AI, Detroit, 20 Aug. 1989.

[353] Y.-J. Lin. *A Parallel Implementation of Logic Programs*. PhD thesis, The Univ. of Texas at Austin, Dept. of Computer Science, May 1988.

[381] D.P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Pitman, London, 1990.

[382] D.P. Miranker and B.J. Lofaso. The organization and performance of a treat-based production system compiler. *IEEE Trans. on Knowledge and Data Engineering*, 3(1):3–10, Mar. 1991.

[383] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of AAAI'92*, pages 459–465, Jul. 1992.

[384] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[385] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:117–133, 1983.

[386] B. Monien, E. Speckenmeyer, and O. Vornberger. Superlinear Speedup for Parallel Backtracking.

[387] S. Mori, C.Y. Suen, and K. Yamamoto. Historical review of OCR research and development. *Proceedings of the IEEE*, 80(7):1029–1058, Jul. 1992.

[388] P. Morris. The breakout method for escaping from local minima. In *Proc. of the 11th National Conf. on Artificial Intelligence*, pages 40–45, Washington, DC, 1993.

[389] B.A. Murtagh and M.A. Saunders. MINOS 5.0 user's guide. Technical Report SOL 83-20, Dept. of Operations Research, Stanford University, Stanford, CA, 1983.

[390] B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[391] B.A. Nadel and J. Lin. Automobile transmission design as a constraint satisfaction problem: A collaborative research project with ford motor Co. Technical report, Wayne State University, 1990.

[392] D. Navinchandra. *Exploration and Innovation in Design.* Springer Verlag, Sadeh, 1990.

[393] D. Navinchandra and D.H. Marks. Layout planning as a consistent labeling optimization problem. In *Proceedings of 4th International Symposium on Robotics and AI in Construction*, 1987.

[394] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[395] L.M. Ni, C. Xu, and T.B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Trans. on Software Engineering*, SE-11(10):1153–1161, Oct. 1985.

[396] A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms.* Academic Press, New York, 1975.

[397] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, California, 1980.

[398] M. Nishizawa. Partitioning of logic units. *Fujitsu Scientific and Technical Journal*, 7(2):1–13, Jun. 1971.

[399] D. Pager. On the efficient algorithm for graph isomorphism. *J. of ACM*, 17(4):708–714, Jan. 1970.

[400] C.H. Papadimitriou and K. Steiglitz. On the complexity of local search for the traveling salesman problem. *SIAM J. on Computing*, 6(1):76–83, 1977.

[401] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, 1982.

[402] C.H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual Symposium of the Foundations of Computer Science*, pages 163–169, 1991.

[403] E. Koutsoupias, and C. H. Papadimitriou. 1992. On the greedy algorithm for satisfiability. *Information Processing Letters* **43**, pp. 53–55.

[404] C.H. Papadimitriou, Private Communications, 1996.

[405] P.M. Pardalos and J.B. Rosen. *Constrained Global Optimization: Algorithms and Applications.* Springer Verlag, New York, 1987.

[406] P. M. Pardalos. *Complexity in numerical optimization.* World Scientific, Singapore and River Edge, N.J., 1993.

[407] A.M. Patel, N.L. Soong, and R.K. Korn. Hierarchical VLSI routing — an approximate routing procedure. *IEEE Trans. on CAD*, CAD-4(2):121–126, Apr. 1985.

[408] W. Patterson. *Mathematical Cryptology.* Rowman & Littlefield, New Jersey, 1987.

[409] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Trans. on CAD*, 8(6):661–679, June 1989.

[410] J. Pearl. *Heuristics.* Addison-Wesley, Reading, 1984.

[411] D. Pehoushek. SAT problem instances and algorithms. Private Communications, Stanford University, 1992.

[412] M.A. Penna. A shape form shading analysis for a single perspective image of a polyhedron. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-11(6):545–554, Jun. 1989.

[413] L.M. Pereira and A. Porto. *Selective Backtracking*, pages 107–114. Academic Press, 1982.

[414] C. Petrie. Revised dependency-directed backtracking for default reasoning. In *Proceedings of AAAI'87*, pages 167–172, 1987.

[415] J. Peysakh, *A fast algorithm to convert Boolean expressions into CNF*, IBM Computer Science RC 12913, No. 57971, T.J. Watson, New York, 1987.

[416] T. Pitassi and A. Urquhart. The complexity of the hajós calculus. *SIAM J. on Disc. Math.*, 8(3):464–483, 1995.

[417] J. Plotkin, J. Rosenthal, and J. Franco. Correction to probabilistic analysis of the Davis-Putnam Procedure for solving the Satisfiability problem. *Discrete Applied Mathematics*, 17:295–299, 1987.

[418] D.P. La Potin and S.W. Director. Mason: a global floorplanning approach for VLSI design. *IEEE Trans. on CAD*, CAD-5(4):477–489, Oct. 1986.

[419] D. K. Pradhan, editor. *Fault-Tolerant Computing: Theory and Practice*. Prentice-Hall, Englewood Cliffs, 1986.

[420] D. Prawitz. An improved proof procedure. *Theoria*, 26(2):102–139, 1960.

[421] D. Pretolani. A linear time algorithm for unique Horn satisfiability. *Information Processing Letters* 48:61–66, 1993.

[422] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. *DIMACS Series Volume: Clique, Graph Coloring, and Satisfiability — Second DIMACS Implementation Challenge*. Editors: D.S. Johnson and M.A. Trick, American Mathematical Society, 1996.

[423] P. Purdom. Survey of average time SAT performance. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[424] P.W. Purdom. Tree size by partial backtracking. *SIAM J. on Computing*, 7(4):481–491, Nov. 1978.

[425] P.W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[426] P.W. Purdom. Solving satisfiability with less searching. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 6:510–513, 1984.

[427] P.W. Purdom. A survey of average time analyses of satisfiability algorithms. *J. of Information Processing*, 13(4):449–455, 1990.

[428] P.W. Purdom. Average time for the full pure literal rule. *Information Sciences*, 1994 78:269–291.

[429] P.W. Purdom and C.A. Brown. An analysis of backtracking with search rearrangement. *SIAM J. on Computing*, 12(4):717–733, Nov. 1983.

[430] P.W. Purdom and C.A. Brown. The pure literal rule and polynomial average time. *SIAM J. on Computing*, 14:943–953, 1985.

[431] P.W. Purdom and C.A. Brown. Polynomial-average-time satisfiability problems. *Information Science*, 41:23–42, 1987.

[432] P.W. Purdom, C.A. Brown, and E.L. Robertson. Backtracking with multi-level dynamic search rearrangement. *Acta Informatica*, 15:99–113, 1981.

[433] P.W. Purdom and G.N. Haven. Backtracking and probing. Technical Report No. 387, Dept. of Computer Science, Indiana University, Aug. 1993.

[434] P.W. Purdom and G.N. Haven. Probe Order Backtracking. Accepted in *SIAM J. on Computing*, to appear in 1997.

[435] R. Puri and J. Gu. An efficient algorithm to search for minimal closed covers in sequential machines. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 12(6):737–745, Jun. 1993.

[436] R. Puri and J. Gu. An efficient algorithm for computer microword length minimization. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 12(10):1449–1457, Oct. 1993.

[437] R. Puri and J. Gu. Area efficient synthesis of asynchronous interface circuits. In *Proc. of IEEE International Conference on Computer Design*, 1994.

[438] R. Puri and J. Gu. A divide-and-conquer approach for asynchronous circuit design. In *Proceedings of the 7th IEEE International Symposium on High-Level Synthesis*, pages 118–125, May 1994.

[439] R. Puri and J. Gu. A BDD SAT solver for satisfiability testing. *Annals of Mathematics and Artificial Intelligence*, Vol. 17, pp. 315–337, 1996.

[440] C. D. V. P. Rao and N. N. Biswas. On the Minimization of Wordwidth in the Control Memory of a Microprogrammed Digital Computer. *IEEE Trans. on Computers*, C-32(9):863–868, Sept. 1983.

[441] C. V. S. Rao and N. N. Biswas. Minimization of Incompletely Specified Sequential Machines. *IEEE Trans. on Computers*, C-24:1089–1100, Nov. 1975.

[442] R.C. Read and D.G. Corneil. The graph isomorphism disease. *J. of Graph Theory*, 1:339–363, 1977.

[443] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. In *Proc. of ICALP'86*, pages 314–323. Springer LNCS 226, 1986.

[444] J. Reed, A. Sangiovanni-Vincentelli, and M. Santomauro. A new symbolic channel router: YACR2. *IEEE Trans. on CAD*, CAD-4(3):208–219, July 1985.

[445] M. Reichling. A simplified solution of the n queens' problem. *Information Processing Letters*, Vol. 25:253–255, June 1987.

[446] M. Resende and T. Feo. A GRASP for MAX-SAT. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[447] R.L. Rivest. *Cryptography. In* Handbook of Theoretical Computer Science. *J.V. Leeuwen, editor*, chapter 13, pages 719–756. The MIT Press, Cambridge, 1990.

[448] R.L. Rivest and C.M. Fiduccia. A 'greedy' channel router. In *Proc. of the 19th ACM/IEEE Design Automation Conference*, pages 418–424, Jun. 1982.

[449] J.A. Robinson. A machine–oriented logic based on the resolution principle. *Journal of the ACM*, pages 23–41, 1965.

[450] A. Rosenfeld. Computer vision: Basic principles. *Proceedings of the IEEE*, 76(8):863–868, Aug. 1988.

[451] A. Rosenfeld, R. A. Hummel, and S. W. Zucker. Scene labeling by relaxation operations. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-6(6):420–433, June 1976.

[452] W. Rosiers and M. Bruynooghe. Empirical study of some constraint satisfaction problems. In *Proceedings of AIMSA'86*, North Holland, Sept. 1986.

[453] A.E. Ruehli, P.K. Wolff, and G. Goertzel. Analytical power/timing optimization technique for digital system. In *Proc. of the 14th ACM/IEEE Design Automation Conference*, pages 142–146, Jun. 1977.

[454] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, 1995.

[455] R.L. Russo. On the tradeoff between logic performance and circuit-to-pin ratio for LSI. *IEEE Trans. on Computers*, C-21:147–153, 1972.

[456] R.L. Russo, P.H. Oden, and P.K. Wolff. A heuristic procedure for the partitioning and mapping of computer logic graphs. *IEEE Trans. on Computers*, C-20:1455–1462, Dec. 1971.

[457] Y.G. Saab and V. B. Rao. Combinatorial optimization by stochastic evolution. *IEEE Transactions on CAD*, CAD-10(4):525–535, Apr. 1991.

[458] A. Saldanha, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. A Framework for Satisfying Input and Output Encoding Constraints. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 170–175, 1991.

[459] A. Samal. *Parallel Split-Level Relaxation.* PhD thesis, Dept. of Computer Science, Univ. of Utah, Aug. 1988.

[460] A. Samal and T. C. Henderson. Performance of arc consistency algorithms on the Cray. Technical Report UUCS-TR-87-017, Dept. of Computer Science, Univ. of Utah, July 23 1987.

[461] A. Samal and T.C. Henderson. Parallel consistent labeling algorithms. *International Journal of Parallel Programming*, 1988.

[462] T.J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 216–226, 1978.

[463] W.L. Schiele. Improved compaction by minimized length of wires. In *Proc. of the 20th ACM/IEEE Design Automation Conference*, pages 121–127, 1983.

[464] I. Schiermeyer. Solving 3-Satisfiability in less than $O(1.579^n)$ steps. *Lecture Notes in Computer Science* **702**, pages 379–394, 1993.

[465] I. Schiermeyer. Pure literal lookahead: an $O(1.497^n)$ 3-Satisfiability algorithm. In *Proc. of the Workshop on Satisfiability*, Universitá delgi Studi, Siena, Italy, pages 63–72, 1996.

[466] J. S. Schlipf, F. Annexstein, J. Franco, and R. Swaminathan. On finding solutions for extended Horn formulas. *Information Processing Letters* **54**, pages 133–137, 1995.

[467] B.M. Schwartzschild. Statistical mechanics algorithm for Monte Carlo optimization. *Physics Today*, 35:17–19, 1982.

[468] P. Schwarz and A. Spector. Synchronizing shared abstract types. *ACM Trans. on Computer Systems*, Aug. 1984.

[469] M. G. Scutella. A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability. *Journal of Logic Programming* **8**, pages 265–273, 1990.

[470] C. Sechen and A.L. Sangiovanni-Vincentelli. The TIMBERWOLF placement and routing package. In *Proc. of the 1984 Custom Integrated Circuit Conf.*, May 1984.

[471] B. Selman. Private Communications, Aug. 1992.

[472] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI'92*, pages 440–446, Jul. 1992.

[473] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proc. of the 13th Int'l Joint Conf. on Artificial Intelligence*, pages 290–295, 1993.

[474] B. Selman, H.A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. *DIMACS Series Volume: Clique, Graph Coloring, and Satisfiability — Second DIMACS Implementation Challenge*. American Mathematical Society, pp. 290–295, 1996.

[475] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. of the 12th National Conf. on Artificial Intelligence*, pages 337–343, Seattle, July 1994.

[476] B. Selman, Private Communication, 1995.

[477] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS : A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Dept. of EECS, Univ. of California, Berkeley, May, 1992.

[478] M.J. Shensa. A computational structure for the propositional calculus. In *Proceedings of IJCAI*, pages 384–388, 1989.

[479] R.C.-J. Shi, A. Vannelli, and J. Vlach. An improvement on Karmarkar's algorithm for integer programming. COAL Bulletin of the Mathematical Programming Society, 21:23–28, 1992.

[480] H. Shin, A.L. Sangiovanni-Vincentelli, and C.H. Sequin. Two-dimensional compaction by 'zero refining'. In *Proc. of the 23rd ACM/IEEE Design Automation Conference*, pages 115–122, Jun. 1986.

[481] K.G. Shin and M.-S. Chen. On the number of acceptable task assignments in distributed computing systems. *IEEE Trans. on Computers*, 39(1):99–110, Jan. 1990.

[482] P. Siegel. *Representation et Utilization de la Connaissances en Calcul Propositionel.* PhD thesis, University Aix-Marseille II, 1987.

[483] J.C. Simon. Off-line cursive word recognition. *Proceedings of the IEEE*, 80(7):1150–1161, Jul. 1992.

[484] R. Sosič and J. Gu. Quick n-queen search on VAX and Bobcat machines. CS547 AI Course Project, Winter Quarter, Feb. 1988.

[485] R. Sosič and J. Gu. How to search for million queens. Technical Report UUCS-TR-88-008, Dept. of Computer Science, Univ. of Utah, Feb. 1988.

[486] R. Sosič and J. Gu. A polynomial time algorithm for the n-queens problem. *SIGART Bulletin*, 1(3):7–11, Aug. 1990, ACM Press.

[487] R. Sosič and J. Gu. Fast search algorithms for the n-queens problem. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-21(6):1572–1576, Nov./Dec. 1991.

[488] R. Sosič and J. Gu. 3,000,000 queens in less than one minute. *SIGART Bulletin*, 2(2):22–24, Apr. 1991, ACM Press.

[489] R. Sosič and J. Gu. A parallel local search algorithm for satisfiability (SAT) problem. *Submitted for publication*, 1993.

[490] R. Sosič and J. Gu. A parallel local search algorithm for the n-queen problem. *Submitted for publication*, 1993.

[491] R. Sosič and J. Gu. Efficient local search with conflict minimization. *IEEE Trans. on Knowledge and Data Engineering*, 6(5):661–668, Oct. 1994.

[492] R. Sosič, J. Gu, and R. Johnson. The Unison algorithm: Fast evaluation of Boolean expressions. Accepted for publication in *Communications of the ACM* in 1992. *ACM Transactions on Design Automation of Electronic Systems*, (1)4:456–477, Oct. 1996.

[493] R. Sosič, J. Gu, and R. Johnson. A universal Boolean evaluator. *IEEE Trans. on Computers*, accepted for publication in 1992.

[494] J. Soukup. Circuit layout. In *Proceedings of the IEEE*, pages 1281–1304, Oct. 1981.

[495] E. Speckenmeyer. Personal Communication. 1996.

[496] R. Stallman and G.J. Sussman. Forward reasoning and dependency directed backtracking. *Artificial Intelligence*, 9(2):135–196, 1977.

[497] J. Stankovic, K. Ramamritham, and S. Cheng. Evaluation of a bidding algorithm for hard real-time distributed systems. *IEEE Trans. on Computers*, C-34(12):1130–1143, Dec. 1985.

[498] J.A. Stankovic. Real-time computing systems: The next generation. Manuscript. Feb. 18, 1988.

[499] L. Sterling and E. Shapiro. *The Art of Prolog, Advanced Programming Techniques*. The MIT Press, Cambridge, Massachusetts, 1986.

[500] H.S. Stone and P. Sipala. The average complexity of depth-first search with backtracking and cutoff. *IBM J. Res. Develop.*, 30(3):242–258, May 1986.

[501] H.S. Stone and J.M. Stone. Efficient search techniques – an empirical study of the n-queens problem. *IBM J. Res. Develop.*, 31(4):464–474, July 1987.

[502] C.Y. Suen, M. Berthod, and S. Mori. Automatic recognition of hand printed characters – the state of the art. *Proceedings of the IEEE*, 68(4):469–487, Apr. 1980.

[503] S. Sutanthavibul, E. Shragowitz, and J.B. Rosen. An analytical approach to floorplan design and optimization. *IEEE Trans. on CAD*, 10(6):761–769, June 1991.

[504] R. P. Swaminathan and D. K. Wagner. The arborescence–realization problem. *Discrete Applied Mathematics* **59**, pages 267–283, 1995.

[505] M. Takashima, T. Mitsuhashi, T.Chiba, and K. Yoshida. Programs for verifying circuit connectivity of MOS/VLSI artwork. In *Proc. of the 19th ACM/IEEE Design Automation Conference*, pages 544–550, 1982.

[506] A. Thayse and M. Davio, *Boolean differential calculus and its application to switching theory*, IEEE Trans. on Computers **22** (1973), 409–420.

[507] H. Tokuda, J. Wendorf, and H. Wang. Implementation of a time driven scheduler for real-time operating systems. In *Proc. of the Real-Time Systems Symposium*, Dec. 1987.

[508] A. Törn and A. Žilinskas. *Global Optimization*. Springer-Verlag, 1989.

[509] P. P. Trabado, A. Lloris-Ruiz, and J. Ortega-Lopera. Solution of Switching Equations Based on a Tabular Algebra. *IEEE Trans. on Computers*, C-42:591–596, May 1993.

[510] K. Truemper. Alpha-balanced graphs and matrices and GF(3)-representability of matroids. *Journal of Combinatorial Theory B* **32**, pages 112–139, 1982.

[511] K. Truemper. Monotone Decomposition of Matrices. Technical Report UTDCS-1-94, University of Texas at Dallas. 1994.

[512] K. Truemper. Polynomial algorithms for problems over d-systems. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[513] K. Truemper. In preparation, expected 1997. Effective Logic Computation.

[514] G.S. Tseitin. *On the Complexity of Derivations in Propositional Calculus. In* Structures in Constructive Mathematics and Mathematical Logic, Part II, *A.O. Slisenko, ed.*, pages 115–125. 1968.

[515] K. J. Turner. *Computer Perception of Curved Objects Using a Television Camera*. PhD thesis, Univ. Edinburgh, 1974.

[516] J.D. Tygar and R. Ellickson. Efficient netlist comparison using hierarchy and randomization. In *Proc. of the 22nd ACM/IEEE Design Automation Conference*, pages 702–708, 1985.

[517] J. E. Tyler. Parallel computer architectures and problem solving strategies for the consistent labeling problem. Master's thesis, Dept. of Elec. Engg., Virginia Polytech. Inst. State Univ., Blacksburg, Virginia, Oct. 1983.

[518] J.D. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, 1982.

[519] J.R. Ullman. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.

[520] J.R. Ullman. A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 20(2):141–147, Feb. 1977.

[521] S.D. Urban and L.M.L. Delcambre. Constraint analysis: A design process for specifying operations on objects. *IEEE Trans. on Knowledge and Data Engineering*, 2(4):391–400, Dec. 1990.

[522] A. Urquhart. Hard examples for resolution. *J. of ACM*, 34:209–219, 1987.

[523] A. Urquhart. The complexity of gentzen systems for propositional logic. *Theoretical Computer Science*, 66:87–97, 1989.

[524] A. Urquhart. The relative complexity of resolution and cut-free gentzen systems. *Annals of Mathematics and Artificial Intelligence*, 6:157–168, 1992.

[525] A. Urquhart. The complexity of propositional proofs. *The Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

[526] M. Valtorta. Response to "Explicit Solutions to the N-Queens Problem for all N." *SIGART Bulletin*, 2(4):4, Aug. 1991, ACM Press.

[527] M. van der Woude and X. Timermans. Compaction of hierarchical cells with minimum and maximum compaction constraints. In *Proc. Int'l Symposium on Circuits and Systems*, pages 1018–1021, 1983.

[528] P. Vanbekbergen, G. Goossens, F. Catthoor, and H. De Man. Optimized Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications. *IEEE Trans. on CAD*, 11(11):1426–1438, Nov. 1992.

[529] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A Generalized State Assignment Theory for Transformations on Signal Transition Graphs. In *Proc. of ICCAD*, pages 112–117, 1992.

[530] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A Generalized State Assignment Theory for Transformations on Signal Transition Graphs. *J. of VLSI Signal processing*, 1993. In Press.

[531] B. W. Wah, G. J. Li, and C. F. Yu. Multiprocessing of combinatorial search problems. *IEEE Computer*, 18(6):93–108, June 1985. Also in *Tutorial: Computers for Artificial Intelligence Applications*, ed. B. W. Wah, IEEE Computer Society, 1986, pp. 173-188.

[532] B. Wah and G.J. Li. *Computers for Artificial Intelligence Applications*. IEEE Computer Society Press, Washington D. C., 1986.

[533] B.W. Wah, editor. *New Computers for Artificial Intelligence Processing. IEEE Computer*, volume 20, number 1. IEEE Computer Society Press, 1987.

[534] B.W. Wah, M.B. Lowrie, and G.-J. Li. Computers for symbolic processing. *Proceedings of the IEEE*, 77(4):509–540, Apr. 1989.

[535] B. W. Wah, G. J. Li, and C. F. Yu. Multiprocessing of combinatorial search problems. In L. Kanal, V. Kumar, and P. S. Gopalakrishnan, editors, *Parallel Algorithms for Machine Intelligence and Pattern Recognition*, pages 102–145. Springer-Verlag, New York, NY, 1990.

[536] B. W. Wah and L.-C. Chu. Combinatorial search algorithms with meta-control: Modeling and implementations. *Int'l J. of Artificial Intelligence Tools*, 1(3):369–397, September 1992.

[537] B. W. Wah and Y. Shang. A comparison of a class of IDA* search algorithms. *Int'l J. of Artificial Intelligence Tools*, 3(4):493–523, October 1995.

[538] B. W. Wah and Y. Shang. A discrete lagrangian-based global-search method for solving satisfiability problems. In Ding-Zhu Du, Jun Gu, and Panos Pardalos, editors, *Proc. of the DIMACS Workshop on Satisfiability Problem: Theory and Applications*. American Mathematical Society, March 1996.

[539] B. W. Wah and C. F. Yu. Stochastic modeling of branch-and-bound algorithms with best-first search. *Trans. on Software Engineering*, SE-11(9):922–934, September 1985.

[540] B. Wah and Y.-J. Chang. Trace-Based Methods for Solving Nonlinear Global Optimization and Satisfiability Problems. *J. of Global Optimization*. Submitted in July 1995 and accepted to appear in 1996.

[541] D. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271, MIT, Nov. 1972.

[542] D. Waltz. *Understanding Line Drawings of Scenes with Shadows. In* P. H. Winston, *The Psychology of Computer Vision*, chapter 2, pages 19–92. McGraw-Hill Book Company, New York, 1975.

[543] W. Wang, J. Gu, and T. C. Henderson. A pipelined architecture for parallel image relaxation operations. *IEEE Trans. on Circuits and Systems*, CAS-34(11):1375–1384, Nov. 1987.

[544] W. Wang, J. Gu, and K.F. Smith. A regular layout for concurrent discrete relaxation computation. In *Proceedings of 1987 IEEE International CAS Symposium*, pages 927–933, May 1987.

[545] W. Wang and C.K. Rushforth. An adaptive local search algorithm for channel assignment problem. *IEEE Trans. on Vehicular Technology*, Vol. 45, No. 3, pp. 459–466, Aug. 1996.

[546] W. Wang and C.K. Rushforth. Structured partitioning for channel assignment problem. *IEEE Trans. on Vehicular Technology*, 1996.

[547] M. B. Wells. *Elements of Combinatorial Computing.* Pergamon Press, Oxford, 1971.

[548] H.P. Williams. Linear and integer programming applied to the propositional calculus. *Systems Research and Information Sciences*, 2:81–100, 1987.

[549] P.H. Winston. *Artificial Intelligence.* Addison-Wesley, Reading, 1984.

[550] E.E. Witte, R.D. Chamberlain, and M.A. Franklin. Parallel simulated annealing using speculative computation. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):483–494, Oct. 1991.

[551] L. Wos, G. A. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy. *Journal of the ACM*, 12:536–541, 1965.

[552] L.C. Wu and C.Y. Tang. Solving the satisfiability problem by using randomized approach. *Information Processing Letters*, 41:187–190, 1992.

[553] T. Y. Young and K. S. Fu, editors. *Handbook of Pattern Recognition and Image Processing.* Academic Press, Orlando, 1986.

[554] C. F. Yu. *Efficient Combinatorial Search Algorithm.* PhD thesis, School of Electrical Engineering, Purdue University, West Lafayette, IN, Dec. 1986.

[555] C. F. Yu and B. W. Wah. Stochastic modeling of branch-and-bound algorithms with best-first search. *IEEE Trans. on Software Engineering*, 11(9):922–934, Sept 1985.

[556] C.F. Yu and B.W. Wah. Efficient branch-and-bound algorithms on a two-level memory system. *IEEE Trans. on Software Engineering*, 14(9):1342–1356, Sept 1988.

[557] R. Zabih and D. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of AAAI'88*, pages 155–160, 1988.

[558] V.N. Zemlyachenko, N.M. Korneeko, and R.I. Tyshkevich. Graph isomorphism problem. *J. of Soviet Mathematics*, 29:1426–1481, 1985.

[559] S. Zhang and A. G. Constantinides. Lagrange programming neural networks. *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, 39(7):441–452, 1992.

[560] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Trans. on Software Engineering*, SE-14(9):1327–1341, Sept. 1988.

[561] S. W. Zucker, R. A. Hummel, and A. Rosenfeld. An application of relaxation labeling to line and curve enhancement. *IEEE Trans. on Computers*, C-26:394–403,922–929, 1977.

DEPT. OF ELECTRICAL AND COMPUTER ENGINEERING, UNIV. OF CALGARY, CALGARY, ALBERTA T2N 1N4, CANADA
E-mail address: gu@enel.ucalgary.ca, jungu@dimacs.rutgers.edu

DEPT. OF COMPUTER SCIENCE, INDIANA UNIVERSITY, BLOOMINGTON, INDIANA 47405, U.S.A.
E-mail address: pwp@cs.indiana.edu

COMPUTER SCIENCE DEPARTMENT, UNIVERSITY OF CINCINNATI, CINCINNATI, OH 45221-0008, U.S.A.
E-mail address: John.Franco@UC.Edu

DEPT. OF ELECTRICAL AND COMPUTER ENGINEERING, UNIV. OF ILLINOIS AT URBANA-CHAMPAIGN, URBANA, IL 61801-3082, U.S.A.
E-mail address: b-wah@uiuc.edu