An Exact Parallel Algorithm for the Maximum Satisfiability Problem

by

Judith D. Furman

Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Science in Mathematics with Operations Research Option

New Mexico Institute of Mining and Technology

Socorro, New Mexico

August, 1996

ACKNOWLEDGEMENT

I'd like to thank the members of my committee for their support. In particular I would like to thank my advisor, Brian Borchers, for his support and help in all aspects of my degree at New Mexico Tech. I would also like to thank my husband, Ben, for his love, support, and many hours of reading my rough drafts. I would also like to extend my thanks to those professors that spent time getting to know me and who encouraged me when I struggled. Thanks.

This thesis was typeset with LaTeX[1] by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for the New Mexico Institute of Mining and Technology thesis format was adapted by Gerald Arnold from the LaTeX macro package for The University of Texas at Austin by Khe-Sing The.

ABSTRACT

I present a parallel algorithm for solving the maximum satisfiability problem. I will compare the parallel algorithm to a sequential algorithm. Both algorithms use a two–phase approach. The first phase uses the GSAT heuristic to obtain a good upper bound on the number of unsatisfied clauses. The second phase uses a Davis-Putnam-Loveland like algorithm to solve the problem. The parallel algorithm uses a "master–slave" paradigm. The master process keeps a list of subproblems and gives a slave process a new subproblem when it finishes its current one. Efficiency was between 15% and 85% for most problems. Linear speedup was not achieved.

Table of Contents

List of Tables

Chapter 1

Introduction

Propositional calculus is a type of logic that involves atomic propositions and various logical connectives, such as ‘not’, ‘and’, ‘or’, and ‘implies’, whereas predicate calculus also includes quantifiers such as ‘for every’, and ‘for some’ [23, 26]. We are only interested in propositional calculus when dealing with satisfiability problems, so we will limit our discussion to propositional calculus.

A propositional logic formula F can always be written in conjunctive normal form as the conjunction of m clauses, where each clause is the disjunction of a set of literals and each literal is either a variable or the negation of a variable. Since a clause is the disjunction of a set of variables and a set of negated variables, clauses can be written in the form:

$$C = (\vee_{y_i \in C^+} y_i) \vee (\vee_{y_i \in C^-} \overline{y_i})$$

where y_i is a logical variable and $\overline{y_i}$ is the negation of y_i. A clause is true if any of the variables in C^+ are true or if any of the variables in C^- are false.

The *satisfiability* problem (SAT) is to determine whether every element in a collection of clauses can be satisfied simultaneously. Simply stated, can the formula $C_1 \wedge \cdots \wedge C_k$ be made true?

It has been shown that there is an algorithm for the 2–SAT problem which is linear as per the number of variables specified [15]. However, the general satisfiability problem with 3 or greater variables per clause is a typical NP-complete problem [15, 27].

The satisfiability problem can also be formulated as a 0-1 integer programming problem. For each clause in the satisfiability problem, we construct a constraint in which the logical variable becomes a 0-1 variable x_i. If x_i is negated in the satisfiability problem, then it is written as $(1 - x_i)$ in the constraint. For example, the clause

$$\overline{x_1} \vee x_2 \vee \overline{x_4}$$

becomes the constraint

$$(1 - x_1) + x_2 + (1 - x_4) \geq 1$$

In general, the satisfiability problem becomes an integer programming feasibility problem of the form

$$\begin{array}{rl} & \min \quad 0 \\ \text{subject to} & \displaystyle\sum_{y_i \in C_j^+} x_i + \sum_{y_i \in C_j^-} (1 - x_i) \geq 1 \quad j = 1, .., m \\ & x \quad \text{binary} \end{array}$$

where m is the number of constraints or clauses of the SAT problem [6]. If the integer programming problem has a feasible solution, then the SAT problem is satisfiable. If not, then the problem is unsatisfiable.

Now that satisfiability has been defined, we look at another problem. Many problems are not satisfiable, so the question one might ask of them is:

"How many of the clauses can be satisfied?" This problem is the *Maximum Satisfiability* problem (MAX–SAT). The goal of MAX–SAT is to find a truth assignment which satisfies the largest possible number of a given set of clauses. This can be stated as follows: Given a set of clauses $\{C_1, \ldots, C_k\}$ what is the maximum number that can be true simultaneously?

Since MAX–SAT is the optimization version of the NP–complete SAT problem, MAX–SAT falls into the class of NP–hard problems. Even though there is an algorithm for the 2–SAT problem which is linear in the number of variables, MAX–SAT is NP–hard even if there are only 2 variables per clause.

The MAX–SAT problem can also be formulated as an integer programming problem [6]. We introduce an auxiliary variable z_j for each clause and let $z_j = 0$ if clause j is satisfied and $z_j = 1$ if clause j is not satisfied. The integer programming formulation is:

$$\min \sum_{j=1}^{m} z_j$$

$$\text{subject to} \sum_{y_i \in C_j^+} x_i + \sum_{y_i \in C_j^-} (1 - x_i) + z_j \geq 1 \quad j = 1, .., m$$

$$x, z \text{ binary.}$$

Satisfiability problems have been solved using various methods. Some of the methods used are resolution [29, 12, 22, 23], the Davis-Putnam-Loveland Procedure(DPL) [22, 24, 21, 14], Branch and Bound [22, 3], cutting planes [22, 23], and heuristic methods, some of which are GSAT [25, 32, 31, 30, 33], GRASP [28], and simulated annealing [31, 33]. There have also been several parallel algorithms written [4, 19].

Some methods used to solve MAX–SAT problems are GSAT [25, 31], a modification of Davis-Putnam-Loveland algorithm [5], a Branch and Cut algorithm [6], simulated annealing [20, 32], and a modified tabu search called Steepest Ascent Mildest Descent [20]. Also, Goemans and Williamson [17] have described an .7584-approximation algorithm for MAX–SAT problems which uses semidefinite programming. Of the above methods, there are two complete methods: the modification of the Davis-Putnam-Loveland algorithm and the Branch and Cut Algorithm. Cheriyan et. al. [9] also have a linear programming approach for solving MAX–2–SAT problems. At this point in time, there are no published parallel algorithms for the MAX–SAT problem.

Chapter 2

Survey of Literature

2.1 Resolution

Resolution, developed by Robinson in 1965 [29], is a complete symbolic method that has been used to solve satisfiability problems. It was designed to solve first–order predicate calculus problems but has also been used to solve propositional calculus problems, and resolution applied to propositional calculus is called ground resolution [22, 26].

Resolution works in the following way: Two clauses (parents) have a resolvent when x_i appears in one clause and $\overline{x_i}$ appears in another clause. Their resolvent is a clause that contains all of the literals in either of the two parent clauses except for x_i and $\overline{x_i}$, which cancel. For example the following parents,

$$x_1 \vee \overline{x_2} \vee x_3$$

and

$$\overline{x_1} \vee \overline{x_2} \vee x_4$$

produce the resolvent,

$$\overline{x_2} \vee x_3 \vee x_4.$$

Note that the resolvent follows logically from the conjunction of the parent clauses. Another routine used together with resolution is called absorption. A clause C_k is said to be *absorbed* by another C_i if all the literals in C_k appear in C_i. For example, the clause $x_3 \vee x_4$ would be absorbed by the clause $\overline{x_2} \vee x_3 \vee x_4$. Because of this redundancy, we can remove absorbed clauses.

The resolution algorithm contains three steps that are executed iteratively. The first step is to determine if any clause in our set S is the empty clause, in which case the problem is *not* satisfiable, and we can stop. At the second step, we remove all absorbed clauses. By checking for absorption and removing absorbed clauses, we ensure that the procedure terminates [22]. The third step requires that we generate all resolvents with parents in the set of clauses S. If no resolvents exist, then the problem is satisfiable. Otherwise, we delete any clause from S which is absorbed by any of the new resolvents to simplify the problem. Then, we add the new resolvents to the set of clauses S and start over at step 1.

For example in the given set of clauses

$$x_1$$

$$\overline{x_1}$$

$$x_1 \vee x_2$$

the first two clauses resolve to an empty clause. Thus the set of clauses is not satisfiable.

However, the set of clauses

$$x_1 \vee x_2$$

$$\overline{x_1} \vee x_2$$

produce the resolvent

$$x_2$$

giving us the new set of clauses

$$x_1 \vee x_2$$

$$\overline{x_1} \vee x_2$$

$$x_2$$

Since there are no more resolution possibilities, the problem is satisfiable. It has been shown that resolution with absorption is a logically complete algorithm [22].

The drawback to using resolution is the time complexity of the method. Researchers have shown that, using resolution, running time is not only exponential in the number of variables in the worst case, but it also tends to become rapidly impractical in practice as the size of the problem increases [22].

2.2 Cutting Planes

Resolution has connections to an integer programming approach called cutting planes. The integer programming formulation of the satisfiability and the maximum satisfiability problem can be solved in several ways, one of which is by cutting planes.

Resolvents in an inference problem form a subset of the class of Chvatal cuts for linear programming relaxation problems [22, 23]. Chvatal cutting planes are formed by taking positive linear combinations of the constraints, and rounding the coefficients of the new constraints in a way that ensures that integer feasible solutions are not cut off. These cuts are used to further reduce the convex hull of the solution, where the convex hull is a bounded polyhedron, or polytope, which contains the solutions to the linear programming problem. In our case this will be an integral polytope. These cuts are used to further define this polytope and narrow the search area. If the linear programming relaxation of the integer programming problem ($x \geq 0$) is solved, and we obtain a non–integer solution, then we must somehow "cut" that solution out. Cutting planes accomplish this. This procedure solves the linear programming relaxation of the integer programming problem and adds cutting planes until the linear programming relaxation problem produces an integer solution. If no feasible solution is reached, then the set of constraints is unsatisfiable.

The following is an example of how resolvents form a subset of the class of Chvatal cuts for the linear programming relaxation problem [22]. We saw earlier that the two clauses

$$x_1 \vee \overline{x_2} \vee x_3$$

and

$$\overline{x_1} \vee \overline{x_2} \vee x_4$$

produced the resolvent

$$\overline{x_2} \vee x_3 \vee x_4.$$

The first two clauses are equivalent to the following two constraints respectively

$$x_1 + (1 - x_2) + x_3 \geq 1$$

and

$$(1 - x_1) + (1 - x_2) + x_4 \geq 1.$$

If we add the two constraints together, we get the following inequality, which is a Chvatal cut:

$$1 + 2(1 - x_2) + x_3 + x_4 \geq 2.$$

Upon simplifying we get

$$2(1 - x_2) + x_3 + x_4 \geq 1.$$

We can drop the factor of 2 from the $(1 - x_2)$ term because if x_2 was equal to 1, then one of the other variables must equal 1 to make the inequality true, and if x_2 was equal to 0, then the inequality is satisfied whether or not we have the factor of 2. So, we are left with:

$$(1 - x_2) + x_3 + x_4 \geq 1.$$

If we convert this inequality, which is a Chvatal cutting plane, to logical form, we get

$$\overline{x_2} \vee x_3 \vee x_4$$

which was the resolvent obtained from the first resolution example. So from this example, we can see that resolvents form a subset of the class of Chvatal cutting planes.

2.3 Davis-Putnam-Loveland

One of the oldest and most popular exact methods for solving SAT problems is the Davis–Putnam–Loveland procedure [21]. The algorithm begins with a set of clauses, S. The algorithm proceeds with several subroutines, although one of the subroutines used is optional. The optional algorithm can help to further simplify the problem.

There is one subroutine of importance when solving this problem. It is unit clause resolution or forward chaining. Firstly, a unit clause is a clause which contains only one variable. In unit clause resolution, pick any unit clause u in the set, S, of clauses. If none exist, then exit the subroutine. Otherwise, force u to be true by removing any clauses that contain u and removing the occurrence of $\overline{u}$ from any clauses. Repeat the procedure until all unit clauses have been found. If there are any empty clauses, then S is unsatisfiable. If there are no clauses left, then S is satisfiable.

If there are clauses left, then we use a second (optional) subroutine called monotone variable fixing [21]. In monotone variable fixing, we look for monotone variables, which are variables that appear negated in every occurrence or posited in every occurrence. If x_i is a monotone variable that is negated in every occurrence, then we set x_i = FALSE and remove all the clauses containing x_i. If x_i is posited in every occurrence, then set x_i = TRUE and remove all clauses containing x_i. If there are none, then exit the subroutine.

The main routine starts with the set of clauses, S, and applies unit clause resolution and monotone variable fixing. Although monotone variable fixing is optional, it is useful in helping to simplify the problem. If there are no clauses left, then S is satisfiable. If not, then continue by selecting a variable on which to branch, say x_i. Branch by creating two subsets of clauses, $S \cup \{x_i\}$ and $S \cup \{\overline{x_i}\}$. The first subset forces x_i to be true, and the second subset forces

x_i to be false. Upon solving these two subproblems, if either subset generates no clauses upon applying unit resolution and monotone variable fixing, then the original set of clauses, S, is satisfiable. If both subsets generate empty clauses when applying unit resolution and monotone variable fixing, then the original set of clauses S is unsatisfiable. For the subset(s) that still contain clauses, we must pick a new variable on which to branch and continue until we generate an empty set of clauses (no clauses are left after unit resolution and monotone variable fixing) in which case either, S, is satisfiable or we generate an empty clause in which case S is unsatisfiable [21, 22].

2.4 Branch and Bound

The integer programming formulations of SAT and MAX–SAT can also be solved by Branch and Bound [3, 22]. This method is guaranteed to solve the problem exactly [3].

Branch and bound algorithms begin by solving the linear programming (LP) relaxation problem, which is attained by making all the variables continuous on [0,1]. If a variable, x_i, is found to be fractional at optimality, then the algorithm constructs two new subproblems in which the fractional variable, x_i, is fixed at 0 or 1. The algorithm continues solving subproblems and creating new subproblems as they arise until all subproblems have been eliminated from consideration [6].

A subproblem can be eliminated from consideration for the following reasons: 1) it is infeasible, 2) the solution to the subproblem has a higher objective function value than the smallest known integer solution, or 3) the solution to the subproblem is an integer solution. After all subproblems have been considered, the optimal solution is the best integer solution obtained while solving subproblems [6, 22].

For SAT problems, the set of clauses, S, is satisfiable if and only if the objective function of the best integer solution is 0. For MAX–SAT problems, the objective function associated with the best integer solution is the number of unsatisfied clauses in the best solution. Because we have enumerated all possible subproblems, we know that the solution obtained contains no error.

2.5 Branch and Cut

Another exact method for solving MAX–SAT problems is branch and cut [6]. The algorithm proceeds much like branch and bound algorithms. The algorithm begins by running GSAT to obtain an initial upper bound on the solution.

At each node of the tree, we solve the linear programming relaxation problem. At this point, cuts are generated. Joy, Mitchell, and Borchers [6] used resolution cuts and odd cycle inequalities. Resolution cuts have been explained in the resolution section, and information about odd cycle inequality cuts can be read in Joy, Mitchell, and Borchers [6]. After cuts have been made, bounding is applied by fixing variables (i.e. monotone variable fixing). If there remain any fractional variables after cuts have been applied and variables fixed, branching is performed as in branch and bound. Two new subproblems are created and solved in the above fashion. This procedure is continued until no fractional variables appear at any node.

Like branch and bound, the objective function value of the best integer solution gives the information that we need about our MAX–SAT solution.

2.6 GSAT

Other than exact methods, heuristics may also be used to gain solutions to SAT and MAX–SAT problems. Heuristics are techniques which seek good (i.e. near–optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close a feasible solution is to optimality [1].

One such method used to solve SAT and MAX–SAT problems is GSAT [31, 25, 30, 32, 33, 25]. GSAT is a local search heuristic, meaning that it can find locally optimal solutions which may or may not be globally optimal. The algorithm is as follows: Pick a random solution with which to begin. Determine how many clauses are satisfied by this random solution. To pick the next solution we do the following: For each variable, determine how many clauses would be satisfied and how many would be unsatisfied by flipping the variable from true to false or from false to true. Flip the variable which gives the greatest net increase in the number of satisfied clauses. If more than one variable gives the same maximum amount of increase, randomly pick a variable, from that set, to flip. By randomly choosing the variable to flip we should not generate the same solution over and over again. The number of clauses satisfied is updated with this information and the flip is calculated by changing appropriate values in the data structures.

This algorithm flips *Max–Flips* number of variables before the algorithm gives up and restarts, and it is normally restarted *Max–Tries* number of times, where the user specifies the variables *Max–Flips* and *Max–Tries*. This does not work very well for problems whose SAT structures give no clue about the location of maximums. The reason for this is that by only flipping one variable we are searching a local neighborhood of solutions. If we did not start anywhere near the globally optimal solution, and the neighborhood does not give information about the globally optimal solution, then we will not find the globally optimal solution but a local one. However, it works well to find a local maximum, so by starting at random solutions there is a possibility of finding a satisfying assignment to the SAT problem we are looking at. Still, if it returns an assignment that does not satisfy all the clauses, we cannot conclude that the problem is unsatisfiable because heuristics cannot guarantee that the solution found is optimal [32].

For MAX–SAT the algorithm is essentially the same except we are not just trying to determine if a formula is satisfiable, but we want to know the maximum number of clauses that can be satisfied. So we keep track of the number of satisfied clauses at any given time and determine how many can be satisfied. Again, if we find NUM clauses satisfied, then we are not guaranteed that this is the optimal number of clauses that can be satisfied (unless NUM = number of clauses, in which case the formula is satisfiable) because GSAT cannot guarantee that the solution found is optimal [32].

There have been several modifications made to GSAT to try to get better accuracy when solving SAT and MAX–SAT problems. One modification is introducing a “random walk”. Every time we need to flip a variable, we either choose a variable according to the GSAT strategy explained above or, with some probability, p, we choose a variable occurring in some unsatisfied clause which we flip regardless of whether the number of satisfied clauses increases or decreases. This may allow us to move in a direction that will lead us to a global maximum, rather than getting stuck in a local maximum [31].

2.7 Simulated Annealing

Another heuristic approach to solving MAX–SAT problems is simulated annealing [33, 20]. In this approach, we use a GSAT-like algorithm but choose to flip variables according to an annealing schedule. Starting with a random solution we repeatedly pick a random variable which we would like to flip and compute δ, the change in the number of unsatisfied clauses. If $\delta \leq 0$ (a downhill or sideways move), we make the flip. Otherwise, we flip the variable with probability $e^{-\delta/T}$, where T is the parameter called the *temperature*. The temperature may either be held constant, in which case the annealing corresponds to the Metropolis algorithm, or the temperature may be decreased slowly from a high temperature to near zero according to an annealing schedule. Most annealing schedules change the temperature by multiplying by some constant factor that is less than 1. If we do not make either of these moves, then we go back and pick another variable which we would like to flip.

Another consideration has to do with how to stop the annealing algorithm. One way is to stop the algorithm when only sideways moves have been made for a set number of flips. Another way is to limit the number of flips to *Max–Flips* before restarting the algorithm from another random solution. Given a finite cooling schedule, simulated annealing is not guaranteed to find a global optimum, i.e. an assignment that satisfies all of the clauses or in the case of MAX–SAT it is not guaranteed to find the optimal number of clauses that can be satisfied [32].

According to experimental results from Selman, Levesque, and Mitchell [32], GSAT with a random walk strategy generally performed significantly better than GSAT or simulated annealing. It was reported that no temperature could be found that performed better than GSAT on any of the problems tested.

2.8 Steepest Ascent Mildest Descent

Steepest Ascent Mildest Descent is a local search heuristic very similar to what is known as Tabu search [20]. The algorithm starts with an initial solution. We make changes along the direction of steepest ascent until we reach a local optimum. If this local optimum is better than any previous solution, we save the new solution and its value. Then we make changes along the direction of mildest descent where a reverse change is forbidden for a given number of iterations. By restricting reverse changes, we hope to "climb out" of a local maximum, thereby finding a new one. The algorithm terminates when no better solution has been found for a set number of iterations, which is predetermined by the programmer.

Hansen and Jaumard [20] compared simulated annealing to steepest ascent mildest descent (SAMD). They found that SAMD performed better and had a lower computing time than simulated annealing on the maximization of quadratic 0-1 functions and quadratic assignment problems (less than 30 facilities). SAMD also outperformed simulated annealing for random MAX–2–

SAT, MAX–3–SAT, and MAX–4–SAT problems.

2.9 Parallel Algorithms

The complexity of the Satisfiability problem leads to a desire to find a faster method of solving problems. There are many problems which cannot be solved exactly because there is not enough time to enumerate all possible solutions. Hence, parallel satisfiability algorithms can lend to solving larger problem instances. Another benefit of parallel algorithms is that we can solve the medium sized problems in less computational time.

Some important concepts in parallel computing involve determining how well the parallel algorithm compares to the sequential algorithm. Two of the most prominent methods of comparing parallel algorithms are *speedup* and *efficiency*. Speedup is defined as "the gain in computation speed achieved by using P processors with respect to a single processor" [18]. If T_1 is the time required by a "good" sequential algorithm and T_P is the time required by the parallel algorithm with P processors, then the speedup, SU, can be written as $SU = T_1/T_P$. The efficiency, E, of the parallel algorithm is defined as the effective utilization of the computing resources, or the ratio of the speedup to the number of processors used ($E = SU/P$). Speedup does not always increase linearly with the number of processors because there is only so much that one can parallelize.

When writing parallel algorithms, the goal is to have an efficiency rating of 1 or very close to 1. This means that the algorithm obtains linear speedup. However, because most problems have inherently sequential pieces which cannot be parallelized, linear speedup is very hard to obtain. Another reason for not obtaining linear speedup is communications overhead involved in the parallel code. Therefore, in order to obtain linear speedup, one must be very careful with the amount of work done by the parallel code and the amount of communication required to maintain the parallel workings of the code. It would be unwise to use parallel code that runs slower than the "good" sequential algorithm.

In parallel search algorithms, such as branch and bound, the speedup can greatly differ over executions, because the tree may be searched differently by different processors for different trials [18]. For example, an execution of the parallel version of a search algorithm may obtain a solution by visiting fewer nodes than the sequential code, or may search more nodes than the sequential code. The above two scenarios are referred to as speedup anomalies [13, 18]. The first scenario results in what is termed as a superlinear speedup or acceleration anomaly, which is evident when a speedup of greater than P is obtained, when P processors are used. The second scenario results in a deceleration anomaly, occurring when a speedup of less than P is obtained due to excessive work.

Parallel algorithms have been written for many related problems including: theorem proving using a divide and conquer strategy [8], local search [18], and branch and bound [16, 13]. Specifically, parallel algorithms have been written for the Satisfiability problem [19, 4]. As of yet, there have been no documented parallel algorithms written for the Maximum Satisfiability problem.

2.9.1 Parallel Branch and Bound

Several parallel algorithms have been written for solving branch and bound problems [13, 16]. Since there are many different architectures, discussion of all the different branch and bound algorithm is beyond the scope of this work. For a more complete discussion, please refer to Gendron and Crainic [16]. Here, we discuss a basic parallel branch and bound algorithm.

Most parallel branch and bound algorithms use the same bounding rules as sequential algorithms. One problem with parallel branch and bound algorithms deals with when to start the parallel processors. They cannot be started immediately because there are very few subproblems at the start of branch and bound. Therefore, it is necessary to perform some of the branch and bound algorithm sequentially. It is necessary to experiment with the amount of sequential work done to find the best strategy. The amount of sequential work done is dependent upon the problem and the architecture used.

Another main issue in parallel branch and bound algorithms is in determining how to distribute the work load. Again, this is dependent on the architecture used. Several different methods are used. Most store subproblems in a "global" list, which is normally in shared memory. This list stores either unsolved subproblems, solved subproblems, or a combination of both. In a message passing system, one (or more) master process keeps track of the list of subproblems and assigns subproblems to slave processes to be solved.

In more sophisticated implementations, the slaves can divide problems, and share the subproblems with other processes. Workload balancing also occurs when two (or more) processors share workload amongst themselves.

If one processor is finished, it can take some load from a processor that is very busy. In this way, we can eliminate processors from becoming idle.

The algorithms follow a particular procedure. First, some part of the problem is solved sequentially. When there are enough subproblems for all the processors, the parallel processors are "given" a subproblem(s) which they solve. Some of the subproblems created by the parallel processors are added to the global (or local) list. When a processor runs out of work, it returns to the list for another problems. The routine ends when all the subproblems have been solved. The above procedure is a very basic parallel branch and bound.

2.9.2 Parallel SAT Algorithms

As with parallel branch and bound algorithms, parallel SAT algorithms are very similar to their sequential counterparts. Böhm and Speckenmeyer [4] presented a fast SAT solver to a competition at the University of Paderborn in 1991/1992 [7]. At this competition, Böhm and Speckenmeyer's sequential SAT code had the best performance of 35 algorithms. Böhm and Speckenmeyer have taken their fast SAT solver algorithm and parallelized it. Although other parallel SAT codes have been written on different architectures, we will devote our attention to discussing the algorithm presented by Böhm and Speckenmeyer. Information about the other parallel SAT algorithms can be found in a paper by Gu [19].

Böhm and Speckenmeyer's parallel SAT solver is very similar to a parallel version of the Davis–Putnam–Loveland algorithm. The algorithm they use for choosing the next variable, is to choose the variable which appears most often in the smallest clauses. The hope is to reduce small clauses to size 1 which collapse during unit clause tracking. This should help the code to run faster. The data structures used are designed so that access is fast.

One of the distinguishing elements of Böhm and Speckenmeyer's work is that they use a workload balancing scheme that seems to work very well. Workload balancing comes into play when a processor does not have enough work to do. Workload balancing is performed with the following steps: 1) each processor calculates the amount of work that it has; 2) the last processor adds the amount of work over all processors; 3) the last processor calculates the optimal load that each processor should have (within some tolerance); 4) each processor then calculates the overload; and 5) depending on the amount of overload or lack thereof will determine if load is sent or received and which neighbor to send to or receive from.

The algorithm proceeds with the following steps: The input is divided into subproblems and distributed to all the processors. Each processor actually runs two processes in parallel. One of the processes performs the fast sequential code for solving the subproblems. The other process is a balancer. The "worker" processor communicates with the "balancer" processor when it needs more work, or when the "balancer" wants to give the "worker" more work. Periodically, the "balancer" estimates the workload of its "worker." Based on this information, the workload balancing procedure is performed. Also communicated to the "balancer" process are updated solutions or whether it is time to quit.

Experiments were performed on random k–SAT formulas, for $k = 2$, 3, and 4. The ratio of clauses to variables was varied, and it was found that the code obtained very near to linear speedup.

Chapter 3

A Parallel Davis-Putnam-Loveland (DPL) Algorithm for MAX-SAT

3.1 Davis-Putnam-Loveland with GSAT – A two-phase approach

Another method for solving the MAX–SAT problem is a two–phase algorithm which combines a heuristic technique to find a good solution and then uses an exact method to enumerate all possible choices. The code written by Borchers and Furman [5] first uses the GSAT procedure to obtain a good upper bound *ub* on the number of unsatisfied clauses in an optimal solution and then uses a modification of the Davis–Putnam–Loveland procedure that enumerates all possible truth assignments.

In this procedure, we have a partially specified truth assignment in which some subset of the variables have been set to values of true and false. If this solution produces a better upper bound on the number of unsatisfied clauses, then we update *ub* as the new record solution. We must also keep track of *unsat*, the number of clauses left unsatisfied by the current solution.

If $unsat \geq ub$ then the current partial solution cannot be further extended to yield a solution better than the current incumbent solution. We discard this partial solution and backtrack to another partial solution. We backtrack by finding the variable that was set to false most recently, set it to true, and then continue processing.

If $unsat = ub - 1$ then we perform unit clause tracking. A unit clause is one in which all but one of the literals are fixed at false, or fixed at true if they are negated. Any literals left in a unit clause can be fixed at true (or false if they are negated) to make the clause true. If not, then $unsat$ would increase to a value greater than ub. After unit clause tracking, we update the value for ub.

After unit clause tracking, we perform monotone variable fixing. A monotone variable is one which appears negated or posited in every occurrence. If x_i is monotone and posited in every occurrence, then set x_i to true and remove all clauses containing x_i. If x_i is monotone and negated in every occurrence, then set x_i to false and remove all clauses containing $\overline{x_i}$. Exit the routine when there are no more monotone variables.

If $unsat < ub$ then we must continue processing by adding another logical variable to the current partial solution. Since this variable must be tried at both true and false, this creates two new subproblems, which we refer to as "branching". The algorithm executes this branching by first selecting the clauses with the smallest number of unfixed literals and then by selecting the unfixed variable which appears in the largest number of these clause. We then continue to the next iteration of the algorithm.

3.2 Parallel MAX-SAT Algorithm

This parallel MAX–SAT code uses the "master–slave" paradigm. The basic algorithm that is used is the two phase Davis–Putnam–Loveland algorithm (dpl) [5], which was described earlier in Section 3.1. A brief sketch of

```
Step 1   Spawn slave tasks
Step 2   Send the following to the slaves:
             number of variables
             number of clauses
             an array with the problem
Step 3   Receive best_num_sat, from the slaves' GSAT routine
Step 4   Send slaves the following:
             a set of six variables occurring in the most clauses
             the best of the GSAT solutions
Step 5   Loop until all subproblems have been sent out:
             receive the following information from a slave:
                 number of clauses left unsatisfied in an optimal solution
                 number of branches performed by the subproblem
             send a new subproblem to the slave
Step 6   Wait for the slaves to send information about all outstanding subproblems
Step 7   Process the last information about subproblems received from the slaves
Step 8   Send exiting signal to the slaves
Step 9   Print out results and quit
```

Table 3.1: Outline of the "master" algorithm

the master and slave "modules" are listed in Tables 3.1 and 3.2 respectively.

This code is run in a parallel environment, called PVM. Researchers in conjunction with Oak Ridge National Laboratories developed PVM, or Parallel Virtual Machine [2]. PVM allows a diverse collection of machines to simulate the parallel environment. The machines appear as one large distributed–memory computer. PVM supplies routines that allow function tasks to start up other tasks and to communicate with one another. Applications can be parallelized by using the message passing constructs which are similar to those in most distributed–memory machines [2].

```
Step 1   Start up the slave task
Step 2   Receive information from the master
             number of clauses
             number of variables
             an array with the problem
Step 3   Read the array into the structure used for solving problems
Step 4   Run the GSAT routine and report findings
Step 5   Run dpl on an initial subproblem and report findings to the master
Step 6   Loop until told to stop by the master
             Receive a new subproblem
             Run dpl and report findings to master
Step 7   Exit the parallel environment
```

Table 3.2: Outline of the “slave” algorithm

In the master program, slaves are spawned in the PVM environment. Initial information like the number of clauses, the number of variables, and an array with the problem are reported to the slave processes. The slaves then run the GSAT heuristic. When the slaves are done with their GSAT computations, the master combines the knowledge received and sends the slaves the best number of clauses satisfied as well as a list of n variables, which will be used in the Davis–Putnam–Loveland routine. The n variables chosen are those which occur in the largest number of clauses. These n variables allow the problem to be split into 2^n subproblems.

Now we enter a loop where we receive results from the slaves computations and assign new subproblems to slaves until all 2^n subproblems have been assigned. Subproblems are assigned to slaves that have completed their current subproblem and have reported their information (number of unsatisfied clauses and number of branches traversed) to the master process. When all 2^n

subproblems have been assigned, the master waits for the slaves to finish their last subproblem, collects the information (number of unsatisfied clauses and number of branches traversed) when they are done, and sends them a terminating signal. After all information is collected, the master makes the final calculations, and then quits.

The slave process receives start up information from the master and becomes part of the parallel virtual machine environment. The slave receives the number of clauses, the number of variables, and an array containing the problem from the master. The slave then reads the problem into the structures used in the algorithm.

After reading in the problem, the algorithm performs the GSAT heuristic as described earlier in Section 2.6. The best solution obtained from GSAT is relayed to the master who in turn sends information used in the Davis–Putnam–Loveland (dpl) routine, such such as the best number of clauses satisfied and the set of n variables which are chosen by the master.

To begin the algorithm, the master sends a signal to the slave. This signals either more work or completion of work (termination). If the signal is for termination, the slave exits the PVM environment. If the signal is for more work, the slave enters into a loop. Inside the loop, the slave requests and receives a new subproblem chosen by the master process. The slave sets up this new subproblem by setting the appropriate variables and performs the Davis–Putnam–Loveland routine. When it is done, it reports the number of unsatisfied clauses in the best solution as well as the number of branches traversed to the master. It then receives a new signal. The loop continues receiving new subproblems and solving them until the slave receives a signal of termination. At this point, the slave exits the PVM environment.

This algorithm runs in exponential time as per the number of variables specified. In the worst case, $2^{\text{num vars}}$ branches will be searched.

Chapter 4

Experimental Results and Discussion

In this section, we compare the parallel MAX–SAT algorithm described in Chapter 3 and a sequential Davis–Putnam–Loveland algorithm, which was described in Section 2.7. Both algorithms were written in C. The parallel code was run on a collection of seven 486DX2/66 PC's under LINUX, connected in a LAN via ethernet. PVM was used to create links between the seven machines, thereby creating the virtual parallel environment. The code was tested on both random and benchmark problems. The time reported is elapsed time in seconds. We use elapsed time, because the code runs 99% efficiently on the PC machines. For the parallel code, we use $n = 6$.

4.1 Random Problems

We are interested in solving MAX–2–SAT and MAX–3–SAT problems. According to Crawford and Auton [11, 10], a ratio of 1 variable to 4.258 clauses will produce hard MAX–3–SAT problems. A MAX–SAT problem becomes difficult when it has many unsatisfiable clauses. By hard maximum satisfiability problems, we mean that if the ratio of clauses to variables is smaller than 4.258 for MAX–3–SAT problems then the problem is likely to be satisfied. If the ratio is larger than 4.258 for MAX–3–SAT, then the problem is almost certainly unsatisfiable. For this paper, we wish to create hard maximum satisfiability problems. So, we will use ratios of 1 variable to 2 clauses and 1 variable to 2.5 clauses for the MAX–2–SAT problems. And we will use ratios of 1 variable to 5 clauses and 1 variable to 5.5 clauses for MAX–3–SAT problems. We will set the number of variables to 75 and 100. With these above factors a 2^2 factorial design will be performed for both the MAX–2–SAT problems and MAX–3–SAT problems with efficiency being used as the response. We are interested in determining what factors affect the efficiency of the parallel algorithm.

Lists of the random MAX–2–SAT and MAX–3–SAT problems and their characteristics are given in Tables 4.1 and 4.2.

4.1.1 Speedup and Efficiency

Computational results for the random MAX–2–SAT and MAX–3–SAT problems are given in Tables 4.3 and 4.4.

The 2^2 factorial design for MAX–2–SAT problems was computed at a 95% confidence level, and the following results were obtained: The effect for the number of variables was 0.22785 ± 0.125243. The effect for the ratio of variables to clauses was 0.06145 ± 0.199383. And the two–way interaction was 0.03275 ± 0.213213. Because the first confidence interval does not include 0, we conclude that the number of variables is significant. Thus, efficiency tends to increase as the number of variables is increased. However, since the effect for the ratio of variables to clauses and the two–way interaction confidence intervals included 0, we can draw no conclusions about either of the two effects.

The 2^2 factorial design for MAX–3–SAT problems was computed at

Problem	Variables	Clauses	Optimal Solution
rand2-0_2-no-1	75	150	3
rand2-0_2-no-2	75	150	1
rand2-0_2-no-3	75	150	3
rand2-0_2-no-4	75	150	6
rand2-0_2-no-5	75	150	4
rand2-0_2.5-no-1	75	188	9
rand2-0_2.5-no-2	75	188	5
rand2-0_2.5-no-3	75	188	6
rand2-0_2.5-no-4	75	188	6
rand2-0_2.5-no-5	75	188	7
rand2-1_2-no-1	100	200	4
rand2-1_2-no-2	100	200	4
rand2-1_2-no-3	100	200	6
rand2-1_2-no-4	100	200	5
rand2-1_2-no-5	100	200	4
rand2-1_2.5-no-1	100	250	10
rand2-1_2.5-no-2	100	250	4
rand2-1_2.5-no-3	100	250	10
rand2-1_2.5-no-4	100	250	12
rand2-1_2.5-no-5	100	250	11

Table 4.1: Characteristics of random MAX–2–SAT problems

Problem	Variables	Clauses	Optimal Solution
rand3-0_5-no-1	75	375	1
rand3-0_5-no-2	75	375	2
rand3-0_5-no-3	75	375	2
rand3-0_5-no-4	75	375	2
rand3-0_5-no-5	75	375	2
rand3-0_5.5-no-1	75	413	4
rand3-0_5.5-no-2	75	413	5
rand3-0_5.5-no-3	75	413	4
rand3-0_5.5-no-4	75	413	4
rand3-0_5.5-no-5	75	413	4
rand3-1_5-no-1	100	500	2
rand3-1_5-no-2	100	500	2
rand3-1_5-no-3	100	500	3
rand3-1_5-no-4	100	500	2
rand3-1_5-no-5	100	500	3
rand3-1_5.5-no-1	100	550	4
rand3-1_5.5-no-2	100	550	7
rand3-1_5.5-no-3	100	550	4
rand3-1_5.5-no-4	100	550	4
rand3-1_5.5-no-5	100	550	5

Table 4.2: Characteristics of random MAX–3–SAT problems

Problem	parallel		sequential		
	time (sec.)	branches	time (sec.)	branches	Efficiency
rand2-0_2-no-1	4	416	3	195	0.107
rand2-0_2-no-2	8	0	3	1	0.054
rand2-0_2-no-3	5	1776	3	119	0.086
rand2-0_2-no-4	16	118344	35	68941	0.313
rand2-0_2-no-5	5	2958	4	1357	0.114
rand2-0_2.5-no-1	66	795600	381	839585	0.825
rand2-0_2.5-no-2	5	1673	6	3816	0.171
rand2-0_2.5-no-3	6	17389	15	19570	0.357
rand2-0_2.5-no-4	5	12507	9	9487	0.257
rand2-0_2.5-no-5	7	53108	23	41936	0.469
rand2-1_2-no-1	7	14799	8	4323	0.163
rand2-1_2-no-2	6	1291	7	3261	0.167
rand2-1_2-no-3	65	520102	108	156821	0.237
rand2-1_2-no-4	14	67986	20	20619	0.204
rand2-1_2-no-5	6	16303	8	4168	0.190
rand2-1_2.5-no-1	1050	8086248	3027	4566444	0.412
rand2-1_2.5-no-2	7	3482	6	2035	0.122
rand2-1_2.5-no-3	223	1509229	1318	2151487	0.844
rand2-1_2.5-no-4	3284	26750852	18285	30564870	0.795
rand2-1_2.5-no-5	3591	54997679	35051	61765758	1.394

Table 4.3: Computational Results for random MAX–2–SAT problems

Problem	parallel		sequential		Efficiency
	time (sec.)	branches	time (sec.)	branches	
rand3-0_5-no-1.cnf	4	54	5	66	0.179
rand3-0_5-no-2.cnf	3	2954	7	1663	0.333
rand3-0_5-no-3.cnf	7	2326	6	1162	0.122
rand3-0_5-no-4.cnf	6	1765	6	1130	0.143
rand3-0_5-no-5.cnf	4	1071	5	900	0.179
rand3-0_6-no-1.cnf	17	138208	102	117873	0.857
rand3-0_6-no-2.cnf	68	621700	340	505878	0.714
rand3-0_6-no-3.cnf	26	209601	143	152945	0.786
rand3-0_6-no-4.cnf	22	118336	70	80930	0.50
rand3-0_6-no-5.cnf	33	203238	134	167198	0.580
rand3-1_5-no-1.cnf	7	6445	12	3715	0.245
rand3-1_5-no-2.cnf	7	9100	12	3871	0.245
rand3-1_5-no-3.cnf	18	75536	92	69766	0.730
rand3-1_5-no-4.cnf	8	6245	12	4092	0.214
rand3-1_5-no-5.cnf	23	82663	96	70289	0.527
rand3-1_6-no-1.cnf	349	827411	662	540636	0.271
rand3-1_6-no-2.cnf	20421	145431947	81249	102067740	0.568
rand3-1_6-no-3.cnf	374	679370	411	348381	0.157
rand3-1_6-no-4.cnf	352	677523	447	399568	0.181
rand3-1_6-no-5.cnf	2263	6696734	4049	4073839	0.256

Table 4.4: Computational Results for random MAX–3–SAT problems

a 95% confidence level, and the following results were obtained: The effect for the number of variables was 0.09765 ± 0.092922. The effect for the ratio of variables to clauses was -0.04995 ± 0.0673. And the two–way interaction was -0.15045 ± 0.134527. Because the confidence interval for the ratio of variables to clauses included 0, we can draw no conclusions about this effect. Because the confidence intervals of the other two effects do not include 0, we conclude that the number of variables and the two–way interaction are both significant. Thus, efficiency tends to decrease as the number of variables and the ratio of variables to clauses are increased simultaneously. However, efficiency tends to increase as the number of variables alone increases.

We are not confident that these 95% confidence intervals are very accurate, considering that only five replications were made at each design point. Also, the efficiency values over the five replications seem to be more exponentially distributed than normally distributed. For better accuracy in both efficiency and the factorial design, 15 – 25 more replications should be run for each design point.

4.2 Benchmarks

The benchmark problems were chosen so that there would be a diverse set of structured problems which are unsatisfiable. The characteristics of the benchmark problems used can be found in Table 4.5. The benchmark problems are structured problems from real applications. These problems come from DIMACS. The SSA problem is an instance from circuit fault analysis. The dubois problems are instances from the gensathard.c code. The jnh problems are a set of random instances generated to be difficult by rejecting unit clauses

Problem	Variables	Clauses	Optimal Solution
aim-100-1_6-no-3	100	160	1
aim-100-1_6-no-4	100	160	1
dubois20	60	160	1
dubois23	69	184	1
dubois26	78	208	1
dubois27	81	216	1
hole8	72	297	1
hole9	90	415	1
hole10	110	561	1
jnh13	100	850	2
jnh14	100	850	2
jnh202	100	800	1
jnh203	100	800	1
jnh302	100	900	4
jnh303	100	900	3
ssa0432-003	435	1027	1

Table 4.5: Characteristics of MAX–SAT benchmark problems

and setting the density to a hard value. The hole problems are instances of the pigeon hole problem. And lastly, the aim problems are artificially generated 3–SAT instances.

4.2.1 Speedup and Efficiency

Computational results for the benchmark problems can be found in Table 4.6. Looking at the efficiency ratings, we observe that the parallel code is not 100% efficient in all but one instance. The efficiency fluctuated from 7.2 % to 125%. There seems to be a trend in terms of efficiency between the different classes of problems. For example, the dubois problems all had an efficiency

Problem	parallel		sequential		
	time (sec.)	branches	time (sec.)	branches	Efficiency
aim-100-1_6-no-3	729	11169266	5012	14348099	0.982
aim-100-1_6-no-4	256	5850904	2237	6669356	1.25
dubois20	67	3145664	257	2097155	0.548
dubois23	542	25165760	2151	16777219	0.567
dubois26	4550	201326528	18011	134217731	0.565
dubois27	9408	402653120	36643	268435459	0.556
hole8	7	53854	15	40319	0.306
hole9	44	454766	123	362879	0.399
hole10	539	4345880	1357	3628799	0.360
jnh13	7	2108	12	696	0.245
jnh14	10	3016	11	540	0.157
jnh202	9	34	9	10	0.143
jnh203	8	47	12	721	0.214
jnh302	98	263499	126	39655	0.184
jnh303	30	56724	77	24305	0.367
ssa0432-003	228	35961	115.4	24457	0.072

Table 4.6: Computational Results for MAX–SAT benchmark problems

of about 55%, while the jnh problems had an efficiency closer to 20%. This difference could be a result of the difference in the types of problems and their structure.

Chapter 5

Conclusions and Future Work

The efficiency for most problems was between 15% and 85%. There are several possible reasons for this lack of efficiency. One reason would be that the harder subproblems are solved late in the algorithm process. This would cause one or two processors to run for a long time while other processors were idle. It is also possible that the choice of six variables should have been dynamic rather than restricted, as it was, to a selection of those six variables which occurred in the most clauses. This selective choice of variables results in the number of branches searched by the parallel algorithm exceeding the number searched by the sequential algorithm. A good example of this is dubois27. The parallel code took 402,000,000 branches while the sequential code took only 268,000,000 branches.

Efficiency of the smaller random problems was between 5% and 20%, as opposed to 15% — 95% for larger random problems. This was expected because of the loss in efficiency due to communications overhead and the fact that the sequential code is as fast as it is.

One thing should be noted about every problems that was run: GSAT obtained the optimal solution.

Future work should include more work in the load balancing portion of the algorithm. In particular, it should include dynamic load balancing instead of static load balancing. By this I mean that the master program should begin by running a part of the sequential algorithm, in order to generate some subproblems. When enough subproblems have been generated, the master sends the subproblems to the slaves to be solved. When a slaves has competed its work, the master should query the slaves and find out how much work each has. Then, the master will take work from one or more of the slaves and give that work the the slave who is finished.

References

[1] J. Beasley, K. Dowsland, F. Glover, M. Laguna, C. Peterson, C. Reeves, and B. Soderberg. *Modern Heuristic Techniques for Combinatorial Problems.* Halsted Press, New York, 1993.

[2] A. Beist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, Massachusetts, 1994.

[3] C. E. Blair, R. G. Jeroslow, and J. K. Lowe. Some Results and Experiments in Programming Techniques for Propositional Logic. *Computers and Operations Research*, 13(5):633–645, 1986.

[4] M. Böhm and E. Speckenmeyer. A Fast Parallel SAT-Solver—Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence*, 1995.

[5] Brian Borchers and Judith Furman. A two-phase exact algorithm for MAX–SAT and weighted MAX–SAT Problems. Submitted to *Journal of Combinatorial Optimization*, 1996.

[6] Brian Borchers, Steven W. Joy, and John E. Mitchell. A Branch and Cut Algorithm for MAX–SAT and Weighted MAX–SAT. To Appear in *Proceedings of The DIMACS Workshop on the Satisfiability Problem: Theory and Practice*, 1996.

[7] M. Buro and H.K Buning. Report on a SAT Competition. *EATCS Bulletin*, 49:143–151, 1993.

[8] Wen-Tsuen Chen and Lung-Lung Liu. A Parallel Approach for Theorem-Proving in Propositional Logic. *Information Sciences*, 41:61–76, 1987.

[9] J. Cheriyan, W. H. Cunningham, L. Tunçel, and Y. Wang. A Linear Programming and Rounding Approach to MAX 2-SAT. In *Second DIMACS Implementation Challenge: Cliques, Coloring, and Satisfiability, D. S. Johnson and M. A. Trick (eds), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS, 1996*, 1996.

[10] J.M. Crawford and L.D. Auton. Experimental Results on the Crossover Point in Satisfiability Problems. *AAAI-93*, pages 21–27, 1993.

[11] J.M. Crawford and L.D. Auton. Experimental Results on the Crossover Point in Random 3SAT. *Artificial Intelligence*, 81(1/2):31, March 1996.

[12] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[13] J. Eckstein. Parallel Branch and Bound Algorithms for General Mixed Integer Programming on the CM–5. Technical Report TMC-257, Thinking Machines Corporation, 245 First Street, Cambridge, MA, 02142, September 1993.

[14] G. Gallo and G. Urbani. Algorithms for Testing the Satisfiability of Propositional Formulae. *Journal of Logic Programming*, 7:45–61, 1989.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[16] B. Gendron and T.G. Crainic. Parallel Branch and Bound Algorithms: Survey and Synthesis. *Operations Research*, 42:1042–1066, 1994.

[17] Michel X. Goemans and David P. Williamson. Improved Approximations Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *Journal of the ACM*, 42:1115–1145, 1995.

[18] A. Grama and V. Kumar. Parallel Search Algorithms for Discrete Optimization Problems. *ORSA Journal on Computing*, 7(4):365–395, Fall 1995.

[19] J. Gu. Parallel Algorithms for Satisfiability Problem. In *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, volume 22, pages 105–161, 1995.

[20] P. Hansen and B. Jaumard. Algorithms for the Maximum Satisfiability Problem. *Computing*, 44:279–303, 1990.

[21] F. Harche, J.N. Hooker, and G.L. Thompson. A Computational Study of Satisfiability Algorithms for Propositional Logic. *ORSA Journal on Computing*, 6(4):423–435, Fall 1994.

[22] J. N. Hooker. A Quantitive Approach to Logical Inference. *Decision Support Systems*, 4:45–69, 1988.

[23] J. N. Hooker. Resolution vs. Cutting Plane Solution of Inference Problems: Some Computational Experience. *Operations Research Letters*, 7(1):1–7, 1988.

[24] J.N. Hooker. Solving the Incremental Satisfiability Problem. *Journal of Logic Programming*, 15:177–186, 1993.

[25] Yuejun Jiang, Henry Kautz, and Bart Selman. Solving Problems with Hard and Soft Constraints Using a Stochastic Algorithm for MAX–SAT. Presented at the 1st International Joint Workshop on Aritificial Intelligence and Operations Research, June 1995.

[26] D. Loveland. *Automated Theorem–Proving: A Logical Basis.* North Holland, New York, 1978.

[27] C. H. Papadimitriou and M. Yannakakis. Optimization, Approximation, and Complexity Classes. *Journal of Computers and System Sciences*, 43:425–440, 1991.

[28] Mauricio G. C. Resende and Thomas A. Feo. A GRASP for Satisfiability. To appear in "Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge", David S. Johnson and Michael A. Trick (eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1995.

[29] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[30] B. Selman and H. Kautz. Domain-Independant Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of IJCAI*, Murray Hill, NJ 07974, 1993.

[31] B. Selman, H.A. Kautz, and B. Cohen. Local Search Strategies for Satisfiability Testing. DRAFT of paper presented at Second DIMACS Challenge Workshop on Cliques, Coloring, and Satisfiability, Rutgers University, October 1993.

[32] B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA*, pages 440–446, July 1992.

[33] Bart Selman and Henry A. Kautz. An Empirical Study of Greedy Local Search for Satisfiability Testing. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93), Washington, DC*, pages 46–51, 1993.