Exploiting Variable Dependency in Local Search

Henry Kautz, David McAllester, and Bart Selman

AT&T Laboratories
180 Park Avenue
Florham Park, NJ 07932
{kautz, dmac, selman}@research.att.com
http://www.research.att.com/~{kautz, dmac, selman}

Abstract

Stochastic search has recently been shown to be successful for solving large boolean satisfiability problems. However, systematic methods tend to be more effective in problem domains with a large number of dependent variables: that is, variables whose truth values are directly determined by a smaller set of independent variables. In systematic search, truth values can be efficiently propagated from the independent to the dependent variables by unit propagation. Such propagation is more expensive in traditional stochastic procedures. In this paper we propose a mechanism for effectively dealing with dependent variables in stochastic search. We also provide empirical data showing the procedure outperforms the best previous stochastic and systematic search procedures on large formulas with a high ratio of dependent to independent variables.

1 Introduction

Recent years have seen significant progress in our ability to solve large propositional satisfiability problems. Randomly generated problem instances have proven to be a useful tool for testing and developing algorithms. In addition, however, researchers have now began to develop and solve propositional encodings of interesting, real-world problems. Such encodings were not even being considered a few years ago, because they were thought to be far too large to be handled by any method. However, between 1991 and 1996 the size of hard satisfiability problems that could be feasibly solved grew from ones involving less than 100 variables to ones involving over 10,000 variables. Now SAT algorithms are being applied to such problems as constraint-based planning (Blum and Furst 1995; Kautz and Selman 1996), problems in finite algebra (Fujita *et al.* 1993), verification of hardware and software, scheduling (Crawford and Baker 1994), circuit synthesis and diagnosis (Larrabee 1992), and many other domains, including natural language processing and machine learning. In fact, one can argue that the value of research on propositional reasoning ultimately depends on our ability to find suitable SAT encodings of real-world problems.

Propositional encodings of problems from other domains turn out to have distinct computational properties from random-CNF instances. Encoded problems contain *structure* that arises both from the semantics of the source problem domain, as well as from the particular *conventions* used to encode the problem. Algorithms that take advantage of the kinds of structure that appear in a particular class of encoded problems can often offer a higher level of performance.

Some of the structure in a problem instances is revealed in its syntactic form. In particular, one can extract many of the *dependencies* between variables. SAT encodings of real-world problems often contain large numbers of variables whose values are constrained to be a simple Boolean function of other variables. We call these *dependent* variables. Variables whose values can not be easily determined to be a simple function of other variables are called *independent*. For a given SAT problem there may be many different ways to classify the variables as dependent and independent. But for most SAT encodings there is a natural division between dependent and independent variables, which is expressed by conventions for encoding *definitions*.

A search algorithm for SAT testing can greatly reduce its search space by indentifying and effectively handling defined variables. Since an assignment to the independent variables determines a truth value for each dependent variable, the number of assignments that need be considered by even an exhaustive search is at most 2^n where n is the number of *independent* variables. It is fairly straightforward to take advantage of variable dependencies in *systematic* search, such as the Davis-Putnam procedure (Davis *et al.* 1962). In systematic search methods, basic dependencies are propagated in

linear time using unit propagation. Branching on defined variables can be avoided by using a variable-ordering heuristic that places the independent variables ahead of the dependent ones. Further improvements can be made through the use of even more sophisticated heuristics based on detecting and propagating definitions (Silva 1995).

By contrast, it appears to be more difficult to establish that identification of defined variables improves stochastic search. Local search procedures for satisfiability (Selman *et al.* 1992, Gu 1992) take longer to propagate dependencies. A rough rule of thumb is that about $O(n^2)$ steps are required for the values of the dependent variables to become aligned with independent variables. Although there is no complete theoretical explanation for this observation, there is an intuition that it comes from the time required to propagate dependencies through Horn clauses involved in the definitions. It is known that stochastic search solves sets of Horn clauses in expected $O(n^2)$ time (Papadimitriou 1991; 1993).)

The contribution of this paper is to describe a method for improving local search for problems with defined variables. We will describe the architecture of the *Dagsat* system, in which search concentrates only on the independent variables, and a fast mechanism handles dependent variables. Dagsat can be viewed as a generalization of previous local search algorithms for clausal satisfiability. There have been some previous work on generalizing these algorithms, particularly that by Sebastiani (1994) on non-clausal GSAT. Our approach differs from Sebastiani's in at least two key aspects:

- The structures Dagsat handles are strictly more general than non-clausal propositional logic. Any non-clausal formula can be handled by Dagsat without an increase in size, but converting from Dagsat's format to non-clausal CNF (over the same set of independent variables) can lead to an exponential blowup. Another way of stating this is that non-clausal formulas are the special case where each defined variables is only *used* a single time.

- As we will describe below, Dagsat is able to restrict its search space by only selecting variables to flip that contribute to the *falsehood* of the formula under consideration.

Finally, we will also present empirical evidence that Dagsat efficiently solves large, hard problems containing a high ratio of dependent to independent variables. Our initial benchmarks include encodings of protocol verfication and circuit synthesis problems, as well as a set of randomly-generated *structured* problems.

2 Boolean Dags

We are concerned with the problem of determining satisfiability for Boolean formulas represented as directed acyclic graphs (dags). More formally, we take a *definition* be an expression of the form $x \equiv E$ where E is either a conjunction or disjunction of literals (where a literal is either a Boolean variable or the negation of a Boolean variable). For example, $x \equiv (y \vee \neg z \vee w)$ defines x to be the disjunction of y, $\neg z$, and w. We define a *Boolean dag* to be a nonempty finite sequence of definitions with the property that for each definition $x \equiv E$ in the sequence, x does not appear in E or in any prior definition. This last constraint ensures that definitions are well founded — no variable is defined in terms of itself. Variables appearing on the left hand of definitions are called the *defined* variables and all other variables are called the *independent* variables. Note that if we are given a truth assignment to the indepent variables then a truth assignment for all defined variables can be computed by "executing" the definitions in the order given. The last variable defined in the dag will be called the *root variable* of the dag. A truth assignment to the independent variables *satisfies* a given Boolean dag if the corresponding computed value of the root variable is true. A Boolean dag is *satisfiable* if it is satisfied by some truth assignment to the independent variables.

A Boolean dag represents a Boolean expression involving only the independent variables. This expression can be derived by starting with the root variable and iteratively replacing defined variables by their definition. However, the written length of this Boolean expression can be exponentially longer than the dag. Consider the sequence of definitions $x_1 \equiv (y_0 \vee x_0)$, $y_1 \equiv (x_0 \wedge y_0)$, $\ldots$, $x_n \equiv (x_{n-1} \vee y_{n-1})$, $y_n \equiv (x_{n-1} \wedge y_{n-1})$. This dag has only two independent variables and is satisfied when they are both true. However, the dag represents a formula with binary operations and uniform depth n — the written length of this formula is $O(2^n)$ even though the size of the dag is $O(n)$. Exponential explosions in the written length of the represented formula is common in practice. Hence any file format for representing Boolean dags should be based on sequences of definitions rather than textual representations of highly nested Boolean expressions.

Our implementation uses a standard file format for CNF SAT problems to represent Boolean dags. We take the representation of the definition $x \equiv (y_1 \wedge \ldots \wedge y_n)$ to be the following sequence of clauses.

$$x \vee \neg y_1 \vee \ldots \vee \neg y_n, \quad \neg x \vee y_1, \ \ldots \ \neg x \vee y_n$$

Similarly, we take the representation of $x \equiv (y_1 \vee \ldots \vee y_n)$ to be the following.

$$\neg x \vee \neg y_1 \vee \ldots \vee \neg y_n, \quad x \vee y_1, \ \ldots \ x \vee y_n$$

We take the SAT representation of Boolean dag to be the sequence of these clausal representations of the definitions in the dag followed by a unit clause stating that the root variable must be true. Note that there is a one to one correspondence between the assignments satisfying a given Boolean dag and the assignments satisfying its SAT representation. In particular, the SAT representation is satisfiable if and only if the Boolean dag is satisfiable. It should also be clear that any Boolean dag can be easily recovered from its SAT representation. Given the SAT representation of Boolean dags we can use the standard file formats for CNF SAT problems as a file format for Boolean dags. This allows standard tools for CNF SAT problems, such as existing systematic solvers, to be used directly on Boolean dags. Our implementation also allows an arbitrary CNF SAT problem to be interpreted as a Boolean dag — we heuristically find "definitions" in the SAT problem and then define the root variable to be the conjunction of all the clauses which are not parts of the heuristically recovered definitions. Of course, this heuristic recovery is done in such a way that a given dag is guaranteed to be uniquely recovered from its SAT representation.

3 The Walksat Architecture

The Walksat (or WSAT) architecture defines a class of local search procedures for (CNF) Boolean satisfiability. At each point in time we have a particular complete truth assignment to the Boolean variables. In the Walksat architecture we select a violated clause; select a variable occuring in that clause; and then flip the value of the selected variable. Different procedures within this architecture use different heuristics for selecting clauses and variables. In most applications it seems best to select the violated clause at random and then select a literal in noisy manner with a bias favoring choices leading to a lower number of violated clauses. The number of violated clauses is used as the cost (or energy) function guiding the stochastic search.

The Walksat architecture seems to be uniformly superior to the earlier GSAT architecture which used the same cost function but flipped variables not occuring in violated clauses (Selman *et al.* 1994). The Walksat architecture can be viewed as a definition of a local search space with locally irreversable moves — if a given variable occurs in only one violated clause, and flipping that variable does not lead to new violated clauses, then the flip cannot be immediately undone. However, for any solvable problem there is always a path to a solution — for any violated clause there exists a choice of literal which reduces the Hamming distance to the nearest solution by one. It is also interesting to note that, although the Walksat moves are irreversable, there always remains a path to a global optimum in any MAXSAT problem — if we are not at a global optimum then there is a violated clause which is satisfied in the nearest (in Hamming distance) global optimum. So there always exists a local move that reduces the Hamming distance to the nearest global optimum in a MAXSAT problem. The irreversibility of the local moves implies that the set of states (truth assignments) reachable from the current point can become smaller, but never larger, as the search progresses. Although there is guaranteed to always be an accessible solution, the set of accessible states can "ratchet down" so that the search space becomes smaller over time.

4 Boolean Dags as Clause Sets

The first step in generalizing the Walksat architecture is defining an appropriate cost function to guide the local search. In the Walksat architecture the cost function is the number of violated clauses. But consider the SAT representation of a given Boolean dag. All clauses in the SAT representation are parts of definitions except the final unit clause which expresses the constraint that root variable must be true. If we take an arbitrary truth assignment to the independent variables, then compute a value for each defined variable according to its definition, all clauses become satisfied except possibly the final unit clause. This means that for *any* truth assignment to the independent variables the number of violated clauses can be driven down to one simply by computing the approriate value for the defined variables. Clearly the number of violated clauses in the SAT representation is not a good cost function.

To define a more useful cost function we first rewrite the dag into a certain normal form. First, we force the root of the dag to be a conjuction — if the root is a disjunction then we construct a new root which is defined to be a conjunction with a single argument which is the original root. Next we expand argument of the root which are themselves conjunctions. For example, letting r be the root variable, we replace

$$\ldots,\ x \equiv (y_1 \wedge \ldots \wedge y_n),\ \ldots,\ r \equiv (\ldots \wedge x \wedge \ldots)$$

by

$$\ldots,\ x \equiv (y_1 \wedge \ldots \wedge y_n),\ \ldots,\ r \equiv (\ldots \wedge y_1 \wedge \ldots \wedge y_n \wedge \ldots)$$

(note that x may be used elsewhere). By repeatedly applying this and other transformations we can, in linear time, put the dag in a form where the root is a conjunction of "top level clauses" each of which is a disjunction of at least two literals. If a unit disjunction appears as a top level constraint, then the dag can be simplified by replacing all occurances of the variable involved by the

appropriate truth value and promoting the definition of that variable, if it has one, to a top level constriant. We call this "root normal form".

A dag in root normal form can be viewed as a sequence of definitions plus a set of top level clauses. Any truth assignment to the independent variables induces a truth assignment on all variables. We can now construct a generalization of Walksat. Each truth assignment is determined by the truth of the independent variables. For each truth assignment we can count the number of violated top level clauses and use this as the cost function in guiding the stochastic search. We can now repeatedly select a violated top level clause and then heuristically select an *independent* variable implicitly appearing in the violated clause — we say that x appears implicitly in an expression E if it either appears in E (in the normal sense) or there exists a variable y appearing in E with definition $y \equiv E'$ and x appears implicitly in E'. This set of local moves is similar to Walksat's in that moves can be locally irreversable and yet from any assignement there exists a sequence of local moves leading to a solution. In the full Dagsat archicture the set of possible local moves is restricted even further.

Whenever an independent variable is flipped the dependent variables are updated so that each dependent variable has the truth value given by its definition. The update of the truth values can be done incrementally. We say that a variable undergoes a net change during an update if the truth value after the update is different from the truth value before the update. We say that a variable y with definition $y \equiv E$ *requires consideration* if some variable appearing (explicitly) in E undergoes a net change during the update. One can use a priority queue (a heap) to process the variables requiring consideration in the order in which their definitions appear in the dag. This way we only visit variables requiring consideration, *i.e.*, those whose definitions involve variables undergoing a net change. Furthermore, each variable requiring consideration is only visited once. If we assume that definitions have a bounded number of arguments (say two) then the time required for this update operation is $n \log n$ where n is the number of variables requiring consideration (each heap operation can take $O(\log n)$ time).

5 The Dagsat Architecture

The Dagsat architecture is defined by the set of local moves allowed from a given truth assignment. In this section we define the architecture by defining the more restricted set of allowed moves mentioned in the previous section.

Restricted moves, and various other concepts, are most easily described by assuming the dag is in positive normal form. Intuitively, we achieve positive normal form by using de Morgan's law to push negations to the leaves of the dag. More formally, in a positive normal form dag the top level constraints are all disjunctions of defined variables and each definition is of one of the following forms where each y_i must be a defined variable and z must be an independent variable.

$$x \equiv (y_1 \wedge \ldots \wedge y_n), \quad x \equiv (y_1 \vee \ldots \vee y_n), \quad x \equiv z, \quad x \equiv \neg z$$

We now describe how to put any Boolean dag into positive normal form in linear time. For each variable x we introduce two new variables x' and x'' representing x and its negation respectively. For any literal L we let $\neg L$ be the opposite literal. For example, $\neg(\neg z)$ is the literal z. We define L' by the condition L' is x' if L is x and L' is x'' if L is $\neg x$. For any literal L we have that L' is a variable. If z is an independent variable then we add the definitions $z' \equiv z$ and $z'' \equiv \neg z$. Now each definition of the form $x \equiv (L_1 \wedge \ldots \wedge L_n)$ in the original dag is replaced by the two definitions $x' \equiv (L'_1 \wedge \ldots \wedge L'_n)$ and $x'' \equiv ((\neg L_1)' \vee \ldots \vee (\neg L_n)')$. An analogous transformation is used for disjunctive definitions. Finally, each top level clause $L_1 \vee \ldots \vee L_n$ is replaced by $L'_1 \vee \ldots \vee L'_n$.

Conversion to positive normal form greatly simplifies the discussion of Dagsat algorithms and heuristics. However, more complex versions of all such algorithms and heuristics can be constructed for arbitrary dags. Conversion to positive normal form is not necessary in practice and the algorithms generally run more efficiently on the original dag which generally involves only half as many variables. But for the sake of clarity, we now only consider dags which have been converted to positive normal form.

As indicated in the previous section, the Dagsat architecture allows even fewer local moves than those described in the previous section. However, the Dagsat architecture preserves the property that from any assignment there is a local move reducing the Hamming distance to the nearest solution (as measured on the indepenent variables). As before, a local move is made by first selecting a violated top level clause. In a positive normal form dag this clause is always of the form $y_1 \vee \ldots \vee y_n$ where each y_i is a defined variable which is false under the current truth asignment. One then selects a *source of falsehood* for some y_i. If y is defined by $y \equiv (x_1 \wedge \ldots \wedge x_i)$ then a source of falsehood for y is selected by selecting a source of falsehood for some *currently false* x_i. If y is a currently false variable defined by $y \equiv (x_1 \vee \ldots \vee x_i)$ then each x_i must be currently false and we select a source of falsehood for y by selecting a source of falsehood for some x_i. Finally, if y is defined by $y \equiv L$ where L is a literal involving the independent variable z then z is the unique source of falsehood for y. In general the sources of falsehood for a currently false

variable y will be a proper subset of the variables that appear implicitly in the definition of y. Thus, restricting the allowed local moves to sources of falsehood reduces the number of allowed local moves. It is interesting to note that the set of allowed local moves could be equivalently defined simply as the set of sources of falsehood for the root where the root variable is defined as the conjunction of the top level clauses.

Note that for each violated top level constraint some source of flasehood for that constraint must have the opposite value in the nearest solution (under Hamming distance). So, as in Walksat, there always exists a local move that reduces the Hamming distance to the nearest solution.

6 The Virtual Clause Heuristic

A heuristic within the Dagsat architecture is a method of selecting a local move. We have experimented with a variety of heuristics and one we call "vclause" has turned out to be the most successful on the problems we tried. A given Boolean dag defines a set of "virtual clauses" on the independent variables. This is the set of clauses that would be derived by converting the dag to a CNF formula by repeatedly distributing disjunctions over conjunctions in the expression represented by the dag. In general there can be exponentially many virtual clauses. However, it is possible to efficiently compute a single violated virtual clause. The vclause heuristic selects a local move by first constructing a violated virtual clause and then selecting a variable in that clause using some standard Walksat heuristic. Various choices of the possible Walksat heuristic leads to variations on the vclause heuristic. For example, the vclause-G heuristic uses the G Walksat heuristics to select a literal from the constructed violated clause. Our most successful heuristic was vclause-tabu which uses the tabu Walksat heuristic. In the tabu Walksat heuristic a variable is called *tabu* if it has been flipped within the last t moves where t is the "tenure" parameter of the heuristic. The tabu heuristic selects, from among the nontabu literals in the (virtual) clause, the literal that minimizes the cost of the next assignment (ties are broken by randomly selecting a member of the set best choices). If all literals are tabu then a literal is selected at random. All effective Walksat heuristics measure the change in the number of violated clauses that would occur for each possible choice of literal from the clause. In our implementation this is done by actually flipping each variable in the virtual clause and measuring the change in the number of violated top level clauses.

To construct a violated virtual clause on the independent variables we first randomly select a violated top level clause $y_1 \vee \ldots \vee y_n$. For each y_i we then construct a clause from the expansion of the formula represented by y_i and return the union of these clauses. If y is defined by $y \equiv (x_1 \wedge \ldots \wedge x_n)$ then to construct a violated clause from the expansion of y we select a currently false x_i at random and recursively construct a violated virtual clause from the expansion of x_i. If y is defined by $y \equiv (x_1 \vee \ldots \vee x_n)$ then to construct a violated clause from the expansion of y we recursively construct a violated clause from the expansion of each x_i, and return the union of the clauses constructed in this way. If y is defined by $y \equiv L$ where L is a literal involving an independent variable, then we return the singleton clause containing L.

7 Conspiracy Numbers

Most of the properties of Boolean dags described in the previous sections can be computed directly from the single root variable of the dag without treating the root as a special case. For example, the local moves can be defined as the set of sources of falsehood of the root variable. A virtual clause can be computed from the root variable by simply treating the root as a defined conjunction. But the cost function is defined, essentially, to be the number of false children of the root. This treats the root as a special case. Conspiracy numbers give a cost function which does not treat the root variable as a special case. Conspiracy numbers were first introduced as a measure of confidence in the min/max value of nodes in game search trees. They have been used effectively in "solving" a variety of simple games.

The conspiracy number of a defined variable is a kind of generalized "violation count" for that variable. The conspiracy number of a variable defined to be a conjunction of other variables is always at least as large as the number of false children. So the conspiracy number of the root variable is always at least as large as the number of violated top level constraints. But if one particular top level constraint is itself very difficult to satisfy then this makes the root variable even more difficult to satisfy since all top level constraints must be satisfied. More generally, the conspiracy number of any variable that is currently true is 0. If y is defined by $y \equiv (x_1 \wedge \ldots \wedge x_n)$ then the conspiracy number of y is sum of the conspiracy numbers of the x_i (which equals the sum of the conspiracy numbers of the x_i which are currently false). If y is defined by $y \equiv (x_1 \vee \ldots \vee x_n)$ then the conspiracy number of y is the minimum of the conspiracy numbers of the variables x_i. If y is currently false and defined by $y \equiv L$ where L is a literal involving an independent variable then the conspiracy number of y is 1. When a variable is flipped the conspiracy numbers can be incrementally updated in much the same way as the truth values in $O(n \log n)$ time where n is the number

formulas				DAGSAT		WALKSAT		NTAB		
id.	ind.var	dep.var	top.cls.	time	flips	time	flips	time	bcktr.	u.prop.
struc1	25	475	400	0.02	44	0.3	35×10^3	0.1	194	12×10^3
struc2	50	450	850	8.3	13,903	16.5	2×10^6	0.6	990	47×10^3
struc3	50	950	850	1.5	1,649	16.9	4×10^6	7.9	10,289	0.9×10^6
struc4	75	1425	1000	1.7	1,833	23.1	5×10^6	41	36,542	4×10^6
struc5	100	1900	1500	4.9	4,336	105	23×10^6	—	—	—
struc6	100	7900	1000	1.9	559	—	—	—	—	—
pipe	98	2695	47	0.4	599	5.5	0.5×10^6	—	—	—
add	121	352	242	68	0.1×10^6	0.1	1927	—	—	—

Table 1: Results of on a range problem instances.

of variables requiring examanination (as in the method of updating truth values).

Although the conspiracy number of the root seems like the right cost function, in most problems we tried this conspiracy number was almost always equal to the number of false children of the root, *i.e.*, the number of violated top level constraints. Those problems where the conspiracy numbers were significantly larger than the number of top level violations, and hence where conspiracy numbers might have provided more guidance, proved to be too difficult to solve using the methods we tried. It remains to be seen whether conspiracy numbers can be used to effectively in local search.

8 Empirical Evaluation

In Table 1, we present an empirical evaluation of Dagsat. The "struc" instances are randomly generated structured SAT problems. They consist of a ground layer of independent variables, followed by layers of definitions. The first instance, "struc1", consists of 25 independent variables, followed by 19 layers of definitions, each containing 25 defined variables. Each definition is either an "and" or an "or" node (with equal probability), with two children from any of the variables in the layers below. Finally, the top level clauses each contain 5 variables, chosen from any of the variables. We chose this particular structure because it is similar to the structure observed in SAT encodings of state-based planning problems.

We compared the performance of Dagsat against Walksat and a highly optimized implementation of the Davis-Putnam Procedure called "NTAB" (Crawford and Auton 1993; 1996). The parameters of both Dagsat and Walksat were tuned by hand. Dagsat used the vclause-tabu heuristic, with a tabu length of 3, and up to 100,000 flips per run. Walksat used a noise level of 20%, and up to 50,000,000 flips per run. The success rate (chance that a single run would solve the problem) was 95% for Dagsat and 90% for Walksat.

We see that problems struct1 to struct6 grow increasingly harder. On the harder instances Dagsat clearly outperforms the other algorithms both in terms of absolute time and the number of flips (or backtracks). (All experiments were run on a 200Mhz R10000 SGI Challenge. Instances and code are available from the authors.) An hypen in the table indicates the method was run for over an hour without success.

It is interesting to note how the dependencies affect the performance of Dagsat and Walksat. The number of flips Dagsat performs appears to be roughly quadratic in the number of independent variables. By contrast, the number of flips Walksat requires is closer to quadratic in the *total* number of variables. We also see that NTAB does very large numbers of unit propagation, an indication of the high number of dependent variables.

The next problem instance, "pipe", is an encoding of a protocol verification problem. Dagsat still dominates the other methods: Walksat still does well in terms of absolute time, even though it requires a much larger number of flips. Finally, "add" is an encoding of a circuit synthesis problem. Walksat clearly outperforms Dagsat on this instance. This problem has a much lower ratio of defined to independent variables, which may explain why Dagsat cannot take much advantage of the problem's structure. Note that NTAB is also unable to uncover the structure in this instance.

9 Conclusions

We have presented a architecture for extending stochastic local search methods to Boolean satisfiability problems containing definitional structure. This approach generalizes earlier work on both clausal and nonclausal satisfiability testing. The method was designed to address the problem of the large number of dependent variables that naturally occur in SAT encodings problems from other domains.

Finally, we reported on initial experiments with our implementation, that showed the approach outperforms state of the art systematic and stochastic algorithms on problems with a high ratio of defined variables to independent variables.

References

Battiti, R., and Protasi, M. (1996). Reactive Search, a history-based heuristic for MAX-SAT. Technical report, Dipartimento di Matematica, Univ. of Trento, Italy.

Blum, A. and Furst, M.L. (1995). Fast planning through planning graph analysis. *Proc. IJCAI-95*, Montreal, Canada.

Crawford, J.M. and Auton, L.D. (1993) Experimental Results on the Cross-Over Point in Satisfiability Problems. *Proc. AAAI-93*, Washington, DC, 21–27. Extended version, *Artificial Intelligence* (1996).

Crawford, J. and Baker, A.B. (1994). Experimental results on the application of satisfiability algorithms to scheduling problems. *Proc. AAAI-94*, Seattle, WA.

Dechter, R. and Rish, I. (1994). Directional resolution: the Davis-Putnam procedure, revisited. *Proc. KR-94*, Bonn, Germany.

Dubois, O. , Andre, P., Boufkhad, Y., and Carlier, J. (1996). A-SAT and C-SAT. *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*. (to appear)

Gent, I., and Walsh, T. (1993). Towards an understanding of hill-climbing procedures for SAT. *Proc. AAAI-93*, 28–33.

Gu, J. (1992) Efficient local search for very large-scale satisfiability problems. *Sigart Bulletin* 3(1), 8–12.

Kautz, H. and Selman, B. (1992) Planning as Satisfiability. *Proc. ECAI-92*, Vienna, Austria, 1992, 359–363.

Kautz, H. and Selman, B. (1996) Pushing the envelope: planning, propositional logic, and stochastic search. *Proc. AAAI-96*, Portland, OR, 1996.

Kautz, H., McAllester, D., and Selman, B. (1996). Encoding Plans in Propositional Logic. In preparation.

Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. (1983). Optimization by simulated annealing. *Science*, 220 (1983) 671–680.

Mazure, B., Sais, L., and Gregoire, E. (1996). Bootsing Complete Techniques Thanks to Local Search Methods. *Proc. Math & AI-96*.

Minton, S., Johnston, M.D., Philips, A.B., and Laird, P. (1992) Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence,* (58)1–3, 1992, 161–205.

Papadimitriou, C.H. (1991). On selecting a satisfying truth assignment. *Proc. of 32th Conference on the Foundations of Computer Science*, 1991, 163– 169.

Papadimitriou, C.H. (1993). *Computational Complexity.* Addison Wesley, 1993.

Sebastiani, R. (1994). Applying GSAT to Non-Clausal Formulas (Research Note). *Journal of Artificial Intelligence Research (JAIR)*, 1, 309-314.

Sebastiani, R. (1996). Personal communication.

Selman, B. , Kautz, H., and Cohen, B. (1994). Noise Strategies for Local Search. *Proc. AAAI-94*, Seattle, WA, 1994, 337–343.

Selman, B., Levesque, H., and Mitchell, D. (1992). A New Method For Solving Hard Satisfiability Problems. *Proc. AAAI-92*, San Jose, CA, 1992, 440-446.

Silva, J.P.M. (1995). *Search Algorithms for Satisfiability Problems in Combinatorial Switching Circuits.* Ph.D. Thesis, Univ. of Michigan.

Trick, M. and Johnson, D. (Eds.) (1993) *Proc. DIMACS Challenge on Satisfiability Testing.* Piscataway, NJ, 1993. (DIMACS Series on Discr. Math.)