Journal of Automated Reasoning **15:** 359–383, 1995.

Branching Rules for Satisfiability*

J. N. HOOKER
Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, U.S.A.
e-mail: jh38@andrew.cmu.edu

and

V. VINAY
Centre for Artificial Intelligence and Robotics, Bangalore, India
e-mail: vinay@yantra.ernet.in

(Received: 7 June 1994)

Abstract. Recent experience suggests that branching algorithms are among the most attractive for solving propositional satisfiability problems. A key factor in their success is the rule they use to decide on which variable to branch next. We attempt to explain and improve the performance of branching rules with an empirical model-building approach. One model is based on the rationale given for the Jeroslow–Wang rule, variations of which have performed well in recent work. The model is refuted by carefully designed computational experiments. A second model explains the success of the Jeroslow–Wang rule, makes other predictions confirmed by experiment, and leads to the design of branching rules that are clearly superior to Jeroslow–Wang.

Key words: branching, algorithms, satisfiability, Jeroslaw–Wang rule.

Recent computational studies [2, 7, 13, 21] suggest that branching algorithms are among the most attractive for solving the propositional satisfiability problem. An important factor in their success – perhaps the dominant factor – is the *branching rule* they use [13]. This is a rule that decides, at each node of an enumeration tree, which variables should be set to true or false in order to generate the children of that node. A clever branching rule can reduce the size of the search tree by several orders of magnitude.

One rule that has been found to be particularly effective in a wide variety of problems [13] is the Jeroslow–Wang rule [17], which we define below. Another promising rule is the shortest positive clause rule used by Gallo and Urbani in their Horn relaxation algorithm [11]. There is little understanding, however, of when and why these rules work well.

Our purpose here is to try to improve our understanding of branching rules and to design better ones. We will show that the original motivation for the J–W rule, namely that it takes a branch in which one is most likely to find a satisfying truth assignment, does not explain its performance. A proper explanation is

* GSIA Working Paper 1994-09. The first author is partially supported by ONR grant N00014-92-J-1028. The authors wish to thank Ajai Kapoor for assistance in computational p667 testing and statistical analysis.

considerably more nuanced and reveals that the original motivation produces a good rule only through a remarkable coincidence.

Furthermore, our analysis leads us to a "two-sided" J–W rule that, in computational tests, is significantly better than J–W. We find that the shortest positive clause rule is inferior to the 2-sided J–W rule but has the interesting feature that it branches only on variables that occur in positive clauses. When this feature is added to the two-sided rule, the latter's performance appears improved, although statistical analysis does not permit us to establish improvement with 95% confidence.

We implement each branching rule within the same basic Davis–Putnam–Loveland (DPL) algorithm [5, 20], with a slight modification when shortest clause branching is used. The algorithm performs unit resolution and monotone variable fixing at each node. For the sake of providing a controlled testing environment, we omit numerous other devices that might accelerate performance. Our computation times should therefore not be taken as the best that might be achieved, and they are in fact generally longer than those obtained by DPL-based algorithms that were tested as part of the Second DIMACS Challenge [7, 9, 23, 25]. Because the DIMACS algorithms differ from each other and from ours in several respects, it is impossible to identify the factors that account for differences in performance. By using a rudimentary DPL algorithm, we sacrifice performance but isolate the influence of the branching rule.

Although we focus on a DPL-based algorithm, it is reasonable to believe that the insights gained here could be profitably applied to other algorithms that involve branching, such as the Horn relaxation algorithm, the branch-and-cut algorithm of Hooker and Fedjki [15], and the hypergraph algorithms of Gallo and Pretolani [10].

A second purpose of this study is to demonstrate some elements of the empirical paradigm for the study of algorithms recommended in [14]. Rather than simply compare branching rules in computational tests, we formulate models that purport to explain the behavior of branching rules. We view these models as empirical theories analogous to those developed in physics or chemistry. They do not and are not intended to lead to mathematical theorems, but they make certain testable predictions. We design computational experiments to test the predictions and analyze the results statistically. Negative results refute a theory, whereas positive results provide some degree of confirmation.

We begin in Section 1 by stating a generic branching algorithm for the satisfiability problem. In Section 2 we describe our experimental design, defend our procedures for statistical analysis, and summarize the computational results. The remainder of the paper refers to these results repeatedly in order to test various predictions.

In Section 3 we formulate a Satisfaction Hypothesis that seems best to capture the traditional motivation for the J–W rule. On this hypothesis the rule works because it takes a branch in which the probability of satisfiability is maximized.

The hypothesis, however, does not explain the behavior of the J–W rule on unsatisfiable problems. It does lead to a probabilistic model from which one can derive the J–W rule as an approximation. But the model makes a prediction that is contradicted by experience. Furthermore, a rule derived in Section 4 from a more refined model, one based on a generalized Lovász local lemma, does not result better performance as it should.

In Section 5 we propose a Simplification Hypothesis that explains a rule's performance in terms of how effectively it simplifies the problem. We develop a Markov Chain model that estimates the degree of simplification. We find that the model explains observations that refute the satisfaction model and correctly predicts superior performance for a two-sided branching rule. In Section 6 we discuss the clause branching rule and the value of its strategy of branching only on variables that occur in a positive clause. Section 7 summarizes the results and conclusions.

1. A Generic Branching Algorithm

The satisfiability problem (SAT) of propositional logic is generally given in conjunctive normal form, or *clausal* form. A clause is a disjunction of *literals*, each of which is an atomic proposition or its negation. The following is a clause,

$$x_1 \vee \neg x_2 \vee \neg x_3,$$

where $\vee$ means "or," $\neg$ means "not," and the x_j's are atomic propositions (atoms) that must be either true or false. (A clause may not contain more than once occurrence of any given atom.) The SAT problem is to determine whether some assignment of truth values to atoms makes a given conjunction of clauses true, or equivalently, whether some assignment makes every clause in a set S of clauses true. It is the original NP-complete problem [3].

Any propositional formula may be converted to clausal form in linear time, possibly by adding new atoms [26].

The Davis–Putnam algorithm [5], as modified by Loveland [20], is a generic branching method for solving SAT. It searches a tree in which the root node is associated with the original problem. It applies monotone variable fixing and unit resolution (explained below) at each node to simplify the problem and perhaps fix some variables to true or false. The leaf nodes are those at which unit resolution finds a contradiction, or the variables fixed so far satisfy all the clauses. Each nonleaf node has two children associated with simpler subproblems that are obtained by setting some variable to true and then to false. The search is depth-first and terminates when it reaches a node at which all clauses are satisfied or until it backtracks to the root node, in which case the problem is unsatisfiable. A *branching rule* determines which variable is fixed to true and false at each node, and which child is explored first.

```
Davis–Putnam–Loveland Algorithm.

Procedure Branch(S, k)
   Perform Monotone Variable Fixing on S.
   Perform Unit Resolution on S.
   If a contradiction is found, return.
   If S is empty,
      declare problem to be satisfiable and stop.
   Branch :
      Pick a literal L containing a variable
         that occurs in S.
      Perform Branch(S ∪ {L},k + 1).
      Perform Branch(S ∪ {¬L},k + 1).
End.
```

```
Procedure Monotone Variable Fixing.
   While S contains a monotone literal L do :
      Fix L to true.
      Remove from S all clauses containing L.
End.

Procedure Unit Resolution
   While S contains a unit clause L do :
      Fix L to true.
      Remove from S all clauses containing L.
      Remove from S all occurrences of ¬L.
      If ¬L is removed from a unit clause,
         return with contradiction.
End.
```

A statement of the algorithm appears above. The algorithm is initiated with the procedure call **Branch**(S,1), where S is the set of clauses to be checked for satisfiability. k is the current level in the search tree. The branching rule determines the choice of the literal L. For convenience we say that the algorithm *branches on* the variable in L and *branches to* L. S is unsatisfiable if and only if the algorithm never declares it satisfiable.

Monotone variable fixing simply fixes to true any variable that always occurs posited, and to false any variable that always occurs negated. It deletes all clauses containing the fixed variables and repeats the process. *Unit resolution* fixes to true any literal that belongs to a unit clause (a clause containing exactly one literal).

This allows the problem to be simplified. The process continues until a variable is fixed to both true and false (contradiction), or no unit clauses remain.

2. Experimental Design and Analysis

There is no generally accepted approach to the design and analysis of computational experiments, as only a handful of papers in the literature use rigorous methods (e.g., [1, 12, 19]). The first rigorous treatment of which we are aware is that of Lin and Rardin [19]. They use a traditional factorial design in which the response variable (in our case, the computation time or the number of nodes in the search tree) is influenced by two factors, namely the algorithm type and the problem type. A fixed number of random problems are generated in each cell (i.e., each algorithm/problem type combination). The problem type is specified by setting parameters in the problem generator.

Lin and Rardin recommend "blocking on problems" for comparing algorithms. This approach runs each algorithm on the same set of random problems and therefore removes one source of noise that may obscure the relative performance of the algorithms. Golden and Stewart [12] also use blocking on problems, and Amini and Racer [1] use it in the context of a split plot design.

We use a similar factorial design with blocking on problems. The factors are the branching rule and the problem type. The problems themselves (152 in number) are taken from a library of satisfiability problems collected by the DIMACS project mentioned earlier. They are publicly available via anonymous ftp to dimacs.rutgers.edu. Some of the problems in the library could not be solved within the memory limitations of our computer. We deleted still other problems that fit into memory but could not be solved within two hours by any of the branching rules. The remaining problems appear in Tables I and II.

The intent of using multiple factors in a design is to isolate the effect of such nuisance parameters as problem type and problem size from the effect of the branching rule. We did not use problem size as a separate factor, because the test problems of a given type are generally of similar size or else difficult to classify in size categories. When size classes could be distinguished we split a problem type into two subtypes, based on size. This yielded 11 problem types, listed in Tables I and II. We tested 9 branching rules and thereby obtained 99 cells.

An analysis of variance (ANOVA) is traditionally used to determine which factor levels have a statistically significant effect on the response. We did not use ANOVA, for two reasons. First, it requires that each cell contain an equal number of problems. This is not the case for the DIMACS library, whose problem classes differ in size. In such cases Petersen [22] recommends that one use multiple regression with dummy variables, and we followed this advice.

Secondly, the traditional ANOVA is based on an interpretation of the blocked design that seems strained here. The notion of a block is inspired by agricultural experimentation, in which each of several plots of land receives various

TABLE I. List of satifiability problems solved

Problem class	No. of problems	Problem names*
aim100	21	aim-100-[1_6-[no,yes1]-[1,2,3,4], 2_0-[no-[2,3,4],yes1-[1,2,3,4]], [3_4-yes1-[3,4], 6_0-yes1-[1,2,3,4]]]
aim200	15	aim-200-[[1_6, 3_4,6_0]-[1,2,3,4], 2_0-[1,3,4]]
aim50	24	aim-50-[1_6, 2_0]-[no,yes1]-[1,2,3,4], aim-50-[3_4, 6_0]-yes1-[1,2,3,4]
Dubois	3	dubois[20,21,100]
ii32	11	ii32[a1,b[1,2,3,4],c[1,2,3],d1,e[4,5]]
ii8	11	ii8[a[1,2,3,4],b[2,3],c[1,2],d[1,2],e1]
jnh	48	jnh[2,3, ..., 8,10, ..., 20,201, ..., 220,301, ..., 310]
par16	4	par16-[1,1-c,2,2-c]
par8	9	par8-[1,2,2-c,3,3-c,4,4-c,5,5-c]
pret	4	pret60_[25,40,60,75]
ssa	2	ssa[7552-038,0432-003]

*Brackets indicate multiple problems. For instance, a[b,c] indicates problems ab and ac; a[b,c[d,e]] indicates problems ab, acd, and ace; a[[b,c]d[e,f],g] indicates abde, abdf, acde, acdf and ag.

TABLE II. Problem characteristics

Problem class	Number of variables	Number of clauses	Number sat	Number unsat	Description*
aim100	100	160-600	14	7	Problems generated by method of Iwama,
aim200	200	320-1200	15	0	Albeta and Miyano [16]; satisfiable
aim50	50	80-300	16	8	instances have exactly one solution.
Dubois	60-300	160-598	1	2	Hard random instances [6]
ii32	225-522	1280-11636	11	0	Inductive inference problems coded
ii8	66-950	186-6689	11	0	as SAT instances [18]
jnh	100	800-900	15	33	Random instances [13, 15]
par16	317-1015	1264-3374	4	0	Parity learning problems coded as
par 8	67-350	266-1171	9	0	SAT instances [4]
pret	60	161	0	4	Graph 2-coloring problems [23]
ssa	435-1501	1027-3575	1	1	Circuit stuck-at fault analysis [25]
Total			97	55	

*Further information is available by anonymous ftp to dimacs.rutgers.edu, or from the sources cited.

treatments (fertilizer type, watering level, etc.). Each treatment is used once in each plot. The rationale is that the soil in a given plot (block) is likely to be homogeneous, so that the treatments get a fair comparison. This approach obviously implies that each treatment is used an equal number of times (i.e., each cell contains the same number of data).

In a computational context, the problem type and the algorithm must be regarded as treatments applied to an underlying block of problems. This is reasonable in the case of algorithms but not in the case of problem types. There is no underlying class of problems $1, \ldots, n$ to which one applies different treatments to obtain problems $A_1, \ldots, A_n$ of type A, problems $B_1, \ldots, B_n$ of type B, and so on. One merely generates n problems of types A, n more problems of type B, etc.

The regression approach, which we used, regards problem type simply as an attribute that may affect the response variable. There is no need to generate an equal number of problems of each type. We used the following regression model, which accounts for interactions between the problem type and branching rule.

$$Z = \mu + \sum_{i=1}^{10} b_i X_i + \sum_{j=1}^{8} c_j Y_j + \sum_{i=1}^{10} \sum_{j=1}^{8} d_{ij} X_i Y_j.$$

The response variable Z is the computation time or node count. The dummy variables are,

$$X_i = \begin{cases} 1, & \text{if problem type is } i, \\ 0, & \text{otherwise,} \end{cases}$$
$$Y_j = \begin{cases} 1, & \text{if branching rule } j \text{ is used,} \\ 0, & \text{otherwise.} \end{cases}$$

Note that in the summations, i ranges over all but one of the problem classes (numbered $0, \ldots, 10$), and j over all but one of the branching rules (numbered $0, \ldots, 8$). Thus if Z is computation time,

μ = predicted time required by rule 0 on problem class 0,
$\mu + b_i$ = predicted time required by rule 0 on class i $(i > 0)$,
$\mu + c_j$ = predicted time required by rule j on class 0 $(j > 0)$,
$\mu + b_i + c_j + d_{ij}$ = predicted time required by rule j on class i $(i, j > 0)$,

and similarly if Z is the node count. We will refer to these quantities as the *summed effects*. The regression problem has a unique solution because the number of parameters (99) equals the number of cells.

Missing data points are a perennial problem in computational testing, because computation must be cut off if it runs impracticably long. Lin and Rardin discuss various ways of estimating missing data. But these approaches make assumptions about the distribution of computation times that are unrealistic here, due to the extreme outliers that satisfiability problems typically generate. We simply used the cutoff time (two hours) as a surrogate for the true time. In the context of our

TABLE III. Summed effects on computation time (seconds)

	Problem Class					
	0	1	2	3	4	5
Branching Rule	aim100	aim200	aim 50	Dubois	ii32	ii8
0. Random	4787	5713	7.37	1515	514	0.6
1. J–W	461	3151	0.50	1698	16	2001
2. 1st order	4378	5555	6.69	1555	45	3399
3. 2nd order	2839	5003	5.07	4826	958	5205
4. Reverse J–W	414	3134	0.46	1694	143	0.9
5. 2-sided J–W	183	1465	0.27	1673	10	1920
6. Clause	2162	3055	1.89	579	135	499
7. Pos J–W	325	2518	0.33	1424	7	1023
8. 2-sided pos J–W	162	1431	0.24	1410	5	1111

	6	7	8	9	10
Branching Rule	jnh	par16	par 8	pret	ssa
0. Random	23.44	6660	0.129	1462	478
1. J–W	0.76	358	0.041	1298	5
2. 1st order	11.63	4511	0.101	1174	5
3. 2nd order	222.33	6648	4.014	7200	392
4. Reverse J–W	0.87	357	0.041	1297	172
5. 2-sided J–W	0.39	355	0.037	1280	170
6. Clause	3.70	335	0.048	665	3603
7. Pos J–W	0.80	317	0.041	711	3
8. 2-sided pos J–W	0.40	314	0.038	701	53

study this tends to provide a conservative test. Whenever we conclude that one algorithm is significantly better than another, the better algorithm is cut off less often than the worse. It is likely therefore that their difference would be even more significant if the true times were used. We note below, however, that a different approach is required in the case of the second-order branching rule.

The problems were solved on an HP Series 9000 workstation using a code written by J. Hooker in C and compiled with the HP UX C compiler with optimization. The computation times exclude time required to set up the data structures and read the problem into memory. In the case of satisfiable problems, they reflect the time required to find one solution only; no attempt is made to find all solutions. Each of the 9 branching rules was applied to 152 problems, resulting in 1368 data points.

The resulting estimates for the summed effects are displayed in Table III. For instance, the predicted computation time for solving a problem in Class 0 with the 2-sided positive J–W branching rule is 162 seconds. (The branching

TABLE IV. Branching rules. The rules are more precisely defined in subsequent sections. For a given literal L, the quantity $J(L)$ is defined to be the sum of 2^{-n_i} over all clauses C_i containing L, where n_i is the number of literals in C_i

0. Random	Random branching: randomly select unfixed literal.
1. J–W	Jeroslow–Wang rule: maximize $J(L)$ over literals L.
2. 1st order	First Order Probability rule: maximize $J(L) - J(\neg L)$, a 1st-order estimate of satisfaction probability.
3. 2nd order	Second Order Probability rule: maximize a 2nd-order estimate of satisfaction probability, based on a generalization of the Lovász local lemma.
4. Reverse J–W	Reverse Jeroslow–Wang rule: maximize $J(\neg L)$.
5. 2-sided J–W	Two-Sided Jeroslow–Wang rule: maximize $J(x_j) + J(\neg x_j)$ over variables x_j.
6. Clause	Shortest Positive Clause Branching: Branch on the literals in a shortest clause containing all positive literals.
7. Pos J–W	Positive Jeroslow–Wang rule: J–W rule, but branch only on literals that occur in an all-positive clause.
8. 2-sided pos J–W	Two-Sided Positive Jeroslow–Wang rule: 2-sided J–W, but branch only on literals that occur in a positive clause.

rules are defined in Table IV.) Summed effects on the number of nodes in the search tree are displayed in Table V. It is clear from these tables that the choice of branching rule can make an enormous difference in the behavior of the algorithm, sometimes two or three orders of magnitude.

The statistical significance of each coefficient b_i, c_j, d_{ij} can be estimated using t statistics. 32 of the 80 interaction coefficients d_{ij} are significantly different from zero at the 95% confidence level. This indicates that the relative performance of the branching rules differs significantly from one problem class to another. Their performance should therefore be examined in each problem class individually.

It is also possible, using t statistics for the difference between coefficients, to determine whether one rule is significantly better than another within a given problem class. It turns out that rather large differences fail to be significant at the 95% level. For instance, Rule 0 is not significantly better than Rule 4 in problem class 0, even though its predicted running time is only a third as much. This is due to the fact that problems in a single class tend to differ widely in difficulty (e.g., by factors of 1000 or more). This introduces intra-class variation that reduces the ability of the regression to detect significant differences between branching rules even when controlling for problem type.

To alleviate this difficulty, we followed Golden and Stewart [12] in using Wilcoxon's signed rank test, a nonparametric test that can be used to determine whether one algorithm is better than another on a common set of problems. Since it measures differences between two branching rules by rank rather than

TABLE V. Summed effects on node count (thousands)

	Problem Class					
	0	1	2	3	4	5
Branching Rule	aim100	aim200	aim 50	Dubois	ii32	ii8
0. Random	18286	9590	39.9	3515	35.8	631
1. J–W	1337	4847	2.0	6291	1.6	1445
2. 1st order	17302	9582	34.6	6291	6.1	2876
3. 2nd order	883	249	2.3	538	0.02	27
4. Reverse J–W	1227	4863	1.8	6291	4.2	0
5. 2-sided J–W	507	1825	1.0	6291	0.8	1061
6. Clause	8047	6974	9.5	3146	37.5	83
7. Pos J–W	983	4217	1.3	5243	0.8	594
8. 2-sided pos J–W	482	1990	0.9	5243	0.6	480

	6	7	8	9	10
Branching Rule	jnh	par16	par 8	pret	ssa
0. Random	9.19	2761	0.105	4814	526.9
1. J–W	0.18	67	0.017	5542	3.7
2. 1st order	4.62	1059	0.068	5542	3.8
3. 2nd order	0.46	6	0.020	1074	5.0
4. Reverse J–W	0.20	67	0.019	5542	47.4
5. 2-sided J–W	0.08	67	0.017	5542	46.6
6. Clause	1.26	79	0.029	3111	733.4
7. Pos J–W	0.19	64	0.019	3207	2.1
8. 2-sided pos J–W	0.08	64	0.019	3207	16.4

actual values, it is not affected by the extreme outliers typical of satisfiability problems. For instance, if rule A is 1 second faster than rule B on problem 1, 10 seconds faster on problem 2, and 100,000 seconds faster on a very hard problem 3, the differences would be equated with their respective ranks 1, 2, 3 rather than their actual values. We found that the Wilcoxon test is somewhat more likely to judge differences between rules to be statistically significant. It permits only pairwise comparison (Friedman's test can be used for multiwise comparison), but this is adequate for our purposes. We computed Wilcoxon statistics for all pairs of branching rules within each of the 11 problem classes. *All significance results quoted henceforth will be those of the Wilcoxon test at the 95% confidence level.*

The meaningfulness of significance testing obviously rests on the assumption that the problem sample is random in some sense. The DIMACS problems may represent a biased sample, and another problem set could yield different results. But these problems resulted in performance variations so large that only very

pronounced differences between branching rules could pass the significance tests. These tests may therefore screen out many of the results that are likely to be artifacts of the problem sample.

3. The Satisfaction Hypothesis

We first attempt to capture the rationale for the Jeroslow–Wang rule in an empirical hypothesis. The hypothesis is somewhat imprecise but will shortly provide the motivation for two precise models that make testable predictions.

Satisfaction Hypothesis: Other things being equal, a branching rule performs better when it creates subproblems that are more likely to be satisfiable.

First, we describe the J–W rule itself. It branches to a literal that occurs in a large number of short clauses. To state it more precisely, let S contain the clauses $C_1, \ldots, C_m$.

1. Jeroslow–Wang Rule

Branch to a literal L that maximizes

$$J(L) = \sum_{\substack{i \\ L \in C_i}} 2^{-n_i},$$

over all literals L, where n_i is the number of literals in C_i.

Table III shows that the J–W rule (Rule 4) can indeed result in much more intelligent branching than a random choice of branching variable (Rule 1). It is better than random branching in 9 out of 11 problem classes. The superiority is statistically significant in 6 classes (0, 1, 2, 4, 6, 7) and often substantial (an order of magnitude or more).

Jeroslow and Wang justify their rule as one that tends to branch to a subproblem that is most likely to be satisfiable ([17, pp. 172–173]). They reason that clause C_i rules out 2^{n-n_i} truth valuations, so that all the clauses remaining after branching to L rule out at most $2^n \sum_i 2^{-n_i} = 2^n J(L)$ valuations. By maximizing the number of valuations ruled out by the clauses that are deleted, they maximize the number that are not ruled out by those clauses remaining in the subproblem. This presumably makes a satisfiable subproblem more likely.

To begin with this, this motivation does nothing to explain the performance of the J–W rule on unsatisfiable problems.

But this aside, a more careful analysis shows that the motivation is problematic even for satisfiable problems. Let X_i be an indicator random variable that is 1 when a random truth assignment falsifies C_i and 0 otherwise. Clearly

$$\Pr(X_i = 1) = E(X_i) = 2^{-n_i}.$$

Here $E(X_i)$ is the expected value of X_i. If we define $X = X_1 + \cdots + X_m$, $\Pr(X = 0)$ is the probability that a random truth assignment satisfies all the clauses. Since this probability is hard to compute, we can approximate it with the following well-known lower bound:

$$\Pr(X = 0) \geqslant 1 - E(X). \tag{1}$$

The right-hand side gives a kind of first-order approximation of $\Pr(X = 0)$. (We will discuss higher order approximations in Section 4.) The expected number of falsified clauses $E(X)$ is easy to compute, since by the linearity of expectations,

$$E(X) = \sum_{i=1}^{m} 2^{-n_i}.$$

Thus we can maximize an approximation of $\Pr(X = 0)$ by minimizing $E(X)$. In particular, we obtain a satisfiable problem if one exists, since $\Pr(X = 0) > 0$ if $E(X) < 1$.

To derive a branching rule, we suppose that S is the problem at the current node and consider the effect of branching to a literal L. The expected number of falsified clauses in the resulting subproblem is

$$E(X|L) = \sum_{\substack{i \\ \neg L \in C_i}} 2^{-(n_i - 1)} + \sum_{\substack{i \\ L, \neg L \notin C_i}} 2^{-n_i}, \tag{2}$$

because the clauses containing L are removed, whereas the clauses containing $\neg L$ contain one less literal. We wish to branch to a literal L that minimizes $E(X|L)$. Since the above expression may be rewritten as

$$E(X|L) = E(X) - \sum_{\substack{i \\ L \in C_i}} 2^{-n_i} + \sum_{\substack{i \\ \neg L \in C_i}} 2^{-n_i}, \tag{3}$$

we have the following branching rule.

2. First-Order Probability Rule

Branch to a literal L that maximizes

$$\sum_{\substack{i \\ L \in C_i}} 2^{-n_i} - \sum_{\substack{i \\ \neg L \in C_i}} 2^{-n_i} = J(L) - J(\neg L).$$

over all literals L.

Note that the J–W rule neglects the term $J(\neg L)$. Thus it fails to consider the fact that setting L to true not only removes some clauses from the subproblem, but it shortens some clauses that remain in the subproblem (those containing $\neg L$). Thus the J–W rule does not even maximize a first-order approximation of satisfaction probability, but a truncation of the first-order formula.

If the Satisfaction Hypothesis is correct, one might initially expect the full first-order rule to be superior to the J–W rule, since it provides a more accurate estimate of the probability of satisfaction. Table III shows, however, that the full first-order rule is worse than the truncated rule in 9 of 11 problem classes. The difference is statistically different in 6 classes (0, 1, 2, 4, 6, 7) and often substantial. The full rule is never significantly better than J–W.

The poor performance of the full first-order rule might be traced, however, to a property peculiar to this rule rather than to a weakness in the Satisfaction Hypothesis. Namely, if L maximizes $E(X|L) = J(L) - J(\neg L)$, then $\neg L$ *minimizes* it. Thus if the algorithm branches to the best literal L and is obliged to backtrack, it must branch to the *worst* literal $\neg L$.

If one anticipates the possibility of backtracking and considers the sum of the first-order probability estimates for both L and $\neg L$, one obtains a constant:

$$\begin{aligned} E(X|L) + E(\neg X|L) &= [E(X) - J(L) + J(\neg L)] \\ &\quad + [E(X) - J(\neg L) + J(L)] = 2E(X). \end{aligned}$$

So in a first-order approximation, the advantage of branching to L always exactly complements the advantage of branching to $\neg L$.

The J–W rule, one might argue, avoids this peculiarity while still maximizing an approximation, however crude, of satisfaction probability. The J–W rule also shares a desirable property with the first-order rule.

LEMMA 1. *The first-order and J–W rule branch to a literal L for which*

$$E(X|L) \leqslant E(X).$$

Thus both rules select a literal that does not increase the expected number of falsified clauses.

Proof. From (3) it suffices to show that $J(L) \geqslant J(\neg L)$. This follows from the way L is chosen in the J–W rule. It also follows from the first-order rule, which ensures that $J(L) - J(\neg L) \geqslant J(\neg L) - J(L)$ and therefore $J(L) \geqslant J(\neg L)$. □

It is more difficult, however, to reconcile another prediction of the Satisfaction Hypothesis with experience. A truncated version of a rule that *minimizes* the first-order probability of satisfaction should, on this hypothesis, result in very poor behavior. To minimize the first-order criterion is to maximize $J(\neg L) - J(L)$. A truncated version, which we might call a "reverse" J–W rule, would maximize $J(\neg L)$. One might expect this rule to be worse than random branching, because it picks the worst branch as measured by one term of the first-order criterion. In any case it should be substantially worse than J–W.

Table III shows that the performance of a reverse J–W rule (Rule 7) is actually about the same as that of the J–W rule in 8 problem classes, significantly better in one, and worse in two; it is significantly worse in only one (class 4). Furthermore, the reverse J–W rule is *better* than random branching in 9 of 11 classes, and significantly better in five (0, 1, 2, 6, 7).

4. A Second-Order Branching Rule

A further test of the Satisfaction Hypothesis is to branch so as to maximize a more accurate estimate of satisfaction probability than provided by either the J–W or first-order rule. The hypothesis implies that such a rule should be superior to J–W.

By estimating $\Pr(X = 0)$ to be $1 - E(X)$, the first-order rule assumes that $X_1, \ldots, X_m$ are mutually exclusive events. One way to attempt to correct this oversimplification is to note that $1 - E(X)$ contains the first two terms of an inclusion-exclusion series:

$$\Pr(X = 0) = 1 - E(X) + \sum_{ij} E(X_i X_j) + \cdots + (-1)^m E(X_1 X_2 \cdots X_m). \tag{4}$$

If the third term is used as well, one obtains a branching rule that accounts for two-way interactions among the X_i's. The resulting approximation, however, is no longer a lower bound on $\Pr(X = 0)$ as in (1). Also the approximation is often poor because the second order term tends to dominate, due to the large number of clauses relative to the size of any $E(X_i X_j)$.

We will therefore take the opposite approach of assuming independence of $X_1, \ldots, X_m$:

$$\Pr(X = 0) = \prod_{i=1}^{m} (1 - E(X_i)),$$

and again making a partial correction by taking account of pairwise dependencies. Our vehicle for doing this is a straightforward generalization of the Lovász local lemma [8, 24].

To state the generalized lemma, let $\{A_1, \ldots, A_m\}$ be a collection of events. A *dependency graph* is a graph defined over a set of vertices $\{1, \ldots, m\}$, in which vertices i and j are connected by an edge if and only if A_i and A_j are dependent. Ordinarily the graph is taken to be a directed graph, but in our application it may be assumed to be undirected.

LEMMA 2. *Let $\{A_1, \ldots, A_m\}$ be a set of events, with respective probabilities $p_1, \ldots, p_m$, that define an m-vertex dependency graph G such that*

$$\sum_{\{i,j\} \in E} p_j < \frac{1}{4}, \quad i = 1, \ldots, m,$$

where E is the edge set of G. (The edges are written $\{i, j\}$ because G is undirected.) Then

$$\Pr(\overline{A}_1 \cap \cdots \cap \overline{A}_m) > \prod_{i=1}^{m} (1 - 2p_i) \geqslant 0,$$

where the last inequality holds if $p_i \leqslant \frac{1}{2}$ for $i = 1, \ldots, m$.

Proof. The proof is similar to that of the symmetric version of the local lemma. We first show the following.

$$\Pr(A_i | \cap_{j \in T} \overline{A}_j) \leqslant 2p_i, \quad \text{for all } T \subseteq \{1, \ldots, m\}. \tag{5}$$

As in the original proof, we prove the claim by induction on the size of T. When T is empty, there is nothing to prove. For the inductive step, assume by a suitable relabelling that T is the first s events. Then,

$$\Pr(A_{s+1} | \overline{A}_1 \cap \cdots \cap \overline{A}_n) = \frac{\Pr(A_{s+1} \cap \overline{A}_1 \cap \cdots \cap \overline{A}_d)}{\Pr(\overline{A}_1 \cap \cdots \cap \overline{A}_d | \overline{A}_{d+1} \cap \cdots \cap \overline{A}_s)}, \tag{6}$$

where the first d events of T may depend on A_{s+1}. Now,

$$\begin{aligned} \Pr(\overline{A}_1 \cap \cdots \cap \overline{A}_d | \overline{A}_{d+1} \cap \cdots \cap \overline{A}_s) &\geqslant 1 - \sum_{i=1}^{d} \Pr(A_i | \overline{A}_{d+1} \cap \cdots \cap \overline{A}_s) \\ &\geqslant 1 - 2 \sum_{i=1}^{d} p_i, \end{aligned}$$

using the induction hypothesis. Applying this and the fact that the numerator of (6) is bounded above by p_{s+1}, we obtain from (6) that

$$\Pr(A_{s+1} | \overline{A}_1 \cap \cdots \cap \overline{A}_n) < 2p_{s+1},$$

which proves (5).

To complete the proof of the lemma, observe that

$$\begin{aligned}\Pr(\overline{A}_1 \cap \cdots \cap \overline{A}_n) &= \Pi_{i=1}^{m} \Pr(\overline{A}_i | \overline{A}_1 \cap \cdots \cap \overline{A}_{i-1}) \\ &> \Pi_{i=1}^{m}(1-2p_i) \geqslant 0.\end{aligned}$$

The last expression is clearly nonnegative when $p_i \leqslant 1/2$ for all i. □

To apply this result to a set of clauses, we let event A_i be the falsification of a clause C_i, i.e., $X_i = 1$. Thus $p_i = E(X_i) = 2^{-n_i}$. Two events are dependent if the corresponding clauses have a common atom. The lemma now tells us that if

$$\sum_{\{i,j\} \in E} 2^{-n_j} < \frac{1}{4}, \quad i = 1, \ldots, m, \tag{7}$$

then,

$$\Pr(X = 0) > \prod_{i=1}^{m}(1 - 2E(X_i)) \geqslant 0.$$

The last inequality clearly holds because each $p_i = 2^{-n_i} \leqslant 1/2$, due to the fact that the clauses are nonempty.

It is reasonable to design a branching rule that minimizes the sum of the quantities on the left of (7) over all i, after a literal L is fixed. We therefore introduce a potential function

$$\phi(G) = \sum_{i} \sum_{\substack{j \\ \{i,j\} \in E}} p_j.$$

It is a simple matter to check that

$$\phi(G) = \sum_{\{i,j\} \in E} p_i + p_j.$$

Now consider the effect of setting literal L to true.

$$\begin{aligned}\phi(G|L) = &\sum_{\substack{\{i,j\} \in E \\ \neg L \in C_i, C_j \\ C_i \cap C_j \neq \{\neg L\}}} 2(p_i + p_j) + \sum_{\substack{i \\ \neg L \in C_i}} \sum_{\substack{j \\ \{i,j\} \in E \\ L, \neg L \notin C_j}} (2p_i + p_j) \\ &+ \sum_{\substack{\{i,j\} \in E \\ L, \neg L \notin C_i, C_j}} (p_i + p_j),\end{aligned}$$

which, by suitable regrouping, may be written as

$$\phi(G|L) = \phi(G) - \sum_{\substack{\{i,j\} \in E \\ L \in C_i \cup C_j}} (p_i + p_j) + \sum_{\substack{i \\ \neg L \in C_i}} \sum_{\substack{j \\ \{i,j\} \in E \\ L \notin C_j}} p_i$$

$$- \sum_{\substack{\{i,j\} \in E \\ C_i \cap C_j = \{\overline{L}\}}} (p_i + p_j). \tag{8}$$

Since the last term is negligible, we obtain the following branching rule.

3. Second-Order Branching Rule

Branch to a literal L that maximizes

$$\sum_{\substack{\{i,j\} \in E \\ L \in C_i \cup C_j}} (2^{n_i} + 2^{n_j}) - \sum_{\substack{i \\ \neg L \in C_i}} \sum_{\substack{j \\ \{i,j\} \in E \\ L \notin C_j}} 2^{n_i}$$

over all literals L.

It is not too difficult to see that for the literal L chosen by the rule, the first term in (8) dominates the second term. This leads to an improvement property similar to the one we obtained for the J–W and first-order rules.

LEMMA 3. *The second-order branching rule selects an L such that*

$$\phi(G|L) \leqslant \phi(G).$$

Computational testing of the second-order branching rule is complicated by the substantial burden of evaluating its criterion function. Comparison of Tables III and V reveals that the second-order rule requires from 10 to 1000 times more computation per node (or even more) than other rules. This is because its complexity is $O(mN)$ rather than $O(N)$, where m is the number of clauses and N the number of literal occurrences.

Clearly the second-order rule is unsuitable for practical use because its total consumption of computer time is much greater than that of J–W. But it is unfair to conclude on this basis that the second-order rule is a less intelligent branching rule than J–W, and thereby to refute the satisfaction model. A fairer test of the model is to compare the number of nodes generated under either branching rule.

Unfortunately a comparison of node count is difficult, because the second-order rule consumes the entire allotment of two hours in a large fraction of cases.

This means that its search is often cut off long before completion, whereas the J–W search is only rarely cut off. Thus Table V seriously underestimates the number of nodes generated by the second-order rule.

Five of the problem classes, however, are easy enough so that the second-order algorithm is never cut off (classes 2, 4, 6, 8, 10). One can therefore obtain a fairer comparison of the node counts in these classes. This does not completely remove the bias against J–W, because it selects classes in which the second-order rule is known to be faster than in other classes. Nonetheless J–W generates about the same or fewer nodes in 4 of the 5 classes. The superiority of the J–W rule is statistically significant only in class 6, but the evidence clearly fails to confirm the implication of the Satisfaction Hypothesis that a more accurate estimate of probability should result in a better rule.

5. A Simplification Model

Having found that the computational experience refutes or fails to confirm the Satisfaction Hypothesis, we propose another line of explanation. It views the J–W rule as choosing the branch that most simplifies the problem after unit resolution. Simplification will be more precisely defined in the model to follow, but basically a simpler problem is one with fewer and shorter clauses.

Simplification Hypothesis: Other things being equal, a branching rule works better when it creates simpler subproblems.

The basic motivation for this hypothesis is that a branching rule that produces simpler subproblems is more likely to resolve the satisfiability of subproblems without a great deal of branching.

This motivation becomes more compelling when one reasons as follows. Let a *single branch* node be a node only one of whose children is visited, and a *double branch* node at which both children are explored. Clearly, if there are n atoms, at most $n - 1$ nodes visited will be single branch nodes. Since far more than $n - 1$ nodes are visited in most searches, even when clever branching rules are used, the great majority of nonleaf nodes are double branch nodes in most problems. In other words, the subtrees rooted at most visited nodes contain no solution.

A branching rule that maximizes the probability of satisfaction may cause the algorithm to backtrack from fewer nodes before finding a solution (assuming the problem is satisfiable to begin with). But since the subtrees rooted at most nodes will contain no solution in any case, it makes sense to branch in such a way that these subtrees are as small as possible. This leads to the Simplification Hypothesis.

A branching rule simplifies a problem when it allows unit resolution to eliminate more literals and clauses. We therefore propose a probabilistic model that provides a partial analysis of unit resolution.

When a literal L is fixed to true, two-literal clauses containing $\neg L$ are reduced to unit clauses, allowing elimination of several clauses. So it is reasonable to choose an L for which $\neg L$ occurs in the most two-literal clauses. But the resulting unit clauses, when fixed to true, create still more unit clauses. Therefore it is more reasonable to fix a literal that will result in the creation of a maximum number of unit clauses at some point in the unit resolution algorithm.

Let U_i be a random variable that takes the value 1 if C_i becomes a unit clause at some point during unit resolution after L is fixed to true. $U = U_1 + \cdots + U_m$ is the total number of unit clauses generated. We wish to choose L so that the this number is maximized. We will analyze unit resolution as a random process for which we can calculate the expected number $E(U|L)$ of unit clauses created if L is fixed to true. Since $E(U|L) = \sum_{i=1}^{m} E(U_i|L)$, it suffices to compute $E(U_i|L) = \Pr(U_i = 1|L)$ for an arbitrary clause C_i.

The random process is a Markov chain. Let the state be the number of literals remaining in C_i. A state transition occurs each time the unit resolution algorithm fixes the value of a literal. If we assume that the fixed literal is randomly selected from $\{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$ and that C_i contains $k > 1$ literals in a given step, the transition probabilities are:

$$\Pr(C_i \text{ eliminated}) = k/2n,$$
$$\Pr(C_i \text{ reduced to } k-1 \text{ literals}) = k/2n,$$
$$\Pr(C_i \text{ unchanged}) = 1 - k/n.$$

We suppose that the process terminates when C_k is eliminated or reduced to a unit clause.

In reality, the fixed literal is not randomly chosen, and the process is prematurely terminated when a contradiction is found or no more unit clauses remain. But we will determine empirically whether the model correctly predicts computational performance despite its simplifying assumptions.

The transition probability matrix P for this Markov chain has the following form.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 1 & 0 & 0 & 0 & \\ 2/2n & 2/2n & 1-2/n & 0 & 0 & \\ 3/2n & 0 & 3/2n & 1-3/n & 0 & \\ 4/2n & 0 & 0 & 4/2n & 1-4/n & \\ \vdots & & & & & \end{bmatrix}.$$

If $\pi = (\pi_1, \ldots, \pi_n)$ is the vector of probabilities π_k that C_i contains k literals at a given stage, then πP is the vector of probabilities after one more transition.

Lengths 0 and 1 are the two absorbing states. When C_i eventually leaves its current state, it is eliminated (length 0) with probability $1/2$ and shortened by one with probability $1/2$. Obviously, if C_i begins with k literals, it eventually

reaches state 1 (becomes a unit clause) with probability $\Pr(U_i|L) = 2^{-(k-1)}$. Thus

$$\begin{aligned} E(U|L) &= \sum_{\substack{i \\ \neg L \in S}} 2^{-(n_i-2)} + \sum_{\substack{i \\ L, \neg L \notin S}} 2^{-(n_i-1)} \\ &= E(U) - \sum_{\substack{i \\ L \in S}} 2^{-(n_i-1)} + \sum_{\substack{i \\ \neg L \in S}} 2^{-(n_i-1)} \\ &= E(U) - 2J(L) + 2J(\neg L). \end{aligned}$$

Since the factor of two can be ignored, we obtain precisely the opposite of the first order probability rule stated earlier! The best literal for one rule is the worst for the other.

The simplification criterion also shares the property that when it branches to the best literal L and backtracks, it must branch to the worst literal $\neg L$. This property is particularly damaging here, because the simplification criterion branches on a literal precisely because backtracking is likely.

In fact it seems desirable to evaluate both L and $\neg L$ simultaneously, since branching to one is likely to be followed by a branch to the other. That is, one should choose a *variable* x_j that maximizes $E(U|x_j) + E(U|\neg x_j)$. But as in the case of the first order probability criterion, $E(U|x_j) + E(U|\neg x_j)$ is a constant, $2E(U)$, for all L.

One option is to use the strategy used earlier in an attempt to justify the J–W rule, namely delete the negative term in $E(U|L)$.

4. Reverse Jeroslow–Wang Rule

Branch to a literal L that maximizes

$$J(\neg L)$$

over all literals L.

We have already found that, contrary to the Satisfaction Hypothesis, this is an effective branching rule, and now we are beginning to understand why. In fact, if all nonleaf nodes were double branch nodes (and most are), maximizing $J(L)$ (the J–W rule) and maximizing $J(\neg L)$ have the same effect. At single branch nodes, one should take the branch most likely to be satisfiable, which is better predicted by $J(L)$ than $J(\neg L)$. On balance, then, one should generally obtain slightly better performance by maximizing $J(L)$ than maximizing $J(\neg L)$. As noted earlier, computation testing found that the former is about the same as the latter in 8 of 11 problem classes, better in two, and significantly worse in one. So observation is at least consistent with the prediction.

Most importantly, the Simplification Hypothesis predicts the good performance from the J–W rule that several investigators have observed. That the Satisfaction Hypothesis inspired a good rule is a stroke of luck.

In the meantime, we should be able to improve both the J–W and Simplification rules by considering both L and $\neg L$; that is, by branching on the variable x_j that maximizes $J(x_j) + J(\neg x_j)$. The first branch should be to the literal L (x_j or $\neg x_j$) for which the satisfaction probability is higher. Measuring the satisfaction probability by the First-Order Probability rule, we branch to L if $J(L) - J(\neg L) \geqslant J(\neg L) - J(L)$; i.e., if $J(L) \geqslant J(\neg L)$. This yields

5. Two-Sided Jeroslow–Wang Rule

Branch on a variable x_j that maximizes

$$J(x_j) + J(\neg x_j)$$

over all variables in S. Branch first to x_j if

$$J(x_j) \geqslant J(\neg x_j),$$

and otherwise first to $\neg x_j$.

Table III reveals that the 2-sided J–W rule (Rule 8) is better than J–W in every problem class but one. The difference is statistically significant in classes 0, 1, 2, 6, 7.

6. Shortest Positive Clause Branching

An interesting variation of the Davis–Putnam–Loveland algorithm branches on positive clauses rather than on variables [11]. Rather than setting some x_j to true and then to false in order to generate successor nodes, it picks a shortest positive clause, such such as $x_1 \vee x_2 \vee x_3$. (A positive clause is one with all positive literals.) It then creates a successor node for each literal in the clause, three in this case. For the first node, x_1 is set to true. For the second, x_2 is set to true and x_1 to false (to avoid regenerating solutions in which x_1 is true). For the third, x_3 is set to true and both x_1 and x_2 to false. A statement of the algorithm follows.

The algorithm is actually very similar to a DPL algorithm with an approximation of the J–W branching rule. By branching to a literal in a shortest positive clause, it branches to a literal L for which $J(L)$ is likely to be fairly large. The three successor nodes generated by a clause $x_1 \vee x_2 \vee x_3$ would in effect be generated by DPL if one branches on x_1, x_2 and x_3 in that order. The second branch would occur upon branching to $\neg x_1$ and then to x_2, and the third upon branching to $\neg x_1$, then to $\neg x_2$ and then to x_3.

6. Shortest Positive Clause Branching

Procedure **Branch**(S,k)
  Perform **Monotone Variable Fixing** on S.
  Perform **Unit Resolution** on S.
  If a contradiction is found, return.
  If S is empty,
    declare problem to be *satisfiable* and stop.
  Branch:
    If S contains no positive clauses,
      declare problem to be *satisfiable* and stop.
    Else pick a positive clause $x_{j_1} \vee \cdots \vee x_{j_p}$ in S.
    For $i = 1, \ldots, p$ do:
      Perform **Branch**($S \cup \{\neg x_{j_1}, \ldots, \neg x_{j_{i-1}}, x_{j_i}\}, k+1$).
End.

The positive clause branching rule has one feature lacked by branching rules hitherto considered. By branching only on positive clauses, it exploits the fact that there is no need to branch on any variable that never occurs in a positive clause. This is because if only such variables remain, the remaining clauses can always be satisfied by setting all variables to false.

The computational results provide little motivation to use clause branching rather than DPL. The positive clause branching rule is better than J–W in 4 problem classes but never significantly better. It is worse than J–W in 7 problem classes and significantly worse in classes 2 and 6. It is worse than the 2-sided J–W rule in 7 problem classes and significantly worse in classes 0, 1, 2, 4, 6 (while significantly better in no class).

The strategy of branching only on variables that occur in some positive clause, however, could prove useful in a DPL algorithm. We modified the J–W rule and the 2-sided J–W rule to incorporate it.

7. Positive Jeroslow–Wang Rule

Branch to a literal L that maximizes $J(L)$ over all variables in S that occur in some clause that contain all positive literals.

The two-sided rule (Rule 8) is better than the 2-sided J–W rule in every problem class, although the difference is statistically significant only in class 4.

8. Positive Two-Sided Jeroslow–Wang Rule

Branch on a variable x_j that maximizes

$$J(x_j) + J(\neg x_j)$$

over all variables in S that occur in some clause that contain all positive literals. Branch first to x_j if

$$J(x_j) \geqslant J(\neg x_j),$$

and otherwise first to $\neg x_j$. If there is no such x_j, the remaining clauses can be satisfied by setting all remaining variables to false.

7. Conclusions

We found that the Jeroslow–Wang rule is substantially better than random branching, but the original motivation for it (the Satisfaction Hypothesis) does not explain its success, for several reasons.

- It does not explain the rule's behavior on unsatisfiable problems.
- The J–W rule actually maximizes a truncation $J(L)$ of a criterion $J(L) - J(\neg L)$ based on a first-order estimate of satisfaction probability. If this predicts good J–W performance, then a truncation $J(\neg L)$ of the reverse criterion $J(\neg L) - J(L)$ should result in a very poor rule. But the reverse rule is only slightly worse than J–W and much better than random branching.
- A better, second-order estimate of satisfaction probability should, on the Satisfaction Hypothesis, result in better performance. But performance is no better and sometimes worse.

We found that performance is better explained by a Simplification Hypothesis that focuses on a rule's propensity to choose branches that simplify the problem. The reasons are as follows.

- An estimate of simplification based on a Markov chain analysis leads to the criterion $J(\neg L) - J(L)$, whose truncation $J(\neg L)$ was found to perform nearly as well as J–W, contrary to predictions of the Satisfaction Hypothesis.
- The Simplification Hypothesis, as interpreted by our Markov Chain model, implies that both J–W and reverse J–W should perform well, with some preference for the former. This accords with experience.
- The hypothesis also suggests that a 2-sided J–W rule should be superior to J–W, and it is.

We also found that clause branching is worse than the 2-sided J–W rule. But its strategy of branching only on variables that occur in positive clauses appears

to improve the 2-sided J–W rule when added to it, although the statistical analysis cannot confirm this with confidence. In any case, the resulting positive 2-sided Jeroslow–Wang rule appears to be the best examined here and is significantly better than the old Jeroslow–Wang rule.

Finally, our experience does not suggest that more accurate estimates of the branching criterion are worth the expense. The computational overhead of a second-order ($O(mN)$) estimate of satisfaction probability far outweighed any reduction in the size of the search tree relative to a first-order ($O(N)$) estimate.

References

1. Amini, M. M. and Racer, M.: A variable-depth-search heuristic for the generalized assignment problem, *Management Science*, to appear.
2. Böhm, H.: Report on a SAT Competition, Technical report No. 110, Universität Paderborn, Germany, 1992.
3. Cook, S. A.: The complexity of theorem-proving procedures, in *Proc. 3rd Annual ACM Symp. on the Theory of Computing*, 1971, pp. 151–158.
4. Crawford, J.: Problems contributed to DIMACS. For information contact Crawford at AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ, 07974-0636 USA, e-mail jc@research.att.com.
5. Davis, M. and Putnam, H.: A computing procedure for quantification theory, *J. ACM* **7** (1960), 201–215.
6. Dubois, O.: Problems contributed to DIMACS. For information contact Dubois at Laforia, CNRS-Université Paris 6, 4 place Jussieu, 75252 Paris cedex 05, France, e-mail dubois@laforia.ibp.fr.
7. Dubois, O., Andre, P., Boufkhad, Y., and Carlier, J.: SAT versus UNSAT, manuscript, Laforia, CNRS-Université Paris 6, 4 place Jussieu, 75252 Paris cedex 05, France, 1993, e-mail dubois@laforia.ibp.fr.
8. Erdös, P. and Lovász, L.: Problems and results on 3-chromatic hypergraphs and some related questions, in *Infinite and Finite Sets*, North-Holland, Amsterdam, 1975.
9. Freeman, T. W.: Failed literals in the Davis–Putnam procedure for SAT, manuscript, Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 19104 USA, CA, 1993, freeman@gradient.cis.upenn.edu.
10. Gallo, G. and Pretolani, D.: A new algorithm for the propositional satisfiability problem, report TR-3/90, Dip. di Informatica, Universitá di Pisa, *Discrete Applied Mathematics*, to appear.
11. Gallo, G. and Urbani, G.: Algorithms for testing the satisfiability of propositional formulae, *J. Logic programming* **7** (1989), 45–61.
12. Golden, B. L. and Stewart, W. R.: Empirical analysis of heuristics, in Lawler, Lenstra, Rinnooy Kan, and Schmoys (eds), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York, 1985, pp. 207–249.
13. Harche, F., Hooker, J. N., and Thompson, G.: A computational study of sastifiability algorithms for propositional logic, *ORSA J. Computing* **6** (1994), 423–435. For more information contact J. Hooker, email jh38@andrew.cmu.edu.
14. Hooker, J. N.: Needed: An empirical science of algorithms, *Operations Research* **42** (1994), 201–212.
15. Hooker, J. N. and Fedjki, C.: Branch and cut solution of inference problems in propositional logic, *Annals of Mathematics and AI* **1** (1990), 123–139.
16. Iwama, K., Albeta, H., and Miyano, E.: Random generation of satisfiable and unsatisfiable CNF predicates, in *Proc. of 12th IFIP World Computer Congress*, 1992, pp. 322–328. For further information contact Eiji Miyano, Dept. of Computer Science and Communication Engineering, Kyushu University, Fukuoka 812, Japan, e-mail miyano@csce.kyushu-u.ac.jp.
17. Jeroslow, R. and Wang, J.: Solving propositional satisfiability problems, *Annals of Mathematics and AI* **1** (1990), 167–187.

18. Kamath, A., Karmarkar, N., Ramakrishnan, K., and Resende, M.: A continuous approach to inductive inference, *Mathematical Programming* **57** (1992), 215–238. For further information contact Mauricio Resende, AT&T Bell Laboratories, Murray Hill, NJ 07974 USA, e-mail mgcr@research.att.com.
19. Lin, B. W. and Rardin, R. L.: Controlled experimental design for statistical comparison of integer programming algorithms, *Management Science* **25** (1980), 1258–1271.
20. Loveland, D. W.: *Automated Theorem Proving: A Logical Basis*, North-Holland, Amsterdam, 1978.
21. Mitterreiter, I. and Radermacher, F. J.: Experiments on the running time behavior of some algorithms solving propositional logic problems, manuscript, Forschungsinstitut für anwendungsorientierte Wissensverarbeitung, Ulm, Germany, 1991.
22. Petersen, R. G.: *Design and Analysis of Experiments*, Marcel Dekker, New York, 1985.
23. Pretolani, D.: Efficiency and stability of hypergraph SAT algorithms, manuscript, Dip. di Informatica, Univ. di Pisa, Corso Itali 40, 56125 Pisa, Italy. For information on problems contact e-mail pretola@di.unipi.it.
24. Spencer, J.: *Ten Lectures on the Probabilistic Method*, Regional Conference Series in Applied Mathematics **52**, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.
25. Van Gelder, A. and Tsuji, Y. K.: Satisfiability testing with more reasoning and less guessing, manuscript, University of California, Santa Cruz, CA, USA, 1994. For information on problems contact e-mail avg@cs.ucsc.edu or tsuji@cs.ucsc.edu.
26. Wilson, J. M.: Compact normal forms in propositional logic and integer programming formulations, *Computers and Operations Research* **90** (1990), 309–314.