

## The Hazards of Fancy Backtracking

Andrew B. Baker\*

Computational Intelligence Research Laboratory  
1269 University of Oregon  
Eugene, Oregon 97403  
baker@cs.uoregon.edu

### Abstract

There has been some recent interest in intelligent backtracking procedures that can return to the source of a difficulty without erasing the intermediate work. In this paper, we show that for some problems it can be counterproductive to do this, and in fact that such "intelligence" can cause an exponential increase in the size of the ultimate search space. We discuss the reason for this phenomenon, and we present one way to deal with it.

### 1 Introduction

We are interested in systematic search techniques for solving constraint satisfaction problems. There has been some recent work on intelligent backtracking procedures that can return to the source of a difficulty without erasing the intermediate work. In this paper, we will argue that these procedures have a substantial drawback, but first let us see why they might make sense. Consider an example from (Ginsberg 1993). Suppose we are coloring a map of the United States (subject to the usual constraint that only some fixed set of colors may be used, and adjacent states cannot be the same color).

Let us assume that we first color the states along the Mississippi, thus dividing the rest of the problem into two independent parts. We now color some of the western states, then we color some eastern states, and then we return to the west. Assume further that upon our return to the west we immediately get stuck: we find a western state that we cannot color. What do we do?

Ordinary chronological backtracking (depth-first search) would backtrack to the most recent decision, but this would be a state east of the Mississippi and hence irrelevant; the search procedure would only address the real problem after trying every possible coloring for the previous eastern states.

---

\*This work has been supported by the Air Force Office of Scientific Research under grant number 92-0693 and by ARPA/Rome Labs under grant numbers F30602-91-C-0036 and F30602-93-C-00031.

Backjumping (Gaschnig 1979) is somewhat more intelligent; it would immediately jump back to some state adjacent to the one that we cannot color. In the process of doing this, however, it would erase all the intervening work, i.e., it would uncolor the whole eastern section of the country. This is unfortunate; it means that each time we backjump in this fashion, we will have to start solving the eastern subproblem all over again.

Ginsberg has recently introduced *dynamic backtracking* (Ginsberg 1993) to address this difficulty. In dynamic backtracking, one moves to the source of the problem without erasing the intermediate work. Of course, simply retaining the *values* of the intervening variables is not enough; if these values turn out to be wrong, we will need to know where we were in the search space so that we can continue the search systematically. In order to do this, dynamic backtracking accumulates nogoods to keep track of portions of the space that have been ruled out.

Taken to an extreme, this would end up being very similar to dependency-directed backtracking (Stallman & Sussman 1977). Although dependency-directed backtracking does not save intermediate values, it saves enough dependency information for it to quickly recover its position in the search space. Unfortunately, dependency-directed backtracking saves far too much information. Since it learns a new nogood from every backtrack point, it generally requires an exponential amount of memory — and for each move in the search space, it may have to wade through a great many of these nogoods. Dynamic backtracking, on the other hand, only keeps nogoods that are "relevant" to the current position in the search space. It not only learns new nogoods; it also throws away those old nogoods that are no longer applicable.

Dynamic backtracking, then, would seem to be a happy medium between backjumping and full dependency-directed backtracking. Furthermore, Ginsberg has presented empirical evidence that dynamic backtracking outperforms backjumping on the problem of solving crossword puzzles (Ginsberg 1993).

Unfortunately, as we will soon see, dynamic back-

tracking has problems of its own.

The plan of the paper is as follows. The next section reviews the details of dynamic backtracking. Section 3 describes an experiment comparing the performance of dynamic backtracking with that of depth-first search and backjumping on a problem class that has become somewhat of a standard benchmark. We will see that dynamic backtracking is *worse* by a factor exponential in the size of the problem. Note that this will not be simply the usual complaint that intelligent search schemes often have a lot of overhead. Rather, our complaint will be that the effective search space itself becomes larger; even if dynamic backtracking could be implemented without any additional overhead, it would still be far less efficient than the other algorithms.

Section 4 contains both our analysis of what is going wrong with dynamic backtracking and an experiment consistent with our view. In Section 5, we describe a modification to dynamic backtracking that appears to fix the problem. Concluding remarks are in Section 6.

## 2 Dynamic backtracking

Let us begin by reviewing the definition of a constraint satisfaction problem, or CSP.

**Definition 1** A constraint satisfaction problem  $(V, D, C)$  is defined by a finite set of variables  $V$ , a finite set of values  $D_v$  for each  $v \in V$ , and a finite set of constraints  $C$ , where each constraint  $(W, P) \in C$  consists of a list of variables  $W = (w_1, \dots, w_k) \subseteq V$  and a predicate on these variables  $P \subseteq D_{w_1} \times \dots \times D_{w_k}$ . A solution to the problem is a total assignment  $f$  of values to variables, such that for each  $v \in V$ ,  $f(v) \in D_v$  and for each constraint  $((w_1, \dots, w_k), P)$ ,  $(f(w_1), \dots, f(w_k)) \in P$ .

Like depth-first search, dynamic backtracking works with partial solutions; a partial solution to a CSP is an assignment of values to some subset of the variables, where the assignment satisfies all of the constraints that apply to this particular subset. The algorithm starts by initializing the partial solution to have an empty domain, and then it gradually extends this solution. As the algorithm proceeds, it will derive new constraints, or “nogoods,” that rule out portions of the search space that contain no solutions. Eventually, the algorithm will either derive the empty nogood, proving that the problem is unsolvable, or it will succeed in constructing a total solution that satisfies all of the constraints. We will always write the nogoods in directed form; e.g.,

$$(v_1 = q_1) \wedge \dots \wedge (v_{k-1} = q_{k-1}) \Rightarrow v_k \neq q_k$$

tells us that variables  $v_1$  through  $v_k$  cannot simultaneously have the values  $q_1$  through  $q_k$  respectively.

The main innovation of dynamic backtracking (compared to dependency-directed backtracking) is that it only retains nogoods whose left-hand sides are currently true. That is to say that if the above nogood

were stored, then  $v_1$  through  $v_{k-1}$  would have to have the indicated values (and since the current partial solution has to respect the nogoods as well as the original constraints,  $v_k$  would either have some value other than  $q_k$  or be unbound). If at some point, one of the left-hand variables were changed, then the nogood would have to be deleted since it would no longer be “relevant.” Because of this relevance requirement, it is easy to compute the currently permissible values for any variable. Furthermore, if all of the values for some variable are eliminated by nogoods, then one can resolve these nogoods together to generate a new nogood. For example, assuming that  $D_{v_9} = \{1, 2\}$ , we could resolve

$$(v_1 = a) \wedge (v_3 = c) \Rightarrow v_9 \neq 1$$

with

$$(v_2 = b) \wedge (v_3 = c) \Rightarrow v_9 \neq 2$$

to obtain

$$(v_1 = a) \wedge (v_2 = b) \Rightarrow v_3 \neq c$$

In order for our partial solution to remain consistent with the nogoods, we would have to simultaneously unbind  $v_3$ . This corresponds to backjumping from  $v_9$  to  $v_3$ , but without erasing any intermediate work. Note that we had to make a decision about which variable to put on the right-hand side of the new nogood. The rule of dynamic backtracking is that the right-hand variable must always be the one that was most recently assigned a value; this is absolutely crucial, as without this restriction, the algorithm would not be guaranteed to terminate.

The only thing left to mention is how nogoods get acquired in the first place. Before we try to bind a new variable, we will check the consistency of each possible value<sup>1</sup> for this variable with the values of all currently bound variables. If a constraint would be violated, we write the constraint as a directed nogood with the new variable on the right-hand side.

We have now reviewed all the major ideas of dynamic backtracking, so we will give the algorithm below in a somewhat informal style. For the precise mathematical definitions, see (Ginsberg 1993).

### Procedure DYNAMIC-BACKTRACKING

1. Initialize the partial assignment  $f$  to have the empty domain, and the set of nogoods  $\Gamma$  to be the empty set. At all times,  $f$  will satisfy the nogoods in  $\Gamma$  as well as the original constraints.
2. If  $f$  is a total assignment, then return  $f$  as the answer. Otherwise, choose an unassigned variable  $v$  and for each possible value of this variable that would cause a constraint violation, add the appropriate nogood to  $\Gamma$ .
3. If variable  $v$  has some value  $x$  that is not ruled out by any nogood, then set  $f(v) = x$ , and return to step 2.

<sup>1</sup>A value is possible if it is not eliminated by a nogood.

- Each value of  $v$  violates a nogood. Resolve these nogoods together to generate a new nogood that does not mention  $v$ . If it is the empty nogood, then return “unsatisfiable” as the answer. Otherwise, write it with its chronologically most recent variable (say,  $w$ ) on the right-hand side, add this directed nogood to  $\Gamma$ , and call ERASE-VARIABLE( $w$ ). If each value of  $w$  now violates a nogood, then set  $v = w$  and return to step 4; otherwise, return to step 2.

**Procedure ERASE-VARIABLE( $w$ )**

- Remove  $w$  from the domain of  $f$ .
- For each nogood  $\gamma \in \Gamma$  whose left-hand side mentions  $w$ , call DELETE-NOGOOD( $\gamma$ ).

**Procedure DELETE-NOGOOD( $\gamma$ )**

- Remove  $\gamma$  from  $\Gamma$ .

Each variable-value pair can have at most one nogood at a given time, so it is easy to see that the algorithm only requires a polynomial amount of memory. In (Ginsberg 1993), it is proven that dynamic backtracking always terminates with a correct answer.

This is the theory of dynamic backtracking. How well does it do in practice?

### 3 Experiments

To compare dynamic backtracking with depth-first search and backjumping, we will use randomly-generated propositional satisfiability problems, or to be more specific, random 3-SAT problems with  $n$  variables and  $m$  clauses.<sup>2</sup> Since a SAT problem is just a Boolean CSP, the above discussion applies directly. Each clause will be chosen independently using the uniform distribution over the  $\binom{n}{3}2^3$  non-redundant 3-literal clauses. It turns out that the hardest random 3-SAT problems appear to arise at the “crossover point” where the ratio of clauses to variables is such that about half the problems are satisfiable (Mitchell, Selman, & Levesque 1992); the best current estimate for the location of this crossover point is at  $m = 4.24n + 6.21$  (Crawford & Auton 1993). Several recent authors have used these crossover-point 3-SAT problems to measure the performance of their algorithms (Crawford & Auton 1993; Selman, Levesque, & Mitchell 1992).

In the dynamic backtracking algorithm, step 2 leaves open the choice of which variable to select next; backtracking and backjumping have similar indeterminacies. We used the following variable-selection heuristics:

- If there is an unassigned variable with one of its two values currently eliminated by a nogood, then choose that variable.

<sup>2</sup>Each clause in a 3-SAT problem is a disjunction of three literals. A literal is either a propositional variable or its negation.

- Otherwise, if there is an unassigned variable that appears in a clause in which all the other literals have been assigned false, then choose that variable.
- Otherwise, choose the unassigned variable that appears in the most binary clauses. A binary clause is a clause in which exactly two literals are unvalued, and all the rest are false.<sup>3</sup>

The first heuristic is just a typical backtracking convention, and in fact is intrinsically part of depth-first search and backjumping. The second heuristic is unit propagation, a standard part of the Davis-Putnam procedure for propositional satisfiability (Davis, Logemann, & Loveland 1962; Davis & Putnam 1960). The last heuristic is also a fairly common SAT heuristic; see for example (Crawford & Auton 1993; Zabih & McAllester 1988). These heuristics choose variables that are highly constrained and constraining in an attempt to make the ultimate search space as small as possible.

For our experiments, we varied the number of variables  $n$  from 10 to 60 in increments of 10. For each value of  $n$  we generated random crossover-point problems<sup>4</sup> until we had accumulated 100 satisfiable and 100 unsatisfiable instances. We then ran each of the three algorithms on the 200 instances in each problem set. The mean number of times that a variable is assigned a value is displayed in Table 1.

Dynamic backtracking appears to be worse than the other two algorithms by a factor exponential in the size of the problem; this is rather surprising. Because of the lack of structure in these randomly-generated problems, we might not expect dynamic backtracking to be significantly better than the other algorithms, but why would it be *worse*? This question is of more than academic interest. Some real-world search problems may turn out to be similar in some respects to the crossword puzzles on which dynamic backtracking does well, while being similar in other respects to these random 3-SAT problems — and as we can see from Table 1, even a small “random 3-SAT component” will be enough to make dynamic backtracking virtually useless.

### 4 Analysis

To understand what is going wrong with dynamic backtracking, consider the following abstract SAT example:

$$a \rightarrow x \quad (1)$$

$$\Rightarrow \neg a \quad (2)$$

$$\neg a \Rightarrow b \quad (3)$$

$$b \Rightarrow c \quad (4)$$

<sup>3</sup>On the very first iteration in a 3-SAT problem, there will not yet be any binary clauses, so instead choose the variable that appears in the most clauses overall.

<sup>4</sup>The numbers of clauses that we used were 49, 91, 133, 176, 218, and 261 respectively.

Variables	Average Number of Assignments		
	Depth-First Search	Backjumping	Dynamic Backtracking
10	20	20	22
20	54	54	94
30	120	120	643
40	217	216	4,532
50	388	387	31,297
60	709	705	212,596

Table 1: A comparison using randomly-generated 3-SAT problems.

$$c \Rightarrow d \quad (5)$$

$$x \Rightarrow \neg d \quad (6)$$

Formula (1) represents the clause  $\neg a \vee x$ ; we have written it in the directed form above to suggest how it will be used in our example. The remaining formulas correspond to groups of clauses; to indicate this, we have written them using the double arrow ( $\Rightarrow$ ). Formula (2) represents some number of clauses that can be used to prove that  $a$  is contradictory. Formula (3) represents some set of clauses showing that if  $a$  is false, then  $b$  must be true; similar remarks apply to the remaining formulas. These formulas will also represent the nogoods that will eventually be learned.

Imagine dynamic backtracking exploring the search space in the order suggested above. First it sets  $a$  true, and then it concludes  $x$  using unit resolution (and adds a nogood corresponding to (1)). Then after some amount of further search, it finds that  $a$  has to be false. So it erases  $a$ , adds the nogood (2), and then deletes the nogood (1) since it is no longer "relevant." Note that it does *not* delete the proposition  $x$  — the whole point of dynamic backtracking is to preserve this intermediate work.

It will then set  $a$  false, and after some more search will learn nogoods (3)–(5), and set  $b$ ,  $c$  and  $d$  true. It will then go on to discover that  $x$  and  $d$  cannot both be true, so it will have to add a new nogood (6) and erase  $d$ . The rule, remember, is that the most recently valued variable goes on the right-hand side of the nogood. Nogoods (5) and (6) are resolved together to produce the nogood

$$x \Rightarrow \neg c \quad (7)$$

where once again, since  $c$  is the most recent variable, it must be the one that is retracted and placed on the right-hand side of the nogood; and when  $c$  is retracted, nogood (5) must be deleted also. Continuing in this fashion, dynamic backtracking will derive the nogoods

$$x \Rightarrow \neg b \quad (8)$$

$$x \Rightarrow a \quad (9)$$

The values of  $b$  and  $a$  will be erased, and nogoods (4) and (3) will be deleted.

Finally, (2) and (9) will be resolved together producing

$$\Rightarrow \neg x \quad (10)$$

The value of  $x$  will be erased, nogoods (6)–(9) will be deleted, and the search procedure will then go on to rediscover (3)–(5) all over again.

By contrast, backtracking and backjumping would erase  $x$  before (or at the same time as) erasing  $a$ . They could then proceed to solve the rest of the problem without being encumbered by this leftover inference. It might help to think of this in terms of search trees even though dynamic backtracking is not really searching a tree. By failing to retract  $x$ , dynamic backtracking is in a sense choosing to "branch" on  $x$  before branching on  $a$  through  $d$ . This virtually doubles the size of the ultimate search space.

This example has been a bit involved, and so far it has only demonstrated that it is *possible* for dynamic backtracking to be worse than the simpler methods; why would it be worse in the *average* case? The answer lies in the heuristics that are being used to guide the search.

At each stage, a good search algorithm will try to select the variable that will make the remaining search space as small as possible. The appropriate choice will depend heavily on the values of previous variables. Unit propagation, as in equation (1), is an obvious example: if  $a$  is true, then we should immediately set  $x$  true as well; but if  $a$  is false, then there is no longer any particular reason to branch on  $x$ . After  $a$  is unset, our variable-selection heuristic would most likely choose to branch on a variable other than  $x$ ; branching on  $x$  anyway is tantamount to randomly corrupting this heuristic. Now, dynamic backtracking does not really "branch" on variables since it has the ability to jump around in the search space. As we have seen, however, the decision not to erase  $x$  amounts to the same thing. In short, the leftover work that dynamic backtracking tries so hard to preserve often does more harm than good because it perpetuates decisions whose heuristic justifications have expired.

This analysis suggests that if we were to eliminate the heuristics, then dynamic backtracking would no longer be defeated by the other search methods. Table 2 contains the results of such an experiment. It is important to note that all of the previously listed heuristics (including unit resolution!) were disabled for the purpose of this experiment; at each stage, we simply chose the first unbound variable (using some

Variables	Average Number of Assignments		
	Depth-First Search	Backjumping	Dynamic Backtracking
10	77	61	51
20	2,243	750	478
30	53,007	7,210	3,741

Table 2: The same comparison as Table 1, but with all variable-selection heuristics disabled.

Variables	Average Number of Assignments		
	Depth-First Search	Backjumping	Dynamic Backtracking
10	20	20	20
20	54	54	53
30	120	120	118
40	217	216	209
50	388	387	375
60	709	705	672

Table 3: The same comparison as Table 1, but with dynamic backtracking modified to undo unit propagation when it backtracks.

fixed ordering). For each value of  $n$  listed, we used the same 200 random problems that were generated earlier.

The results in Table 2 are as expected. All of the algorithms fare far worse than before, but at least dynamic backtracking is not worse than the others. In fact, it is a bit better than backjumping and substantially better than backtracking. So given that there is nothing intrinsically wrong with dynamic backtracking, the challenge is to modify it in order to reduce or eliminate its negative interaction with our search heuristics.

## 5 Solution

We have to balance two considerations. When backtracking, we would like to preserve as much nontrivial work as possible. On the other hand, we do not want to leave a lot of “junk” lying around whose main effect is to degrade the effectiveness of the heuristics. In general, it is not obvious how to strike the appropriate balance. For the propositional case, however, there is a simple modification that seems to help, namely, undoing unit propagation when backtracking.

We will need the following definition:

**Definition 2** *Let  $v$  be a variable (in a Boolean CSP) that is currently assigned a value. A nogood whose conclusion eliminates the other value for  $v$  will be said to justify this assignment.*

If a value is justified by a nogood, and this nogood is deleted at some point, then the value should be erased as well. Selecting the given value was once a good heuristic decision, but now that its justification has been deleted, the value would probably just get in the way. Therefore, we will rewrite DELETE-NOGOOD as follows, and leave the rest of dynamic backtracking intact:

### Procedure DELETE-NOGOOD( $\gamma$ )

1. Remove  $\gamma$  from  $\Gamma$ .
2. For each variable  $w$  justified by  $\gamma$ , call ERASE-VARIABLE( $w$ ).

Note that ERASE-VARIABLE calls DELETE-NOGOOD in turn; the two procedures are mutually recursive. This corresponds to the possibility of undoing a cascade of unit resolutions. Like Ginsberg’s original algorithm, this modified version is sound and complete, uses only polynomial space, and can solve the the union of several independent problems in time proportional to the sum of that required for the original problems.

We ran this modified procedure on the same experiments as before, and the results are in Table 3. Happily, dynamic backtracking no longer blows up the search space. It does not do much good either, but there may well be other examples for which this modified version of dynamic backtracking is the method of choice.

How will this apply to non-Boolean problems? First of all, for non-Boolean CSPs, the problem is not quite as dire. Suppose a variable has twenty possible values, all but two of which are eliminated by nogoods. Suppose further that on this basis, one of the remaining values is assigned to the variable. If one of the eighteen nogoods is later eliminated, then the variable will still have but three possibilities and will probably remain a good choice. It is only in the Boolean problems that an assignment can go all the way from being totally justified to totally unjustified with the deletion of a single nogood. Nonetheless, in experiments by Jónsson and Ginsberg it was found that dynamic backtracking often did worse than depth-first search when coloring random graphs (Jónsson & Ginsberg 1993). Perhaps some variant of our new method would help on these

problems. One idea would be to delete a value if it loses a certain number (or percentage) of the nogoods that once supported it.

## 6 Conclusion

Although we have presented this research in terms of Ginsberg's dynamic backtracking algorithm, the implications are much broader. Any systematic search algorithm that learns and forgets nogoods as it moves laterally through a search space will have to address—in some way or another—the problem that we have discussed. The fundamental problem is that when a decision is retracted, there may be subsequent decisions whose justifications are thereby undercut. While there is no *logical* reason to retract these decisions as well, there may be good heuristic reasons for doing so.

On the other hand, the solution that we have presented is not the only one possible, and it is probably not the best one either. Instead of erasing a variable that has lost its heuristic justification, it would be better to keep the value around, but in the event of a contradiction remember to backtrack on this variable instead of a later one. With standard dynamic backtracking, however, we do not have this option; we always have to backtrack on the most recent variable in the new nogood. Ginsberg and McAllester have recently developed *partial-order dynamic backtracking* (Ginsberg & McAllester 1994), a variant of dynamic backtracking that relaxes this restriction to some extent, and it might be interesting to explore some of the possibilities that this more general method makes possible.

Perhaps the main purpose of this paper is to sound a note of caution with regard to the new search algorithms. Ginsberg claims in one of his theorems that dynamic backtracking “can be expected to expand fewer nodes than backjumping provided that the goal nodes are distributed randomly in the search space” (Ginsberg 1993). In the presence of search heuristics, this is false. For example, the goal nodes in *unsatisfiable* 3-SAT problems are certainly randomly distributed (since there are not any goal nodes), and yet standard dynamic backtracking can take orders of magnitude longer to search the space.

Therefore, while there are some obvious benefits to the new backtracking techniques, the reader should be aware that there are also some hazards.

## Acknowledgments

I would like to thank all the members of CIRL, and especially Matthew Ginsberg and James Crawford, for many useful discussions.

## References

Crawford, J. M., and Auton, L. D. 1993. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 21–27.

Davis, M., and Putnam, H. 1960. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery* 7:201–215.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the ACM* 5:394–397.

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

Ginsberg, M. L., and McAllester, D. A. 1994. GSAT and dynamic backtracking. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*.

Ginsberg, M. L. 1993. Dynamic backtracking. *Journal of Artificial Intelligence Research* 1:25–46.

Jónsson, A. K., and Ginsberg, M. L. 1993. Experimenting with new systematic and nonsystematic search procedures. In *Proceedings of the AAAI Spring Symposium on AI and NP-Hard Problems*.

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

Selman, B.; Levesque, H.; and Mitchell, D. 1992. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 440–446.

Stallman, R. M., and Sussman, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* 9:135–196.

Zabih, R., and McAllester, D. 1988. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 155–160.