TECHNICAL REPORT NO. 311

Average Time Analysis Of Clause Order Backtracking

by

Khaled Bugrara, Northeastern University
and
Paul W. Purdom, Jr., Indiana University

Revised: January 1992

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

Average Time Analysis of Clause Order Backtracking

Khaled Bugrara, Northeastern University and Paul Walton Purdom, Jr., Indiana University

Abstract: Backtracking algorithms solve problems by selecting a variable and assigning each possible value to the variable. The resulting subproblems are simplified and solved recursively. Simple backtracking selects variables in a fixed order. Clause order backtracking selects variables from the first nontrivial clause that has not yet been satisfied. Formulas are given for the average time used by clause order backtracking when solving random CNF satisfiability problems, where the problem sets have v variables, t clauses, and a probability p of a literal being in a clause. The average time for clause order backtracking is always less than that for simple backtracking. It leads to polynomial time under many conditions where simple backtracking uses exponential average time. Cases where clause order backtracking uses average time less than v^n (in the limit of v going to infinity) include $p < 1/(2v)e^{[(n-1)\ln v - \ln t]/t}$ and $p > \sqrt{[\ln t + (\ln v)/2]/v}$. (The second result needs a slight increase in the coefficient of $\ln t$ when t increases faster than $v^{\ln v}$.)

†This work was supported by Northeastern University, Indiana University, and FAW (Research Institute for Applied Knowledge Processing at the University of Ulm).

1 Constraint satisfaction and backtracking

Many interesting problems require determining whether a set of constraints on variables with discrete values can be satisfied. Let $R_1(x_1, \ldots, x_v)$, ..., $R_t(x_1, \ldots, x_v)$ be a set of relations and x_1, ..., x_v be a set of variables, where each variable has a finite set of possible values. A constraint satisfaction problem consists of determining whether the variables can be set in a way that makes all of the relations *true*. Such problems can be quite difficult even when each relation is simple. For example, if each relation is a clause, then the constraint satisfaction problem becomes the classical Conjunctive Normal Form (CNF) satisfiability problem. Many special forms of the constraint satisfaction problem are NP-complete [3, 6, 12].

Searching is one common way to solve constraint satisfaction problems. The basic idea of searching is to choose a variable and generate subproblems by assigning each possible value to the variable. In each subproblem the relations are simplified by plugging in the value of the selected variable. If any subproblem has a solution, then the original problem has a solution. Otherwise, the original problem has no solution. The subproblems are solved by applying the technique recursively. *Simple search algorithms* stop the recursion when all variables have values. They use exponential time when used to find all solutions.

If any relation of a constraint satisfaction problem is always *false*, then the problem has no solution. *Backtracking* improves over plain search by immediately reporting no solution for problems with a false relation. Often this short cut saves a huge amount of time. Backtracking can take either exponential or polynomial average time, depending on the set of problems being solved [1, 11].

Simple backtracking has a fixed order for selecting variables. Sometimes it wastes time by assigning values to variables that do not appear in the problem. Even when the original problem uses all the variables, some of the simplified subproblems may use only a few of them. For problems with a few short clauses simple backtracking *frequently* assigns values to absent variables.

Clause order backtracking reports no solution if the problem has a relation that is always *false*. Otherwise, it selects the first relation that is not always *true*. For the first variable that affects the value of this relation, each possible value is plugged in, the predicate simplified, and the resulting subproblem is solved by recursive application of the algorithm. Each solution of the subproblem (along with the the partial assignments of values that lead to the subproblem) gives a solution to the original problem. If the original problem has no nontrivial relations, then every assignment of values to the remaining variables results in a solution.

The following is a precise statement of the version of the algorithm that we analyze. This version is tailored for CNF satisfiability problems. The algorithm finds every solution to the given CNF problem, but it reports the solutions in a compressed form. A clause is always *false* if and only if it is empty (contains no literals). A clause is a *tautology* if and only if it contains a variable and its negation. A tautological clause always evaluates to *true*.

Clause Order Backtracking Algorithm for CNF problems.

1. If the CNF problem has an empty clause, return with an empty set of solutions, and charge one time unit.
2. If the first remaining clause of the CNF problem is a tautology, then remove it from the problem. Repeat this part of the step as long as it applies. If there are no clauses, then return with the current assignment of values to the variables as a solution (each assignment of values for the remaining unset variables results in a solution) and charge one time unit.
3. Let k be the number of unset variables in the first remaining clause. (Step 1 ensures that $k \geq 1$, and Step 2 ensures that each variable occurs in at most one literal of the clause.) For j starting at 1 and increasing to at most k, generate the j^{th} subproblem by setting the first $j-1$ variables of the clause so that their literals are *false* and setting the j^{th} variable so that its literal is *true*. Use the assignment of values to simplify the CNF problem (remove each false literal from its clause and remove from the problem each clause with a true literal). Apply the algorithm recursively to solve the simplified problem. If setting the first $j-1$ literals of the first remaining clause to *false* results in some clause being empty, then stop generating subproblems. The set of solutions for the original problem is the union of the set of solutions for the subproblems. If the loop stops with $j = h$, then charge $h+1$ time units.

The cost in time units has been defined to be the same as the number of nodes in the backtrack tree generated by the algorithm. The actual running time of the algorithm depends on how cleverly it is

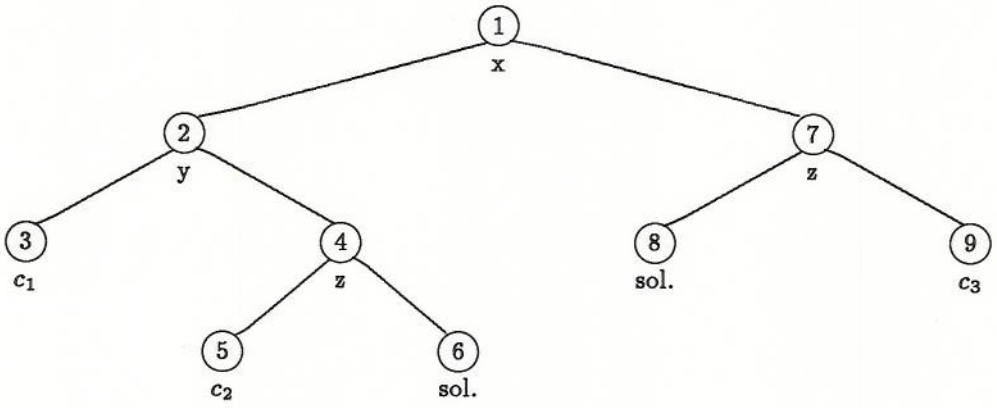

Fig. 1. The backtrack tree for the predicate consisting of the three clauses $c_1 = x \vee y$, $c_2 = x \vee z$, $c_3 = \bar{x} \vee \bar{z}$.

implemented, but a good implementation will result in a time that is proportional to the number of nodes multiplied by a factor that is between 1 and tv, where v is the number of variables and t is the number of clauses.

The backtrack tree includes nodes for determining that the first remaining clause is empty. The computation associated with these nodes can be done quickly. In the analysis we briefly consider the effect of not charging for these nodes (giving an upper limit of k time units for Step 3). A cost of up to $k+1$ units at Step 3 is natural when comparing the algorithm with simple backtracking, but a limit of k units is more natural when comparing the algorithm with unit clause backtracking. (Unit clause backtracking selects variables from clauses of length one when possible.)

Fig. 1 illustrates the counting of nodes. Each internal node is labelled with the variable that is set at the corresponding point in the computation, the left branch corresponds to setting the variable to *false*, and the right branch corresponds to setting it to *true*. The first clause, $c_1 = x \vee y$, contributes nodes 1, 2 and 3. The second clause, $c_2 = x \vee z$, contributes nodes 4 and 5. Node 6 represents the solution $\{x = false,\ y = true,\ z = true\}$. When x is *true*, both the first and second clauses are satisfied, but the third clause, $c_3 = \bar{x} \vee \bar{z}$ contributes nodes 7 and 9. Node 8 represents the set of solutions with $\{x = true,\ z = false\}$. All values of y are permitted at node 8. If the predicate had a fourth clause, $c_4 = x \vee \bar{y}$, then the predicate would be *false* at node 4 and clause c_2 would contribute only node 4. Nodes 5 and 6 would not be in the tree. With the three clause problem of Fig. 1, nodes 3, 5, and 9 would not be charged for using the alternate charging method of the previous paragraph.

2 Probability model

The *random clause model* generates random CNF satisfiability problems using a set of v variables. A random clause is formed by independently selecting each of the $2v$ literals with probability p. A random predicate is formed by independently forming t random clauses.

In this model some variables may not occur in a particular problem. Some clauses may be empty. Some clauses may be tautologies. Some clauses may be duplicates of others. Each of these features may be considered to be a defect in the model, since problems initially given to a satisfiability algorithm are not likely to have them. However, these defects are not as important as it may first appear because most of the features do occur in the subproblems that are produced by searching algorithms. The model has the advantage that the distribution of clauses does not change much when setting a variable. This leads to a relatively easy analysis. As a result more satisfiability algorithms have been analyzed with this model than with any other. The problems generated by this model can be either easy or hard (for the algorithms that have been analyzed so far) depending on how the parameters are set [2, 4, 5, 7, 8, 9, 10, 11]. The model is useful for comparing strengths and weaknesses of various satisfiability algorithms.

3 Discussion of results

This section discusses the the average time performance of clause order backtracking and compares it

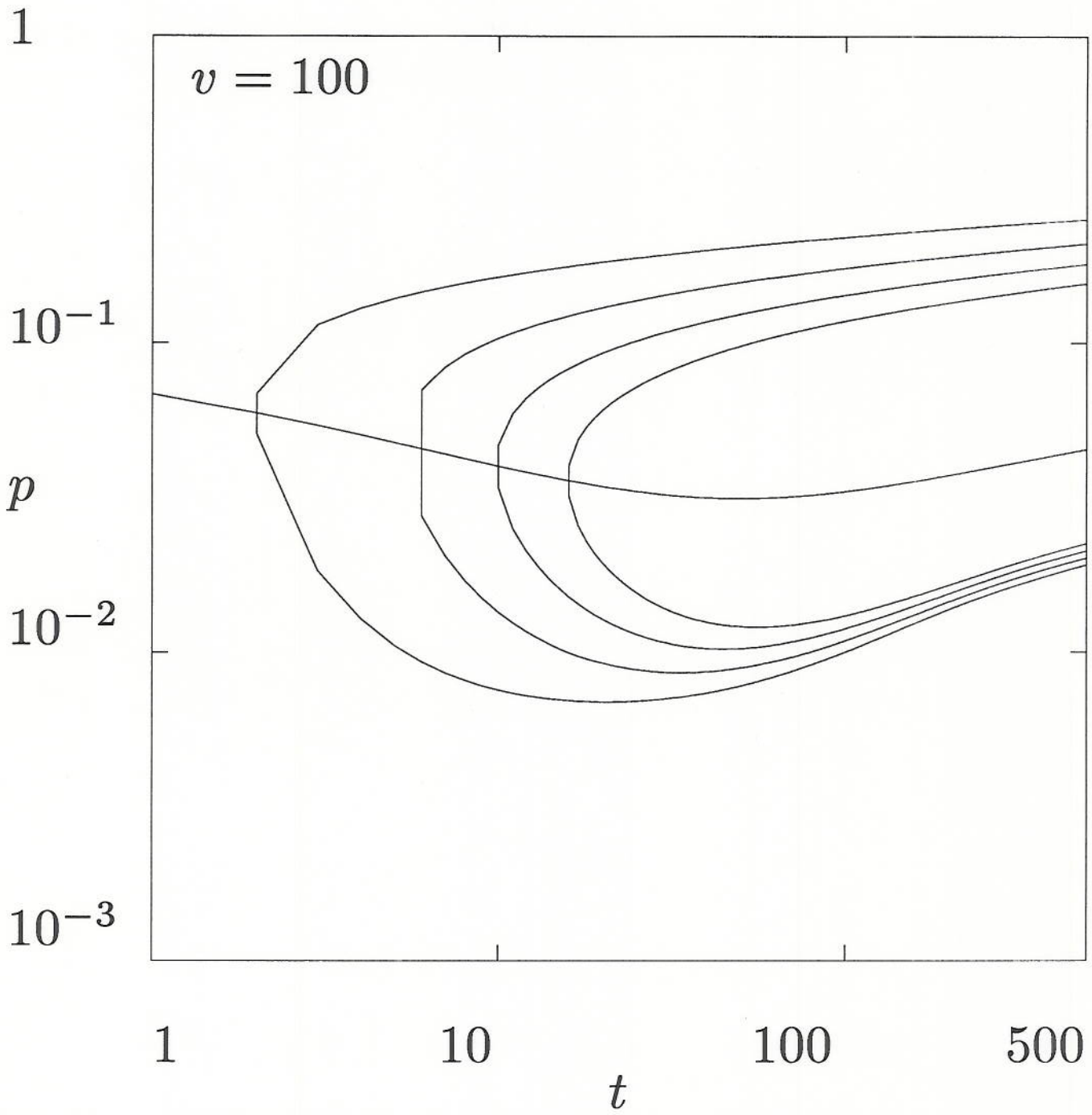

Fig. 2. The outer contour shows for each t the value of p that results in an average of 100 nodes. Proceeding inward, contours for an average of 100^2, 100^3, and 100^4 are also shown. The line from the left side to the right side shows for each t which value of p results in the largest average number of nodes. Consecutive points on a contour are connected by a straight line.

with that of other satisfiability algorithms. The following sections contain the detailed derivation of the results.

A contour plot of the performance for problems with one hundred variables is given in Fig. 2. A similar plot for fifty variables is given in [9]. The line that runs from the left to right side shows for each t, the value of p that leads to the largest average number of nodes. The remaining contours denote constant running time

and are as follow. The outer contour in Fig. 2 shows the conditions where the average number of nodes per problem is 100. Each contour inward shows the conditions where the average number of nodes per problem is a factor of 100 larger. Thus, the contours show the conditions where the average number of nodes per problem is v, v^2, v^3, and v^4. The actual average running time is bounded by a low degree polynomial times the average number of nodes, where the details of the polynomial depend on just how cleverly one programs the algorithm.

The average number of nodes for clause order backtracking is never more than the average number of nodes for the version of simple backtracking that reports every solution. Under many conditions, clause order backtracking is much faster, but when pv is small and t/v is large the improvement is not important. (This is the lower right region of Fig. 2.) The comparison with simple backtracking [11] implies that the average number of nodes for clause order backtracking is bounded by v^n for large v when

$$\frac{t}{v} > \frac{\ln 2 - n(\ln v)/v}{-\ln(1-e^{-pv})} \quad \text{and} \quad pv < \ln 2 \tag{1}$$

and also when

$$\frac{t}{v} > \frac{pt[2\ln 2 - n(\ln v)/v]}{(\ln 2)\ln(1+pt/\ln 2) + pt\ln[1+(\ln 2)/pt]} \quad \text{and} \quad pv > \ln 2 \quad \text{provided } pv/\ln v \text{ goes to zero.} \tag{2}$$

There are additional cases where clause order backtracking is fast. Under most of these conditions the average number of solutions per problem is exponential. The reason that clause order backtracking is able to run in polynomial time and report every solution is that solutions are reported in compressed form. When the algorithm reports a solution, it often does not assign values to all the variables. Any truth assignment to the remaining variables is a solution.

For small p we show that the average number of nodes is no more than v^n when

$$p \le \frac{1}{2v} e^{[(n-1)\ln v - \ln t - O(1)]/t}. \tag{3}$$

This limit approaches $1/(2v)$ when t grows faster than a constant times $\ln v$. This result corresponds to the fact that in Fig. 2 the contours all remain above $p = 1/(2v)$. The result in eq. (3) is better than the one in eq. (1) when

$$\frac{t}{v} \le 0.743 < \frac{\ln 2}{-\ln(1-e^{-1/2})}. \tag{4}$$

For large p we show that the average number of nodes is no more than v^n when

$$p \ge \sqrt{\frac{(1+\delta)\ln t + (\ln v)/2}{v}}, \tag{5}$$

for any $\delta > 0$ and large v. When $\ln[(\ln t)/(\ln v)] < \ln v$ (i.e., when $t < v^{\ln v}$), δ can be replaced with zero. This result corresponds to the upper branch of the contours in Fig. 2. Iwama's counting algorithm [7] for satisfiability has an average number of nodes bounded by v^n when

$$p > \sqrt{\frac{\ln t - \ln n}{v}}. \tag{6}$$

This is quite similar to eq. (5), but somewhat better. Exact calculations for particular cases [9] confirm the suggestion of these upper bound calculations, i.e., Iwama's algorithm is faster for large p. (It is slow for small p). Iwama's algorithm provides a count of the number of solutions, but does not give the solutions. When p is above the bound in eq. (5) clause order backtracking quickly provides a complete list of solutions (in compressed form).

Eqs. (3) and (5) show that for all p we have polynomial average time when

$$t < 2(n-1) - O([\ln\ln v]/\ln v). \tag{7}$$

Other algorithms are known to be faster for small t, but none of them report all solutions. For example, the upper bound for the pure literal rule algorithm [10] is better than the upper bound for clause order backtracking for small t. (Comparing upper bounds does not always show which algorithm is best, but the measurements in [9] suggest that the upper bounds are close to the true values.) Comparing eq. (3) with eq. (28) of [10] shows that for small p, the pure literal rule algorithm has a better upper bound for

$$t < 2(nv \ln v)^{1/2}. \tag{8}$$

Eq. (18) of [10] shows that the pure literal rule has a better upper bound for large p when

$$t < \frac{n-2}{\ln 2} \ln v \left[1 + O\left(\frac{1}{v}\right)\right]. \tag{9}$$

The fastest analyzed algorithm [5] for small t combines the unit clause rule, the pure literal rule, and resolution to eliminate variables that occur no more than two times. The use of resolution results in polynomial average time when t is below $v^{2/3}$ so long as p is not too large (when p is too large for this result, Iwama's algorithm is fast).

Among the analyzed satisfiability algorithms that report all solutions, clause order backtracking is one of the fastest. Selecting short clauses before long clauses would clearly improve the speed but complicate the analysis. A version of clause order backtracking where unit clauses are selected before longer clauses has been partly analyzed [9], but it is significantly faster than simple clause order backtracking only when pv is small. Resolution based algorithms are much faster than clause order backtracking for small t and some values of p. For some applications the fact that they do not report all solutions is a disadvantage. Clause order backtracking runs in low degree polynomial average time for both large and small p. Only intermediate values of p lead to problems that are difficult for the algorithm.

4 Analysis

In the analysis, we associate clauses with nodes in the backtrack tree in a way that is equivalent to that described in Section 1 but different in detail. The backtrack tree always has a root. Each variable that is set when processing a clause results in two nodes, one for setting it to *true* and one for setting it to *false*. Thus, in Fig. 1, this way of counting associates node 1 with the root, nodes 2, 7, 3, and 4 with clause c_1, nodes 5 and 6 with clause c_2, and nodes 8 and 9 with clause c_3.

We now derive a sum that gives the average number of nodes generated by clause order backtracking. First, we consider the probability that the first clause attempts to contribute j nodes to the all false branch of the search tree (the branch where all variables are set to *false*). Then, we consider the probability that the first clause contributes zero nodes to the branch. (This happens when the first clause is a tautology.) Next, we consider the probability that the clauses following the first clause frustrate its attempt to contribute j nodes (by becoming *false* before j variables have been set). Finally, we use the fact that the probabilities are the same on all branches to obtain the sum for the average number of nodes.

The probability that a clause contains the first variable appearing positively but not negatively is $p(1-p)$. The probability that the first variable appears either positive or negatively, but not both is $2p(1-p)$. The probability that a clause contains k literals and the clause is not a tautology is

$$\binom{v}{k} 2^k (p - p^2)^k (1-p)^{2v-2k} = \binom{v}{k} 2^k p^k (1-p)^{2v-k}. \tag{10}$$

Suppose i variables are set to *false*. The probability that a random clause contains no true literals, contains k unset literals, and is not a tautology is

$$\binom{v-i}{k} 2^k p^k (1-p)^{2v-i-k}. \tag{11}$$

The probability that such a clause contains its first true literal when j of its variables have been set to *false* is 2^{-j} (provided $j \leq k$). Let $a_j(i)$ be the probability that a random clause is not a tautology, that it contains

no true literals initially, and that it first contains a true literal after setting the first j of its previously unset variables. Summing eq. (11) times 2^{-j} gives

$$a_j(i) = \sum_{k \geq j} \binom{v-i}{k} 2^{k-j} p^k (1-p)^{2v-i-k} \quad \text{for } j \geq 1. \tag{12}$$

This is the probability that the first clause attempts to contribute j nodes to the all false branch of the backtrack tree.

The probability that a clause contains no true literals and is not a tautology is

$$\sum_k \binom{v-i}{k} 2^k p^k (1-p)^{2v-i-k} = (1-p)^v (1+p)^{v-i}. \tag{13}$$

Let $a_0(i)$ be the probability that a random clause evaluates to *true* after i of its variables have been set to *false*. Any clause meets these conditions except those considered in eq. (13). Thus,

$$a_0(i) = 1 - (1-p)^v (1+p)^{v-i}. \tag{14}$$

This is the probability that a clause is skipped in Step 2 of the algorithm, and thus contributes zero nodes to the all false branch of the backtrack tree.

A clause evaluates to *false* if it contains only false literals. It can not contain true or unset literals. The probability that a clause contains no such literals in $(1-p)^{2v-i}$. Thus, the probability that a clause evaluates to *false* is

$$1 - (1-p)^{2v-i}. \tag{15}$$

Consider a particular predicate and a particular node in the associated backtrack tree. Let n be the number of clauses that contribute nodes to the path from the root to the selected node. (These are the clauses that are skipped in Step 2 and those that are used in Step 3 of the algorithm). Let j_m be the number of nodes that the m^{th} clause contributes to the path. Then $j_m \geq 0$ for $1 \leq m < n$, and $j_n \geq 1$. Let s_m be the number of nodes contributed before those of the m^{th} clause. That is,

$$s_m = \sum_{1 \leq k < m} j_k. \tag{16}$$

Consider a particular set of j_m's and the set of all predicates that generate backtrack trees with that set of j_m's. Assume that each set variable is assigned the value *false*. The m^{th} clause must contribute j_m nodes to the all false branch, with s_m variables having already been set. The remaining $t-n$ clauses (those that are not associated with any m) must not be *false* after $s_{n+1} - 1$ variables have been set. If all these conditions are met, then the backtrack tree has a node for both values of the last literal: one value results in the current clause containing a true literal while the other value results in each literal of the current clause being either *false* or *unset*. Let $B(n, j_1, \ldots, j_n)$ be the probability that the backtrack tree for a random predicate contains a node where the m^{th} clause contributes j_m (for $1 \leq m \leq n$) nodes to the path and where the last set variable makes the n^{th} clause *true*. Since each condition pertains to one clause, and each clause is independently selected,

$$B(n, j_1, \ldots, j_n) = [1 - (1-p)^{2v-s_{n+1}+1}]^{t-n} \prod_{1 \leq m \leq n} a_{j_m}(s_m). \tag{17}$$

The a_{j_1} factor in this formula gives the probably that the first clause of a random predicate contributes j_1 nodes, the a_{j_2} factor gives the probability that the second clause contributes j_2 nodes, etc. The $[1 - (1-p)^{2v-s_{n+1}+1}]^{t-n}$ factor gives the probability that the clauses that have not contributed nodes have not previously evaluated to *false*. If you consider node 5 in Fig. 1, it was reached by the first clause contributing two nodes (nodes 2 and 4), the second clause contributing one node (node 5). To reach node 5 it was also necessary for the remaining clauses (clause c_3 in this case) not to evaluate to *false* at node 4. If the

noncontributing clauses were not *false* at node 4, then they could not have been *false* at any of the earlier nodes on the path. The factor $a_2(0)$ gives the probability that the first random clause will contribute two nodes, and the $a_1(2)$ factor gives the probability that the second clause will contribute one node (even though two variables have already be set). The factor $[1-(1-p)^4]^1$ gives the probability that the one remaining clause does not evaluate to *false* even though two variables have been set at the parent of node 5.

Eq. (17) gives the probability that the tree has a node with the specified characteristics. In particular, the last variable has been set so that its literal in the selected clause evaluates to *true*. With the same probability the tree has a node where that variable is set so that the literal evaluates to *false*. Thus, the average number of nodes corresponding to a particular $j_1, \ldots, j_n$, with each variable being assigned any preselected value (such as *false*), is $2B(n, j_1, \ldots, j_n)$.

In the random clause model each branch of the backtrack tree has the same probability. There are 2^i branches that have i variables set. Also each backtrack tree has a root node. Thus, the expected number of nodes in the backtrack tree is

$$N(t) = 1 + \sum_{1\le n\le t} \sum_{j_1\ge 0} \cdots \sum_{j_{n-1}\ge 0} \sum_{j_n\ge 1} 2^{s_{n+1}+1} B(n, j_1, \ldots, j_n). \tag{18}$$

The initial 1 counts the root of the backtrack tree, and the remaining terms count the nodes on level s_n for $1 \le s_n \le v$. Using $b_j(i) = 2^j a_j(i)$ and eq. (17), eq. (18) can be written as

$$N(t) = 1 + 2 \sum_{1\le n\le t} \sum_{j_1\ge 0} \cdots \sum_{j_{n-1}\ge 0} \sum_{j_n\ge 1} [1-(1-p)^{2v-s_{n+1}+1}]^{t-n} \prod_{1\le m\le n} b_{j_m}(s_m). \tag{19}$$

4.1 Recurrence equation

The number of terms needed to evaluate $N(t)$ with eq. (19) increases rapidly with t and v. We now derive a recurrence equation that is quicker to evaluate. Define

$$N(t,i) = 1 + 2 \sum_{1\le n\le t} \sum_{j_1\ge 0} \cdots \sum_{j_{n-1}\ge 0} \sum_{j_n\ge 1} [1-(1-p)^{2v-i-s_{n+1}+1}]^{t-n} \prod_{1\le m\le n} b_{j_m}(i+s_m). \tag{20}$$

Note that $N(t) = N(t,0)$.

Giving separate consideration to the $n=1$ case in eq. (20) results in

$$\begin{aligned} N(t,i) = 1 &+ 2\sum_{j\ge 1} [1-(1-p)^{2v-i-j+1}]^{t-1} b_j(i) \\ &+ 2 \sum_{2\le n\le t} \sum_{j_1\ge 0} \cdots \sum_{j_{n-1}\ge 0} \sum_{j_n\ge 1} [1-(1-p)^{2v-i-s_{n+1}+1}]^{t-n} \prod_{1\le m\le n} b_{j_m}(i+s_m). \end{aligned} \tag{21}$$

Giving separate consideration to the j_1 sum, results in

$$\begin{aligned} N(t,i) = 1 &+ 2\sum_{j\ge 1} [1-(1-p)^{2v-i-j+1}]^{t-1} b_j(i) \\ &+ 2\sum_j b_j(i) \sum_{2\le n\le t} \sum_{j_2\ge 0} \cdots \sum_{j_{n-1}\ge 0} \sum_{j_n\ge 1} \\ &\qquad [1-(1-p)^{2v-i-j-s'_{n+1}+1}]^{t-n} \prod_{2\le m\le n} b_{j_m}(i+j+s'_m), \end{aligned} \tag{22}$$

where $s'_k = s_k - j$. Now replace n by $n+1$, m by $m+1$, j_i by j_{i-1}, and s'_{i+1} by s_i. As a result the new s_i is the sum of the new j_1 through j_n, and

$$\begin{aligned} N(t,i) = 1 &+ 2\sum_{j\ge 1} [1-(1-p)^{2v-i-j+1}]^{t-1} b_j(i) \\ &+ \sum_j b_j(i) \sum_{1\le n\le t-1} \sum_{j_1\ge 0} \cdots \sum_{j_{n-1}\ge 0} \sum_{j_n\ge 1} \\ &\qquad 2[1-(1-p)^{2v-i-j-s_{n+1}+1}]^{t-n-1} \prod_{1\le m\le n} b_{j_m}(i+j+s_m). \end{aligned} \tag{23}$$

The part of eq. (23) to the right of the second $b_j(i)$ is $N(t-1,i+j)-1$, so N obeys the recurrence equation

$$N(t,i) = 1 + 2\sum_{j\geq 1}[1-(1-p)^{2v-i-j+1}]^{t-1}b_j(i) + \sum_j b_j(i)[N(t-1,i+j)-1] \tag{24}$$

$$= 1 + b_0(i)[N(t-1,i)-1] + \sum_{j\geq 1} b_j(i)\{2[1-(1-p)^{2v-i-j+1}]^{t-1} + N(t-1,i+j)-1\}. \tag{25}$$

Eq. (20) gives the boundary condition $N(0,i)=1$.

With eq. (25), the average time can be computed in time $O(tv^2)$. Fig. 2 shows the results of calculations for $v=100$.

The recurrences in this paper were checked by comparing their solutions with the actual number of nodes generated by programs for the algorithms. Each recurrence was solved algebraically for $1\leq t\leq 3$, $1\leq v\leq 3$ using Macsyma. (A modification of eq. (25) was checked for $1\leq t\leq 6$, $1\leq v\leq 6$, $tv\leq 12$.) Also, each of the 2^{2tv} SAT problems was generated and solved with a program that counted the number of nodes that were generated. A problem with i literals has probability $p^i(1-p)^{2tv-i}$. Multiplying the count by the probability and summing over all problems gives a formula for the average number of nodes. (This formula is a polynomial in p with integer coefficients.) The formulas generated from the recurrences were identical with the ones generated by the programs.

4.2 Related recurrences

The number of nodes that results when a clause of length k is allowed to contribute at most k nodes can be obtained by replacing $2a_{j_n}$ in the previous derivations with $a_{j_n}+a'_{j_n}$, where a' is defined by eq. (12) with the $k=j$ term omitted from the sum. Let $T(t,i)$ be the solution to

$$T(t,i) = 1 + \sum_{j\geq 1}[1-(1-p)^{2v-i-j+1}]^{t-1}[b_j(i)+b_{j+1}(i)] + \sum_j b_j(i)[T(t-1,i+j)-1] \tag{26}$$

with boundary condition $T(0,i)=1$. The average number of nodes for the alternate counting method is $T(t)=T(t,0)$.

The recurrence for the number of solutions (in compressed form) is given by eq. (25) with the constant term dropped. Let $S(t,i)$ be the solution to

$$S(t,i) = \sum_j b_j(i)S(t-1,i+j) \tag{27}$$

with the boundary condition $S(0,i)=1$. The average number of solutions is given by $S(t)=S(t,0)$.

5 Comparison with simple backtracking

Theorem 1. The average number of nodes generated by simple backtracking is greater than or equal to the average number of nodes generated by clause order backtracking for any distribution where the probability of a predicate is not changed by permuting the variable names in a single clause.

Proof. Consider an arbitrary node, q, in a possible backtrack tree. Refer to this node with a string of T's and F's, where the i^{th} symbol says which way to set the i^{th} variable. (The important point is that the node can be described without reference to which variables are set on the path to the node.) For each predicate, P, consider whether or not node q occurs in the clause order backtrack tree for P. If node q does not occur, it is either because one of the clauses used to determine the order of assigning values to variables becomes *false* or because one of the other clauses becomes *false*. For the comparison between clause order backtracking and simple backtracking, construct a predicate P' in the following way. If node q does not occur in the clause order backtrack tree for P, then P' is the same as P. Otherwise, let k be the number of variables on the path from the root to node q, let v_i be the i^{th} variable set by clause order backtracking on the path from the root to q, and let c_i be the clause that introduced v_i into the clause order backtrack tree. Form predicate P' from P by making the following change in variable names in the indicated clauses.

For i from 1 to k, interchange variable names v_i and i in all clauses after c_i. For any node q, this mapping from P to P' is well defined, each P' is the image of a unique P, and P and P' have the same probability in the model for generating random clauses. Finally, if node q occurs in the clause order backtrack tree for P, then it also occurs in the simple backtrack tree for P'. Summing over all predicates P, we find that the probability of node q occurring in the clause order backtracking tree is no larger than the probability of it occurring in the simple backtracking tree. (For most distributions the node is less likely to be in the clause order tree, because a node q that does not appear in the clause order backtrack tree for P may still appear in the simple backtrack tree for P'.) □

7 Asymptotics

Let $U(t)$ be an upper bound on $N(t,i)-1$ so that $U(t) \geq N(t,i)-1$ for $0 \leq i \leq v$ and $t \geq 0$. Since all the coefficients in eq. (25) are positive

$$N(t,i)-1 \leq b_0(i)U(t-1) + \sum_{j\geq 1} b_j(i)\{2[1-(1-p)^{2v-i-j+1}]^{t-1} + U(t-1)\}. \tag{28}$$

Using

$$\sum_{j\geq 1} b_j(i) = 2p(v-i)(1-p)^v(1+p)^{v-i-1} \tag{29}$$

and

$$1-(1-p)^{2v-i} \geq 1-(1-p)^{2v-i-j+1} \qquad \text{for } j \geq 1 \tag{30}$$

gives

$$N(t,i)-1 \leq [1-(1-p)^v(1+p)^{v-i}]U(t-1)+2p(v-i)(1-p)^v(1+p)^{v-i-1}\{2[1-(1-p)^{2v-i}]^{t-1}+U(t-1)\}, \tag{31}$$

which can be written as

$$\begin{aligned} N(t,i)-1 \leq\ & 4p(v-i)(1-p)^v(1+p)^{v-i-1}[1-(1-p)^{2v-i}]^{t-1} \\ & + \{[2p(v-i)-1-p](1-p)^v(1+p)^{v-i-1}+1\}U(t-1). \end{aligned} \tag{32}$$

Replacing the i's on the right side of eq. (32) with the value that maximizes each part and replacing two factors of $(1+p)^{-1}$ with 1 gives

$$N(t,i)-1 \leq 4pv(1-p^2)^v[1-(1-p)^{2v}]^{t-1} + \{1+[2pv-1](1-p^2)^v\}U(t-1), \tag{33}$$

where we have assumed that $2pv-1$ is nonnegative. (When $2pv-1$ is negative, a slight modification of the current derivation shows that $N(t,i)$ is small, so that case is not considered further.)

Now let $U(t)$ be the solution to the recurrence

$$U(t) = 4pv(1-p^2)^v[1-(1-p)^{2v}]^{t-1} + \{1+[2pv-1](1-p^2)^v\}U(t-1) \tag{34}$$

with boundary condition $U(0)=0$. Eq. (34) and the boundary for eq. (25) imply that $U(t)$ is an upper bound on $N(t,i)-1$. Let $a = 1+(2pv-1)(1-p^2)^v$ and $b = 1-(1-p)^{2v}$. Then,

$$U(t) = 4pv(1-p^2)^v \frac{a^t-b^t}{a-b}. \tag{35}$$

Since $(a^t-b^t)/(a-b) = a^{t-1}+a^{t-2}b+a^{t-3}b^2+\cdots+ab^{t-2}+b^{t-1}$, $a \geq b \geq 0$, and $a \geq 1$, we have

$$U(t) \leq 4ptv(1-p^2)^v a^t. \tag{36}$$

Also,

$$U(t) \leq \frac{4pv(1-p^2)^v}{a-b} a^t. \tag{37}$$

Fig. 3 shows the contours from the approximation

$$N(t,i) \le 1 + \frac{4pv(1-p^2)^v}{1+p} \frac{[1+(2pv-1)(1-p^2)^v/(1+p)]^t - [1-(1-p)^{2v}]^t}{(2pv-1)(1-p^2)^v/(1+p) + (1-p)^{2v}}. \tag{38}$$

This approximation is eq. (35) modified by including the $(1+p)^{-1}$ factors that were dropped in deriving eq. (35). When eq. (38) gives a small result, it appears to have the main features of the real results (except in the lower right region where p is small and t is large). The $(1+p)^{-1}$ factors that are including in eq. (38) but omitted in eq. (35) have a noticeable effect in the upper part of Fig. 3, but the following derivations show that the factors are not significant when $v^{1/2}$ is large.

The right side of eq. (38) is large when a is above 1, and it is small when a is near 1. Thus, it is small when pv is near $1/2$ (so that $2pv-1$ is small) and when p^2v is large (so that $2pv(1-p^2)^v$ is small). When p is larger than v^{-1} but smaller than $v^{-1/2}$, the bound is exponentially large (provided t is not small).

6.1 The small p case

From eq. (36) we obtain

$$N(t,i) \le 1 + 4ptv(1-p^2)^v[1+(2pv-1)(1-p^2)^v]^t. \tag{39}$$

When p is small, the $(1-p^2)^v$ term is not important. Replacing it with the upper limit of 1 gives

$$N(t,i) \le 1 + 4ptv(2pv)^t. \tag{40}$$

The number of nodes is no more than v^n when

$$v^n \ge 1 + 4ptv(2pv)^t. \tag{41}$$

Rearranging and taking logarithms gives

$$\ln(2pv) \le \frac{n\ln v - \ln(4ptv) - O(v^{-n})}{t}. \tag{42}$$

Since $p \le 1$, the number of nodes is no more than v^n when

$$p \le \frac{1}{2v} e^{[(n-1)\ln v - \ln t - O(1)]/t}. \tag{43}$$

6.2 The Large p case

From plugging in the values of a and b into eq. (37) we obtain

$$N(t,i) \le 1 + \frac{4pv(1-p^2)^v}{(2pv-1)(1-p^2)^v + (1-p)^{2v}}[1+(2pv-1)(1-p^2)^v]^t. \tag{44}$$

Dropping terms that lead to an even large bound gives

$$N(t,i) \le 1 + \frac{4pv(1-p^2)^v}{(2pv-1)(1-p^2)^v}[1+2pv(1-p^2)^v]^t \tag{45}$$

$$\le 1 + \frac{2}{1-1/(2pv)}[1+2pv(1-p^2)^v]^t \tag{46}$$

$$\le 1 + \left[1+O\left(\frac{1}{pv}\right)\right][1+2pv(1-p^2)^v]^t. \tag{47}$$

The number of nodes is no more than v^n when

$$v^n \ge 1 + 2\left[1+O\left(\frac{1}{pv}\right)\right][1+2pv(1-p^2)^v]^t. \tag{48}$$

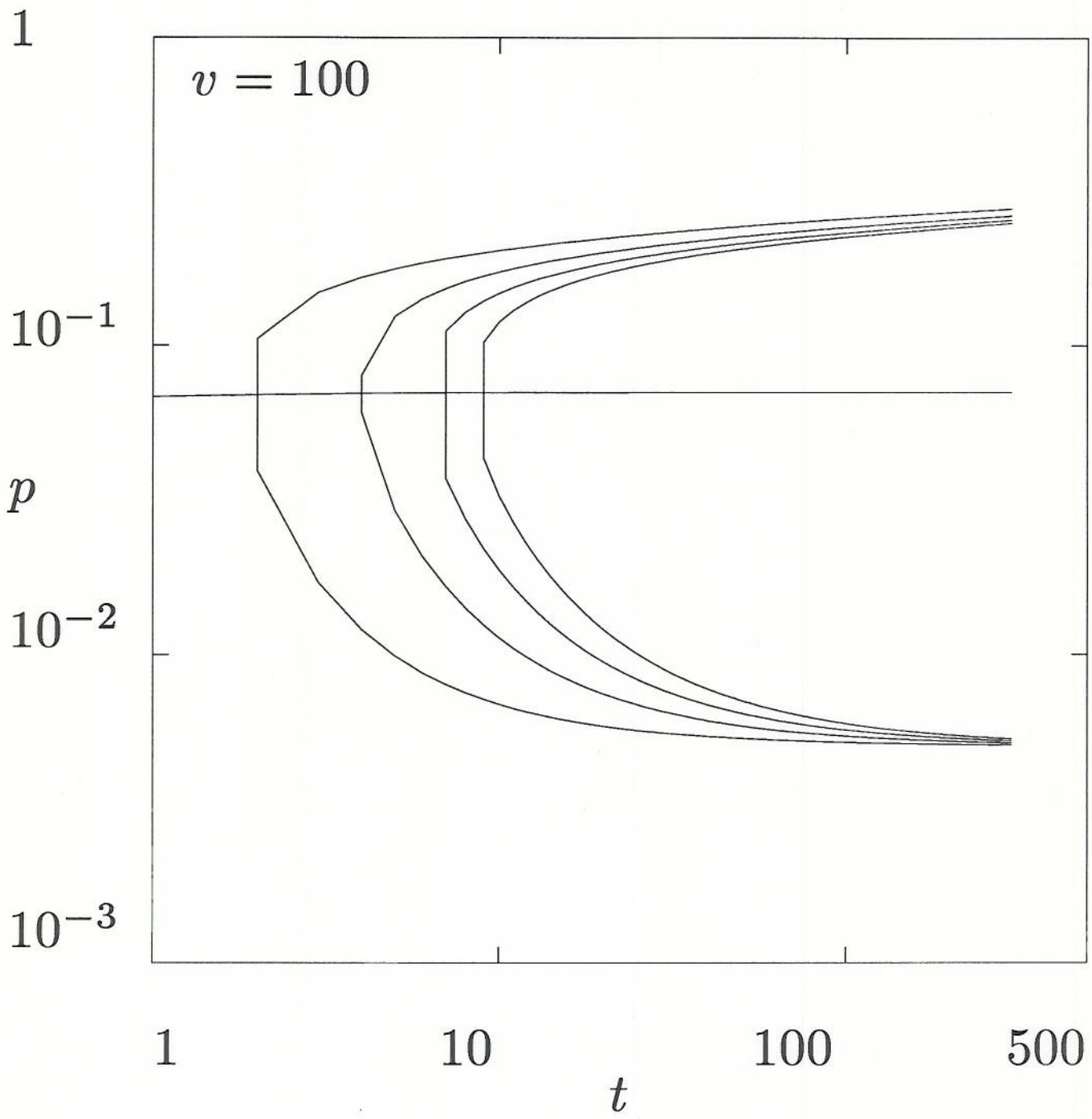

Fig. 3. The contours for the approximation to the average number of nodes.

Writing this as $v^n - 1 \geq 2[1+O(1/(pv))][1+2pv(1-p^2)^v]^t$ and taking logarithms gives $[n \ln v - \ln(1-v^{-n})]/t \geq \ln[1 + 2pv(1-p^2)^v] + \ln 2 + \ln[1 + O(1/(pv))]$, which can be written as

$$e^{[n \ln v - O(1)]/t} - 1 \geq 2pv(1-p^2)^v. \tag{49}$$

Expanding the exponential in a power series gives

$$\frac{n \ln v}{2ptv}\left[1 + O\left(\frac{n \ln v}{t}\right)\right] \geq (1-p^2)^v. \tag{50}$$

Taking logarithms gives

$$\ln p + \ln t + \ln v - \ln\ln v - O\left(\frac{n \ln v}{t}\right) \leq v \ln(1-p^2). \tag{51}$$

Dividing by v and replace $\ln(1-p^2)$ with $-p^2 - O(p^4)$.

$$p^2 \geq \frac{\ln p + \ln t + \ln v - \ln\ln v}{v} - O(p^4) - O\left(\frac{n \ln v}{tv}\right). \tag{52}$$

For any $\delta > 0$ and large v, eq. (52) is satisfied by

$$p \geq \sqrt{\frac{(1+\delta)\ln t + (\ln v)/2}{v}}. \tag{53}$$

When $\ln[(\ln t)/(\ln v)] < \ln v$ (i.e., when $t < v^{\ln v}$), the bound on p is also satisfied with $\delta = 0$.

To obtain conditions where the average time is barely exponential (bounded by $(1+\epsilon)^v$ for small positive ϵ), replace $n \ln v$ with ϵ in eqs. (3) and (52).

Acknowledgement: We wish to thank the referees for their carefully reading of the paper and their helpful suggestions.

References

1. Cynthia A. Brown and Paul W. Purdom, *An Average Time Analysis of Backtracking*, SIAM J. Comput. **10** (1981) pp 583–593.
2. Khaled Bugrara, Youfang Pan, and Paul Purdom, *Exponential Average Time for the Pure Literal Rule*, SIAM J. Comput. **18** (1988) pp 409–418.
3. Stephen A. Cook, *The complexity of Theorem-Proving Procedures*, Proc. 3rd ACM Symp. on Theory of Computing, ACM, New York (1971) pp 151–158.
4. John Franco, *On the Occurrence of Null Clauses in Random Instances of Satisfiability*, Indiana University Computer Science Tech. Report 291 (1989).
5. John Franco, *Elimination of Infrequent Variables Improves Average Case Performance of Satisfiability Algorithms*, SIAM J. Comput. **20** (1991) pp 1119–1127.
6. Michael R. Garey and David S. Johnson, *Computers and Intractability*, W. H. Freeman, San Francisco (1979) pp 38–44, 48–50.
7. Kazuo Iwama, *CNF Satisfiability Test by Counting and Polynomial Average Time*, SIAM J. Comput. **18** (1989) pp 385–391.
8. Paul W. Purdom, *Search Rearrangement Backtracking and Polynomial Average Time*, Artificial Intelligence **21** (1983) pp 117–133.
9. Paul W. Purdom, *A Survey of Average Time Analyses of Satisfiability Algorithms*, Journal of Information Processing, **13** (1990) pp 449–455. An earlier version appeared as *Random Satisfiability Problems*, Proc. of the International Workshop on Discrete Algorithms and Complexity, The Institute of Electronics, Information and Communication Engineers, Tokyo (1989) pp 253–259.
10. Paul W. Purdom And Cynthia A. Brown, *The Pure Literal Rule and Polynomial Average Time*, SIAM J. Comput. **14** (1985) pp 943–953.
11. Paul W. Purdom and Cynthia A. Brown, *Polynomial-Average-Time Satisfiability Problems*, Information Sciences **41** (1987) pp 23–42.
12. Paul W. Purdom, Cynthia A. Brown, and Edward L. Robertson, *Backtracking with Multi-Level Dynamic Search Rearrangement*, Acta Informatica **15** (1981) pp 99–113.