Algorithms for the Satisfiability (SAT) Problem: A Survey[1]

Jun Gu,[2] Paul W. Purdom,[3] John Franco,[4] and Benjamin W. Wah[5]

July 1993
Rev. 1994, 1995, 1996

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 the SAT problem as a discrete, constrained decision problem. Recent years, many optimization methods, parallel algorithms, and practical techniques have been developed for the SAT problem. In this survey, we present a general framework, an algorithm space, that integrates the existing SAT algorithms into a unified perspective. We will describe sequential and parallel SAT algorithms including variable setting, resolution, local search, global optimization, mathematical programming, and practical SAT algorithms for the SAT problem. This survey will provide a set of practical applications of the SAT problem.

Keywords: Constraint satisfaction problem (CSP), search, optimization, NP-complete problems, satisfiability (SAT) problem.

[1]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 Grants OGP0046423, NSERC Strategic Grants MEF0045793 and STR0167029, NSF Grant CCR-9203941, ONR Grant N00014-94-1-0382, NSF Grant MIP-92-18715, and 1996 NSF/DIMACS workshop on the satisfiability (SAT) problems.

[2]Dept. of Electrical and Computer Engineering, Univ. of Calgary, Calgary, Alberta T2N 1N4, Canada. gu@enel.ucalgary.ca
[3]Dept. of Computer Science, Indiana University, Bloomington, Indiana 47405, U.S.A. pwp@cs.indiana.edu
[4]Computer Science Department, University of Cincinnati, Cincinnati, OH 45221-0008, U.S.A. John.Franco@UC.Edu
[5]Coordinated Science Lab, Univ. of Illinois at Urbana-Champaign, Urbana, IL 61801-3082, U.S.A. b-wah@uiuc.edu

Preliminary Release Note

Due to time limit, we are not able to finish this survey paper before the 1996 DIMACS SAT workshop. We are releasing this preliminary version. The final, complete version of this survey will incorporate recent results to be presented in the DIMACS workshop and will be finished after the workshop.

If you have any comment on this survey, we appreciate that you will send it to the authors.

Contents

1 Introduction

The satisfiability (SAT) problem has three components [89, 170]:

- 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* ($\vee$) connectors.

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 form (*CNF*) formula satisfiable:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m, \tag{1}$$

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

The SAT problem is a *core* of a large family of computationally intractable NP-complete problems [89, 170]. 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. However, clever algorithms can rapidly solve many SAT problems of practical interest. There has been great interest in designing efficient algorithms to solve most SAT problems.

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 12). Therefore, methods to solve the SAT problem play an important role in the development of efficient computing systems.

Traditional methods treat the SAT problem 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 the existing SAT algorithms into a unified perspective. We will describe sequential and parallel SAT algorithms and give performance comparisons among major SAT algorithms including variable setting, resolution, local search, global optimization, mathematical programming, and practical SAT algorithms to the SAT problem. At the end of this survey, we will give a collection of practical applications of the satisfiability problem.

In the constraint satisfaction problem (CSP) one has a set of constraints which depend on a set of discrete variables. One seeks a set of value assignments to variables that satisfies all of the given constraints. Thus, the SAT problem is a special case of CSP. Also, any CSP problem can be re-expressed as a SAT problem. Due to this close relationship, any SAT algorithm can be transformed into a CSP algorithm, and this can usually be done in a way that maintains the efficiency of the algorithm. (See Section 5.6: Historical Notes).

The rest of the paper is organized as follows: In the next section, we give a general framework, an algorithm space, that puts existing SAT algorithms into a unified perspective. This is followed with a brief overview of the basic SAT algorithm classes and a discussion of the general performance evaluation approach for the SAT algorithms. In Section 3, some SAT problem instance models and preliminaries for the paper are given. Section 4 describes the variable setting and resolution procedures for the SAT problem. Local search algorithms for the SAT problem are discussed in Section 5. Global optimization techniques for the SAT problem are presented in Section 6. The integer programming approaches to the SAT problem is given in Section 7. Section 8 discusses special subclasses of the SAT problem. Some advanced techniques for the SAT problem are described in Section 9. Section 10 gives probabilistic and average case analysis of the SAT problem. Partial experimental results and performance comparisons of several major SAT algorithms are given in Section 11. Section 12 summarizes applications of the SAT problem. Future work for SAT research is discussed in Section 13. Finally, Section 14 concludes the paper.

2 An Algorithm Space Perspective of SAT Algorithms

In this section, we give an algorithm space perspective which provides some insights to develop efficient algorithms for the SAT problem. Following this, we give a brief overview of the basic sequential and parallel SAT algorithms. The basic algorithm category and algorithm performance evaluation methods will also be discussed.

2.1 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 1 shows a typical *algorithm space* that unifies a variety of search and optimization algorithms in terms of variable domain, constraint character, and the parallelism in the algorithms [194].

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 variable improvement which do not satisfy the constraints (typically, these algorithms assign some cost to not satisfying constraints and then look for the least cost solution). This defines a vertical axis showing constraint characteristics in the space. Most SAT algorithms are sequential, some have been implemented in parallel. A third axis indicating the parallelism in the algorithms is added in the algorithm space. Following the three axes, the algorithm space is divided into eight octants, representing 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 2 gives some typical examples for four sequential classes of SAT algorithms in the space (this figure is a 2-dimensional cross section of the algorithm space cut at the sequential side). In the discrete search space (left half of 2), variables, values, constraints, and the objective functions are defined with

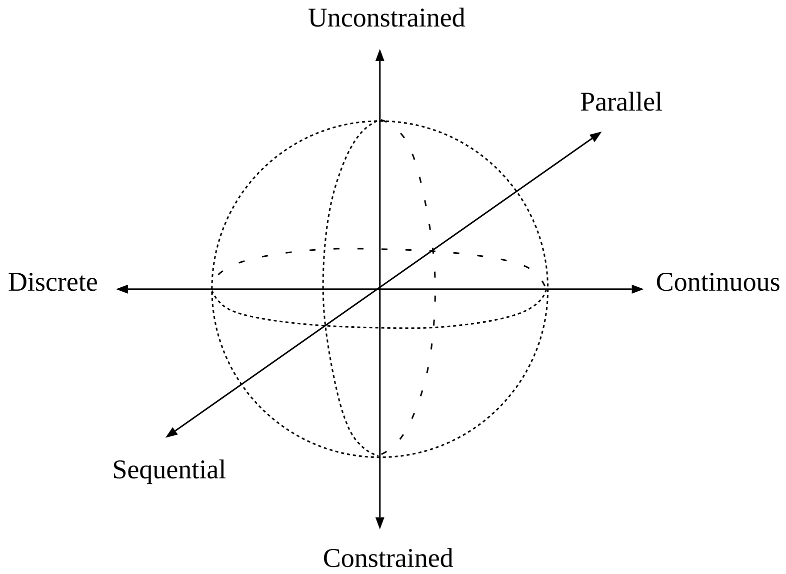

Figure 1: The algorithm space is a unified framework for discrete search algorithms and continuous optimization algorithms. The octants represent eight basic classes of algorithms.

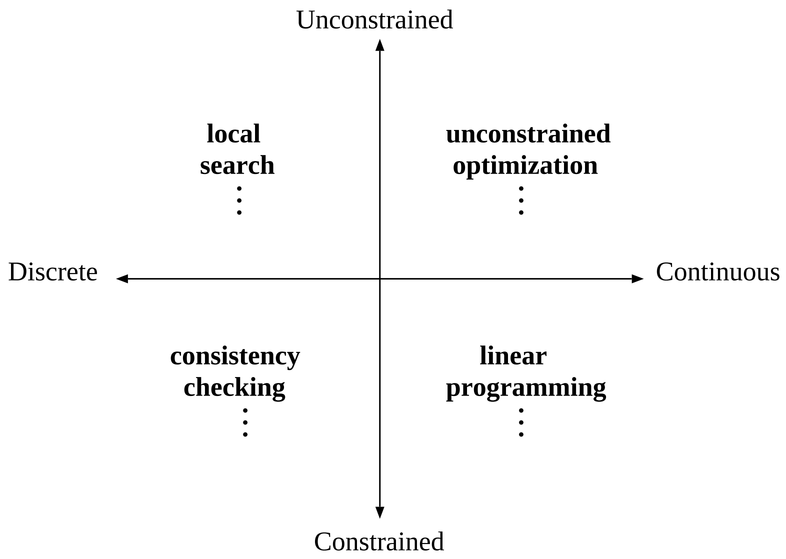

Figure 2: A 2-dimensional cross section of the algorithm space cut at the sequential side. It indicates a unified framework for some discrete search algorithms and continuous optimization techniques for the SAT problem.

discrete values. If one handles a discrete search problem with consistency checking or constraint resolution, the approach belongs to the *discrete constrained methods* [201, 311, 329, 463]. 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. [190, 191, 347, 418, 422].

In the continuous search space (right half of 2), variables, values, constraints, and the objective functions are defined quantitatively with real values. If one solves a continuous optimization problem with explicitly constraints, one uses the *continuous constrained methods*, e.g., constrained minimization, primal methods, and cutting plane methods [309]. If the problem constraints are converted into an objective function, then the problem is transformed into an unconstrained one. The latter can be solved by the *non-continuous constrained methods*, such as the descent methods, conjugate direction methods, and Newton methods [194, 193, 309].

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 often solves the problem efficiently. Furthermore, local search works in a 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 problem [29, 193, 194, 224, 249, 248].

The algorithm space provides a unified and global perspective to the development of search and optimization algorithms for the SAT problem. For a given search problem in general, 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 2, for example, once we had consistency algorithms and local search algorithms for the SAT problem, it would be natural to think about unconstrained optimization algorithms for the SAT problem for 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 the SAT problem [194].

There are other ways of looking at a variety of SAT algorithms. A different algorithm space for the SAT problem that incorporates algorithm completeness was given in [194] (see Figure 3).

2.2 Basic SAT Algorithm Classes

Following the algorithm space, a number of major SAT algorithm classes can be identified. They are given in Figure 4 in their chronological order. Most existing SAT algorithms can be grouped into these categories.

- *Discrete, constrained algorithms.* Algorithms in this category treat the SAT problem as a constrained decision problem, solving the problem with discrete search and inference procedures. One straightforward way to solve the SAT problem is to enumerate all possible truth assignments to see if one satisfies the formula. Many improved techniques, such as consistency algorithms [201, 311], backtracking algorithms [28, 47, 56, 284, 363], term-rewriting [115, 234], production system [414], multi-valued logic

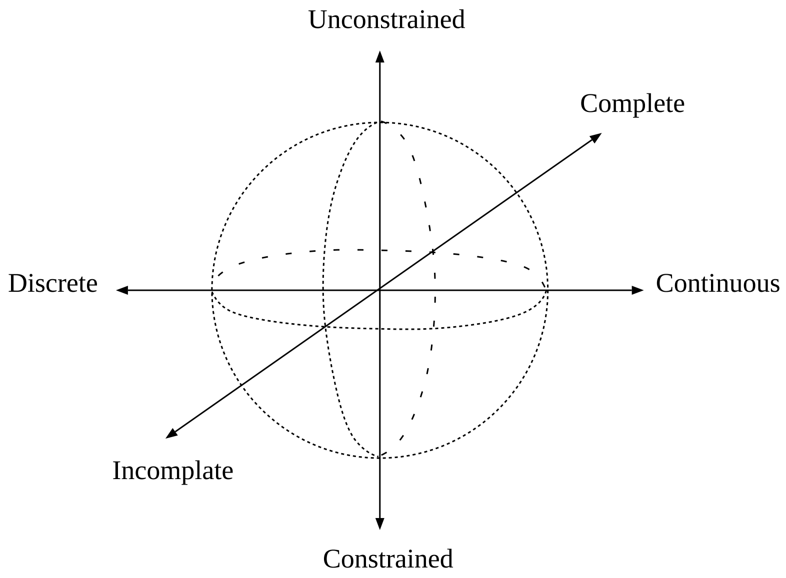

Figure 3: An algorithm space incorporating algorithm's completeness for the SAT problem. Each octant represents one class of SAT algorithms.

[410], Binary Decision Diagrams [53, 15], chip and conquer [174], resolution and regular resolution [176, 307, 341, 386, 445, 470], independent set algorithm [245], and matrix inequality system [443], 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 procedure [103], or by assigning some variable each possible value and generating a subproblem for each value, as was done in Loveland's modification to the Davis-Putnam procedure [102, 307]. Resolution generates only one new problem, but in the worst case the number of clauses in that new problem will be proportional to the square of the number of clauses in the original problem. Assigning values to a variable (often called searching) generates two new problems. For random problems, resolution methods are fast when the number of clauses is small compared to the number of values [151, 75], while search methods are fast except when the number of clauses is such that the expected number of solutions is near one [371]. The two approaches can be combined, using resolution on some variables and search on others.

Other specific algorithms using these principles include simplified Davis-Putnam algorithms [164, 182, 368], and a simplified Davis-Putnam algorithm with strict ordering of variables [235]. The Davis-Putnam algorithm improved in certain aspects over Gilmore's proof method [177]. 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 the simple algorithms are fast, related practical algorithms are also fast. (It is difficult to tell whether the practical algorithm is slow under conditions that make the simplified algorithm be slow.)

- Discrete
  - Constrained
    - 1960: Davis-Putnam (DP) algorithm [103]
    - 1965: Resolution [386]
    - 1971: Consistency algorithms [83, 185, 237, 462]
    - 1978: Loveland's Davis-Putnam (DPL) [102, 307]
    - 1986: Parallel consistency chips [185, 199, 200]
    - 1986: Binary decision diagrams (BDD) [15, 53]
    - 1988: Chip and conquer [174]
    - 1990: DPL plus heuristic (DPLH) [248]
    - 1989: Local search & backtracking [191]
    - 1993: Backtracking and probing [371]
    - 1994: Parallel DP algorithm [32]
    - 1994: Matrix inequality system [443]
  - Unconstrained
    - 1987: *CNF* Local search (SAT1.0) [186, 191]
    - 1987: *DNF* Local search (SAT1.4) [186, 194]
    - 1987: Parallel local search (SAT1.6) [186, 191]
    - 1987: Local search hardware [186]
    - 1988: Local search for n-queen [185, 423, 416]
    - 1990: *Unison* algorithm and hardware [424, 425]
    - 1990: Local search with partial assignment [418, 422]
    - 1991: Local search for 2SAT [346]
    - 1991: Local search complexity [196, 347]
    - 1992: Local search — GSAT [408]
    - 1994: GRASP [383]
- Continuous
  - Constrained
    - 1986: Branch-and-bound (APEX) [29]
    - 1988: Programming models [29, 249]
    - 1988: Cutting plane [225, 223]
    - 1989: Branch-and-cut [226]
    - 1989: Interior point method [262, 260]
  - Unconstrained
    - 1987: *UniSAT* models [186, 189, 194]
    - 1987: Global optimization (SAT6) [186, 194]
    - 1989: Neural net models [253, 66]
    - 1990: Global optimization & backtracking [194]
    - 1991: SAT14 algorithms [194]

Figure 4: Some typical algorithms for the SAT problem.

A number of special SAT problems, such as 2-satisfiability and Horn clauses, are *solvable* in polynomial time [5, 89, 341]. Several linear time algorithms [16, 139], a polynomial time algorithm [346], and a polynomial time bound [399] exist.

- *Discrete, unconstrained algorithms.* In this approach, the number of unsatisfiable *CNF* (or satisfiable *DNF*) clauses is formulated as the value of the objective functions, so the SAT problem is transformed into a discrete, unconstrained minimization problem to the objective function. Local search is a major class of discrete, unconstrained search methods [190, 191, 201, 196, 347, 408]. It can be used to solve the transformed problem (see Section 5). A special case of local search is simulated annealing [206].

- *Constrained programming algorithms.* Methods in this class were developed based on the fact that the SAT problem can be written as integer program and one could try to solve it as a linear programming problem [29, 224, 225, 249, 262, 260, 345, 469]. Many approaches, including branch-and-bound [29], cutting plane method [225, 223], branch-and-cut [226], interior point algorithm [262, 260], and improved interior point algorithm [411], 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 but they do not possess robust convergent property and often fail to solve hard satisfiability problems [29, 224, 225, 249, 262, 260].

- *Unconstrained, global optimization algorithms.* Special models are formulated that transform a discrete SAT problem on Boolean space $\{0,1\}^n$ into a quantitative, unconstrained *UniSAT* problem on real space E^n. Thus, the original decision problem is transformed into an unconstrained global optimization problem which can be solved by existing global optimization methods [186, 190, 189, 194, 201] (see Section 6).

2.3 Parallel SAT Algorithms

In practice, most sequential SAT algorithms can be mapped onto parallel computer systems, resulting in parallel SAT algorithms [195]. A speed up greater than the number of processor sometimes occurs because of correlations among variable settings that lead to solutions [331]. Accordingly, as given in Figure 5, there are *four* classes of parallel algorithms for the SAT problem.

- *Parallel, discrete, constrained algorithms.* Many discrete, constrained SAT/CSP algorithms have been implemented in parallel algorithms or put on special-purpose, hardware VLSI architectures. These include parallel consistent labeling algorithms [448, 320], parallel discrete relaxation (DRA) chips [199, 185, 200], parallel arc consistency (PAC) algorithms [398], parallel constrained search architectures [185, 200], parallel *Unison* algorithms [424], parallel *Unison* architectures [425], parallel Davis-Putnam algorithms [32, 76, 141], and parallel logical programming languages [87, 303, 458, 460, 461].

- *Parallel, discrete, unconstrained algorithms.* A number of discrete local optimization algorithms were implemented with parallel computing machines. These include *CNF* local search [186, 191], *DNF* local

- Discrete
  - Constrained
    - 1983: Parallel CLP algorithms [448, 320]
    - 1986: Parallel DRA chips [185, 199, 200]
    - 1987: Parallel DP algorithm [76]
    - 1988: Parallel AC algorithms [398]
    - 1988: Parallel CSP architectures [185, 200]
    - 1990: *Unison* algorithm and hardware [424, 425]
    - 1992: Vectorized DP algorithm [141]
    - 1994: MIMD DP algorithm [32]
  - Unconstrained
    - 1987: *CNF* local search [186, 191]
    - 1987: *DNF* local search [186, 194]
    - 1987: Parallel local search [186, 191]
    - 1991: Discrete $\alpha\beta$ relaxation [188]
    - 1993: Multiprocessor local search [421, 420]
- Continuous
  - Constrained
    - 1989: Interior point method [262, 260]
  - Unconstrained
    - 1987: *UniSAT* models [186, 194]
    - 1987: Global optimization (SAT6) [186, 194]
    - 1991: Continuous $\alpha\beta$ relaxation [188]
    - 1991: SAT14 algorithms [194]
    - 1991: Parallel global optimization [189, 194]
    - 1992: Neurocomputing [195]

Figure 5: Some parallel SAT/CSP algorithms.

- General-Purpose Programming
  - Sequential Machines
    - VAX-8600
    - SUN workstations
  - Parallel Machines
    - 1983: Multicomputers [448, 320]
    - 1987: CRAY [397]
    - 1988: BBN butterfly plus [396]
    - 1988: Connection machine [90]
    - 1989: KORBX vector computer [262, 260]
    - 1992: ETA10Q Vector computer [141]
    - 1994: INMOS Transputer [32]
- Special-Purpose Architectures
  - Sequential Machines
    - 1980: Analog processor [322]
    - 1986: DRA1 architectures [275, 199]
  - Parallel Machines
    - 1986: DRA architectures [185, 199, 200]
    - 1986: mDRA architectures
    - 1987: CSP architectures [185]
    - 1987: mCSP architectures [185]
    - 1988: DRA model architecture [90]
    - 1989: DRA model architecture [302]
    - 1990: *Unison* architectures [424, 425]

Figure 6: Computer architectures used for running SAT/CSP algorithms.

search [186, 194], parallel local search [186, 191], and multiprocessor local search [421, 420]. A new $\alpha\beta$ relaxation technique was developed within a parallel and distributed environment [188].

- *Parallel, constrained programming algorithms.* Kamath et al. implemented an interior point zero-one integer programming algorithm on a *KORBX(R)* parallel/vector computer [262, 260].

- *Parallel, unconstrained, global optimization algorithms.* Several of these algorithms have been implemented: *UniSAT* models [186, 194], parallel, continuous $\alpha\beta$ relaxation [188], and parallel global optimization algorithms [189, 194].

Computer architectures affect the data structures, implementation details, and thus the performance of the SAT algorithms. A variety of computer systems have been used for running the SAT algorithms (Figure 6). Most early studies of CSP/SAT algorithms were performed on sequential computers. Recent work has been concentrated on parallel programming on the multiprocessor machines. McCall et al. [320, 448] 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 [397] and an 18-node BBN Butterfly Plus MIMD, shared-memory, homogeneous parallel processor [396]. Cooper and Swain implemented parallel AC algorithms on a Connection Machine [90]. Kamath and

Karmarkar et al. implemented an interior point zero-one integer programming algorithm for SAT on a *KORBX(R)* parallel/vector computer [262, 260]. Recently, Fang and Chen implemented a vectorized DP algorithm on an *ETA10Q* vector computer [141]. Speckenmeyer and Böhm have experimented with the parallelization of variants of the Davis-Putnam-Loveland procedure on a message based MIMD Transputer system built with 320 (INMOS T800/4MB) processors [32]. They found that, 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 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 [32].

Research works continue by building special-purpose VLSI architectures to speed up the SAT/CSP computations. For an n-variable and m-value CSP problem, Wang and Gu [464, 465] gave an $O(n^2m^2)$ time parallel DRA2 algorithm and an SIMD DRA2 architecture. Furthermore, Gu and Wang [199] gave an $O(n^2m)$ time parallel DRA3 algorithm and a dynamic DRA3 architecture for solving general DRA problems. Later, Gu and Wang [185, 200] developed an $O(nm)$ time, massively parallel DRA5 algorithm and a parallel DRA5 VLSI architecture. For problems of practical interest, parallel DRA algorithms running on special-purpose, hardware VLSI architectures offer many orders of magnitude performance improvement over the sequential algorithms.

Recently, Sosič, Gu, and Johnson have developed a number of parallel algorithms and architectures for differential, non-clausal inference of the SAT problem [424, 425].

An extreme example of parallel processing is to NP-complete calculation using chemistry with DNA molecules. This would appear to lead a factor of about 10^{23} 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, 305].

2.4 Algorithm Categories

Some SAT algorithms are *complete* (they definitely determine whether a problem has a solution or does not have one) [53, 103, 102, 307, 386], while others are *incomplete* (they sometimes determine whether or not the problem has a solution, but in other cases they can not find one) [191, 194, 260, 346].

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 problem has no solution or the algorithm did not search hard enough. In the strict sense of the word *algorithm*, the incomplete algorithms are not algorithms at all but such procedures are of particular interest for those problems that are so difficult that a complete algorithm can not solve them in reasonable time.

Complete algorithms can (1) determine whether or not a solution exists, (2) give the variable settings for one solution, (3) find all solutions or an optimal solution, or (4) prove that there is no solution. Algorithms of the first type would be of only theoretical interest 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 problem. The algorithms are essential to many important practical application problems that are NP-hard in nature. Recently, Major et. al. used SAT to precede a program to calculate chemical interaction energies in their program to predict RNA folding [312]. Gu and Puri developed an efficient complete SAT algorithm for asynchronous computer circuit design, aiming at producing the minimal circuit structure [198, 375]. Incomplete algorithms can not optimize solution quality, playing little role in solving practical optimization problems.

Requiring that a program produces each solution in explicit form ensures that the worst case time will be exponential whether or not P = NP (because some problems 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., giving the settings of some variables with the understanding that the remaining variables are *don't cares* which can have any value. For some problems, using this approach to represent all solutions is much more compact than an explicit representation [56, 324]. Binary Decision Diagrams (BDD) are more sophisticated and compact way to represent the set of all solutions [53, 15]. Some SAT problems, however, have a structure where it is faster to generate the solution to various subsets of the constraints (which depend on a subset of the variables) and then test whether those various solution sets have any thing in common rather that to try and solve the entire problem at once. This type of SAT algorithm shows greater efficiency improvements for certain practical engineering design problems [376].

A discrete optimization problem has a function of discrete variables. The goal is to find the variable setting that leads to the lowest value of the function. 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 SAT problem sets, the probability that the problems have solutions 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 problem instances generated from the 3-SAT model as an example. Figure 7 shows, for 50 variables, the real execution results of the Davis-Putnam algorithm for 100 to 500 clauses. The computing time used by a program for the DP algorithms ([103, 102, 307]) is shown for the 3-SAT problem model (solid line) and the average 3-SAT problem model (dotted line) [186, 191, 328, 93]. The SAT problem instances generated in the left region are usually satisfiable, and the procedure is fast. For the problem instances in the right region are usually unsatisfiable, and the procedure is fast. For problems in the middle many are satisfiable and many are unsatisfiable; the procedure is slow. Because the Davis-Putnam algorithm is a complete algorithm, it is able to verify satisfiability and unsatisfiability. So it gives results for problem instances in all three regions.

The results of the Davis-Putnam algorithm may not hold for a different SAT algorithm. An incomplete algorithm that searches for solutions may often find a solution much more quickly than the Davis-Putnam algorithm but it does not always verify satisfiability and it can not prove unsatisfiability. In particular, it gives no answer if a *CNF* formula is not satisfiable. Thus, for most problem instances in the peak region and nearly all problem instances in the right region, an incomplete algorithm will not terminate in

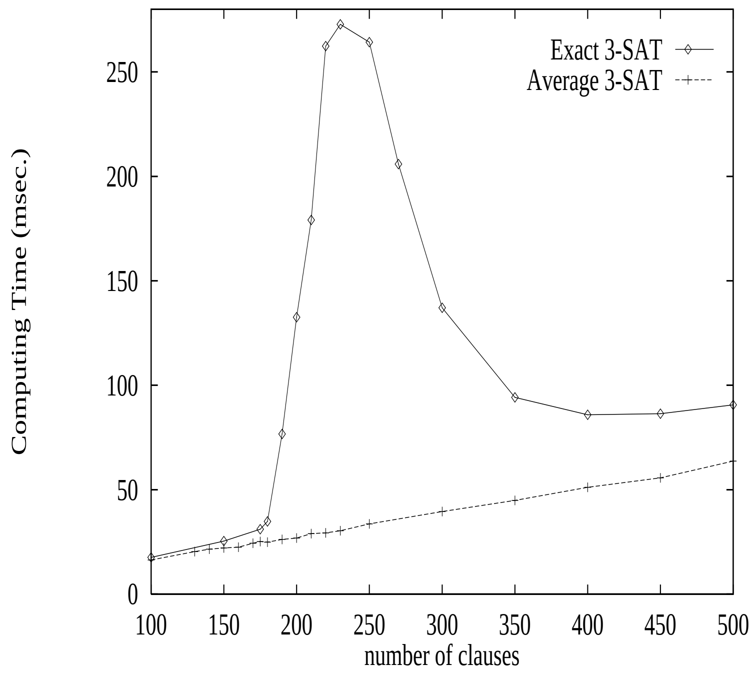

Figure 7: Computing time for the exact and the average 3-SAT problem models (with 50 variables) on a SUN SPARC 1 workstation.

a reasonable amount of time.

2.5 Algorithm Performance Evaluation

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

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. Analytic 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 analytic result on the simplified algorithm is 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: analytic 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 analytic studies of SAT algorithms [62, 61]. The two top winners were associated with both experimental and analytic studies of SAT algorithms. The analytic studies of SAT algorithms invoke the following three aspects.

1. Worst Case Studies. Unless P = NP, all SAT algorithms have a worst-case time that is greater than polynomial in the problem size [89]. A number of works have concentrated on the worst case studies of the variable setting algorithms for the SAT problem [174, 330, 276].

2. Probabilistic Studies. Since the typical performance of the better satisfiability algorithms is much better than any proven worst case results, there is considerable interest in probabilistic performance of these algorithms. These studies use some model for generating random problem instances and then calculate the performance of algorithms on those problems. 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 problem. The algorithm must solve each problem for the average to be defined. In *probabilistic time* studies, the algorithm has a deadline, and one studies the fraction of problems that are solved within the deadline. Probabilistic time studies can be done on algorithms which give up on some fraction of the problems so long as that fraction is less than the goal fraction.

For typical or random problems, one can study either the average time or the probabilistic time of the given problems. For incomplete algorithms, the average time is not defined so only the fraction of problems 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 problems.

A number of researchers studied the average time and probabilistic time performance of certain SAT algorithms [47, 153, 152, 180, 181, 182, 235, 362, 370, 365]. Despite the worst-case complexity of the SAT problem, algorithms and heuristics with polynomial average time complexities have constantly been reported [71, 72, 149, 150, 363, 367, 368, 369, 472]. This subject is treated in more detail in Section 10.2.

3. The Number of Solutions. Some researchers investigated the number of solutions from the SAT problem instances. Extending Iwama's work [245] Dubois gave a combinatorial formula computing the number of solutions of a set of any clauses [133]. Dubois and Carlier also studied the mathematical expectation of the number of solutions for a probabilistic model [134].

During the past two decades, many search algorithms' performance were studies through sampling techniques [270, 362, 433],[6] experimental simulations [48, 173, 207], theoretical studies [47, 159, 160, 370, 363, 367, 432], as well as the combined effort of the above approaches [173, 232, 270, 362, 432].

3 SAT Problem Models

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

3.1 Random Problem Models

The time of a SAT algorithm depends on the type of problem being solved. The following SAT problem models are often used by researchers in the area.

- **Hardest problems**. Generate that problem that is most difficult for the algorithm being measured. This approach is often used for theoretical studies. There are too many possible problems to use this approach in experimental studies. See [57] to see what effects these minor variations have on the average time needed by the simple backtracking algorithm.

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

Most analytic studies use the following two basic models to generate random *CNF* formulas. Each model has many variations dependent of whether identical clause are permitted, whether a variable and its negation can occur in clause, etc.

[6] Knuth [270] first showed how to measure the size of a backtrack tree by repeatedly following random paths from the root. Purdom [362] 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 [433] 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.

- **The l-SAT model.** In the l-SAT model, a randomly generated *CNF* formula consists of independently generated random clauses. Each clause contains exactly l literals. Each literal q is uniformly chosen from $X = \{x_1, \ldots, x_m\}$ without replacement, i.e., for $x_i \in X$, $\Pr[q = x_i] = 1/|X|$, where $|X|$ denotes the number of elements in X, and then q is negated with probability p. This model is something called the *fixed length clause* model. Similar models were used in [47, 153, 191, 194, 328, 324, 367].

- **The average l-SAT model.** In the average l-SAT model, a randomly generated *CNF* formula consists of independently generated random clauses. Each of the n variables occurs positively with probability $p(1-p)$, negatively with probability $p(1-p)$, both positively and negatively with probability p^2, absent with probability $(1-p)^2$. The average number of literals in a clause is $l = 2pn$. This model is also called the *random clause* model. This model and its variations were used in [182, 190, 191, 194, 225, 226, 262, 260, 411].

Both the l-SAT and average l-SAT models have many variations depending on whether the clauses are required to be distinct, whether the literals in a clause are required to be distinct, whether a variable and its negation can occur in the same clause, etc. Most papers use just one model, but the performance of simple backtracking has been considered under a number of related models [57].

Real world problems usually have various clause lengths, but have no clauses of length 0 or 1. This last point makes them more like the problems generated by the l-SAT model. On the other hand the mathematics associated with average time analyses for l-SAT models is much more difficult, so not many algorithms have been analyzed with that model. 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.

Much of the difference in mathematical difficulty between l-SAT and average l-SAT models is associated with the effects that steps in an algorithm have on the distribution of clause sizes. The most important step is often assigning a value to a variable. When the initial set of clauses comes from the average l-SAT model such a step produces a set of clauses which also comes from the average l-SAT model (but with one less variable). However, when the initial set of clauses comes from the l-SAT model the resulting clauses do not have fixed length.

Various sources of random problems can be adapted to provide random SAT problems by transforming the original problem into a SAT problem. The distribution generated by such a process is usually quite different from that generated by the above standard random models.

3.2 Hardness

Various SAT algorithms differ greatly in the amount of time they need to solve particular SAT problems. For example, Iwama's algorithm [245] is fast for random problems with lots of solutions and slow for random problems with few solutions, while simple backtracking [369] is fast on problems with few solutions and slow on problems with many solutions. Therefore, the *hard-and-easy distributions* of the SAT problem instances depend not only on the inherent property of the SAT problem models but also on the algorithms

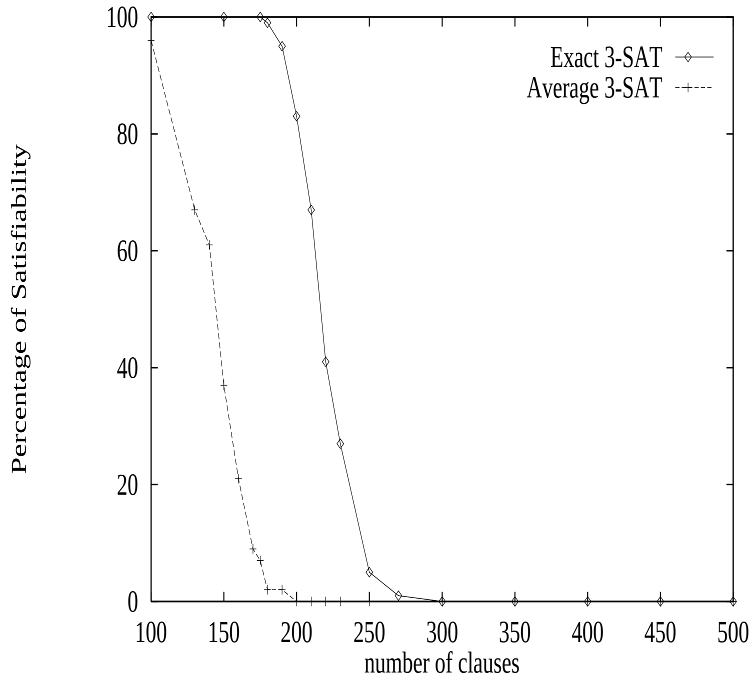

Figure 8: Percentage of satisfiability for the 3-SAT and the average 3-SAT problem models with 50 variables.

used to solve the problem instances. Any particular SAT problem is easy for some algorithm (for example a table lookup algorithm with that problem in its lookup table). Thus, hardness is a *property* of large sets of problems rather than individual problems.

For problem sets generated by random models with parameters, the probability of a solution varies with the parameter settings. Those problem sets generated with parameter sets in the regions where solutions are going from unlikely to common are particularly difficult for all algorithms that have been studied (see Figure 8).

For random l-SAT problems the hardness of the SAT problem increases, up to a point, when m/n increases or the number of literals l ($l > 3$) in each clause decreases. That is, fewer literals and larger number of clauses reduce the possibility of making all clauses jointly satisfiable. The 3-SAT problem is thought to be a difficult problem to solve. An informal indication of hardness can be derived from the percentage of sat/unsat statistics of the SAT problem instances.

3.3 The Comparison of SAT Problem Models

Figures 7 and 8 show, for 50 variables, the real computing time and percentage of satisfiability of a complete SAT ($SAT14.11$) algorithm[7] for problem instances generated from the 3-SAT model and the average 3-SAT

[7] $SAT14.11$ is a backtracking algorithm combined with coordinate descent in the real space [194].

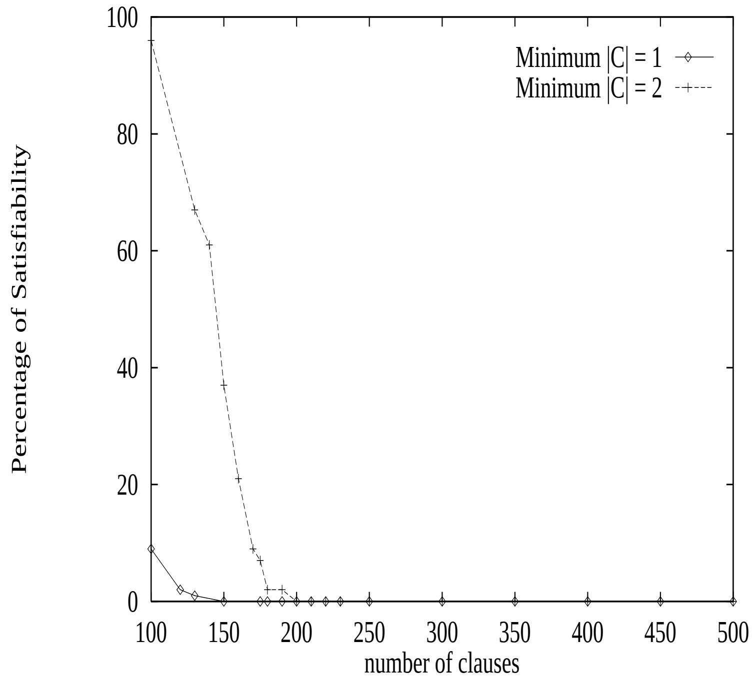

Figure 9: Percentage of satisfiability for two average 3-SAT problem models with $|C|_{min} = 1$ and $|C|_{min} = 2$ (50 variables). Problem instances generated from a model with smaller minimum number of literals have much lower percentage of satisfiability.

model [186, 191, 193, 194]. For this algorithm the average 3-SAT model is much easier than the 3-SAT model since it takes a complete algorithm much less computing time to solve problem instances generated from the average 3-SAT model.

For many incomplete algorithms, however, the situation is different. As Figure 8 indicates that, for the same problem sizes, problem instances generated from the average 3-SAT model have much lower percentage of satisfiability, as compared to those generated from the 3-SAT model. The sat-and-unsat boundary of the average 3-SAT model is drawn by smaller m/n values than those for the 3-SAT problem model. In this case, since most problem instances generated from the average 3-SAT model are unsatisfiable, it is much harder for an incomplete algorithm to handle the problem instances generated from the average 3-SAT problem than those generated from the 3-SAT problem.

In general, for different problem instance models, the m/n values that separate the sat and unsat regions differ considerably. Even for the same problem model, a tiny variation to any parameter of the models will significantly change the m/n values and thus the sat-and-unsat boundary. For the average 3-SAT problem model (see Figure 9), problem instances generated from a model with minimum number of literals being 1 have much lower percentage of satisfiability than the model with minimum number of literals being 2.

The m/n values that determine the sat-and-unsat boundary also depend on the algorithms used to solve the problem. Apparently, an incomplete algorithm can not decide a sat-and-unsat boundary for any SAT problem instance model.

Experimenters who do experiments to measure the speed of the best complete algorithms have these impressions (for random problems with the parameters set to be near the SAT-unSAT boundary):

1. average l-SAT problems are easy for the best algorithms,
2. l-SAT problems are difficult for the best algorithms, and
3. both models are of similar difficulty when the average clause length is large.

Obviously, there is some conflict in these beliefs.

3.4 Practical Problem Models

The random problem models are suitable for theoretical studies of tha SAT algorithms. Structured problem instances and practical SAT application problems are essential to benchmark the real performance of the SAT algorithms.

- **The regular SAT models.** Models such as the graph coloring and n-queen problems are used to assess the performance of the SAT algorithms [189, 194].
- **Practical applications problems.** Models derived from practical application domains, for instance, from 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 12).

Real experiments strongly suggest that there is little correlation between the performance of a SAT algorithm tested through random problem models and the performance of the same algorithm tested through practical problem models. Local search is faster for some random problems, but can be hundreds times slower than a complete SAT algorithm, for asynchronous computer circuit design problems [198, 375]. The sat-and-unsat boundary and transition phenomenon discussed in the random models have not yet been observed in the practical problem models.

The practical applications are ultimately the *most important*, although it is difficult for people outside the area of application to understand how important or difficult particular application problems are. It is also difficult to develop a general theory of 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.

3.5 Preliminaries

To simplify our discussion, throughout this paper, let:

- F be a *CNF* Boolean formula in disjunctive normal form. It is the logical *and* of clauses each of which is the logical *or* of literals, • m be the number of clauses in F,
- n be the number of variables in F,
- C_i be the ith clause,
- $|C_i|$ be the number of literals in clause C_i,
- Q_{ij} 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:

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

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

On real space E^n, let:

- y_j be the jth variable,
- $\mathbf{y}$ be a vector of n variables,
- $c_i(\mathbf{y})$ be the ith clause function,
- $q_{ij}(\mathbf{y})$ be the jth literal function of the ith clause function, and
- $f(\mathbf{y})$ be a real function from E^n to E,

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 [309], 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 forms a vector function $\mathbf{f} = (f_1, f_2, ..., f_m)$ whose ith components 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

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

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

Two aspects of the iterative optimization algorithms are global convergence and local convergence rate [309]. *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 converges to the solution.

4 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 SAT formulas. Recursion continues until one or more primitive formulas are generated and solved to determine the satisfiability of the original. One way to achieve this is through splitting.

In splitting, a variable v existing in a formula is selected and the formula is replaced by one subformula for each of two possible truth assignments to v. Each subformula 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 subformula contains v and the original formula has a satisfying truth assignment if and only if either subformula has one. Splitting insures that a search for a solution terminates with a result.

Another effective paradigm is based on resolution [386]. 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 resolvants using a particular variable at one time. When this is done, one may drop the original clauses that contain the variable and its negation. An algorithm may use both splitting and resolution.

Early examples of these approaches are the two forms of the Davis Putnam procedure. The original *Davis-Putnam (DP) procedure* used resolution [103] while the revised version, i.e., the *Davis-Putnam-Loveland (DPL) procedure*, used splitting [102, 307]. Combining splitting with depth-first search in the DPL procedure avoids a memory explosion that occurs on many problems when they are solved with the original DP procedure.

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

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

The following subsections present various SAT algorithm, organized by the basic approach that the 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 problems.

4.1 Resolution

Given two clauses $C_1 = (v \vee x_1 \vee ... \vee x_N)$ and and $C_2 = (\bar{v} \vee y_1 \vee ... \vee y_M)$, where all of the x_i and y_j are distinct, the resolvent of C_1 and C_2 is the clause $(x_1 \vee ... \vee x_N \vee y_1 \vee ... \vee y_M)$, 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. Thus, adding the resolvent to the clauses of the original problem does not change its set of solutions. Furthermore, if the original problem has no solution, repeatedly adding resolvents eventually generates the empty clause.

Resolution [386]: For each variable x and each pair of clauses C_1 and C_2 such that C_1 contains the literal x and C_2 contains the literal not x, add the resolvent of C_1 and C_2 with respect to x to the set of clauses of the problem. Repeat until the empty clause is obtained (in which case the problem is unsatisfiable) or until not new resolvents can be computed (in which case the problem is satisfiable).

There are two useful and simple refinements to the basic resolution algorithm. First, once all the resolvents with respect to a particular variable have been formed, the clauses containing that variable can be dropped. Doing this does not change whether or not the problem is satisfiable, 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 problem is satisfiable. At the next to last step the problem has just one variable, so you can test each value to see which one satisfies the problem (perhaps both will). Pick a satisfying value and plug it into the formula for the proceeding step, converting it into a one variable problem. Solve that problem and proceed in this manner until an assignment for all variables is found.

For some problems the order in which clauses are resolved can have a big effect on how much effort is needed to solve a problem. The worst-case associated with the best possible order (the order is selected after the problem is given) has received considerable study [164, 445, 453]. These studies all used problems 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 problems need exponential time when the ratio of clauses to variables is above a constant (whose value depends on l) [75]. The constant is such that nearly all of the problems in the set have no solution.

The second simple refinement is called 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.

Although resolution can be applied to SAT problems, the main reason for interest in resolution is that it can be applied to the more difficult problem of solving first order predicate calculus problems. There is a vast literature on that subject. Bibel has a good book on the topic [26].

A literal is *pure* if all of its occurrences are positive or if all of its occurrences are negative. A pure literal results in no resolvents. Resolution on a pure literal does not change the problem. Resolution on a pure literal followed by the discarding the original clauses leads to a smaller problem. An important improvement to the basic resolution algorithm is to select pure literals for resolution (before selecting non-pure literals) [103].

4.2 Backtracking

Backtracking algorithms are based on splitting. During each iteration, the procedure selects a variable and generates two subproblems by assigning the two values, *true* and *false*, to the selected variable. In each subproblem, 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 the false 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 problem. The internal nodes of the tree correspond to subproblems that can not be solved directly, the leaf nodes correspond to subproblems that can be solved directly. The nodes are connected with arcs that can be labeled with variable assignments.

Simple backtracking [47]: If the problem has an empty clause (a clause which is always false) then exit and report that the problem has no solution. If the problem has no variables, then exit and report that the problem has a solution. (The current assignment of values to variables is a solution to the original problem.) Otherwise, select the first variable that does not yet have a value. Generate two subproblems by assigning each possible value to the selected variable. Solve the subproblems recursively. Report a solution if any subproblem has a solution, otherwise report no solution.

Backtracking is an old algorithm. It is relatively quick when most of the subproblems are skipped.

In the average l-SAT model, the average number of solutions per problem is approximately

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

Thus, when m/n and pn are such that the exponent is negative, problems have very few solutions, but when the exponent is positive problems have many solutions, on average. When m/n is below $\ln 2$, the average number of subproblems 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 subproblems and large values lead to a huge number of subproblem, 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 [369].

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 subproblem in those cases where the first subproblem 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 problems that have solutions, the analysis in [369] 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.

Unit clause backtracking [363]: This algorithm is the same as simple backtracking except for how variables are selected. If some clause depends on only one of the unset variables then select that variable; otherwise, select the first unset variable.

In practice, this improved variable selection often results in much faster backtracking [28].

An average l-SAT analysis of unit clause backtracking [363] shows that the conditions under which it is fast or slow are similar to the conditions underwhich 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 problems occur with large pn values, and so unit clauses are rare.

Clause order backtracking [56]: This algorithm is the same as simple backtracking except for how variables are selected. If this setting does not solve the problem 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 case". Thus, a single solution with unset variables represents the set of solutions obtained by making each possible assignment to the unset variables.

The average l-SAT analysis of this algorithm shows that, in addition to being fast under conditions where simple backtracking is fast, it is under various other conditions. It is fast pn is below 1. When m/n is small, the typical l-SAT problem does not use most of the variables. Thus, most problems 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 problem, 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.

Probe order backtracking [371]: 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 problems 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 [299, 418, 422], probe with several sequences at one time [62, 61], carefully select which variable to set [62, 61], use resolution when it does not increase the problem size [151]. The sixth best prizing winning entry in the 1992 SAT competition used an improvement on probe order backtracking [61].

Franco [150] noticed that a random assignment solves a nonzero fraction of the problems 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 [371], 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., the setting of 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 5.

Shortest clause backtracking : This algorithm is the same as clause order backtracking except for the clause is selected. 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 [28]. No average l-SAT analysis has been done for shortest clause backtracking. (See [324] 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 prizing winning entries in the 1992 SAT competition all used shortest clause backtracking [61] 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.)

4.3 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 subproblem that is no more complex than the original problem, while from the perspective of splitting, pure literals lead to two subproblems, but the solutions to the subproblem where the literal has the value *false* are a subset of the the one where the literal has the value *true*. Therefore, the original problem has a solution if and only if the problem associated with the *true* literal does.

The pure literal rule algorithm [180]: 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 subproblem by assigning the selected variable the value that makes its literals *true*. Otherwise, generate a subproblem for both values of the selected variable. Solve the one or two subproblems.

This is one of the first algorithms to have its average time computed [58, 180, 182, 368]. It has the essence of the pure literal rule from the DP procedure [103]. 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 does rapidly solve a wide class of problems 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 [364] extends this idea by noting that when there are few occurrences of a literal, then the setting that makes that literal *false* leads to a subproblem that is almost a subset of the subproblem obtained by setting the literal to *true*. Thus, if a problem contains one clause that has the only occurrence of a literal, any solutions to the *false* subproblem that are not also solutions to the *true* subproblem have *false* values for all remaining literals in the special clause.

Another case where resolution does not increase the problem 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 [151]. 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.

4.4 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 problem. Iwama showed that combining this idea with inclusion-exclusion and careful programming lead to an algorithm which runs in polynomial average time when $p > \sqrt{(\ln m)/n}$ [245].

If the sum of the area of all the 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 has a tendency to eliminate short clauses. See [155] for a probabilistic analysis of this idea. No average time analysis has been done.

4.5 Improved Techniques for Backtracking

This section considers some refinements that can be added to the basic backtracking and resolutions techniques. Several of these are similar to techniques that have already been discussed.

Branch merging. This is a complementary way to preclusion. Many search trees possess equivalent branches which may have the same effect. They may be merged to reduce search effort [28, 468].

Search rearrangement. This is also known as *most-constrained search* or *nonlexicographic ordering search*. When faced with the choice of several ways of extending the partial solution, it is more efficient to make the next choice that offers the fewest alternatives [28]. 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 [48, 159, 363, 367, 370, 433]. 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 branching rule and its effect with respect to particular problem instances [222]. Böhm and Speckenmeyer experimented with branching effect with a parallel DP procedure implemented on an MIMD machine [32]. Boros, Hammer, and Kogan developed branching rules that aim at the fastest achievement of the q-Horn structures [35]. Several particular forms of search rearrangement were discussed in Section 4.2.

From 2-SAT to general SAT. In many practical application problems, the constraints in the problems are coded as 2-SAT formulas. In SAT problem formulation, very frequent 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 algorithms. This fast operation can significantly reduce the search space. The truth assignment to the rest variables will be handled with a DP procedure. This idea has been used in SAT solver *Stamm* [62, 61], Gallo and Pretolani's 2-SAT relaxation [62, 61], and Larrabee's algorithm [284, 409]. 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 [62, 61]. Dörre further added a limited amount of forward checking to quickly determine 2-SAT formulas in the Eisele-Dörre SAT solver [62, 61]. In the SAT contest [62, 61] the winning programs with 2-SAT solvers where slightly slower than those without.

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 [171, 173]. 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 [199, 200, 311, 334]. 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 [199, 200, 207], forward checking [207, 207], network-based heuristics [311, 113], and discrete relaxation [199, 200, 269, 388].

Intelligent backtracking The intelligent backtracking is performed directly to the variable that causes the failure, reducing the effect of thrashing behavior. Methods in this category include dependency-directed backtracking [428, 130], revised dependency-directed backtracking [356], simple intelligent backtracking [161], and a number of simplifications [50, 104, 105, 106, 108, 110, 172, 207, 389].

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 which require relatively little storage, this method can be used to increase storage requirements in order to decrease the running time of the program. 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 technique increases the speed at which we process nodes in the tree [28].

Backtrack programming. Much work has focused on developing new programming language for backtracking search. This includes the sequential[8] Prolog programming language [82, 431], Prolog with intelligent backtracking scheme [52, 279, 355], and logic programming [215].

Special-purpose architectures. Special-purpose hardware machines were built to prune search space [199, 200, 322], perform backtracking search, and doing AI computations [459, 458, 461].

Parallel processing. Many parallel processing techniques have been developed to speed up search computation [21, 87, 217, 303, 448, 320, 398, 458, 471].

For a SAT problem, the result of assigning values to variables is that the formula is satisfied or it is not. There are no intermediate cases. Many interesting problems involve a complex formula that gives some result for each assignment. The goal is to find the best assignment. When each variable has a discrete set of possible values the problem is usually solved by a branch and bound algorithm.

Branch and bound. Also known as ordered depth-first search. Select a variable. For each possible value of the variable generate a subproblem and compute some quick to compute upper bound on the quality of the solution of the subproblem. Solve recursively all subproblems 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, 284, 289, 296, 352, 232, 476, 475, 470, 474].

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

4.6 Summary of Complexity Results.

The worst-case time for all known SAT algorithms is exponential in the first power of the problem size. The naive algorithm that tries every variable setting requires time 2^n for n variable problems. For l-SAT, the best known bound on worst-case complexity has been worked down from 1.618^n [330] to slightly below 1.5^n obtained by Kullman and also Schiermeyer [427]. Other work on the topic is given in [174].

As with other NP-complete problems there are no exponential lower bound results for the SAT problem. However, it has been proven that all resolution algorithms need time that is exponential in the first power of the problem size [75, 453]. No such lower bound analyses have been done on splitting-based algorithms.

The analysis of particular splitting algorithms indicates that their performance is quite different from that of resolution algorithms. Even simple backtracking [47] uses polynomial under conditions where all resolution algorithms use exponential average time. The results for the fastest analyzed algorithms suggest that the important factors for predicting the average running time are quite different for these two types of algorithms. The most important factor for resolution type algorithms is the relation between the number of variables and the number of clauses. When the number of clauses is greater than a constant times the number of variables, the average time is exponential for a wide range of clause lengths. When the number of clauses is less than the number of variables to the 1/3 power (times some logarithmic factors), the average time is polynomial for any clause length [151, 371]. The average clause length is a factor of secondary importance, but for average clause lengths that are near zero or vary large the best resolution algorithms run rapidly [151, 366].

For the best splitting algorithms the most important factor is the probability that the problem has a solution [371]. If this probability is near 0 or 1, the average time needed to solve random average l-SAT problems is small. Only when the average clause length is within a logarithm factor of the clause length, that results in the probability of a solution being 1/2, is the time needed greater than a polynomial function of the number of variables. The number of clauses is of secondary importance. When the number of clauses is less than the logarithm of the number of variables, the best splitting algorithms are fast [371].

The use of probe order results in an algorithm that is fast when the probability is either small or large (for problems with m clauses and n variables the probability should not be too near $(\ln m)/n$; this is near the boundary in parameter space between the region where solutions are common and where they are rare) and the special treatment of variables with few occurrences results in an algorithm that is fast when the ratio of clauses to variables is small. Both of these algorithms find some but not all of the solutions for the problem. It is believed that stopping at the first solution would greatly reduce the size of the region where the analyses suggest that the algorithms are slow, but so far it has been impossible to calculate the effect of stopping when the first solution is found.

The algorithms that have had their average time computed are not as complex as some of those whose speed as been measured, but all analyzed algorithms are slow even on average l-SAT problem models when m/n is large and the probability is chosen so that the average number of solutions per problem is constant. This suggests that average l-SAT problems may also be difficult when the average clause length is large. (Few experimental measurements have been made in this region of parameter space.)

5 Local Search

Local search procedures are a major class of the discrete, unconstrained optimization procedures that can be applied to a discrete search space. Using local search for the SAT problem, the number of unsatisfiable *CNF* (or satisfiable *DNF*) clauses is formulated as the value of the objective functions, so the SAT problem is transformed into a discrete, unconstrained minimization problem with an objective function [186, 190, 191, 196, 347, 408].

In this section, we give a brief overview of the local search approach and then describe several major local search methods for SAT problem.

5.1 Local Search

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. Local search can be very efficient in favorable cases. However, local search algorithms only work for problems that have solutions, and even then there is no guarantee that they will solve the problem.

Given a minimization (maximization) problem with objective function f and feasible region R, a typical local search procedure requires that, with each solution point $x_i \in R$, there is a predefined *neighborhood* $N(x_i) \subset R$. Given a current solution point $x_i \in R$, the set $N(x_i)$ is searched for a point x_{i+1} with $f(x_{i+1}) < f(x_i)$ ($f(x_{i+1}) > f(x_i)$). If such a point exists, it becomes the new current solution point, and the process is iterated. Otherwise, x_i is retained as a *local optimum* with respect to $N(x_i)$. 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 need only specify the neighborhood and the randomized procedure for obtaining a feasible starting solution.

Since only one search path is followed, the procedure is fast if the path is short. Since the objective function is a small integer, the path will be short if one is so lucky as to obtain improvement at every step of the path. The main weakness is that the procedure may become stuck at a locally optimum configuration (i.e., a local minima). Even if the procedure does not become stuck, the path can be very long if there are many steps with no improvement. (One could consider only paths where every step leads to an improvement, but this leads to a procedure that becomes stuck much more often. Such a procedure gets stuck in flat places as well as at local minima.)

Many search techniques, such as statistical optimization [95, 402], simulated annealing [267], stochastic evolution [394], and conflict minimization [325, 416, 422], 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 [190, 191, 373, 418, 422].

5.2 A *CNF* Local Search Model

Most discrete local search procedures were developed based on a discrete, unconstrained optimization model, the *SAT1 model* [186, 190, 191, 196]. In this model, the truth values assigned to the variables are defined as:

$$x_i = \begin{cases} 1 & \text{if the variable is } True \\ -1 & \text{if the variable is } False \end{cases} \tag{3}$$

The objective function in *SAT1* model is defined as:

$$F(\mathbf{x}) = \sum_{i=1}^{m} C_i(\mathbf{x})|_{C_i=false} \tag{4}$$

$F(\mathbf{x})$ gives the number of unsatisfiable clauses as its objective value.

The *SAT1* model is an unconstrained optimization model for *CNF* formulas. In Section 5.4, we will give a dual model of the $SAT1$ model, i.e., the *SAT4* model, which is an unconstrained optimization model for *DNF* formulas.

5.3 *CNF* Local Search Procedures for the SAT Problem

A Basic Sequential Local Search Procedure. A typical local search procedure for the SAT problem is shown in Figure 10. It consists of an initialization stage and a search stage. In this procedure, at the beginning of search, a SAT problem instance is generated. An initial solution point is chosen. The number of unsatisfiable clauses is computed and is assigned as the value of the objective function. During a single iterative search step, the test to see if the objective function can be minimized is performed by function *test_swap*(). If *test_swap*() returns *true*, the minimization operation is performed by procedure *perform_swap*() and then *evaluate_object_function*() updates the objective function.

The procedure terminates when a solution of the SAT instance is found, i.e., the objective function is reduced to zero. In practice, before the objective function reduces to zero, the procedure may become stuck at a local minima. In most local search procedures, one or more local handlers that improve procedure's convergence performance are added. As is common with most probabilistic local search procedure [91], $SAT1$ does not guarantee termination, although it often exhibits satisfactory performance and robust behavior. If the procedure falls into the local minima points, an unsuccessful attempt occurs and the procedure will call the local handlers.

Random Noise. If the local search procedure gets stuck at a local minimum, further progress may be achieved by using some sort of perturbation method 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 [187]. In simulated annealing, a search process occasionally moves up rather than down in the search space, with large up moves being less likely than small ones. The probability of large up moves is gradually reduced as the search progresses.

```
procedure SAT1.0 ()
begin
    /* initialization */
    get_a_SAT_instance();
    x_0 := select_an_initial_point();
    F(x) := evaluate_object_function(x_0);

    /* search */
    k := 0;
    while F ≠ 0 do
    begin
        for each variable i := 1 to n do
            /* if swap(x_i, ¬x_i) does not increase F */
            if test_swap(x_i, ¬x_i) then
            begin
                x_{k+1} := perform_swap(x_i, ¬x_i);
                F(x) := evaluate_object_function();
            end;
        if local then local_handler();
        k := k + 1;
    end;
end;
```

Figure 10: **SAT1.0:** a local search procedure for the SAT problem.

A variety of local handlers have been designed for use in the local search algorithms [196, 191]. In $SAT1.0$ procedure [186, 196, 190, 191], a local handler that may randomly negate the truth values of up to n variables is used. The basic idea is to generate 'random exchanges' in some current solution points, when the procedure was stuck on a local minima, accepting a modified point as a new current solution not only if the value of the cost function is better but also if it is worse.

Instead of using simulated annealing which requires excessive floating point computation for Boltzmann probability distribution, in the $SAT1.0$ procedure, a definite selection of a new search point is used. This local handler has effectively improved the convergent performance of the $SAT1.0$ local search procedure.

Tunneling Heuristic. The local handlers and their calling conditions have significant effect on the performance, e.g., the run time and average run time, of a local search procedure for satisfiability testing. The conditions for calling local handlers differ from algorithm to algorithm. In the $SAT1.2$ procedure [196, 191], if the objective function does not increase, the local handler is called. In the $SAT1.3$ procedure [196, 191], if the objective function does not increase or the objective function is greater than zero for some iterations, the local handler is called. In these two algorithms, the condition that "objective function does not increase" includes two cases, the objective function reduces its value or the objective function does not change its value, after a local search step.

Instead of making random moves in the vertical direction in search space as in simulated annealing, whenever a local minimum is encountered, we tunnel through the rugged terrain structure of the search space in a horizontal direction, moving from one local minimum valley to another local minimum valley in an attempt to locate a better locally optimal solution. As illustrated in Figure 11, a tunnel can be thought of as a short-cut passing through a mountain separating points of equal elevation. Thus, a tunneling operation can be effectively used to search a region with many local minima. Whenever a local minimum

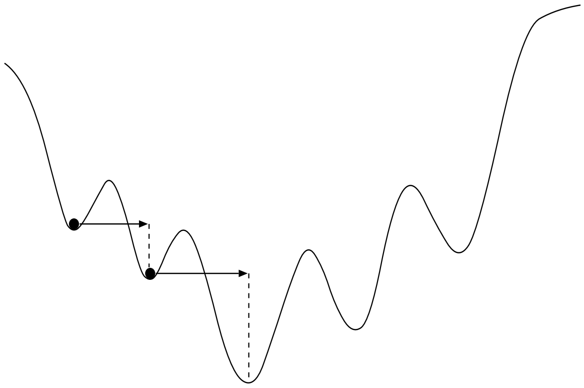

Figure 11: Tunneling

is encountered, a tunnel is made through a mountain to a neighboring valley as long as this does not change/increase the objective function.

Tunneling heuristic was first used in the $SAT1$ algorithm family as early as in 1988. It has been proven effective in solving the SAT problem. The behavior of the local search algorithms with tunneling heuristic illustrates the fact that seemingly innocuous changes in an optimization routine can have a surprisingly large effect on its performance.

Greedy Local Search Algorithms. Local search proceeds by taking *any* solution point that decreases the objective function. Among many neighboring solution points, local search does not take into account its neighbors' relative performance to the objective function. A *greedy* algorithm selects the *best neighbor* that yields the minimum value to the objective function and take this *best neighbor's direction* as the descent direction of the objective function. In a real search space, many continuous optimization algorithms can efficiently locate this best neighbor solution [190, 194].

Greedy local search algorithms search for the best neighbor solution in the discrete search space. This requires, during each iteration, that the *evaluate_object_function* procedure examines all the possible moves and select one that gives the maximum descent [190, 194]. Greedy local search is a special case of the coordinate descent in the real space.

Parallel Local Search. Several parallel algorithms and VLSI architectures have been developed to accelerate satisfiability testing [199, 200, 191, 424]. Depending on implementations, there are a number of alternative ways to group "variables" or "clauses" together and evaluate them simultaneously. In the $SAT1.0$ algorithm, the most frequently used part of computation is function *evaluate_object_function*(). It takes $O(ml)$ time to update the objective function. The execution of *evaluate_object_function* can be done in a simple bit-parallel fashion in $O(m)$ time on a sequential machine.

A computer word of a *SUN* computer has 32 bits (some computers such as *DEC Alpha* have 64 bits in a word). In many *CNF* formulas, however, the number of literals is less than 32. So, in the local search algorithms, one can pack all the literals in one clause into the bits of a computer word and therefore, evaluate all the literals in one clause in parallel. Since there are m clauses, it will take *evaluate_object_function* $O(m)$ time to update the objective function. Occasionally, for a clause having large number of literals, it may be broken into more than one computer words and evaluated simultaneously.

This parallel processing method is general for satisfiability testing. It was implemented in a number of bit-parallel local search algorithms, e.g., the *SAT*1.7 algorithm [186, 191].

Complete Local Search Algorithms. Local search algorithms described above 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 a number of complete local search algorithms to test satisfiability as well as unsatisfiability. The basic idea is to combine local search with systematic search, keeping local search efficiency while maintaining the search completeness with systematic search methods [186, 191]. 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. Probe order backtracking is a simplified version of this idea.

Other Related Works. Recently a number of local search procedures have been developed. Selman et al. used a side-walk (a form of tunneling) heuristic in the *GSAT* procedure [408]. Furthermore, they performed random exchanges with those variables that appear in the unsatisfied clauses [407]. They found that this mechanism improved the *GSAT* procedure. Crawford studied a complete local search algorithm [94]. He used weights assigned to clauses to help choose branch variables and variables occurring in heavily weighted clauses were given precedence.

5.4 A *DNF* Local Search Model

Using the well-known DeMorgan laws, we can obtain an unconstrained optimization model, the *SAT4 model*, for *DNF* formulas [186]: The objective function in the *SAT4* model is defined as:

$$F(\mathbf{x}) = \sum_{i=1}^{m} C_i(\mathbf{x}) \tag{5}$$

where

$$C_i(\mathbf{x}) = \prod_{j=1}^{n} Q_{ij}(x_j) \tag{6}$$

where each literal, $Q_{ij}(x_j)$, is a negation of the corresponding literal in the *CNF* formula. A *CNF* formula is true *if and only if* $F(\mathbf{x})$ takes the global minimum value 0 on the corresponding solution $\mathbf{x}^*$.

With $SAT4$ model, 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:

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

A solution to the *CNF* formula, e.g., $\mathbf{x} = \{1, 0, 0, 0\}$, makes the *CNF*($\mathbf{x}$) to be *true* and makes the *DNF* F($\mathbf{x}$) to be *false*.

5.5 *DNF* Local Search Algorithms for the SAT Problem

A number of local search algorithms were developed for *DNF* formulas. Except the objective function is defined and evaluated differently, they have similar structures as in those *CNF* local search algorithms. In the $SAT1.4$ algorithm [186], 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 be zero. Experimental results indicate that the *DNF* local search algorithm is much faster than the *CNF* local search algorithms.

Similar to previous discussion of *CNF* local search algorithms, a number of *DNF* local search algorithms including parallel *DNF* local search algorithms, directed *DNF* local search algorithms, and complete *DNF* local search algorithms have been developed [186]. In Section 11, we will show the performance of a parallel *DNF* local search algorithm, i.e., the $SAT1.8$ algorithm.

5.6 Historical Notes

There are two historical lines to the development of local search algorithms for the SAT problem (Figure 12): one from operations research (OR) area and one from AI area. A clear historical line suggests the future development.

From an OR point of view, local search is a primitive form of continuous optimization techniques in the real search space. Line search is the backbone of many optimization algorithms in linear and nonlinear programming. Given a solution point $\mathbf{x}_k$ and an objective function $f(\mathbf{x})$ to be minimized, a direction vector, $\mathbf{d}_k$, and a suitable step size, α_k, are sought. A new solution point $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$ is found that reduces the value of the objective function. The process is repeated for all input variables. Find the step size α_k involves solving the problem to minimize $f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)$, which is a one-dimensional search problem in the variable α_k.

Both local search and line search aim at solving a minimization (maximization) problem using local neighborhood search approach. They differ in the way the objective function is reduced: a conventional local search accepts a solution point that reduces the value of the objective function (assume there is no up-move that increases the objective value); a line search admits a solution point that yields the *minimum*

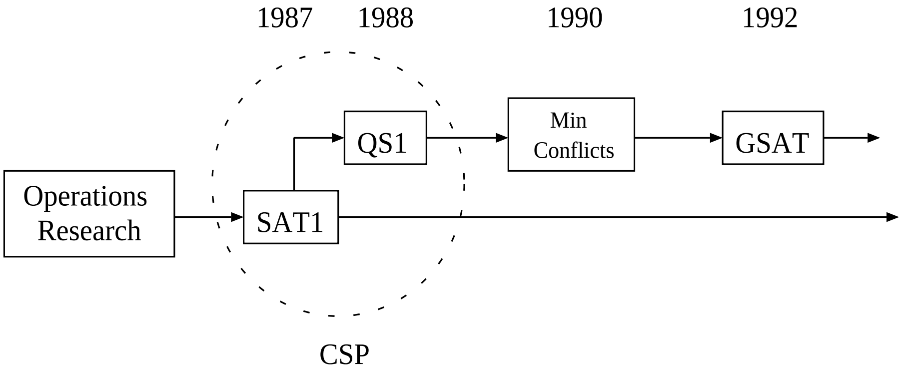

Figure 12: Two main historical lines to the development of local search algorithms for the SAT problem.

value to the objective function. In a broad sense, local search can be viewed as a nondirective line search. This was the original idea behind the early development of the $SAT1$ local search procedures in 1987. From algorithm space perspective (Figure 1), line search and general optimization technique shed light on the early local search method for the SAT problem [186].

The second line of development stemed directly from the close relationship of CSP (constraint satisfaction problem) and SAT problems. The n-queens problem is a typical benchmark problem in CSP. If one can find an efficient search (not an analytical) algorithm for the n-queens problem, then that algorithm can be translated to an efficient efficient algorithm for the SAT problem.

Analytical solutions exist for the n-queen problem with n greater or equal to 4 [2, 9, 24, 140, 219, 382]. They consist of a restricted subset of solutions [9]. If one formulates the n-queen problem as a CSP, we can use backtracking search to search for any general solution. In practice, backtracking search is too slow to solve the n-queens problem for n larger than 96 [433]. A breakthrough point in this direction was due to the local search algorithm for the n-queens problem.

Recent surveys indicate that the first local search algorithms for the n-queens problem were developed in 1988 by Sosič and Gu [277, 278, 256, 391]. Four versions of the algorithms were developed. The *Queen_Search1* ($QS1$) algorithm is an $O(nlogn)$ time local search algorithm [423, 416, 417]. The $QS2$ is a near linear time local search algorithm with an efficient random variable selection/assignment [418]. The $QS3$ is a near linear time local search algorithm with efficient pre- and random variable selection/assignment [418]. The $QS4$ algorithm is a linear time local search algorithm with efficient partial and random variable selection/assignment [419, 422]. Compared to a primitive local search algorithm, these partial and random variable selection/assignment heuristics have significantly increases the search efficiency by orders of magnitude. The $QS4$ algorithm, for example, was able to solve 3,000,000 queens in few seconds. Later, Minton et al. independently reported a similar local search algorithm to the n-queen problem [325].

Following recent work of solving large scale n-queen problems, Selman, Levesque and Mitchell reported the empirical results of a local search algorithm, i.e., the $GSAT$ algorithm, for the SAT problem [408]. The $GSAT$ algorithm is a version of the early $SAT1$ algorithms developed in 1987. Selman has recently addressed [406] that the local search solutions to the large size n-queen problem was "the original impetus" to the *GSAT* algorithm [408].

[To be updated]

6 Global Optimization

Local search proceeds by taking *any* solution point that decreases the value of the objective function as the next solution point. There are many neighboring solution points at each stage of the search. Local search does not take into account its neighbors' relative performance to the objective function. A natural heuristic is to select the *best neighbor point* that yields the minimum value to the objective function. In a real search space, many global optimization algorithms make use of the best-neighbor heuristics.

Using a global optimization method to solve the SAT problem, special models are formulated that transform a discrete SAT problem on Boolean space $\{0,1\}^n$ into a continuous SAT problem on real space E^n. Thus, the discrete decision problem is transformed into an unconstrained, global optimization problem which can be solved by existing global optimization methods [186, 189, 194].

In this section, we briefly overview the global optimization approach. Then we give some transformation models and optimization algorithms for the SAT problem.

6.1 Global Optimization

Global optimization is concerned with the characterization and computation of global minima and maxima of unconstrained nonlinear functions and constrained nonlinear problems [146, 147, 233, 348]. 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) the objective function subject to the constraints imposed upon the values of the decision variables by the problem.

Most optimization algorithms are designed as an iterative refinement process. Typically, in seeking a vector that solves the optimization problem, an initial vector $\mathbf{y}_0$ is selected and the algorithm generates an improved vector $\mathbf{y}_1$. The process is repeated and an even better solution $\mathbf{y}_2$ is found. 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 $\mathbf{y}^*$. This is the basic process of global optimization algorithms for the SAT problem.

6.2 *UniSAT*: Universal SAT Problem Models

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

In the *UniSAT* models, using the universal DeMorgan laws, all Boolean $\vee$ and $\wedge$ connectors in *CNF* formulas are transfromed 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 $F(\mathbf{x})$ 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 problem into an unconstrained global optimization problem:

$$\min_{\mathbf{y} \in \mathbf{E}^n} f(\mathbf{y}) \tag{7}$$

where

$$f(\mathbf{y}) = \sum_{i=1}^{m} c_i(\mathbf{y}). \tag{8}$$

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

$$c_i = \prod_{j=1}^{n} q_{ij}(y_j). \tag{9}$$

In the *UniSAT5* model [186, 190, 194]

$$q_{ij}(y_j) = \begin{cases} |y_j - 1| & \text{if literal } x_j \text{ is in clause } C_i \\ |y_j + 1| & \text{if literal } \bar{x}_j \text{ is in clause } C_i \\ 1 & \text{if neither } x_j \text{ nor } \bar{x_j} \text{ is in } C_i \end{cases} \tag{10}$$

and in the *UniSAT7* model [186, 189, 190, 194]:

$$q_{ij}(y_j) = \begin{cases} (y_j - 1)^2 & \text{if } x_j \text{ is in clause } C_i \\ (y_j + 1)^2 & \text{if } \bar{x}_j \text{ is in clause } C_i \\ 1 & \text{if neither } x_j \text{ nor } \bar{x_j} \text{ is in } C_i \end{cases} \tag{11}$$

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

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

Clearly, $F(\mathbf{x})$ is 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 similar model to *UniSAT5* was proposed independently in the neural network area [253]. A significant difference between the *UniSAT5* and neural network model is their efficiency and practical applicability. The neural network model can only be handled by traditional nonlinear programming methods which were reported extremely slow [253]. The *UniSAT5* can be easily solved in conjunction with the local search approach by simple discrete accounting techniques [186, 194].

The *UniSAT* models transform the SAT problem from a discrete, constrained decision problem into an unconstrained global optimization problem [186, 189, 190, 194]. 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 problem. A *CNF* $F(\mathbf{x})$ is 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*

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

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 to the SAT problem corresponds to a set of global minimum points of the objective function. To find a *true* value of $F(\mathbf{x})$ is equivalent to find a *false* value, i.e., 0, of f($\mathbf{y}$).

The translation of SAT problem into a nonlinear program is quite different from the integer program approach. In the integer programming approach, one views the SAT problem as a 0/1 integer programming problem and tries solving it as a linear programming problem [29, 224, 225, 249, 262, 260, 469]. 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 problem. If the rounded off values do not correspond to a solution, one computes another solution of the linear programming problem.

6.3 A Global Optimization Algorithm for the SAT Problem

Many families of unconstrained global optimization algorithms for the *UniSAT* problem have been developed [186, 189, 194]. A typical global optimization algorithm, i.e., the *SAT6.0* algorithm, is shown in Figure 13. To start, procedure *obtain_a_SAT_instance*() initializes a SAT problem instance or it obtains a practical SAT problem instance. An objective function, f, is formulated according to a given *UniSAT* model. The SAT problem 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 the 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 dimension or in up to n dimensions. Methods able to optimize 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

```
procedure SAT6.0 ()
begin
    /* initialization */
    obtain_a_SAT_instance();
    y_0 := select_an_initial_solution();
    f(y_0) := evaluate_object_function(y_0);

    /* search */
    k := 0;
    while  not(solution_testing()) do
        for some y_{i(k)}s ∈ y_k
        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 13: **SAT6.0:** A general global optimization algorithm for the satisfiability problem.

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 the 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 problem. 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 of the effective local handlers in the $SAT6$ algorithm 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 [186, 189, 194]. These include the basic algorithms, steepest descent methods, modified steepest descent methods, newton 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 starting initial points, scramble search space, formulate higher order local handlers, deflect descent directions, utilize parallelism, and implement hardware architectures to speed up computations.

6.4 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 [309], Gu has recently given a simple algorithm, the $SAT14.5$ algorithm [194, 193], for the *UniSAT5* problem (see Figure 14). The kernel of the $SAT14.5$ algorithm is a discrete *minimizer* which minimizes the 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 the 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, the 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* which are independent to 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 the function f as follows: if $a_j \geq b_j$ then set y_j equal to 1; otherwise set y_j equal to -1.

```
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 14: **SAT14.5:** A global optimization algorithm for the *UniSAT5* problem.

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 the *SAT14.5* algorithm, if the objective function f can no longer be reduced *or* after $b' \log n$ (b' is a constant, see [194, 193]) iterations of the *while* loop f is still at least one, then the *local-handler* is called.

6.5 A Continuous Global Optimization Algorithm

Based on a continuous coordinate descent method [309], Gu, Huang and Du have recently given a $SAT14.7$ algorithm for the *UniSAT7* problem on E^n [194]. For the objective function described in the *UniSAT7* problem model, if only one variable, e.g., x_i, is selected for optimization, then

$$F(x_i) = a_i(x_i - 1)^2 + b_i(x_i + 1)^2 + c_i \tag{12}$$

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

$$x_i = \frac{a_i - b_i}{a_i + b_i}. \tag{13}$$

6.6 Complete Global Optimization Algorithms

The $SAT14.5$, $SAT14.6$, and $SAT14.7$ algorithms are *incomplete* algorithms. In order to achieve computing efficiency and search completeness, in $SAT14.11$ to $SAT14.20$ algorithms, we combine global optimization algorithms with backtracking/resolution procedures and make them complete [186, 194]. Therefore, they are able to verify satisfiability as well as unsatisfiability. Figure 15 gives a typical backtracking global optimization algorithm.

For SAT problem *per se*, backtracking is able to verify unsatisfiability quickly for certain classes of problems but is slow when it comes to verifying satisfiability, as all possible resolutions need to be tried out before concluding that the inference relation holds or that the problem is satisfiable. From our experience, a *combined* global optimization algorithm with backtracking/resolution procedures would perform well for certain classes of satisfiable and unsatisfiable problem instances.

Recently some researchers investigated the number of solutions in the SAT problem instances. Extending Iwama's work [245], Dubois gave a combinatorial formula computing the number of solutions of a set of any clauses [133]. He and Carlier also studied the mathematical expectation of the number of solutions for a probabilistic model [134]. For an incomplete SAT algorithm, the number of solutions has 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 [191, 194].

```
Procedure SAT14.11 ()
begin
    /* initialization */
    get_a_SAT_instance();
    x_0 := select_an_initial_point();
    f := evaluate_object_function(x_0);

    /* backtracking with global optimization */
    x* := backtracking(x_0);
end;

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

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

Figure 15: **SAT14.11:** a complete global optimization algorithm with backtracking.

6.7 Convergence Property and Average Time Complexity

Gu, Gu and Du [197] 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* problem model. They prove that, subject to certain conditions [309], 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 $\mathbf{H}(\mathbf{y})$, respectively.

From these convergence properties, Gu, Gu, and Du roughly estimate that, subject to certain conditions [309], 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 [193]. It shows that, the $SAT14.5$ 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 $SAT14.5$ algorithm is

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

Clearly, the $SAT14.5$ algorithm can give an answer to an unsatisfiable *CNF* formula in $O(2^n)$ time.

7 Integer Programming Method

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

7.1 An Integer Program Formulation for SAT

In order to represent the SAT problem in the framework of mathematical programming, we identify logic value *true* with the integer 1 and *false* with -1. Similar in *UniSAT* models (Section 6.2), all Boolean $\vee$ and $\wedge$ connectors are transformed into $+$ and $\times$ of ordinary addition and multiplication operations, respectively. Using a standard transformation, the ith clause $C_i(\mathbf{x})$ is transformed into a linear inequality [262, 260]:

$$\sum_{j=1}^{n} q_{ij}(w_j) \geq 2 - |C_i|, \tag{14}$$

where $|C_i|$ denotes the number of literals in clause C_i and

$$q_{ij}(w_j) = \begin{cases} w & \text{if literal } x_j \text{ is in clause } C_i \\ -w & \text{if literal } \bar{x}_j \text{ is in clause } C_i \\ 0 & \text{if neither } x_j \text{ nor } \bar{x}_j \text{ is in } C_i \end{cases} \tag{15}$$

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*

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

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 the SAT problem as: finding $w_j = \pm 1$, such that

$$\begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} \geq \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \tag{16}$$

or

$$\begin{pmatrix} -1 & 1 & 0 \\ 1 & -1 & -1 \\ 0 & -1 & -1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{17}$$

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).[9] 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 problem. If the rounded off values do not correspond to a solution, adds a new constraint and computes a solution of the modified linear programming problem. So far most methods developed to solve the integer programs for the SAT problem are indirectly working on the corresponding integer linear programs.

Researchers have observed that the optimal integer-programming solution is not obtained by *rounding* the linear-programming solution. 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 the SAT problem, it is not sufficient simply to round linear-programming solutions.

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

7.2 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

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

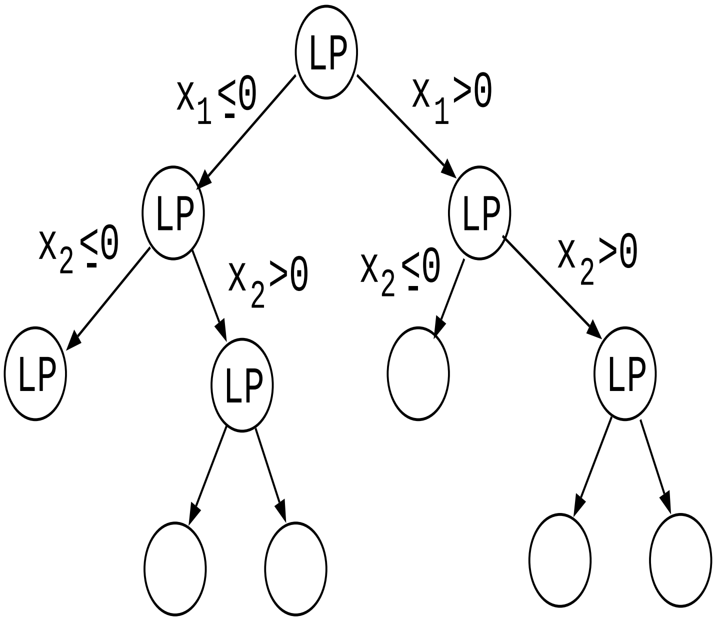

Figure 16: An example of a branch-and-bound tree.

with some sophisticated implementations of Dantzig's Simplex method, such as MINOS [333], or some variations of Karmarkar's interior point method [264].

Hooker early reported that by solving a linear programming of SAT one frequently produces an integer solution [225]. Kamath et al. used MINOS 5.1 to solve linear programming relaxation [262, 260]. They tried some small SAT problem instances and found that the simplex method failed to find integral solutions to the linear programming relaxations in majority of instances tested.

7.3 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 16. The method starts with the fractional solution given by its corresponding LP relaxation. Then a variable of fractional solution is selected. For example, x_1, 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

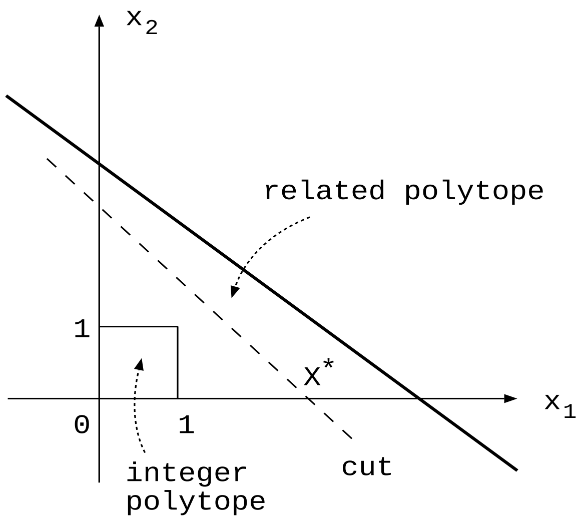

Figure 17: An illustration of the cutting-plane method.

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 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 16. In this way, we implicitly exhausted 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 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.

7.4 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 the 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 17). 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

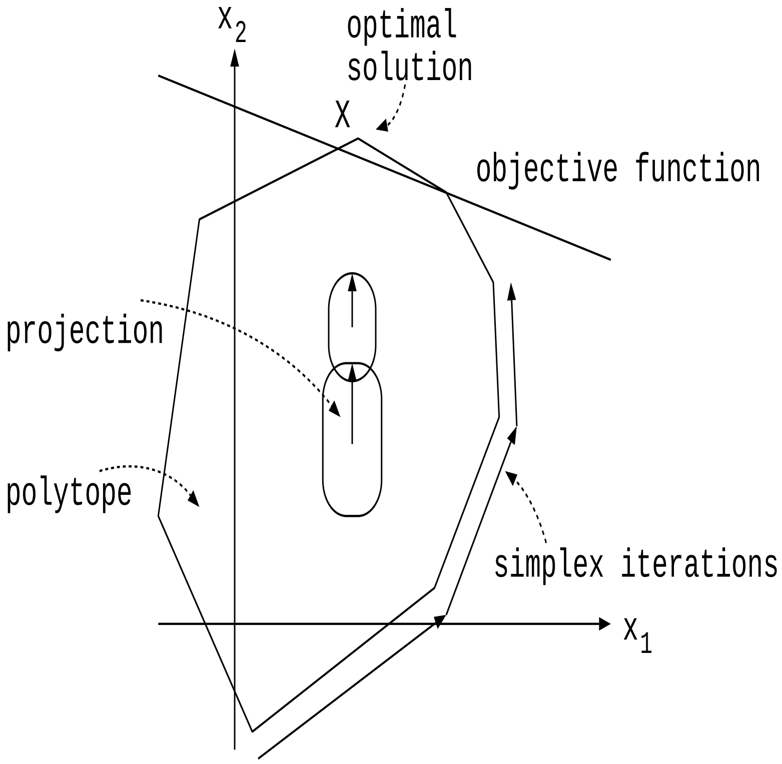

Figure 18: The basic ideas of Simplex method and Karmarkar's method.

excluding any feasible integer points. After the cut, the resulting linear program is solved again. If the optimal 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.

7.5 Interior Point Method

The most important advance in linear programming solution techniques was recently introduced by Karmarkar [264]. As shown in Figure 18, 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 yields an NP-complete complexity, Karmarkar's algorithm has been proven to be a polynomial algorithm.

To apply the 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 project. Therefore it is an incomplete algorithm.

7.6 Improved Interior Point Method

It is expected that a sequence of interior points

$$w^{k+1} = w^k + \alpha \triangle w^* \tag{18}$$

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 the SAT problem instances. If the potential function is well represented by the quadratic approximation around the given point, then if we move along the Newton drection 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 has recently given an improved interior point algorithm [411]. In his algorithm, the step size α is determined by a Golden-Section search [309]. Experiments show significant improvements on the Karmarkar's algorithm.

8 Special subclasses of the SAT problem

Many investigators have identified and explored certain subclasses of the SAT problem that are known to be solved in polynomial time. There are two 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 the SAT problem. If so, a special, fast algorithm can be brought to bear on the formula. Second, study of such subclasses reveals, in part, the nature of easy SAT problem instances. On the other hand, studies of random problems indicate that these known classes contain only a small freaction of the problems that can be solved rapidly. Below, we consider some of the more notable subclasses. When we say apply *unit resolution* we mean apply the unit clause rule to exhaustion.

8.1 2-SAT

Frequently in practical applications, many constraints can be coded as 2-SAT clauses (that is, disjunctions of two literals). An important heuristic for SAT problem solving is to first solve the 2-SAT clauses with fast linear-time algorithms such as described in [16, 139]. Adding a 2-SAT solver to, say, a Davis-Putnam-Loveland procedure can have a significant effect on reducing the search space. This idea has been tried in SAT solver *Stamm* [62, 61], Gallo and Pretolani's 2-SAT relaxation [62, 61], and Larrabee's algorithm [284, 409]. Other variations include Eisele's SAT solver, which weights 2-SAT clauses higher than others [62, 61], and Dörre's addition of a limited amount of forward checking to quickly determine 2-SAT formulas in the Eisele-Dörre SAT solver [62, 61]. In the SAT contest [62, 61] the winning programs with 2-SAT solvers where slightly slower than those without.

8.2 Horn and Extended Horn Formulas

A CNF instance 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 [129, 243, 404].

The class of extended Horn formulas was introduced by Chandru and Hooker [70] who were looking for conditions under which a relaxation of Linear Programming could be used to find solutions to propositional formulas. A theorem of Chandrasekaran [73] 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 algorithm. 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 [436], 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 [70] 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 repeated unit resolutions (assigning values of 0 or 1 as needed), setting values to 1/2 when no unit clauses remain, and rounding the result by a matrix multiplication [70]. These algorithms 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.

8.3 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 [88] 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 [441].

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 [88]. 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 [88]. 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 [88].

8.4 Single-Lookahead Unit Resolution

This class was developed as a generalization of other classes including Horn, extended Horn, simple extended Horn, and balanced formulas [401]. 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 subformulas 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 [444], 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 Davis-Putnam-Loveland 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 2SAT algorithm, because the 2SAT 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.

8.5 q-Horn Formulas

This class of propositional formulas was developed by Boros, Crama, Hammer, Saks, and Sun in [38] and [37]. We choose to characterize the class of q-Horn formulas as a special case of monotone decomposition of matrices [442, 444]. 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 a $(0, \pm 1)$-matrix, 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 [444] 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 [129, 243, 404] 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 colinear 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 [37] is that a q-Horn formula can be recognized in linear time. See [36] 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 [38]. 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:

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

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

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 greater than $1 + \epsilon$, for any $\epsilon > 0$, is NP-complete.

More information on the subject of q-Horn formulas will appear in [444].

8.6 Renamable Formulas

Suppose clauses of a CNF formula F are constructed from a set V of variables and let $V' \subset V$. Define $switch(F, V')$ to be the formula obtained as follows: for every $v \in V'$, reverse all unnegated occurrences of v in F to negated occurrences and all negated occurrences of v to unnegated occurrences. For a given formula F, if there exists a $V' \subset V$ such that $switch(F, V')$ 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 [295] and [17].

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 [401].

8.7 Formula Hierarchies

Classes of formulas presented above can be solved in polynomial time. Some sets of clauses not falling into these classes may be reduced to "equivalent" formulas in a known polynomially solvable class. 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 problems 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 [165]. 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 subformula is at stage $i-1$, and when setting v to *false*, a subformula 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) [55].

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 [165].

For other information on such Hierarchies see, for example, [98, 166].

8.8 Pure Implication Formulas

Pure implication formulas are defined recursively as follows:

1. A variable is a pure implication formula.
2. If F_1 and F_2 are pure implication formulas then $(F_1 \rightarrow F_2)$ is a pure implication formula.

Eliminating parentheses on right to left associativity, a pure implication formula can be written $F_1 \rightarrow F_2 \rightarrow ... \rightarrow F_p \rightarrow z$ 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 [216]; 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. 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.

9 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 been proven to be effective for certain classes of satisfiability problems.

9.1 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., the SAT problem must be expressed as a product of sums of literals. The *CNF* formula is a canonical formula used in most theoretical studies but it 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 [195]. 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 new variables. This will result in clause form representation with substantially larger sizes [174, 357]. While this is not critical in complexity theory, it will have serious impact to solve the 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 [201].

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

9.2 Binary Decision Diagram (BDD)

Ordered Binary Decision Diagrams (OBDDs) [53, 54] is an efficient representation and manipulation method for arbitrary Boolean functions. This representation is defined by imposing restrictions on the Binary Decision Diagrams (BDD) representation introduced by Lee [290] and Akers [8], 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 those problems where one needs to consider every solution to the problem, such as cases where one must to search for an optimal solution. Although the OBDD representation of a function may have size exponential in the number of variables, in practice, many useful functions have more compact representations.

A binary decision diagram 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 formula 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 19, 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 formula F, the

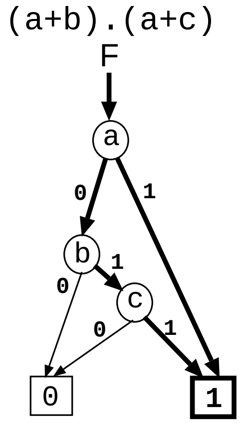

Figure 19: A simple BDD example for $F = (a+b)\cdot(a+c)$.

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 19). Since the early introduction of Binary Decision Diagrams, several extensions have been proposed to reduce the BDD size in practical applications. In an ordered BDD [53, 54], 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 [313] yields small size BDDs for practical Boolean instances. A reduced ordered binary decision diagram is an ordered BDD where each node represents a unique logic function. Bryant showed that the reduced ordered BDD is well-defined and it is a canonical representation of the Boolean function, i.e., two functions are equivalent if their reduced ordered BDDs are isomorphic [53, 54].

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. Binary Decision Diagrams (BDD) 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 SAT problems. For larger size SAT problems, it was suggested that the problem be partitioned into a set of smaller subproblems before applying the BDD algorithms. This approach works well for asynchronous computer circuit design problems [198, 375].

9.3 The *Unison* Algorithms

Based on total differential of a Boolean function, the *Unison* algorithm is capable of evaluating arbitrarily large, complex Boolean functions [195, 425, 424]. 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 the evaluation of 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:

$$dF = F_x dx \oplus F_y dy \oplus F_{xy} dx dy, \tag{19}$$

where $\oplus$ is the *Exclusive-OR* operation [438]. 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 (19), the total differential dF is:

$$dF(G(\vec{x}), H(\vec{y})) = F_x \, dG(x_1, x_2) \oplus F_y \, dH(y_1, y_2) \oplus F_{xy} \, dG(x_1, x_2) \, dH(y_1, y_2). \tag{20}$$

It can be observed from (20) 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 (20) to dG, dH, and their dependent variables, the total differential dF can be evaluated based only on total differentials of the independent variables (see Figure 20).

The *Unison* algorithm works in two phases: initialization and evaluation. The initialization phase computes partial derivatives which 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

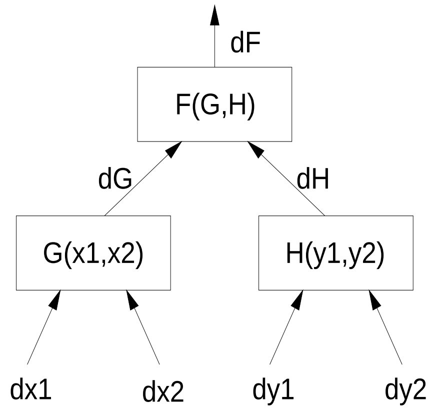

Figure 20: The relation between total differentials.

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 [424], 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 [424].

The *Unison* architecture is built up with a network of multiple universal Boolean elements (UBEs). The connection topology of the *Unison* network specifies the structure of Boolean function that is being evaluated by the *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 the 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 the SAT/CSP problems. In the *Unison* architectures, there is essentially no limits 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 [425, 424].

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 the SAT problem.

9.4 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 problem 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 [187, 201, **?**]. The idea of multispace search was derived from principles of nonequilibrium thermodynamic evolution that structural changes are more fundamental than quantitative changes and evolution depends on the growth of new structure in biological system rather than just information transmission. A search process resembles the evolution process and the structural operations are important to improve the performance of traditional value search methods [187, 201, **?**].

In multispace search, any active component related to the given problem 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 problem 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 21, it also walks across the variable space and other active spaces, dynamically changes the problem structure in terms of variables, parameters, and other components, and systematically *constructs* a sequence of structured, intermediate problem instances. Each intermediate problem instance is solved by an optimization algorithm and the solution found is used as the initial solution to the next intermediate problem instance. By interplaying value optimization with structured operations, multispace search incrementally constructs the final solution to the search problem through a sequence of structured intermediate problem instances. Only at the *last* moment of the search, the reconstructed problem structure approaches the original problem structure, and thus the final value assignment represents the solution to the given search problem.

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 problem instance during each individual

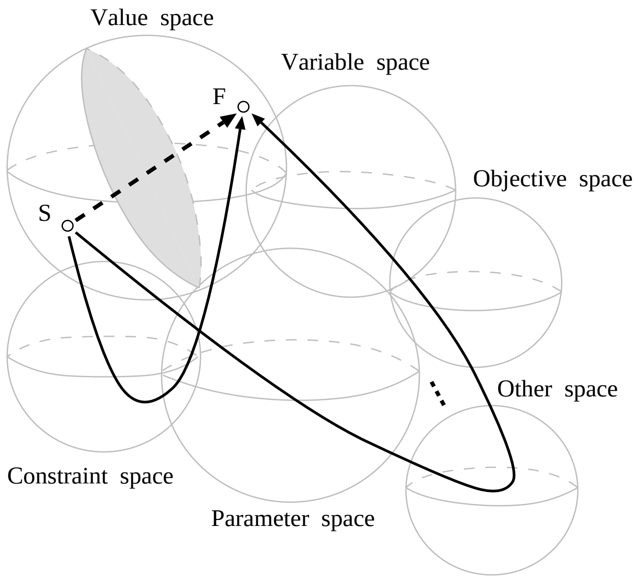

Figure 21: In the value space, a traditional search process (dashed line) cannot pass a "wall" of high cost search states (hatched region). It fails to reach the final solution state, F. A multispace search process (solid lines) scrambles across different search spaces. It could bypass this "wall" through the other search spaces.

search phase. According to the active event in the scrambling schedule [192], the search process enters a specified search space and performs structural operations to the intermediate problem structures, followed by a traditional value search that optimizes the constructed intermediate problem instance. The resulting intermediate solution is then used as the initial problem instance to the next phase of multispace search.

The major structural operations [192] in multispace search include multispace scrambling, extradimension transition (e.g., air bridge, real dimension, and extra dimension), search space smoothing, preprocessing (e.g., compression, partitioning problem size, partitioning variable domain, reorganization, and semi-processing), simulated evolution, and value perturbations (e.g., jumping, tunneling, climbing, annealing, and indexing).

In the next two subsections we describe two preprocessing methods in multispace search, i.e., partitioning problem size and partitioning variable domain, for satisfiability testing.

9.5 Partitioning to Reduce Problem Size

Due to excessive computing time, a larger size NP-hard problem is difficult to solve. Partitioning a large problem into a set of smaller subproblems may permit efficient solving of the problem. There are two partitioning methods. Each consists 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 problem 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 a great interest in the automated design and synthesis of asynchronous circuits [78, 286, 301, 455]. 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 a satisfiability problem to satisfy the complete state coding (CSC) constraints, i.e., the SAT-CSC problem [456].

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-CSC 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 [192, 198, 375]. The partitioning preprocessor, at the beginning, compresses 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 subproblems are integrated together and contribute to the solution of the original problem. 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 the binary decision diagrams (BDD). It is able to find an optimal solution to the asynchronous circuit design problem efficiently.

9.6 Partitioning Variable Domains

Variable domain contains the values to be assigned to the variables. The size of variable domain, along with the number of variables, determines the computational complexity of an optimization algorithm. From the theoretical point of view, even a small amount reduction to the variable domain would result in significant computing efficiency improvements. It is, however, difficult to make use of the variable domain reduction techniques in optimization problem solving. Recently Wang and Rushforth have studied the cellular network structures and developed a novel variable domain reduction technique for channel assignment in the mobile cellular radio networks [467, 466].

The rapid growth of the 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 the cellular radio networks. An FCA algorithm assigns frequency channels to the 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 a satisfiability problem. For a given cellular communication system, there are numerous ways to assign a channel to a request call. 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 [467, 466]. 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 a channel assignment problem with a large number of assignments into two separate channel problems with considerably small numbers of assignments, it dramatically reduces the computational complexity and thus the computing efficiency for solving the problem, in addition to the significantly improved solution results. This novel partitioning technique can be applied to solve channel assignment problem with any existing channel assignment algorithms. During numerous channel assignment experiments, this algorithm outperformed all the available algorithms for solving the practical channel assignment problem benchmarks. Experimental evidence suggests that this partitioning approach is both efficient and effective.

9.7 Parallel SAT Algorithms and Architectures

Many parallel SAT/CSP algorithms have been developed. In a recent survey [195], the following parallel algorithms for the SAT problems were discussed:

1987: Parallel Davis-Putnam algorithms
1986: Parallel discrete relaxation chips
1987: Parallel backtracking architecture
1987: Parallel local search algorithm
1989: Parallel interior point method
1990: Parallel, differential, non-clausal inference
1991: Parallel $\alpha\beta$ relaxation
1991: Parallel global optimization
1992: Neural network approach
1993: Multiprocessor local search

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

For the following two reasons, algorithms running on loosely-coupled, multiprocessor parallel computers offer limited performance improvements for solving the SAT problems. First, in the worst case, a SAT algorithm may suffer from the exponential growth of computing time. In order to solve the SAT problem effectively, a computer is expected to provide a much greater number of speedups. This requires the integration of at least a few million of processors in a tightly-coupled manner. This is infeasible in the current computer system integration technology.

Secondly, as the processor gets much faster, the communication overhead among processors in a parallel machine becomes a bottleneck. It may often take 70 % to even 90 % of the total computing time [254]. Ideally one would expect the speedup on a parallel computer to increase linearly with the increasing number of processors. Due to serious off-processor communication delays, after certain saturation point, adding processors does not increase the 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, the speedup is sometimes greater than the number of processors [331]. Variable setting 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 breaks up the tendency to search similar settings at about the same time.

From our experience, tightly coupled parallel computing, which effectively reduces the off-processor communication delays, is a key to the parallel processing of the SAT problem [195]. In order to use a tightly-coupled parallel architecture for SAT computation, one must map a computing structure to the problem structure and must reduce the total number of sequential computing steps through a large number of symmetrical interactions among simple processors [185, 195]. Several different approaches, e.g., special-purpose parallel VLSI architectures [199, 200], bit-parallel programming on sequential machines [186, 191, 425, 424], and tight programming on a parallel computer system, are the promising alternatives in this direction. They are capable of providing a tight mapping between the problem structure and 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 much larger size SAT problems.

10 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 superpolynomial 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 analytic 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 3), 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.

10.1 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. Variables are placed in clauses independently. Thus, it is possible that null clauses or clauses with complementary literals exist in a random formula. Results presented below are asymptotic (that is, they apply when $n, m \to \infty$).

10.1.1 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 [150], 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 [152]. 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 $pn = c\ln(m) : 1 > c \geq 1/2$, $\lim_{n,m\to\infty} m^{1-c}/n^{1-\epsilon} = \infty$, $\epsilon > 0$.

10.1.2 Polynomial-Time Solvable Classes

Many of the special polynomial-time solvable classes discussed in Section 8 can be identified with regions in the parameter space as well. Here we give some examples taken mainly from [156].

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 $pn < \sqrt{n^{1-\epsilon}/m} : 1 > \epsilon > 0$, $\lim_{n,m\to\infty} m/n \le 1$, 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 $\lim_{n,m\to\infty} m/n \le 1$, 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 Section 10.1.3 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 instances 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.

10.1.3 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 [156]).

A clause is pure if it contains only negated variables or only unnegated variables. Call a formula that has no pure clauses a P-formula. A satisfying truth assignment for any P-formula can be obtained in linear time. If $pn > (1+\epsilon)\ln(m) : \epsilon > 0$, a random formula is a 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 $p^2 n > (1+\epsilon)\ln(m) : \epsilon > 0$, 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 $pn > 1.55\log_2(m)$.

10.2 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 is not enough to insure polynomial-time average complexity but, by exploiting certain properties, 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) $pn < (\epsilon - 1 - \delta)\ln(m)/(2\epsilon) : m = n^\epsilon$, $\epsilon \geq 1$, and, δ is such that $\epsilon - 1 - \delta > 0$; or 2) $pn < (1-\epsilon-\delta)\ln(m)/\epsilon : m = n^\epsilon$, $1 > \epsilon > 2/3$, and, δ is such that $1-\epsilon-\delta > 0$; or 3) $pn \leq (\ln(m)/4)^{1/3} n^{2/3-\epsilon} : m = n^\epsilon$, $2/3 \geq \epsilon > 0$ [151]. 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 Davis-Putnam-Loveland algorithm, called probe-order backtracking, that works well for the half plane $pn > \ln(m)$ exploits the preponderance of P-formulas that results from generating formulas in that region [371]. Given formula F, if an empty clause exists in F, output "unsatisfiable." If there is no clause in F containing only unnegated variables, output "satisfiable." Otherwise, select a clause in 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 [153, 154, 182, 245, 369, 368, 363].

10.3 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 a set of l distinct variables and these 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. In what follows, when we refer to l-SAT, we assume $l \geq 3$ unless specifically stated (as in, for example, 2-SAT).

10.3.1 Satisfiable and Unsatisfiable Formulas

It is easy to show that random l-SAT formulas are unsatisfiable, in probability, if $m/n > -1/\log_2(1 - 2^{-l}) \approx 2^l$ [153]. It has also been shown that a random 2-SAT formula is satisfiable, in probability, if $m/n < 1$ [75, 179]. 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, $f(2) = 1$ [75, 179]. For the 3-SAT case, from above, it is known that random l-SAT formulas are unsatisfiable, in probability, if $m/n > 4.758$ [263]. This has been recently improved to 4.64 [135]. From below, for $l > 2$, it is known that random l-SAT formulas are satisfiable, in probability, if $m/n < max\{2^l/(4l), 1\}$ [80]. This result comes with an algorithm for SAT (explained in Section 10.3.3) that finds a solution in polynomial time, almost always. For 3-SAT, this has been improved to $m/n < 3.003$ [162] (algorithm explained in Section 10.3.3).

10.3.2 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 8.5) 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.

10.3.3 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 [72] (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 [80]. 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 $m/n < 3.003$ [162]. 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 [75] 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 Davis-Putnam-Loveland trees are also exponential in size for $m/n > \log_2(1-2^{-l})$.

10.3.4 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 [250]. 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.

11 Performance and Experiments

The most important measure of a SAT algorithm's performance remains its practical problem-solving ability.

For problems requiring only one solution, both complete algorithms and incomplete algorithms are applicable. For problems requiring all solutions or an optimal solution, only complete algorithms will work. The past two decades have seen the proliferation of different algorithms for the SAT problem: resolution, local search, global optimization, BDD SAT solver, and multispace search, among others. Previous experiences indicated that these techniques complement rather than exclude each other by being effective for particular problem instances.

In this section, we will summarize the experimental performance of several typical SAT algorithms on some random problem instances, the structured problem instances, and the 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.

11.1 Experiments with the Random Problem Instances

[To be updated]

In this section, we give experimental results for the following SAT algorithms for the l-SAT random problem instances and the average l-SAT random problem instances:

SAT1.3:	a sequential *CNF* local search algorithm [186, 196, 190, 191]
SAT1.7:	a parallel *CNF* local search algorithm [186, 191]
SAT1.13:	a complete *CNF* local search algorithm [186, 191]
SAT1.4:	a sequential *DNF* local search algorithm [186]
SAT1.8:	a parallel *DNF* local search algorithm [186]
SAT1.18:	a complete *DNF* local search algorithm [186]
SAT14.6:	an optimized, discrete global optimization algorithm [186, 190, 194]
SAT14.16:	a complete global optimization algorithm [186, 190, 194]
SAT14.7:	a continuous, global optimization [186, 190, 197]
SAT14.11:	a complete, continuous global optimization [186, 190, 189]
DPL:	a Davis-Putnam algorithm in Loveland form [102]
GSAT:	a sequential *CNF* local search algorithm [408]
IP:	a parallel interior point zero-one integer programming algorithm [262, 260].

Real Execution Times. In Table 1 we give real execution times of some local search and global optimization algorithms for the l-SAT problem 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 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 problem instances in the table.

Table 1: Real execution performance of some local and global optimization algorithms on a SUN SPARC 2 Workstation. Average of ten runs. Time Units: seconds. Symbol "G/L" stands for the number of times that 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

Table 2: Performance comparison between the DPL procedure and some optimization algorithms on a SUN SPARC 2 workstation for the 3-SAT problem instances. Average of ten algorithm executions. Time units: seconds. Symbol "S/F" in Column 4 stands for DPL's *success/failure* of giving an answer within a time limit of $120 \times m/n$ seconds. Symbol "G/L" stands for the number of times that the optimization algorithms hit the *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

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

As discussed in [196, 191], beyond a certain range of hardness, for example, for $m = 8500, n = 1000$, and $l = 4$, the computing time of these optimization algorithms starts to increase.

The experimental results shown in Table 1 were collected from early reports in [186, 191, 194]. The present local search and global optimization algorithms are some ten to twenty times faster than their previous versions.

Performance Comparison with the Davis-Putnam Algorithm. The execution results of the DPL algorithm and some local search and global optimization algorithms for the l-SAT problem instances are given in Table 2. Ten algorithm executions were made for each algorithm. The average execution times are reported. Because the DPL algorithm was slow for large size problem instances, a maximum execution

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

time of $120 \times m/n$ seconds was set as the time limit of its execution. Symbol "S/F" in Column 4 stands for DPL's *success/failure* of giving an answer within such a time limit. For the DPL algorithm, the average execution time does not include the maximum execution time limit if some of the ten executions were successful; the 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 these optimization algorithms hit the *global/local* minimum points.

From numerous algorithm executions, we have observed that, for l-SAT random problem instances listed in the table, the Davis-Putnam algorithm was slower than these optimization algorithms. As the problem size increases, the number of failures, F, increases quickly. For some slightly large problem instances, 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 problem instances, such as $m = 10000, n = 1000$, and $l = 10$, the DPL algorithm 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 problem instances efficiently.

Table 2 suggests that DPL algorithm may not be a suitable candidate for large size l-SAT random problem instances. This observation, however, should not be generalized to other application cases. In many other applications, as observed by many researchers [62, 61, 111], DP algorithms performed very well.

Performance Comparison with the $GSAT$ Algorithm. Table 3 gives real performance comparison between some local search and global optimization algorithms running on a SUN SPARC 2 workstation and the $GSAT$ algorithm [408] running on a MIPS computer with comparable computing power [406]. Since $GSAT$ is essentially a version of sequential local search (i.e., $SAT1$) algorithm, for the 3-SAT problem instances generated from the same problem model used in [328], local search and global optimization algorithms performed approximately tens to hundreds times faster than the $GSAT$ algorithm. Among them,

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 the average 3-SAT problem instances. Time units: seconds. Symbol "G/L" stands for the number of times that algorithms hit the *global/local* minimum points.

Problems			Execution Time							
m	n	l	G/L	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

parallel *DNF* local search ($SAT1.8$) algorithm and complete global optimization ($SAT14.16$) performed the best.

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 the SAT problem [262, 260]. They implemented their algorithm in FORTRAN and C languages and ran the algorithm on a *KORBX(R)* parallel/vector computer with problem instances generated from the average 3-SAT problem 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 problem instances (listed in [262, 260]) on a SUN SPARC 2 workstation. The results are given in Table 4. Apparently, 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 are much simpler and they achieve several orders of magnitude of performance improvements in terms of computing time.

Performance Comparison with Pehoushek's Benchmarks. Pehoushek has recently provided an *i-j-regular 3-SAT* problem model [353]. In the model, one puts i positive and j negative copies of each variable in a set, pull out 3 at a time forming a clause. Table 5 shows real execution performance of these optimization algorithms for ten benchmark SAT problem instances provided by Pehoushek. According to Pehoushek's performance measure [353], these optimization algorithms executed *extremely* fast on the

Table 5: Execution performance of some optimization algorithms with Pehoushek's Benchmarks on a SUN SPARC 2 workstation. Time units: seconds. Symbol "G/L" stands for the number of times that the algorithm hits *global/local* minimum points.

Problems				Execution Time					
Names	m	n	l	G/L	SAT1.7	SAT1.13	SAT1.8	SAT14.6	SAT14.16
Prob1	600	180	3	10/0	3.640	4.509	3.314	12.83	8.863
Prob2	600	180	3	10/0	0.501	0.381	0.453	0.714	0.869
Prob3	600	180	3	10/0	0.705	0.682	0.637	1.538	1.723
Prob4	600	180	3	10/0	1.408	2.228	1.279	3.037	1.374
Prob5	600	180	3	10/0	0.727	0.628	0.650	0.744	1.015
Prob6	600	180	3	10/0	0.703	0.480	0.647	0.735	0.768
Prob7	600	180	3	10/0	0.997	6.479	0.901	2.399	2.464
Prob8	600	180	3	10/0	0.814	0.718	0.734	1.392	0.899
Prob9	600	180	3	10/0	1.134	0.839	1.018	0.756	1.054
Prob10	600	180	3	10/0	0.539	0.753	0.486	1.246	1.097

problem instances.

11.2 Experiments with the Structured Problem Instances

[To be updated]

We now take a look at the performance of SAT algorithms for some structured problem instances.

Instances Generated from the N-Queen Problem. To assess the performance of local search and global optimization algorithms with *non-binary* problem instances, we also tested SAT problem instances generated from the n-queen problem. Figures 22 and 23 show performance comparisons between the Davis-Putnam algorithm and some optimization algorithms.

Figure 24 shows some performance comparisons between the Davis-Putnam algorithm and a complete, continuous global optimization algorithm, the $SAT14.11$ algorithm [189]. Due to expensive floating point computation, the execution time of the $SAT14.11$ algorithm is higher than those of the discrete local search and global optimization algorithms.

Instances Generated from Boolean Inference. For the same SAT problem instances generated from Boolean inference problem [261], the performance of a parallel local search ($SAT1.7$) algorithm [191] and a simple backtracking algorithm [284] is shown in Table 6 and Table 7, respectively. An algorithm may be effective for only one type of problem instance. The results suggest that it can be much more efficient if we use several different types of algorithms to handle the same problem instances simultaneously.

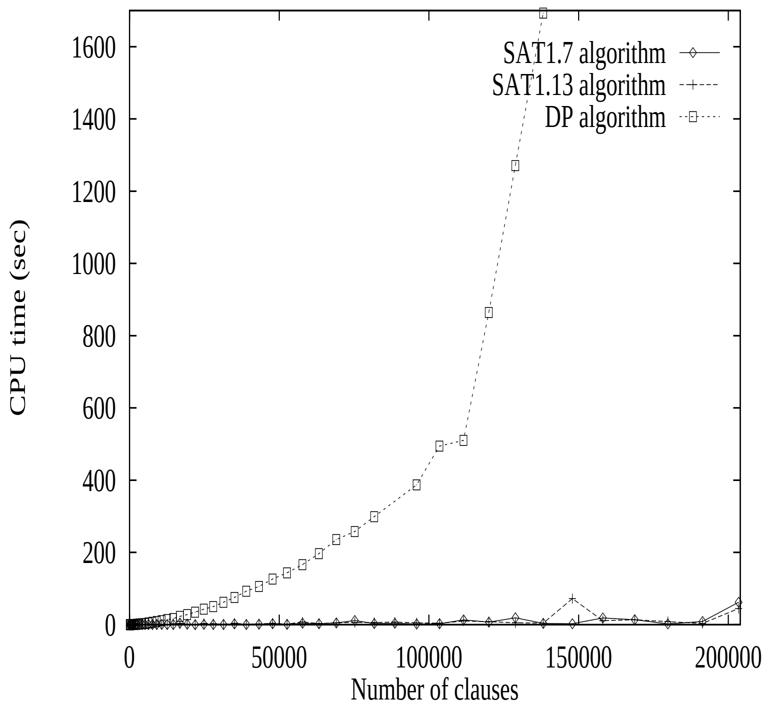

Figure 22: The comparison of the Davis-Putnam algorithm with $SAT1.7$ and $SAT1.13$ algorithms for SAT problem instances generated from a CSP problem.

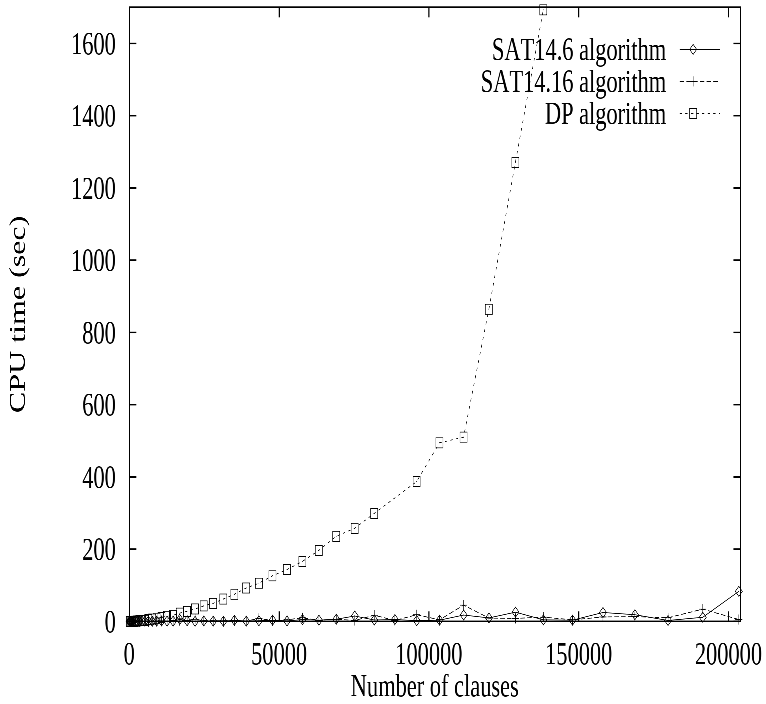

Figure 23: The comparison of the Davis-Putnam algorithm with $SAT14.6$ and $SAT14.16$ algorithms for SAT problem instances generated from a CSP problem.

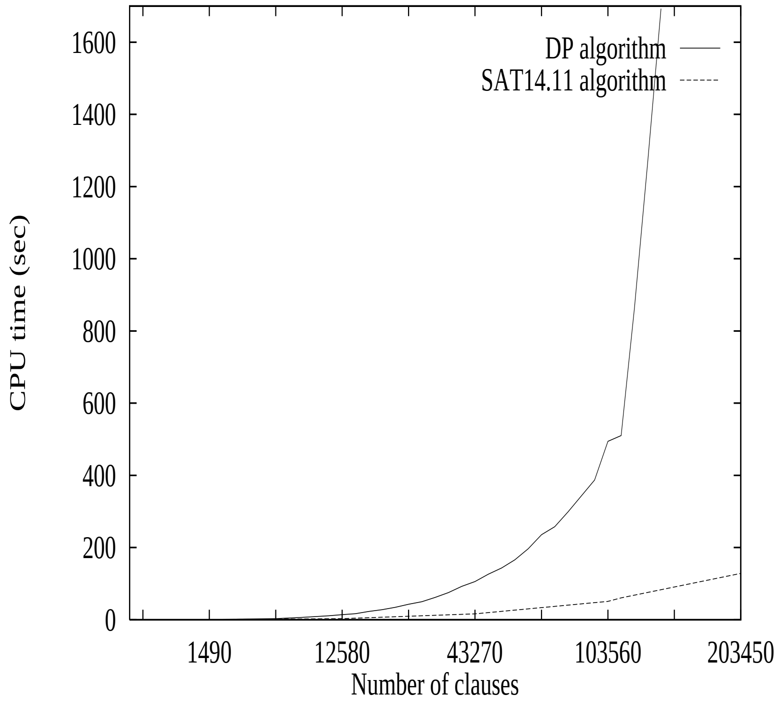

Figure 24: The comparison of the Davis-Putnam algorithm and a $SAT14.11$ algorithm for SAT problem instances generated from a CSP problem.

Table 6: Real execution performance of a $SAT1.7$ algorithm on a SUN SPARC 10 workstation. Time Units: seconds.

Problems			Ten Trials		Execution Time		
Name	Clauses	Variables	Global	SAT	Min	Mean	Max
ii16a1.sat	1650	19368	10/10	YES	10.32000	125.5100	417.4200
ii16b1.sat	1728	24792	10/10	YES	0.610000	6.113000	28.76000
ii16c1.sat	1580	16467	10/10	YES	0.540000	1.874000	3.750000
ii16d1.sat	1230	15901	10/10	YES	0.490000	1.381000	2.820000
ii16e1.sat	1245	14766	10/10	YES	0.530000	0.972000	1.480000
ii16a2.sat	1602	23281		N/A			
ii16b2.sat	1076	16121	10/10	YES	1.900000	39.117999	102.599998
ii16c2.sat	924	13803	10/10	YES	0.350000	14.109001	41.650002
ii16d2.sat	836	12461	10/10	YES	0.330000	19.840001	52.410000
ii16e2.sat	532	7825	10/10	YES	0.500000	6.883000	21.980000
ii32a1.sat	459	9212	10/10	YES	0.360000	3.574000	10.330000
ii32b1.sat	228	1374	10/10	YES	0.110000	0.739000	1.670000
ii32b2.sat	261	2558	10/10	YES	0.100000	1.904000	4.470000
ii32b3.sat	348	5734	10/10	YES	1.640000	10.559000	19.330000
ii32b4.sat	381	9618	10/10	YES	0.510000	2.306000	4.780000
ii32c1.sat	225	1280	10/10	YES	0.010000	0.115000	0.480000
ii32c2.sat	249	2182	10/10	YES	0.060000	0.398000	0.900000
ii32c3.sat	279	3272	10/10	YES	0.690000	5.490000	16.850000
ii32c4.sat	759	20862	10/10	YES	5.520000	361.794995	1496.290039
ii32d1.sat	332	2703	10/10	YES	0.220000	1.068000	3.100000
ii32d2.sat	404	5153	10/10	YES	0.210000	0.914000	2.180000
ii32d3.sat	824	19478	10/10	YES	1.710000	49.522000	109.519997
ii32e1.sat	222	1186	10/10	YES	0.020000	0.326000	1.070000
ii32e2.sat	267	2746	10/10	YES	0.040000	0.113000	0.340000
ii32e3.sat	330	5020	10/10	YES	0.450000	5.270000	13.910000
ii32e4.sat	387	7106	10/10	YES	0.270000	10.734000	46.750000
ii32e5.sat	522	11636	10/10	YES	0.490000	23.424001	84.470001

Table 7: Real execution performance of a simple backtracking algorithm on a SUN SPARC 10 workstation. Time Units: seconds.

Name	Clauses	Variables	SAT	Execution Time
ii16a1.sat	1650	19368	YES	1.285000
ii16b1.sat	1728	24792	YES	1.490000
ii16c1.sat	1580	16467	N/A	1.956250
ii16d1.sat	1230	15901	YES	1.660000
ii16e1.sat	1245	14766	N/A	2.125000
ii16a2.sat	1602	23281	YES	1.430000
ii16b2.sat	1076	16121	YES	1.505000
ii16c2.sat	924	13803	YES	2.016250
ii16d2.sat	836	12461	YES	1.665000
ii16e2.sat	532	7825	N/A	2.051250
ii32a1.sat	459	9212	YES	1.160000
ii32b1.sat	228	1374	YES	1.035000
ii32b2.sat	261	2558	YES	1.035000
ii32b3.sat	348	5734	YES	1.240000
ii32b4.sat	381	9618	YES	1.285000
ii32c1.sat	225	1280	YES	0.000000
ii32c2.sat	249	2182	YES	1.325000
ii32c3.sat	279	3272	YES	1.240000
ii32c4.sat	759	20862	YES	1.695000
ii32d1.sat	332	2703	YES	1.035000
ii32d2.sat	404	5153	YES	1.525000
ii32d3.sat	824	19478	YES	1.755000
ii32e1.sat	222	1186	YES	0.000000
ii32e2.sat	267	2746	YES	1.035000
ii32e3.sat	330	5020	YES	1.565000
ii32e4.sat	387	7106	YES	1.615000
ii32e5.sat	522	11636	YES	1.655000

11.3 Experiments with Practical Industrial Benchmarks

[To be updated]

Performance of the *SAT-CSC* Solver with Partitioning Preprocessing. In Table 8, for industrial asynchronous circuit design benchmarks including the HP and Philips benchmarks, we compare Gu and Puri's SAT solver (having a partitioning preprocessing) [198] with the existing algorithms [287, 456]. 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, compared to the previous methods [287, 456], the SAT-CSC 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 the case of a large circuit example, i.e., $mr0$, it took the SAT-CSC solver 2.80 seconds to solve the problem. It yielded a two-level implementation area with 41 literals.[10] This was compared to 1084.5 seconds and an area of 86 literals with Lavagno et al.'s algorithm. For this example, Vanbekbergen et al.'s algorithm cannot yield a solution within 3600 seconds and the program was aborted due to backtracking limit. For another circuit benchmark $mmu0$, the SAT-CSC solver solved it in 0.87 seconds, as compared to a pre-aborted 406.3 seconds for Vanbekbergen et al.'s approach [456].

Performance of a BDD SAT Solver with Partitioning Preprocessing. The BDD SAT-CSC solver was implemented in C language. In this case, Gu and Puri tested their BDD SAT-CSC 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 [198, 375]. They also compared the performance of their BDD SAT-CSC solver with the well known Lavagno et al.'s [287] asynchronous circuit design technique. The results of these experiments are given in Table 9 and Table 10.

Table 9 compares the execution time of the BDD SAT solver with the execution time of a simple backtracking SAT algorithm of [284]. The experimental results are given for SAT problem instances generated from Gu and Puri's SAT formula partitioning preprocessor [198]. Since the BDD SAT-CSC solver yielded all the solutions, they normalized the execution time of the backtracking algorithm for all the truth assignment. The experimental results (Table 9) show that the BDD SAT solver outperforms the backtracking SAT technique for the practical SAT instances representing asynchronous circuit design.

They also calculated the implementation area of the designed circuits. Table 10 compares their BDD SAT solver with the well known Lavagno et al.'s asynchronous circuit design technique [287]. The BDD SAT-CSC solver yielded a reduced circuit implementation area than the Lavagno et al.'s algorithm for almost all the circuits in the asynchronous benchmark set [287]. 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

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

Table 8: Experimental results of the SAT-CSC solver (with SAT formula partitioning preprocessing), the Vanbekbergen et al.'s algorithm, and the Lavagno et al.'s algorithm, with practical circuit benchmarks on a SUN SPARC-2 workstation. Time unit: seconds.

Circuit Specifications			*Preprocessing [198]*				*Vanbekbergen et al. [456]*				*Lavagno et al. [287]*		
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 9: Experimental results of the BDD SAT-CSC solver and a backtracking SAT algorithm, both with SAT formula partitioning preprocessing, for practical asynchronous circuit benchmarks on a SUN SPARC-2 workstation. Time unit: second.

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

Benchmark Name	*Benchmark* Initial no. of state	Initial no. of states	*BDD SAT Solver* Final no. of signal	Implementation Area (literals)	CPU time sec.	*Lavagno and Moon et al. [287]* Final no. of signal	Implementation 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

some benchmark circuits, e.g., *Pa* and *AlexNonFc*. These results show that, compared to the 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 the complex industrial asynchronous circuit design problems.

12 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., the constraint satisfaction problem and the constrained optimization problems [201]. Due to the *UniSAT* models, some application problems in the real space are related to the SAT problems as well.

- *Mathematics:* finding n-ary relations such as transitive closure [59], detecting graph and subgraph isomorphisms [92, 321, 323, 343, 379, 450, 478], the graph coloring problem [51, 209, 318, 323], mathematical cryptology [350, 384], the automata homomorphism problem [178], finding spanning trees and Euler tours in a graph [340], solving the traveling salesman problem [251, 252, 288, 344], and logical arithmetic [81].

- *Computer science and artificial intelligence:* the constraint satisfaction problem [12, 173, 185, 311, 388], the n-queen problem [173, 208, 416], extended inference [20], logical programming [84, 86, 124, 285], abductive inference for synthesizing composite hypotheses [257], semantic information processing [20, 145, 341], puzzles and cryptoarithmetic [171, 208, 316, 317, 338], truth maintenance [107, 109, 112, 123, 319], production system [242, 326, 327], the soma cube and instant insanity problem [173], theorem proving [235, 272, 365, 477], and neural network computing [12, 13, 114, 218, 304, 238].

- *Machine vision:* image matching problem [20, 77, 387, 473], line and edge labeling problems [67, 143, 214, 446, 462, 480], stereopsis, scene analysis and semantics-based region growing [20, 67, 142, 143, 144, 446, 462], the shape and object matching problem [59, 100, 213], syntactic shape analysis [101, 212], shape from shading problem [11, 44, 157, 228, 229, 231, 230, 240, 314, 354], and image restoration [175].

- *Robotics:* for solving its vision problem [77, 239], packing problem [118], and trajectory and task planning problems [40, 136].

- *Computer-aided manufacturing:* task planning [337], design [335, 336], solid modeling, configuring task [158], design cellular manufacturing system, scheduling [148, 306], and 3-dimensional object recognition [202, 230].

- *Database systems:* operations on objects [449, 452], database consistency maintenance, query-answering and redundancy-checking, query optimization [69, 449], concurrency control [25, 138, 310], distributed database systems [167], truth and belief maintenance [107, 109, 112, 123, 319], the relational homomorphism problem [208, 449], and knowledge organization for recognition system [210].
- *Integrated circuit design automation:* circuit modeling [66, 440], logic minimization [220], state assignment [456, 457], state minimization [183, 378], asynchronous circuit synthesis [198, 375, 374, 376], I/O encoding for sequential machines [395], power dissipation estimation [120], logic partitioning [74, 128, 283, 342, 393], circuit layout and placement [10, 30, 41, 85, 97, 119, 205, 426], scheduling and high-level synthesis [42, 281, 351], pin assignment [39, 392], floorplanning [358, 435], interconnection analysis [126, 127], routing [1, 63, 116, 117, 204, 291, 349, 381, 385, 405], compaction [60, 125, 211, 265, 282, 298, 400, 412, 454], performance optimization [259, 273, 315, 390, 435], testing and test generation [121, 244, 284, 131, 359], and verification [271, 437, 447] are essentially constrained search problems. Many of them demand efficient SAT algorithms.
- *Computer architecture design:* instruction set optimization [4, 99, 184, 247, 373, 377], computer controller optimization [23, 27, 266, 308, 372, 378], arithmetic logic circuit design [68], compiler system optimization [7, 297], scheduling [31, 122, 168, 169, 203, 294], fault-tolerant computing [22, 227, 18], task partitioning and assignment [33, 34, 96, 241, 413], load balancing [300, 339, 479], real time systems [221, 246, 274, 429, 430, 439], data flow consistency analysis [7], data module assignment in memory system [7], and parallel and distributed processing [380, 403].
- *Text processing:* optical character recognition [79, 332, 434], character constraint graph model [236], printed text recognition [19, 236], handwritten text recognition [415], automatic correction of errors in text [451].
- *Computer graphics:* construction of 2-dimensional pictures and 3-dimensional graphical objects from constraints, reasoning of the geometrical features of 3-dimensional objects [49, 163].

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

13 Future Work

A number of future research directions for the satisfiability problem have been discussed recently. They are further emphasized by 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. This will result in clause form representation with substantially larger sizes [174, 357]. While this is not critical in complexity theory, it will have serious impact to the computing efficiency of solving the practical application problems. Thus, efficient representation and manipulation of general Boolean functions is crucial to solve practical application problems.

Theoretical issues. Recent research for the SAT problem has brought up some interesting theoretical problems, such as the average time complexity analysis [191, 235, 361], determining sat-and-unsat boundary [93, 268, 328], global convergence and local convergence rate [193, 197], the structure and hardness of problem instance models [155]. Some of the problems, e.g., the average time complexity analysis, are extremely difficult [293]. So far only some preliminary efforts based on simplified assumptions were given [43, 196, 193, 197].

One of the recent efforts to solve the SAT problem is to find subclasses for which the problem is solvable in polynomial time [137, 166]. Future works in this direction aim 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 the 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 SAT algorithm that performs well for a wide range of SAT problem instances. The existing SAT algorithms complement rather than exclude each other by being effective for particular problem instances. One of the future work is to integrate a number of SAT algorithms, they each work well for a certain range of problem instances. This raises some interesting question as when to apply which algorithm to what problem instances. Another aspect of algorithm integration is to combine incomplete algorithms with complete search framework, making it useful for practical application problems [191, 194, 94].

For important practical applications, there may be significant problem domain information. Efficient SAT algorithms may be developed by exploring the problem- and application-specific structures. Specialized algorithms tailored to particular applications, on the other hand, do give 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 and the advisory/organizing committees of the 1996 DIMACS satisfiability workshop [132]. There has been a strong relationship between the theory, algorithms, and the applications of the SAT problem. A major step in the future is to bring together the theorists, algorithmists, and practitioners working on the SAT problem and on the industrial applications involving the SAT problem, enhancing the interaction between the three research groups. It would be benefit to research community and to the industry if we apply the theoretical and algorithmic results to practical problems and also take the practical problems for theoretical/algorithmic study. In addition to theoretical/algorithmic study, in the future, we will also further concentrate on significant industrial case studies of the SAT problem, practical applications of the 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 the algorithm execution. Not only does it offer faster execution speed on hardware medium, certain sequential portions of the algorithm may be implemented in hardware architectures in parallel form. For the SAT problem *per se*, it has certain granularity at the search tree level, clause level, and variable level which lend itself well to parallel processing. A number of parallel algorithms and architectures for the SAT problem were developed that perform well at different levels of granularity. Two basic approaches were taken in this direction: i.e., implementing parallel SAT inference algorithms on special-purpose VLSI chips [199, 200] or implementing tightly-coupled, parallel SAT algorithms on existing sequential computer machines [186, 191, 191, 425, 424].

Multispace search. Multispace search interplays multispace operations with traditional optimization algorithms [187, 201]. Structural multispace operations disturb the environment of forming local minima and improve the performance of traditional "value" search methods. This makes multispace search a natural approach to handle difficult optimization problems. It is expected that multispace search techniques for NP-hard problems will be developed for a variety of practical application cases.

14 Conclusions

The SAT problem is at the *core* of the class of NP-complete problems. The SAT problem has many practical applications. In recent years, many optimization methods, parallel algorithms, and practical techniques have been developed for the SAT problem. The past two decades have seen the proliferation of many algorithms for the SAT problem: resolution, local search, global optimization, BDD SAT solver, and multispace search, among others. The existing methods complement rather than exclude each other by being effective for particular problem instances. 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 developing SAT algorithms. This survey also provides a set of practical applications of the SAT problem. The area of the SAT research is a rich land of well-developed theory and methods. To apply the 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

Moshe Vardi, Ding-Zhu Du, and Panos Pardalos suggested that this article be written for the 1996 DIMACS satisfiability workshop. Ding-Zhu Du, and Panos Pardalos, Christos Papadimitriou, R.C.T. Lee, Pierre Hansen, Scott Kirkpatrick, Endre Boros, Peter Hammer, Andy Sage, Qianping Gu, Wei Wang, Ruchir Puri, Rok Sosič, Bin Du, Guojie Li, Wei Li, Chen Guo Liang, and Zhiqiang Liu, have provided constructive comments for various early versions of this paper. R.C.T. Lee, Scott Kirkpatrick, Dennis Kibler, Mauricio Resende, Kazuo Iwama, Max Böhm, Ewald Speckenmeyer, Andrei Broder, Rina Dechter, Jon Freeman, James Crawford, Bart Selman, and Vipin Kumar provided us some recent publications.

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] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, Jun. 1978.

[9] W. Ahrens. *Mathematische Unterhaltungen und Spiele (in German).* B.G. Teubner (Publishing Company), Leipzig, 1918-1921.

[10] 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.

[11] J. Aloimonos. Visual shape computation. *Proceedings of the IEEE*, 76:899–916, Aug. 1988.

[12] 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.

[13] J.A. Anderson and E. Rosenfeld, editors. *Neurocomputing: Foundations of Research.* MIT Press, Cambridge, 1988.

[14] 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.

[15] 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.

[16] 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.

[17] B. Aspvall. Recognizing disguised NR(1) instances of the satisfiability problem. *Journal of Algorithms* **1**, pages 97–103, 1980.

[18] A. Bagchi, B. Servatius, and W. Shi. Fault tolerance in massively parallel computers. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[19] H. Baird. Anatomy of a versatile page reader. *Proceedings of the IEEE*, 80(7):1059–1065, Jul. 1992.

[20] D. H. Ballard and C. M. Brown. *Computer Vision.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[21] 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.

[22] A.E. Barbour. Solutions to the minimization problem of falut-tolerant logic circuits. *IEEE Trans. on Computers*, 41(4):429–443, Apr. 1992.

[23] 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.

[24] B. Bernhardsson. Explicit solutions to the n-queens problems for all n. *ACM SIGART Bulletin*, 2(2):7, Apr. 1991, ACM Press.

[25] P.A. Berstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publication Co., 1987.

[26] Wolfgang Bibel. Automated theorem proving. *Vieweg*, 1982.

[27] N. N. Biswas. State Minimization of Incompletely Specified Sequential Machines. *IEEE Trans. on Computers*, C-23:80–84, Jan. 1974.

[28] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Comm. of ACM*, 18(11):651–656, Nov. 1975.

[29] 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.

[30] 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.

[31] J. Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proc. of the International Workshop on Modelling and Performance Evaluation of Computer Systems*, pages 57–65, 1976.

[32] 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.

[33] S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, Jan 1988.

[34] F. Bonomi. On job assignment for a parallel system of processor sharing queues. *IEEE Trans. on Computers*, 39(7):858–869, July 1990.

[35] 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.

[36] 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.

[37] E. Boros, P. L. Hammer, and X. Sun. Recognition of q-Horn formulae in linear time. *Discrete Applied Mathematics* **55**, pages 1–13, 1994.

[38] 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.

[39] H.N. Brady. An approach to topological pin assignment. *IEEE Trans. on CAD*, CAD-3:250–255, July 1984.

[40] 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.

[41] M.A. Breuer. Min-cut placement. *J. of Design Automation and Fault Tolerant Computing*, 1:343–362, 1976.

[42] F. Brewer and D. Gajski. Chippe: A system for constraint driven behavioral synthesis. *IEEE Trans. on CAD*, 9(7):681–695, July 1990.

[43] 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.

[44] M.J. Brooks and B.K.P. Horn. Shape and source from shading. In *Proc. of IJCAI'85*, pages 932–936, Aug. 1985.

[45] C.A. Brown, L.A. Finklestein, and P.W. Purdom. Backtrack Searching in the Presence of Symmetry 6^{th} *International Conference on Algebraic Algorithms and Error Correcting Codes (AAECC)*, Springer Verlag Lecture Notes in Computer Science, 357: 99-110.

[46] C.A. Brown, L.A. Finklestein, and P.W. Purdom. Backtrack Searching in the Presence of Symmetry Nordic Journal of Computing, to appear.

[47] C.A. Brown and P.W. Purdom. An average time analysis of backtracking. *SIAM J. on Computing*, 10(3):583–593, Aug. 1981.

[48] 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.

[49] B. Bruderlin. Constructing three dimensional geometric objects defined by constraints. In *Proceedings of 1986 ACM SIGGRAPH Workshop in Interactive Graphics*, 1986.

[50] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[51] M. Bruynooghe. Graph coloring and constraint satisfaction. Technical Report CW 44, Dept. of Computer Science, Katholieke Universiteit Leuven, Dec. 1985.

[52] M. Bruynooghe and L.M. Pereira. *Deduction Revision by Intelligent Backtracking*, pages 194–215. Ellis Horwood Limited, 1984.

[53] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[54] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[55] H. Kleine Büning. On generalized Horn formulas and k resolution. *Theoretical Computer Science* **116**, pages 405–413, 1993.

[56] K. Bugrara and P. Purdom. Clause order backtracking. Technical Report 311, 1990.

[57] K.M. Bugrara and C.A. Brown. On the average case analysis of some satisfiability model problems. *Information Sciences*, 40:21–37, 1986.

[58] 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.

[59] A. Bundy, editor. *Catalogue of Artificial Intelligence Tools.* Springer-Verlag, Berlin, 1984.

[60] J.L. Burns and A.R. Newton. Efficient constraint generation for hierarchical compaction. In *Proc. Intl. Conf. on Computer Design*, pages 197–200. IEEE Computer Society, Oct. 1987.

[61] M. Buro and H.K. Buning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49:143–151, Feb. 1993.

[62] M. Buro and H.K. Buning. Report on a SAT competition. Technical report, FB-17 — Mathematik/Informatik, Universitat Paderborn, Nov. 1992.

[63] M. Burstein and R. Pelavin. Hierarchical channel router. In *Proc. of the 20th ACM/IEEE Design Automation Conference*, pages 591–597, Jun. 1983.

[64] G. Butler. Computing in Permutation and Matrix Groups II: Backtrack Algorithm *Math. Comp.* 1982 39: 671–680.

[65] G. Butler and C.W.H. Lam A General Backtracking Algorithm for the Isomorphism Problem of Combinatatorial Objects J. Symbolic Computation 1985 1:363-381.

[66] 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.

[67] I. Chakravarty. A generalized line and junction labelling scheme with applications to scene analysis. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-1(2):202–205, Apr. 1979.

[68] 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.

[69] 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.

[70] V. Chandru and J. N. Hooker. Extended Horn sets in propositional logic. *J. ACM* **38**, pages 205–221, 1991.

[71] M.T. Chao and J. Franco. Probabilistic analysis of two heuristics for the 3-satisfiability problem. *SIAM J. on Computing*, 15:1106–1118, 1986.

[72] 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.

[73] 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.

[74] H.R. Charney and D.L. Plato. Efficient partitioning of components. In *Proc. of the 5th Annual Design Automation Workshop*, pages 16–21, 1968.

[75] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *J. of ACM*, 35:759–770, 1988.

[76] W.T. Chen and L.L. Liu. Parallel approach for theorem proving in propositional logic. *Inform. Sci.*, 41(1):61–76, 1987.

[77] R. T. Chin and C. R. Dyer. Model-based recognition in robot vision. *ACM Computing Surveys*, 18(1):67–108, Mar. 1986.

[78] 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.

[79] Y. Chu, editor. *Special Section on Chinese/Kanji Text and Data Processing. IEEE Computer*, volume 18, number 1. IEEE Computer Society Press, 1985.

[80] V. Chvátal and B. Reed. Mick gets some (the odds are on his side). In *Proceedings on the Foundations of Computer Science*, 1992.

[81] J. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2), 1987.

[82] W. F. Clocksin and C. S. Mellish. *Programming in Prolog (2nd Edition).* Springer-Verlag, Berlin, 1984.

[83] M.B. Clowes. On seeing things. *Artificial Intelligence*, 2:79–116, 1971.

[84] J. Cohen, editor. *Special Section on Logic Programming. Comm. of the ACM*, volume 35, number 3. 1992.

[85] J.P. Cohoon and W.D. Paris. Genetic placement. In *Digest of the Intl. Conf. on Computer-Aided Design*, pages 422–425, 1986.

[86] A. Colmerauer. Opening the Prolog III universe. *BYTE Magazine*, pages 177–182, Aug. 1987.

[87] J.S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Boston, 1987.

[88] 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 Internationl Symposium on Mathematical Programming, University of Michigan, 1994.

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

[90] 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.

[91] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[92] 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.

[93] 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.

[94] J.M. Crawford. Solving satisfiability problems using a combination of systematic and local search. Submitted to the DIMACS Challenge II Workshop, 1994.

[95] V. Cěrny. A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm. Technical report, Institute of Physics and Biophysics, Comenius University, Bratislava, 1982.

[96] 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.

[97] W. Dai and E.S. Kuh. Hierarchical floorplanning for building block layout. In *Digest of Intl. Conf. on Computer-Aided Design*, pages 454–457, 1986.

[98] M. Dalal, and D. W. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing letters* **44**, pages 173–180, 1992.

[99] 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.

[100] L. S. Davis. Shape matching using relaxation techniques. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-1(1):60–72, Jan. 1979.

[101] 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.

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

[103] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. of ACM*, 7:201–215, 1960.

[104] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[105] J. de Kleer. Problems with ATMS. *Artificial Intelligence*, 28:197–224, 1986.

[106] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of 11th IJCAI*, pages 290–296, 1989.

[107] J. de Kleer. Exploiting locality in a TMS. In *Proceedings of AAAI'90*, pages 264–271, 1990.

[108] R. Dechter. Learning while searching in constraint satisfaction problems. In *Proceedings of AAAI'86*, 1986.

[109] R. Dechter. A constraint-network approach to truth maintenance. Technical Report R-870009, Computer Science Dept., UCLA, Los Angeles, 1987.

[110] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41(3), 1990.

[111] R. Dechter. Directional resolution: The davis-putnam procedure revisited. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[112] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI'88*, 1988.

[113] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[114] J.S. Denker, editor. *Neural Networks for Computing*, volume 151 of *AIP (Snowbird, Utah) Conference Proceedings*. American Institute of Physics, New York, 1986.

[115] N. Dershowitz, J. Hsiang, N. Josephson, and D. Plaisted. Associative-commutative rewriting. In *Proceedings of IJCAI*, pages 940–944, 1983.

[116] D.N. Deutsch. A 'dogleg' channel router. In *Proc. of the 13th ACM/IEEE Design Automation Conference*, pages 425–433, Jun. 1976.

[117] D.N. Deutsch. Compacted channel routing. In *Digest Intl. Conf. on Computer-Aided Design*, pages 223–225, Nov. 1985.

[118] J. P. A. Deutsch. A short cut for certain combinational problems. In *British Joint Comput. Conference*, 1966.

[119] S. Devadas. Optimal layout via boolean satisfiability. In *Proceedings of ICCAD'89*, pages 294–297, Nov. 1989.

[120] 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.

[121] 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.

[122] S.K. Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, Feb. 1978.

[123] 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.

[124] 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.

[125] J. Doenhardt and T. Lengauer. Algorithm aspects of one-dimensional layout. *IEEE Trans. on CAD*, CAD-6(5):863–878, 1987.

[126] W.E. Donath. Placement and average interconnection lengths of computer logic. *IEEE Trans. on Circuits and Systems*, CAS-26(4):272–277, Apr. 1979.

[127] W.E. Donath. Wire length distribution for placements of computer logic. *IBM J. of Research and Development*, 25(3):152–155, May 1981.

[128] 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.

[129] 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.

[130] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[131] B. Du and J. Gu. Sequential circuit test generation by Boolean satisfiability. 1996, to appear.

[132] 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.

[133] O. Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81:49–64, 1991.

[134] O. Dubois and J. Carlier. Probabilistic approach to the satisfiability problem. *Theoretical Computer Science*, 81:65–75, 1991.

[135] O. DuBois and Y. Boufkhad. 1996. 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.

[136] C. Eastman. Preliminary report on a system for general space planning. *Comm. of ACM*, 15(2):76–87, Feb. 1972.

[137] T. Eiter, P. Kilpelainen, and H. Mannila. Some remarks on renaming and satisfiability hierachies. Summary Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[138] 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.

[139] 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.

[140] B.-J. Falkowski and L. Schmitz. A note on the queens' problem. *Information Processing Letters*, Vol. 23:39–46, July 1986.

[141] 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.

[142] 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.

[143] O. D. Faugeras, editor. *Fundamentals in Computer Vision.* Cambridge University Press, London, 1983.

[144] J. A. Feldman and Y. Yakimovsky. Decision theory and artificial intelligence: I. a semantics-based region analyser. *Artificial Intelligence*, 5:349–371, 1974.

[145] J. D. Findler. *Associative Networks: Representation and Use of Knowledge by Computers.* Academic Press, New York, 1979.

[146] C.A. Floudas and P.M. Pardalos. *A Collection of Test Problems for Constrained Global Optimization Algorithms.* Springer Verlage, New York, 1990.

[147] C.A. Floudas and P.M. Pardalos, editors. *Recent Advances in Global Optimization.* Princeton University Press, New York, 1992.

[148] M.S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Pitman, London, 1987.

[149] J. Franco. Probabilistic analysis of the pure literal heuristic for the satisfiability problem. *Annals of Operations Research*, 1:273–289, 1984.

[150] J. Franco. On the probabilistic performance of algorithms for the satisfiability problem. *Information Processing Letters*, 23:103–106, 1986.

[151] J. Franco. Elimination of infrequent variables improves average case performance of satisfiability algorithms. *SIAM J. on Computing*, 20:1119–1127, 1991.

[152] J. Franco and Y.C. Ho. Probabilistic performance of heuristic for the satisfiability problem. *Discrete Applied Mathematics*, 22:35–51, 1988/89.

[153] J. Franco and M. Paull. Probabilistic analysis of the davis-putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.

[154] J. Franco. 1993. On the occurrence of null clauses in random instances of satisfiability. *Discrete Applied Mathematics* **41**, pp. 203–209.

[155] J. Franco and R. Swaminathan. Toward a good algorithm for determining unsatisfiability of propositional formulas. 1996.

[156] 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.

[157] 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.

[158] F. Frayman and S. Mittal. *Cossack: A Constraints-based Expert System for Configuration Tasks.* Computational Mechanics Publications, Nadel, 1987.

[159] E. C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1):24–32, Jan. 1982.

[160] E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4):755–761, Oct. 1985.

[161] 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.

[162] A. M. Frieze, and S. Suen. 1993. Analysis of simple heuristics for random instances of 3-SAT.

[163] 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.

[164] Z. Galil. On the complexity of regular resolution and the Davis-Putnam procedure. *Theoretical Computer Science*, pages 23–46, 1977.

[165] G. Gallo, and M. G. Scutella. Polynomially solvable satisfiability problems. *Information Processing Letters* **29**, pages 221–227, 1988.

[166] G. Gallo and D. Pretolani. Hierarchies of polynomially solvable SAT problems. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[167] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems*, 8(2), Jun. 1983.

[168] M.R. Garey and D.S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. Comput.*, 4:397–411, 1975.

[169] M.R. Garey and D.S. Johnson. Two-processors scheduling with start-times and deadlines. *SIAM J. on Computing*, 6:416–426, 1977.

[170] 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.

[171] J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings of 12th Annual Allerton Conf. Circuit System Theory*, 1974.

[172] J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of 9th IJCAI*, page 457, 1977.

[173] J. Gaschnig. *Performance Measurements and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, Dept. of Computer Science, May 1979.

[174] A.V. Gelder. A satisfiability tester for non-clausal propositional calculus. *Information and Computation*, 79(1):1–21, Oct. 1988.

[175] 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.

[176] M.R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Los Altos, California, 1987.

[177] P.C. Gilmore. A proof method for quantification theory. *IBM J. Res. Develop.*, 4:28–35, 1960.

[178] A. Ginzberg. *Algebraic Theory of Automata*. Academic Press, New York, 1968.

[179] A. Goerdt. 1992. A threshold for unsatisfiability. *Proceedings of 17th annual Symposium on Mathematical Foundations of Computer Science*, Prague, Czechoslovakia.

[180] A. Goldberg. Average case complexity of the satisfiability problem. In *Proc. Fourth Workshop on Automated Deduction*, pages 1–6, 1979.

[181] A. Goldberg. On the complexity of the satisfiability problem. Technical Report Courant Computer Science No. 16, New York University, 1979.

[182] A. Goldberg, P.W. Purdom, and C.A. Brown. Average time analysis of simplified davis-putnam procedures.[11] *Information Processing Letters*, 15(2):72–75, 6 Sept. 1982.

[183] 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.

[184] 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.

[185] J. Gu. Parallel algorithms and architectures for very fast search. Ph.D Thesis. Technical Report UUCS-TR-88-005, Jul. 1988.

[186] 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.

[187] J. Gu. Optimization by multispace search. Technical Report UCECE-TR-90-001, Dept. of Electrical and Computer Engineering, Univ. of Calgary, Jan. 1990.

[188] J. Gu. An $\alpha\beta$-relaxation for global optimization. Technical Report UCECE-TR-91-003, Dept. of ECE, Univ. of Calgary, Apr. 1991.

[11] Some printer errors in this paper were corrected in *Information Processing Letters*, 16:213, 1983.

[189] J. Gu. The *UniSAT* problem models (appendix). *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(8):865, Aug. 1992.

[190] J. Gu. Efficient local search for very large-scale satisfiability problem. *SIGART Bulletin*, 3(1):8–12, Jan. 1992, ACM Press.

[191] 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.

[192] J. Gu. *Multispace Search: A New Optimization Approach (Summary).* In *Lecture Notes in Computer Science, Vol. 834*, pages 252–260. 1994.

[193] 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.

[194] 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.

[195] 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.

[196] 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. Also in *Lecture Notes in Computer Science*, Vol. 834, pp. 146-154, 1994 and to appear in *IEEE Trans. on Knowledge and Data Engineering.*

[197] J. Gu, Q.P. Gu, and D.-Z. Du. Convergence properties of optimization algorithms for the satisfiability (SAT) problem. *IEEE Trans. on Computers*, Vol. 45, No. 2, pp. 209-219, Feb. 1996.

[198] 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.

[199] 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.

[200] J. Gu and W. Wang. A novel discrete relaxation architecture. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(8):857–865, Aug. 1992.

[201] J. Gu. *Constraint-Based Search.* Cambridge University Press, New York, 1996.

[202] A. Guzman. *Computer Recognition of Three-Dimensional Objects in a Visual Scene.* PhD thesis, MIT, 1968.

[203] 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.

[204] 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.

[205] 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.

[206] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.

[207] R. M. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[208] 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.

[209] F. Harary. *Graph Theory.* Addison-Wesley, Reading, 1969.

[210] W. S. Havens. A theory of schema labelling. *Computational Intelligence*, 1(3 & 4):127–139, 1985.

[211] T. Hedges, W. Dawson, and Y.E. Cho. Bitmap graph build algorithm for compaction. In *Digest Intl. Conf. on Computer-Aided Design*, pages 340–342, Sep. 1985.

[212] T. C. Henderson and L. S. Davis. Hierarchical models and analysis of shape. *Pattern Recognition*, 14(1-6):197–204, 1981.

[213] T. C. Henderson and A. Samal. Multiconstraint shape analysis. *Image and Vision Computing*, 4(2):84–96, May 1986.

[214] T.C. Henderson. *Discrete Relaxation Techniques.* Oxford University Press, New York, 1990.

[215] P. V. Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, Cambridge, 1989.

[216] P. Heusch. The Complexity of the Falsifiability Problem for Pure Implicational Formulas. *Proc. 20th Int. 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.

[217] G.E. Hinton and J.A. Anderson, editors. *Parallel Models of Associative Memory.* Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1981.

[218] 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.

[219] 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.

[220] S.J. Hong and S. Muroga. Absolute minimization of completely specified switching functions. *IEEE Trans. on Computers*, 40(1):53–65, Jan. 1991.

[221] P. Hood and Grover. Designing real time systems in Ada. Technical Report 1123-1, SofTech, Inc., Waltham, Jan. 1986.

[222] J. Hooker and V. Vinay. An empirical study of branching rules for satisfiability. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[223] J.N. Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12:217–239, 1988.

[224] J.N. Hooker. A quantitative approach to logical inference. *Decision Support Systems*, 4:45–69, 1988.

[225] J.N. Hooker. Resolution vs. cutting plane solution of inference problems: Some computational experience. *Operations Research Letter*, 7(1):1–7, 1988.

[226] 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.

[227] 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.

[228] 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.

[229] B.K.P. Horn. Understanding image intensity. *Artificial Intelligence*, 8:301–231, 1977.

[230] B.K.P. Horn and M. J. Brooks, editors. *Shape from Shading.* The MIT Press, Cambridge, 1989.

[231] 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.

[232] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, 1978.

[233] R. Horst and H. Tuy. *Global Optimization: Deterministic Approaches.* Springer Verlage, Berlin, 1990.

[234] J. Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, pages 255–300, 1985.

[235] 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.

[236] X. Huang, J. Gu, and Y. Wu. A constrained approach to multifont character recognition. *IEEE Transactions on Pattern Analysis And Machine Intelligence*, 15(8):838–843, Aug. 1993.

[237] 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.

[238] 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.

[239] K. Ikeuchi. Model-based interpretation of range imagery. In *Proc. of Image Understanding Workshop*, pages 321–339. DRAPA, Feb. 1987.

[240] K. Ikeuchi and B.K.P. Horn. Numerical shape from shading and occluding boundaries. *Artificial Intelligence*, 17(1-3):141–184, 1981.

[241] 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.

[242] T. Ishida. Parallel rule firing in production systems. *IEEE Trans. on Knowledge and Data Engineering*, 3(1):11–17, Mar. 1991.

[243] A. Itai and J. Makowsky. On the complexity of Herbrand's theorem. Working paper 243, Department of Computer Science, Israel Institute of Technology, 1982.

[244] 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.

[245] K. Iwama. CNF satisfiability test by counting and polynomial average time. *SIAM J. on Computing*, pages 385–391, 1989.

[246] 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.

[247] 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.

[248] R.E. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and AI*, 1:167–187, 1990.

[249] R.G. Jeroslow. Computation-oriented reductions of predicate to propositional logic. *Decision Support Systems*, 4:183–197, 1988.

[250] D.S. Johnson. "Approximation Algorithms for Combinatorial Problems". J. of Computer and Systems Sciences, 9 (1974) pp. 256–278.

[251] D.S. Johnson. More approaches to the traveling salesman guide. *Nature*, 330:525, 1987.

[252] 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.

[253] J.L. Johnson. A neural network approach to the 3-satisfiability problem. *J. of Parallel and Distributed Computing*, 6:435–449, 1989.

[254] R. R. Johnson. Critical issues in computer architecture design. Private Communication, 1987-1994.

[255] R. W. Johnson and A. M. McLoughlin Computer-Aided Discorvery of a Fast Matrix-Multiplication Algorithm *NRL Memorandum Report 3994* May 7, 1979.

[256] W. Lewis Johnson. Letter from the editor. *SIGART Bulletin*, 2(2):1, April 1991, ACM Press.

[257] 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.

[258] P.C. Jackson Jr. Heuristic search algorithms for the satisfiability problem. Submitted to the third IEEE TAI Conference, Jul. 1991.

[259] Y.C. Ju, B. Rao, and R.A. Saleh. Consistency checking and optimization of macromodels. *IEEE Trans. on CAD*, 10(8):957–967, Aug. 1991.

[260] 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.

[261] 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.

[262] 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.

[263] 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.

[264] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, (4):373–395, 1984.

[265] G. Kedem and H. Watanabe. Graph optimization techniques for IC layout and compaction. *IEEE Trans. on CAD*, CAD-3:12–20, 1984.

[266] J. Kella. State Minimization of Incompletely Specified Sequential Machines. *IEEE Trans. on Computers*, C-19:342–348, April 1970.

[267] S. Kirkpatrick, C.D. Gelat, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[268] S. Kirkpatrick, G. Gyorgyi, N. Tishby, and L. Troyansky. The statistical mechanics of k-satisfaction. In *Proceedings of Neural Information Processing Systems*, Nov. 1993.

[269] J. Kittler and J. Illingworth. Relaxation labelling algorithms — A review. *Image and Vision Computing*, pages 206–216, 1985.

[270] D. E. Knuth. Estimating the efficiency of backtracking programs. *Mathematics of Computation*, 29(129):121–136, Jan. 1975.

[271] 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.

[272] R. Kowalski. A proof procedure using connection graphs. *J. ACM*, 22(4):572–595, Oct. 1975.

[273] 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.

[274] 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.

[275] D. C-L. Ku. DRA1 chip implementation report. Project Report, Dept. of Computer Science, Univ. of Utah, Mar. 1986.

[276] Oliver Kullmann A systematical approach to 3-SAT-decision, yielding 3-SAT-decision in less than 1.5045^n steps *Theoretical Computer Science*, to appear.

[277] V. Kumar. Algorithms for constraint satisfaction problems: A survey. Technical Report TR-91-28, Dept. of Computer Science, Univ. of Minnesota, 1991.

[278] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *The AI Magazine*, 13(1):32–44, 1992.

[279] V. Kumar and Y.-J Lin. A data-dependency based intelligent backtracking scheme for Prolog. *J. of Logic Programming*, 5(2), June 1988.

[280] J. D. Laderman A noncommutative algorithm for multiplying 3×3 matricies using 23 multiplications *Bull. Amer. Math. Soc.*, 82:126–128.

[281] 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.

[282] G. Lakhani and R. Varadarajan. A wire-length minimization algorithm for circuit layout compaction. In *Proc. of ISCAS'87*, pages 276–279, May 1987.

[283] 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.

[284] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Trans. on Computer-Aided Design*, 11(1):4–15, Jan. 1992.

[285] C. Lassez. Constraint logic programming. *BYTE Magazine*, pages 171–176, Aug. 1987.

[286] 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.

[287] 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.

[288] 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.

[289] E.L. Lawler and D.E. Wood. Brance-and-bound methods: a survey. *Operations Research*, 14(4):699–719, Jul. - Aug. 1966.

[290] C. Y. Lee. Representation of Switching Circuits by Binary Decision Programs. *Bell Systems Technical Journal*, 38:985–999, July 1959.

[291] C. Lee. An algorithm for path connections and its applications. *IEEE Trans. on Electronic Computers*, VEC-10:346–365, Sept. 1961.

[292] E.A. Lee. Consistency in dataflow graphs. *IEEE Trans. on Parallel and Distributed Systems*, 2(2):223–235, Apr. 1991.

[293] R.C.T. Lee. Private Communications, 1992-1993.

[294] J.P. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *ACM Performance Evaluation Review*, 14(1), May 1986. Special Issue.

[295] H. R. Lewis. Renaming a set of clauses as a Horn set. *Journal of the Association for Computing Machinery* **25**, pages 134–135, 1978.

[296] G.-J. Li. *Parallel Processing of Combinatorial Search Problems*. PhD thesis, School of Electrical Engineering, Purdue University, West Lafayette, Dec. 1985.

[297] 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.

[298] 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.

[299] K.J. Lieberherr and E. Specker. Complexity of partial satisfaction. *J. of ACM*, 28:411–421, 1981.

[300] 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.

[301] K. J. Lin and C. S. Lin. Automatic Synthesis of Asynchronous Circuits. In *Proc. of 28th DAC*, pages 296–301, 1991.

[302] 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.

[303] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, The Univ. of Texas at Austin, Dept. of Computer Science, May 1988.

[304] R. P. Lippmann. An introduction to computing with neural net. *IEEE ASSP Magazine*, 4(2):4–22, April 1987.

[305] Richard J. Lipton DNA Solution of Hard Computational Problems *Science*, 268:542-544.

[306] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, Carnegie-Mellon University, May 1985.

[307] D.W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, 1978.

[308] 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.

[309] D.G. Luenberger. *Linear and Nonlinear Programming.* Addison-Wesley, Reading, 1984.

[310] N.A. Lynch. Multi-level atomicity — a new correctness criterion for database concurrency control. *ACM Trans. on Database Systems*, 8(4), Dec. 1983.

[311] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–119, 1977.

[312] 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.

[313] 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.

[314] 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.

[315] D. Marple and A.E. Gamal. Area-delay optimization of programmable logic arrays. Technical report, Stanford University, Sept. 1986.

[316] T.A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, Dec. 1982.

[317] T.A. Marsland and J. Schaeffer. *Computers, Chess, and Cognition.* Springer-Verlag, New York, 1990.

[318] 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.

[319] D. McAllester. Truth maintenance. In *Proceedings of AAAI'90*, pages 1109–1116, 1990.

[320] 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.

[321] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.

[322] 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.

[323] K. Mehlhorn. *Data Structures and Algorithms: Graph Algorithms and NP-Completeness.* Springer-Verlag, Berlin, 1984.

[324] H.-M. Méjean, H. Morel, and G. Reynaud. A variational method for analyzing unit clause search. *SIAM J. on Computing*,

[325] 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.

[326] D.P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Pitman, London, 1990.

[327] 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.

[328] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of AAAI'92*, pages 459–465, Jul. 1992.

[329] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[330] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:117–133, 1983.

[331] Burkhard Monien, Ewald Speckenmeyer, and Oliver Vornberger. Superlinear Speedup for Parallel Backtracking.

[332] 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.

[333] 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.

[334] B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[335] 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.

[336] D. Navinchandra. *Exploration and Innovation in Design.* Springer Verlag, Sadeh, 1990.

[337] 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.

[338] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[339] 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.

[340] A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms.* Academic Press, New York, 1975.

[341] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, California, 1980.

[342] M. Nishizawa. Partitioning of logic units. *Fujitsu Scientific and Technical Journal*, 7(2):1–13, Jun. 1971.

[343] D. Pager. On the efficient algorithm for graph isomorphism. *J. of ACM*, 17(4):708–714, Jan. 1970.

[344] 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.

[345] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, 1982.

[346] 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.

[347] E. Koutsoupias, and C. H. Papadimitriou. 1992. On the greedy algorithm for satisfiability. *Information Processing Letters* **43**, pp. 53–55.

[348] P.M. Pardalos and J.B. Rosen. *Constrained Global Optimization: Algorithms and Applications.* Springer Verlage, New York, 1987.

[349] 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.

[350] W. Patterson. *Mathematical Cryptology.* Rowman & Littlefield, New Jersey, 1987.

[351] 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.

[352] J. Pearl. *Heuristics.* Addison-Wesley, Reading, 1984.

[353] D. Pehoushek. SAT problem instances and algorithms. Private Communications, Stanford University, 1992.

[354] 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.

[355] L.M. Pereira and A. Porto. *Selective Backtracking*, pages 107–114. Academic Press, 1982.

[356] C. Petrie. Revised dependency-directed backtracking for default reasoning. In *Proceedings of AAAI'87*, pages 167–172, 1987.

[357] J. Peysakh, *A fast algorithm to convert boolean expressions into CNF*, IBM Computer Science RC 12913 (#57971) (1987), Watson, New York.

[358] 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.

[359] D. K. Pradhan, editor. *Fault-Tolerant Computing: Theory and Practice.* Prentice-Hall, Englewood Cliffs, 1986.

[360] D. Prawitz. An improved proof procedure. *Theoria*, 26(2):102–139, 1960.

[361] P. Purdom. Survey of average time SAT performance. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[362] P.W. Purdom. Tree size by partial backtracking. *SIAM J. on Computing*, 7(4):481–491, Nov. 1978.

[363] P.W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[364] P.W. Purdom. Solving satisfiability with less searching. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 6:510–513, 1984.

[365] P.W. Purdom. A survey of average time analyses of satisfiability algorithms. *J. of Information Processing*, 13(4):449–455, 1990.

[366] P.W. Purdom. Average time for the full pure literal rule. *Information Sciences*, 1994 78:269–291.

[367] P.W. Purdom and C.A. Brown. An analysis of backtracking with search rearrangement. *SIAM J. on Computing*, 12(4):717–733, Nov. 1983.

[368] P.W. Purdom and C.A. Brown. The pure literal rule and polynomial average time. *SIAM J. on Computing*, 14:943–953, 1985.

[369] P.W. Purdom and C.A. Brown. Polynomial-average-time satisfiability problems. *Information Science*, 41:23–42, 1987.

[370] P.W. Purdom, C.A. Brown, and E.L. Robertson. Backtracking with multi-level dynamic search rearrangement. *Acta Informatica*, 15:99–113, 1981.

[371] P.W. Purdom and G.N. Haven. Backtracking and probing. Technical Report No. 387, Dept. of Computer Science, Indiana University, Aug. 1993.

[372] 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.

[373] 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.

[374] R. Puri and J. Gu. Area efficient synthesis of asynchronous interface circuits. In *Proc. of IEEE International Conference on Computer Design*, 1994.

[375] 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.

[376] R. Puri and J. Gu. A BDD SAT solver for satisfiability testing. *Annals of Mathematics and Artificial Intelligence*, 1996, to appear.

[377] 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.

[378] C. V. S. Rao and N. N. Biswas. Minimization of Incompletely Specified Sequential Machines. *IEEE Trans. on Computers*, C-24:1089–1100, Nov. 1975.

[379] R.C. Read and D.G. Corneil. The graph isomorphism disease. *J. of Graph Theory*, 1:339–363, 1977.

[380] 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.

[381] 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.

[382] M. Reichling. A simplified solution of the n queens' problem. *Information Processing Letters*, Vol. 25:253–255, June 1987.

[383] M. Resende and T. Feo. A GRASP for MAX-SAT. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[384] R.L. Rivest. *Cryptography. In* Handbook of Theoretical Computer Science. *J.V. Leeuwen, editor*, chapter 13, pages 719–756. The MIT Press, Cambridge, 1990.

[385] 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.

[386] J.A. Robinson. A machine–oriented logic based on the resolution principle. *Journal of the ACM*, pages 23–41, 1965.

[387] A. Rosenfeld. Computer vision: Basic principles. *Proceedings of the IEEE*, 76(8):863–868, Aug. 1988.

[388] 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.

[389] W. Rosiers and M. Bruynooghe. Empirical study of some constraint satisfaction problems. In *Proceedings of AIMSA'86*, North Holland, Sept. 1986.

[390] 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.

[391] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, 1995.

[392] 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.

[393] 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.

[394] Y.G. Saab and V. B. Rao. Combinatorial optimization by stochastic evolution. *IEEE Transactions on CAD*, CAD-10(4):525–535, Apr. 1991.

[395] 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.

[396] A. Samal. *Parallel Split-Level Relaxation.* PhD thesis, Dept. of Computer Science, Univ. of Utah, Aug. 1988.

[397] 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.

[398] A. Samal and T.C. Henderson. Parallel consistent labeling algorithms. *International Journal of Parallel Programming*, 1988.

[399] T.J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 216–226, 1978.

[400] W.L. Schiele. Improved compaction by minimized length of wires. In *Proc. of the 20th ACM/IEEE Design Automation Conference*, pages 121–127, 1983.

[401] 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.

[402] B.M. Schwartzschild. Statistical mechanics algorithm for Monte Carlo optimization. *Physics Today*, 35:17–19, 1982.

[403] P. Schwarz and A. Spector. Synchronizing shared abstract types. *ACM Trans. on Computer Systems*, Aug. 1984.

[404] 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.

[405] C. Sechen and A.L. Sangiovanni-Vincentelli. The timberwolf placement and routing package. In *Proc. of the 1984 Custom Integrated Circuit Conf.*, May 1984.

[406] B. Selman. Private Communications, Aug. 1992.

[407] B. Selman and H. Kautz. Escape mechanisms for local search for satisfiability testing. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[408] 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.

[409] 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.

[410] M.J. Shensa. A computational structure for the propositional calculus. In *Proceedings of IJCAI*, pages 384–388, 1989.

[411] 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.

[412] 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.

[413] 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.

[414] P. Siegel. *Representation et Utilization de la Connaissances en Calcul Propositionel.* PhD thesis, University Aix-Marseille II, 1987.

[415] J.C. Simon. Off-line cursive word recognition. *Proceedings of the IEEE*, 80(7):1150–1161, Jul. 1992.

[416] 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.

[417] R. Sosič and J. Gu. A polynomial time algorithm for the n-queens problem. *SIGART Bulletin*, 1(3):7–11, Oct. 1990, ACM Press.

[418] 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.

[419] R. Sosič and J. Gu. 3,000,000 queens in less than one minute. *SIGART Bulletin*, 2(2):22–24, Apr. 1991, ACM Press.

[420] R. Sosič and J. Gu. A parallel local search algorithm for satisfiability (sat) problem. *Submitted for publication*, 1993.

[421] R. Sosič and J. Gu. A parallel local search algorithm for the n-queen problem. *Submitted for publication*, 1993.

[422] R. Sosič and J. Gu. Efficient local search with conflict minimization. *IEEE Trans. on Knowledge and Data Engineering*, 6(5):661–668, Oct. 1994.

[423] R. Sosič and J. Gu. Quick n-queen search on VAX and Bobcat machines. CS 547 AI Class Project, Winter Quarter Feb. 1988.

[424] R. Sosič, J. Gu, and R. Johnson. The Unison algorithm: Fast evaluation of Boolean expressions. *Communications of ACM*, accepted for publication in 1992.

[425] R. Sosič, J. Gu, and R. Johnson. A universal Boolean evaluator. *IEEE Trans. on Computers*, accepted for publication in 1992.

[426] J. Soukup. Circuit layout. In *Proceedings of the IEEE*, pages 1281–1304, Oct. 1981.

[427] E. Speckenmeyer. Personal Communication. 1996.

[428] R. Stallman and G.J. Sussman. Forward reasoning and dependency directed backtracking. *Artificial Intelligence*, 9(2):135–196, 1977.

[429] 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.

[430] J.A. Stankovic. Real-time computing systems: The next generation. Manuscript. Feb. 18, 1988.

[431] L. Sterling and E. Shapiro. *The Art of Prolog, Advanced Programming Techniques.* The MIT Press, Cambridge, Massachusetts, 1986.

[432] 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.

[433] 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.

[434] C.Y. Suen, M. Berthod, and S. Mori. Automatic recognition of handprinted characters – the state of the art. *Proceedings of the IEEE*, 68(4):469–487, Apr. 1980.

[435] 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.

[436] R. P. Swaminathan and D. K. Wagner. The arborescence–realization problem. *Discrete Applied Mathematics* **59**, pages 267–283, 1995.

[437] 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.

[438] A. Thayse and M. Davio, *Boolean differential calculus and its application to switching theory*, IEEE Trans. on Computers **22** (1973), 409–420.

[439] 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.

[440] 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.

[441] K. Truemper. Alpha-balanced graphs and matrices and GF(3)-representability of matroids. *Journal of Combinatorial Theory B* **32**, pages 112–139, 1982.

[442] K. Truemper. Monotone Decomposition of Matrices. Technical Report UTDCS-1-94, University of Texas at Dallas. 1994.

[443] K. Truemper. Polynomial algorithms for problems over d-systems. Presented in the 3rd International Symposium on AI & Mathematics, Jan. 1994.

[444] K. Truemper. In preparation, expected 1997. Effective Logic Computation.

[445] 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.

[446] K. J. Turner. *Computer Perception of Curved Objects Using a Television Camera.* PhD thesis, Univ. Edinburgh, 1974.

[447] 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.

[448] 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.

[449] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, 1982.

[450] J.R. Ullman. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.

[451] 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.

[452] 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.

[453] A. Urquhart. Hard examples for resolution. *J. of ACM*, 34:209–219, 1987.

[454] M. van der Woude and X. Timermans. Compaction of hierarchical cells with minimum and maximum compaction constraints. In *Proc. Intl. Symposium on Circuits and Systems*, pages 1018–1021, 1983.

[455] 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.

[456] 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.

[457] 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.

[458] B. Wah and G.J. Li. *Computers for Artificial Intelligence Applications.* IEEE Computer Society Press, Washington D. C., 1986.

[459] B. W. Wah, G. J. Li, and C. F. Yu. Multiprocessing of combinatorial search problems. *IEEE Computer*, 18(6):93–108, Jun. 1985.

[460] B.W. Wah, editor. *New Computers for Artificial Intelligence Processing. IEEE Computer*, volume 20, number 1. IEEE Computer Society Press, 1987.

[461] B.W. Wah, M.B. Lowrie, and G.-J. Li. Computers for symbolic processing. *Proceedings of the IEEE*, 77(4):509–540, Apr. 1989.

[462] D. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271, MIT, Nov. 1972.

[463] 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.

[464] 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.

[465] 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.

[466] W. Wang and C.K. Rushforth. Structured partitioning for channel assignment problem. *IEEE Trans. on Vehicular Technology*, 1996.

[467] W. Wang and C.K. Rushforth. An adaptive local search algorithm for channel assignment problem. *IEEE Trans. on Vehicular Technology*, 1996, to appear.

[468] M. B. Wells. *Elements of Combinatorial Computing.* Pergamon Press, Oxford, 1971.

[469] H.P. Williams. Linear and integer programming applied to the propositional calculus. *Systems Research and Information Sciences*, 2:81–100, 1987.

[470] P.H. Winston. *Artificial Intelligence.* Addison-Wesley, Reading, 1984.

[471] 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.

[472] L.C. Wu and C.Y. Tang. Solving the satisfiability problem by using randomized approach. *Information Processing Letters*, 41:187–190, 1992.

[473] T. Y. Young and K. S. Fu, editors. *Handbook of Pattern Recognition and Image Processing.* Academic Press, Orlando, 1986.

[474] C. F. Yu. *Efficient Combinatorial Search Algorithm.* PhD thesis, School of Electrical Engineering, Purdue University, West Lafayette, IN, Dec. 1986.

[475] 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.

[476] 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.

[477] R. Zabih and D. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of AAAI'88*, pages 155–160, 1988.

[478] V.N. Zemlyachenko, N.M. Korneeko, and R.I. Tyshkevich. Graph isomorphism problem. *J. of Soviet Mathematics*, 29:1426–1481, 1985.

[479] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Trans. on Software Engineering*, SE-14(9):1327–1341, Sept. 1988.

[480] 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.