In search of the best constraint satisfaction search *

Daniel Frost and Rina Dechter
Dept. of Information and Computer Science
University of California, Irvine, CA 92717
{dfrost,dechter}@ics.uci.edu

Abstract

We present the results of an empirical study of several constraint satisfaction search algorithms and heuristics. Using a random problem generator that allows us to create instances with given characteristics, we show how the relative performance of various search methods varies with the number of variables, the tightness of the constraints, and the sparseness of the constraint graph. A version of backjumping using a dynamic variable ordering heuristic is shown to be extremely effective on a wide range of problems. We conducted our experiments with problem instances drawn from the 50% satisfiable range.

1. Introduction

We are interested in studying the behavior of algorithms and heuristics that can solve large and hard constraint satisfaction problems via systematic search. Our approach is to focus on the average-case behavior of several search algorithms, all variations of backtracking search, by analyzing their performance over a large number of randomly generated problem instances. Experimental evaluation of search methods may allow us to identify properties that cannot yet be identified formally. Because CSPs are an NP-complete problem, the worst-case performance of any algorithm that solves them is exponential. Nevertheless, the average-case performance between different algorithms, determined experimentally, can vary by several orders of magnitude.

An alternative to our approach is to do some form of average-case analysis. An average-case analysis requires, however, a precise characterization of the distribution of the input instances. Such a characterization is often not available.

There are limitations to the approach we pursue here. The most important is that the model we use to generate random problems may not correspond to the type of problems which a practitioner actually encounters, possibly rendering our results of little or no relevance. Another problem is that subtle biases, if not outright bugs, in our implementation may skew the results. The only safeguard against such bias is the repetition of our experiments, or similar ones, by others; to facilitate such repetition we have made our instance generating program available by FTP[1].

In the following section we define formally constraint satisfaction problems and describe briefly the algorithms and heuristics to be studied. We then show that the linear relationship between the number of constraints and the number of variables at the 50% solvable region, observed for 3-SAT problems by (Mitchell, Selman, & Levesque 1992; Crawford & Auton 1983), is observed only approximately for binary CSPs with more than two values per variable. We conducted our experiments with problems drawn from this region. Section 3 describes those studies, which involved backtracking, backjumping, backmarking, forward checking, two variable ordering heuristics, and a new value ordering heuristic called sticking values. The results of these experiments show that backjumping with a dynamic variable ordering is a very effective combination, and also that backmarking and the sticking values heuristic can significantly improve backjumping with a fixed variable ordering. The final section states our conclusions.

2. Definitions and Algorithms

A *constraint satisfaction problem* (CSP) is represented by a *constraint network*, consisting of a set of n variables, $X_1, \ldots, X_n$; their respective value domains, $D_1, \ldots, D_n$; and a set of constraints. A *constraint* $C_i(X_{i_1}, \ldots , X_{i_j})$ is a subset of the Cartesian product $D_{i_1} \times \ldots \times D_{i_j}$, consisting of all tuples of values for a subset $(X_{i_1}, \ldots, X_{i_j})$ of the variables which are compatible with each other. A *solution* is an assignment of values to all the variables such that no constraint is

*This work was partially supported by NSF grant IRI-9157636, by Air Force Office of Scientific Research grant AFOSR 900136 and by grants from Toshiba of America and Xerox.

[1] ftp to ics.uci.edu, login as "anonymous," give your e-mail address as password, enter "cd /pub/CSP-repository," and read the README file for further information.

violated; a problem with a solution is termed *satisfiable*. Sometimes it is desired to find all solutions; in this paper, however, we focus on the task of finding one solution, or proving that no solution exists. A *binary* CSP is one in which each of the constraints involves at most two variables. A constraint satisfaction problem can be represented by a *constraint graph* consisting of a node for each variable and an arc connecting each pair of variables that are contained in a constraint.

Algorithms and Heuristics

Our experiments were conducted with backtracking (Bitner & Reingold 1985), backmarking (Gaschnig 1979; Haralick & Elliott 1980), forward checking (Haralick & Elliott 1980), and a version of backjumping (Gaschnig 1979; Dechter 1990) proposed in (Prosser 1993) and called there *conflict-directed* backjumping. Space does not permit more than a brief discussion of these algorithms. All are based on the idea of considering the variables one at a time, during a *forward* phase, and instantiating the current variable V with a value from its domain that does not violate any constraint either between V and all previously instantiated variables (backtracking, backmarking, and backjumping) or between V and the last remaining value of any future, uninstantiated variable (forward checking). If V has no such non-conflicting value, then a *dead-end* occurs, and in the *backwards* phase a previously instantiated variable is selected and re-instantiated with another value from its domain. With backtracking, the variable chosen to be re-instantiated after a dead-end is always the most recently instantiated variable; hence backtracking is often called *chronological* backtracking. Backjumping, in contrast, can in response to a dead-end identify a variable U, not necessarily the most recently instantiated, which is connected in some way to the dead-end. The algorithm then "jumps back" to U, uninstantiates all variables more recent than U, and tries to find a new value for U from its domain. The version of backjumping we use is very effective in choosing the best variable to jump back to.

Determining whether a potential value for a variable violates a constraint with another variable is called a *consistency check*. Because consistency checking is performed so frequently, it constitutes a major part of the work performed by all of these algorithms. Hence a count of the number of consistency checks is a common measure of the overall work of an algorithm. Backmarking is a version of backtracking that can reduce the number of consistency checks required by backtracking without changing the search space that is explored. By recording, for each value of a variable, the shallowest variable-value pair with which it was inconsistent, if any, backmarking can eliminate the need to repeat unnecessarily checks which have been performed before and will again succeed or fail. Although backmarking *per se* is an algorithm based on backtracking, its consistency check avoiding techniques can be applied to backjumping (Nadel 1989; Prosser 1983). In our experiments we evaluate the success of integrating backjumping and backmarking.

The forward checking algorithm uses a *look-ahead* approach: before a value is chosen for V, consistency checking is done with all future (uninstantiated) variables. Any conflicting value in a future variable W is removed temporarily from W's domain, and if this results in W having an empty domain then the value under consideration for V is rejected.

We used two variable ordering heuristics, min-width and dynamic variable ordering, in our experiments. The *minimum width* (MW or min-width) heuristic (Freuder 1982) orders the variables from last to first by repeatedly selecting a variable in the constraint graph that connects to the minimal number of variables that have not yet been selected. Min-width is a static ordering that is computed once before the algorithm begins. In a dynamic variable ordering (DVO) scheme (Haralick & Elliott 1980; Purdom 1983; Zabih & McAllester 1988) the variable order can be different in different branches of the search tree. Our implementation selects at each step the variable with the smallest remaining domain size, when only values that are consistent with all instantiated variables are considered. Ties are broken randomly, and the variable participating in the most constraints is selected to be first.

We also experimented with a new value ordering heuristic for backjumping called *sticking value*. The notion is to remember the value a variable is assigned during the forward phase, and then to select that value, if it is consistent, the next time the same variable needs to be instantiated during a forward phase. (If the "sticking value" is not consistent, then another value is chosen arbitrarily.) The intuition is that if the value was successful once, it may be useful to try it first later on in the search. This heuristic is inspired by local repair strategies (Minton *et al.* 1992; Selman, Levesque, & Mitchell 1992) in which all variables are instantiated, and then until a solution is found the values of individual variables are changed, but never uninstantiated.

Before jumping to our empirical results, we want to mention that the backjumping algorithm when used with a fixed ordering has a nice graph-based complexity bound. Given a graph G, a *dfs* ordering of the nodes is an ordering generated by a depth first search traversal on G, generating a *DFS* tree (Even 1979). We have shown elsewhere the following theorem:

Theorem 1(Collin, Dechter, & Katz 1991)**:** Let G be a constraint network and let d be a dfs ordering of G whose DFS tree has depth m. Backjumping on d is $O(\exp(m))$.

3. Methodology and Results

The experiments reported in this paper were run on random instances generated using a model that takes

K	N	C	C/N	C	C/N	C	C/N	C	C/N
		T = 1/9		T = 2/9		T = 3/9		T = 4/9	
3	25	199	7.96	89	3.56	51	2.04	31	1.24
3	30	236	7.87	104	3.47	59	1.97	36	1.20
3	35	272	7.77	120	3.43	68	1.94	41	1.17
3	40	310	7.75	137	3.43	76	1.90	45	1.13
3	50	380	7.60	166	3.32	91	1.82	53	1.06
3	60	454	7.57	196	3.27	106	1.77	62	1.03
3	75	565	7.53	244	3.25	132	1.76	74	0.99
3	100	747	7.47	317	3.17	169	1.69	92	0.92
3	125	927	7.42	394	3.15	207	1.66	109	0.87
3	150	1100	7.40	468	3.12	242	1.61	127	0.85
3	175	1290	7.37	546	3.12	281	1.61	146	0.83
3	200	1471	7.36	623	3.11	318	1.59	159	0.80
3	225			697	3.10	353	1.57	176	0.78
3	250			773	3.09	390	1.56	193	0.77
3	275			847	3.08	425	1.54	205	0.75
		T = 4/36		T = 8/36		T = 12/36		T = 16/36	
6	15	**	**	102	6.80	62	4.13	41	2.73
6	25	**	**	165	6.60	100	4.00	65	2.60
6	35	500	14.29	228	6.51	137	3.91	89	2.54
6	50	710	14.20	325	6.50	193	3.87	125	2.51
6	60	852	14.20	389	6.48	231	3.85	150	2.50
		T = 9/81		T = 18/81		T = 27/81		T = 36/81	
9	15	**	**	**	**	79	5.27	53	3.53
9	25	**	**	211	8.44	128	5.12	87	3.48
9	35	**	**	294	8.40	178	5.09	119	3.40

Figure 1: The "C" columns show values of C which empirically produce 50% solvable problems, using the model described in the text and the given values of N, K, and T. The "C/N" column shows the value from the "C" column to its left, divided by the current value for N. "**" indicates that at this setting of N, K and T, even the maximum possible value of C produced only satisfiable instances. A blank entry signifies that problems generated with these parameters were too large to run.

four parameters: N, K, T and C. The problem instances are binary CSPs with N variables, each having a domain of size K. The parameter T (tightness) specifies a fraction of the K^2 value pairs in each constraint that are disallowed by the constraint. The value pairs to be disallowed by the constraint are selected randomly from a uniform distribution, but each constraint has the same fraction T of such incompatible pairs. T ranges from 0 to 1, with a low value of T, such as 1/9, termed a loose or relaxed constraint. The parameter C specifies the number of constraints out of the $N*(N-1)/2$ possible. The specific constraints are chosen randomly from a uniform distribution. This model is the binary CSP analog of the Random KSAT model described in (Mitchell, Selman, & Levesque 1992).

Although our random generator can create extremely hard instances, they may not be typical of actual problems encountered in applications. Therefore, in order to capture a wider variety of instances we introduce another generator, the *chain* model, that creates problems with a specific structure. A chain problem instance is created by generating several disjoint subproblems, called *nodes*, with our general generator described above, ordering them arbitrarily, and then joining them sequentially so that a single constraint connects one variable in one subproblem with one variable in the next.

50% Solvable Points for CSPs

All experiments reported in this paper were run with combinations of N, K, T and C that produces problem instances which are about 50% solvable (sometimes called the "cross-over" point). These combinations were determined empirically, and are reported in Fig. 1. To find cross-over points we selected values of N, K and T, and then varied C, generating 250 or more instances from each set of parameters until half of the problems had solutions. Sometimes no value of C resulted in exactly 50% satisfiable; for instance with $N = 50, K = 6, T = 12/36$ we found with $C = 194$ that 46% of the instances had solutions, while with $C = 193$ 54% did. In such cases we report the value of C that came closest to 50%.

For some settings of N, K and T, all values of C produce only satisfiable instances. Since generally there is an inverse relationship between T, the tightness of each constraint, and C, the number of constraints, this situation occurs when the constraints are so loose that even with C at its maximum value, $N*(N-1)/2$, no unsatisfiable instances result. Our data indicate that this phenomenon only occurs at small values of N.

N	BT+MW	BJ+MW	BT+DVO	BJ+DVO
	K=3 T=1/9			
25	65,413	14,964	2,006	1,977
50	15,248,270	383,321	10,944	10,214
75		8,268,113	51,907	45,014
100		320,587,286	245,974	190,965
125			1,596,655	832,753
150			14,834,004	3,301,619
	K=3 T=2/9			
25	7,489	2,177	571	549
50	895,245	22,153	2,284	1,785
75		243,845	12,581	5,669
100		2,856,423	2,730,226	18,097
125			907,645	32,326
150			4,892,729	199,617
	K=3 T=3/9			
25	2,324	588	254	236
50	1,096,518	2,947	1,493	547
75		11,912	32,604	1,071
100		68,532	1,761,694	2,967
125		341,046		6,329
150		500,734		7,601
	K=3 T=4/9			
25	991	229	124	117
50	43,091,355	642	868	206
75		1,498	141,799	330
100		4,069	1,205,712	855
125		10,722		995
150		14,490		1,916
	K=9 T=9/81			
15	5,844	724	673	673
25	859,802	116,382	1,929	1,924
35		119,547,843	219,601	217,453
	K=9 T=18/81			
15	110,242	48,732	2,428	2,426
25	15,734,382	6,841,255	253,289	252,581
35		392,776,002	17,988,106	17,901,386
	K=9 T=27/81			
15	106,762	73,541	10,660	10,648
25	1,099,838	583,038	55,402	54,885
35		4,868,528	201,658	189,634

Figure 2: Comparison of backjumping and backtracking with min-width and dynamic variable ordering. Each number represents mean consistency checks over 1000 instances. The chart is blank where no experiments were conducted because the problems became too large for the algorithm.

We often found that the peak of difficulty, as measured by mean consistency checks or mean CPU time, is not exactly at the 50% point, but instead around the 10% to 30% solvable point, and the level of difficulty at this peak is about 5% to 10% higher than at the 50% point. We nevertheless decided to use the 50% satisfiable point, since it is algorithm independent. The precise value of C that produces the peak of difficulty can vary depending on algorithm, since some approaches handle satisfiable instances more efficiently.

In contrast to the findings of (Mitchell, Selman, & Levesque 1992; Crawford & Auton 1983) for 3-SAT, we did not observe a precise linear relationship between the number of variables and the number of constraints (which are equivalent to clauses in CNF). The ratio of C to N appears to be asymptotically linear, but it is impossible to be certain of this from our data.

Nodes	BT+MW	BJ+MW	BT+DVO	BJ+DVO
5	17,395,021	13,249	21,564	2,824
10		27,315	83,828	4,707
20		98,260	282,101	8,260
30		294,771	1,201,582	19,882

Figure 3: Comparison of backjumping and backtracking with min-width and dynamic variable ordering, using "chain" problems with 15-variable nodes. K=3, T=1/9, and $N = 15*$ "Nodes". Each number represents mean consistency checks over 1000 instances.

Static and Dynamic Variable Orderings

In our first set of experiments we wanted to assess the merits of static and dynamic variable orderings when used with backtracking and backjumping. As the data from Fig. 2 indicate, DVO prunes the search space so effectively that when using it the distinction between backtracking and backjumping is not significant until the number of variables becomes quite large. An exception to this general trend occurs when using backtracking with dynamic variable ordering on sparse graphs. For example, with $N=100$, $K=3$, and $T=3/9$, C is set to 169, which creates a very sparse graph that occasionally consists of two or more disjoint sub-graphs. If one of the sub-graphs has no solution, backtracking will still explore its search space repeatedly while finding solutions to the other sub-graphs. Because backjumping jumps between connected variables, in effect it solves the disconnected sub-graphs separately, and if one of them has no solution the backjumping algorithm will halt once that search space is explored. Thus the data in Fig. 2 show that backtracking, even with dynamic variable ordering, can be extremely inefficient on large CSPs that may have disjoint sub-graphs.

T	C	$C/2775$	DVO single	MW jmp size
1/9	565	.204	68%	1.92
2/9	244	.088	39%	3.55
3/9	132	.048	27%	5.68
4/9	74	.027	16%	7.25

Figure 4: Data with $N=75, K=3$, drawn from the same experiments as in Fig. 2. The column "C/2775" indicates the ratio of constraints to the maximum possible for $N=75$.

At large N, the combination of DVO and backjumping is particularly felicitous. Backjumping is more effective on sparser constraint graphs, since the average

K	N	T	Backtracking	Forward Checking
3	100	1/9	245,974	252,229
3	100	2/9	2,730,226	5,052,422
3	100	3/9	1,761,694	665,109
6	35	4/36	639,699	646,529
6	35	8/36	78,217	79,527
6	35	12/36	18,404	18,981
6	35	16/36	6,863	7,125
9	25	9/81	1,929	1,935
9	25	18/81	253,289	255,589
9	25	27/81	55,402	56,006
9	25	36/81	17,976	18,274

Figure 5: Comparison of backtracking and forward checking with DVO. Each number is the mean consistency checks over 1000 instances.

size of each "jump" increases with increasing sparseness. DVO, in contrast, tends to function better when there are many constraints, since each constraint provides information it can utilize in deciding on the next variable. We assessed this observation quantitatively by recording the frequency with which backjumping with DVO selected a variable that only had one remaining compatible value. This is the situation where DVO can most effectively prune the search space, since it is acting exactly like unit-propagation in boolean satisfiability problems, and making the forced choice of variable instantiation as early as possible. See Fig. 4, where the column labelled "DVO single" shows how likely DVO was to find a variable with one remaining consistent value, for one setting of N and K. The decreasing frequency of single-valued variables as the constraint graph becomes sparse indicates that DVO has to make a less-informed choice about the variable to choose next.

For the backjumping algorithm with a MW ordering we recorded the average size of the jump at a dead-end, that is, how many variables were passed over between the dead-end variable and the variable jumped back to. With backtracking this statistic would always be 1. This statistic is reported in the "MW jmp size" column in Fig. 4, and shows how backjumping jumps further on sparser graphs.

Dynamic variable order was somewhat less successful when applied to the chain type problems. With these structured problems we were able to experiment with much larger instances, up to 450 variables organized as thirty 15-variable nodes. The data in Fig. 3 show that backjumping was more effective on this type of problem than was DVO, and the combination of the two was over an order of magnitude better than either approach alone.

Forward Checking

A benefit of studying algorithms by observing their average-case behavior is that it is sometimes possible to determine which component of an algorithm is actually responsible for its performance. For instance, forward checking is often acclaimed as a particularly good algorithm (Nadel 1989). We note that it is possible to implement just part of forward checking as a variable ordering heuristic: if instantiating a variable with a certain value will cause a future variable to be a dead-end, then rearrange the variable ordering to make that future variable the next variable. The result is essentially backtracking with DVO. This method does not do all of forward checking, which would require rejecting the value that causes the future dead-end. In Fig. 5 we compare forward checking with backtracking, using DVO for both algorithms. The result is approximately equivalent performance. Thus we suggest that forward checking should be recognized more as a valuable variable ordering heuristic than as a powerful algorithm.

Backmarking and sticking values

The next set of experiments was designed to determine whether backmarking and sticking values, alone or in combination, could improve the performance of backjumping under a static min-width ordering. (We plan to report on backmarking and sticking values with dynamic variable ordering in future work.) Since backmarking and sticking values remember information about the history of the search in order to guide future search, we report on CPU time as well as consistency checks (see Fig. 6). Is the overhead of maintaining additional information less than the cost of the saved consistency checks? Only by examining CPU time can we really tell. We implemented all the algorithms and heuristics described in this paper in a single C program, with common data structures, subroutines, and programmer skill, so we believe comparing CPU times is meaningful, though not definitive.

Our experiments as summarized in Fig. 6 show that both backmarking and sticking values offer significant improvement when integrated with backjumping, usually reducing CPU time by a half or a third. As expected, the improvement in consistency checks is much greater, but both enhancements seem to be cost effective. Backmarking offers more improvement than does sticking values. Both techniques are more effective on the problems with smaller domain sizes; at $K = 9$ the benefit of sticking values in terms of reduced CPU time has almost disappeared. Backmarking helps backjumping over all the problem types we studied. The results from chain problems did not vary significantly from those of the unstructured problems.

4. Conclusions

We have several results from experimenting with larger and harder CSPs than have been reported before. Backjumping with dynamic variable ordering seems in general to be a powerful complete search algorithm. The two components complement each other, with backjumping stronger on sparser, more structured, and possibly disjoint graphs. We have shown that the

			Consistency Checks				CPU Seconds			
K	N	T	BJ	BJ+BM	BJ+ST	BJ+BM+ST	BJ	BJ+BM	BJ+ST	BJ+BM+ST
3	100	1/9	8,268,113	2,600,518	3,800,616	1,423,911	48.954	30.737	23.773	15.198
3	100	2/9	243,835	101,389	129,220	61,326	3.045	2.043	1.664	1.255
3	100	3/9	11,912	6,777	7,599	4,733	0.359	0.302	0.275	0.249
3	100	4/9	1,498	1,096	1,132	873	0.171	0.165	0.163	0.159
6	35	4/36	113,514,082	22,126,240	104,507,721	20,071,376	274.410	140.622	236.639	120.630
6	35	8/36	2,274,267	466,249	1,672,745	438,477	9.410	4.308	6.429	4.003
6	35	12/36	235,842	74,645	215,581	70,725	1.429	0.872	1.212	0.823
6	35	16/36	39,868	15,674	37,963	15,344	0.342	0.224	0.303	0.219
9	25	9/81	116,382	15,157	97,672	13,792	0.250	0.112	0.220	0.100
9	25	18/81	6,786,710	1,260,078	6,514,347	1,239,246	19.117	10.548	18.565	10.393
9	25	27/81	583,038	144,578	566,322	142,904	2.249	1,430	2.247	1.427
9	25	36/81	96,245	30,761	93,870	30,410	0.483	0.339	0.485	0.339
Chain problems with 30 nodes of 15 variables each.										
3	450	1/9	294,771	93,151	188,618	80,264	10.736	6.211	8.003	5.632

Figure 6: Results from experiments with backjumping, backmarking and sticking values. Each number is the mean of 1000 instances, and a min-width ordering was used throughout.

power of forward checking is mostly subsumed by a dynamic variable ordering heuristic. We have introduced a new value ordering heuristic called sticking values and shown that it can significantly improve backjumping when the variables' domains are relatively small. We have also shown that the backmarking technique can be applied to backjumping with good results over a wide range of problems.

One result visible in all our experiments is that among problems with a given number of variables, and drawn from the 50% satisfiable region, those with many loose constraints are much harder than those with fewer and tighter constraints. This is consistent with tightness properties shown in (van Beek & Dechter 1994). The pattern is not always observed for low values of N and T, since there may be no 50% region at all. We have also shown that the linear relationship between variables and clauses observed with boolean satisfiability problems at the cross-over point is not found with CSPs generated by our model.

References

Bitner, J. R., and Reingold, E. 1985. Backtrack programming techniques. *Communications of the ACM* 18:651–656.

Collin, Z.; Dechter, R.; and Katz, S. 1991. On the Feasibility of Distributed Constraint Satisfaction. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 318–324.

Crawford, J. M., and Auton, L. D. 1983. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 21–27.

Dechter, R. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence* 41:273–312.

Even, S. 1979. *Graph Algorithms*. Maryland: Computer Science Press.

Freuder, E. C. 1982. A sufficient condition for backtrack-free search. *JACM* 21(11):958–965.

Gaschnig, J. 1979. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University.

Haralick, R. M., and Elliott, G. L. 1980. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* 14:263–313.

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

Mitchell, D.; Selman, B.; and Levesque, H. 1992. Hard and Easy Distributions of SAT Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 459–465.

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

Prosser, P. 1983. BM + BJ = BMJ. In *Proceedings of the Ninth Conference on Artificial Intelligence for Applications*, 257–262.

Prosser, P. 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9(3):268–299.

Purdom, P. W. 1983. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence* 21:117–133.

Selman, B.; Levesque, H.; and Mitchell, D. 1992. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 440–446.

van Beek, P., and Dechter, R. 1994. Constraint tightness versus global consistency. In *Proc. of KR-94*.

Zabih, R., and McAllester, D. 1988. A Rearrangement Search Strategy for Determining Propositional Satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 155–160.