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

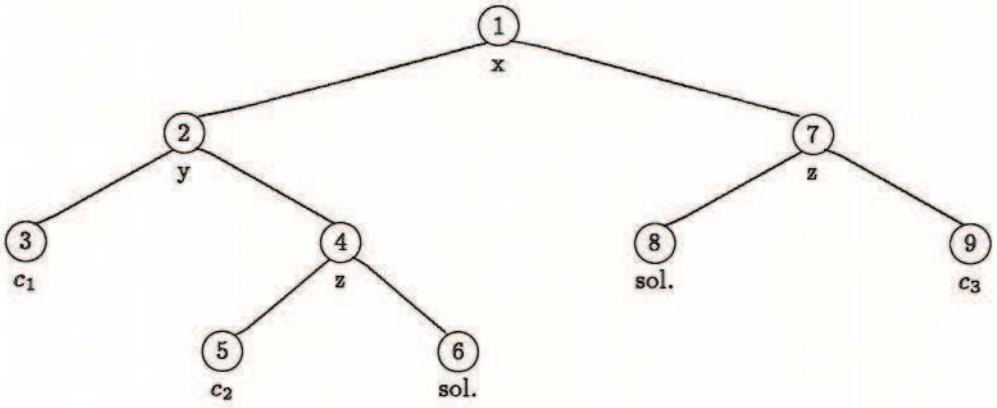

Fig. 1. The backtrack tree for the predicate consisting of the three clauses $c_1 = x \vee y$, $c_2 = x \vee z$, $c_3 = \bar{x} \vee \bar{z}$.

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

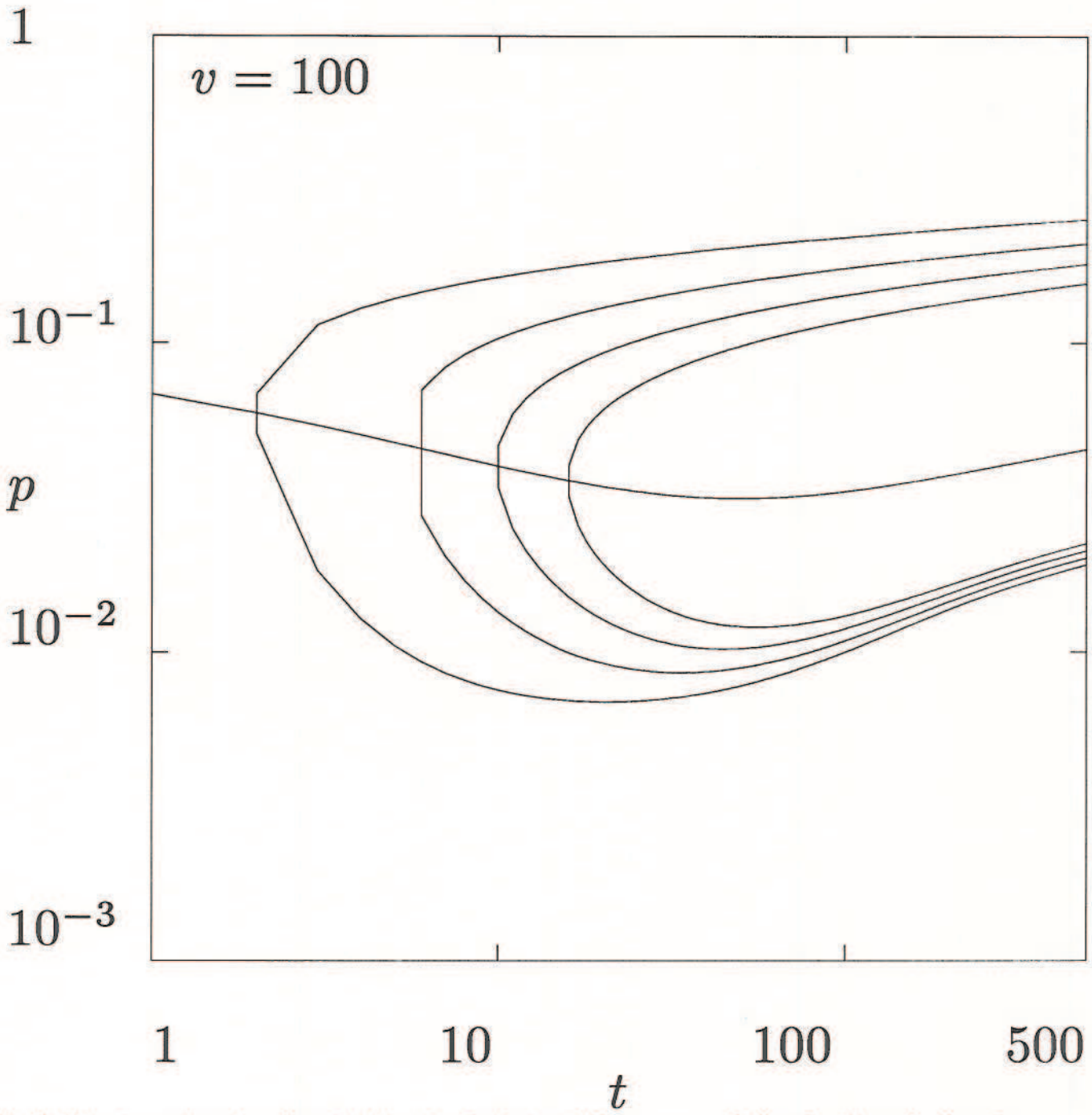

Fig. 2. The outer contour shows for each t the value of p that results in an average of 100 nodes. Proceeding inward, contours for an average of 100^2, 100^3, and 100^4 are also shown. The line from the left side to the right side shows for each t which value of p results in the largest average number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