Journal of Artificial Intelligence Research 1 (1993) 25-46 Submitted 7/93; published 8/93

Dynamic Backtracking

Matthew L. Ginsberg GINSBERG@CS.UOREGON.EDU
CIRL, University of Oregon,
Eugene, OR 97403-1269 USA

Abstract

Because of their occasional need to return to shallow points in a search tree, existing backtracking methods can sometimes erase meaningful progress toward solving a search problem. In this paper, we present a method by which backtrack points can be moved deeper in the search space, thereby avoiding this difficulty. The technique developed is a variant of dependency-directed backtracking that uses only polynomial space while still providing useful control information and retaining the completeness guarantees provided by earlier approaches.

1. Introduction

Imagine that you are trying to solve some constraint-satisfaction problem, or CSP. In the interests of definiteness, I will suppose that the CSP in question involves coloring a map of the United States subject to the restriction that adjacent states be colored differently.

Imagine we begin by coloring the states along the Mississippi, thereby splitting the remaining problem in two. We now begin to color the states in the western half of the country, coloring perhaps half a dozen of them before deciding that we are likely to be able to color the rest. Suppose also that the last state colored was Arizona.

At this point, we change our focus to the eastern half of the country. After all, if we can't color the eastern half because of our coloring choices for the states along the Mississippi, there is no point in wasting time completing the coloring of the western states.

We successfully color the eastern states and then return to the west. Unfortunately, we color New Mexico and Utah and then get stuck, unable to color (say) Nevada. What's more, backtracking doesn't help, at least in the sense that changing the colors for New Mexico and Utah alone does not allow us to proceed farther. Depth-first search would now have us backtrack to the eastern states, trying a new color for (say) New York in the vain hope that this would solve our problems out West.

This is obviously pointless; the blockade along the Mississippi makes it impossible for New York to have any impact on our attempt to color Nevada or other western states. What's more, we are likely to examine every *possible* coloring of the eastern states before addressing the problem that is actually the source of our difficulties.

The solutions that have been proposed to this involve finding ways to backtrack directly to some state that might actually allow us to make progress, in this case Arizona or earlier. Dependency-directed backtracking (Stallman & Sussman, 1977) involves a direct backtrack to the source of the difficulty; backjumping (Gaschnig, 1979) avoids the computational overhead of this technique by using syntactic methods to estimate the point to which backtrack is necessary.

In both cases, however, note that although we backtrack to the source of the problem, we backtrack *over* our successful solution to half of the original problem, discarding our solution to the problem of coloring the states in the East. And once again, the problem is worse than this – after we recolor Arizona, we are in danger of solving the East yet again before realizing that our new choice for Arizona needs to be changed after all. We won't examine every possible coloring of the eastern states, but we are in danger of rediscovering our successful coloring an exponential number of times.

This hardly seems sensible; a human problem solver working on this problem would simply ignore the East if possible, returning directly to Arizona and proceeding. Only if the states along the Mississippi needed new colors would the East be reconsidered – and even then only if no new coloring could be found for the Mississippi that was consistent with the eastern solution.

In this paper we formalize this technique, presenting a modification to conventional search techniques that is capable of backtracking not only to the most recently expanded node, but also directly to a node elsewhere in the search tree. Because of the dynamic way in which the search is structured, we refer to this technique as *dynamic backtracking.*

A more specific outline is as follows: We begin in the next section by introducing a variety of notational conventions that allow us to cast both existing work and our new ideas in a uniform computational setting. Section 3 discusses backjumping, an intermediate between simple chronological backtracking and our ideas, which are themselves presented in Section 4. An example of the dynamic backtracking algorithm in use appears in Section 5 and an experimental analysis of the technique in Section 6. A summary of our results and suggestions for future work are in Section 7. All proofs have been deferred to an appendix in the interests of continuity of exposition.

2. Preliminaries

Definition 2.1 *By a* constraint satisfaction problem (I, V, κ) *we will mean a set* I *of variables; for each* $i \in I$*, there is a set* V_i *of possible values for the variable* i*.* κ *is a set of constraints, each a pair* (J, P) *where* $J = (j_1, \ldots, j_k)$ *is an ordered subset of* I *and* P *is a subset of* $V_{j_1} \times \cdots \times V_{j_k}$*.*

A solution *to the* CSP *is a set* v_i *of values for each of the variables in* I *such that* $v_i \in V_i$ *for each* i *and for every constraint* (J, P) *of the above form in* κ*,* $(v_{j_1}, \ldots, v_{j_k}) \in P$*.*

In the example of the introduction, I is the set of states and V_i is the set of possible colors for the state i. For each constraint, the first part of the constraint is a pair of adjacent states and the second part is a set of allowable color combinations for these states.

Our basic plan in this paper is to present formal versions of the search algorithms described in the introduction, beginning with simple depth-first search and proceeding to backjumping and dynamic backtracking. As a start, we make the following definition of a partial solution to a CSP:

Definition 2.2 *Let* (I, V, κ) *be a* CSP*. By a* partial solution *to the* CSP *we mean an ordered subset* $J \subseteq I$ *and an assignment of a value to each variable in* J*.*

We will denote a partial solution by a tuple of ordered pairs, where each ordered pair (i, v) assigns the value v to the variable i. For a partial solution P, we will denote by $\overline{P}$ the set of variables assigned values by P.

Constraint-satisfaction problems are solved in practice by taking partial solutions and extending them by assigning values to new variables. In general, of course, not any value can be assigned to a variable because some are inconsistent with the constraints. We therefore make the following definition:

Definition 2.3 *Given a partial solution P to a* CSP, *an* eliminating explanation *for a variable i is a pair (v, S) where $v \in V_i$ and $S \subseteq \overline{P}$. The intended meaning is that i cannot take the value v because of the values already assigned by P to the variables in S. An* elimination mechanism ϵ *for a* CSP *is a function that accepts as arguments a partial solution P, and a variable $i \notin \overline{P}$. The function returns a (possibly empty) set $\epsilon(P, i)$ of eliminating explanations for i.*

For a set E of eliminating explanations, we will denote by $\widehat{E}$ the values that have been identified as eliminated, ignoring the reasons given. We therefore denote by $\widehat{\epsilon}(P, i)$ the set of values eliminated by elements of $\epsilon(P, i)$.

Note that the above definition is somewhat flexible with regard to the amount of work done by the elimination mechanism – all values that violate completed constraints might be eliminated, or some amount of lookahead might be done. We will, however, make the following assumptions about all elimination mechanisms:

1. They are *correct*. For a partial solution P, if the value $v_i \notin \widehat{\epsilon}(P, i)$, then every constraint (S, T) in κ with $S \subseteq \overline{P} \cup \{i\}$ is satisfied by the values in the partial solution and the value v_i for i. These are the constraints that are complete after the value v_i is assigned to i.

2. They are *complete*. Suppose that P is a partial solution to a CSP, and there is some solution that extends P while assigning the value v to i. If P' is an extension of P with $(v, E) \in \epsilon(P', i)$, then
$$E \cap (\overline{P'} - \overline{P}) \neq \emptyset \qquad (1)$$
In other words, whenever P can be successfully extended after assigning v to i but P' cannot be, at least one element of $P' - P$ is identified as a possible reason for the problem.

3. They are *concise*. For a partial solution P, variable i and eliminated value v, there is at most a single element of the form $(v, E) \in \epsilon(P, i)$. Only one reason is given why the variable i cannot have the value v.

Lemma 2.4 *Let ϵ be a complete elimination mechanism for a* CSP, *let P be a partial solution to this* CSP *and let $i \notin \overline{P}$. Now if P can be successfully extended to a complete solution after assigning i the value v, then $v \notin \widehat{\epsilon}(P, i)$.*

I apologize for the swarm of definitions, but they allow us to give a clean description of depth-first search:

Algorithm 2.5 (Depth-first search) *Given as inputs a constraint-satisfaction problem and an elimination mechanism* ϵ*:*

1. *Set* $P = \emptyset$*.* P *is a partial solution to the* CSP*. Set* $E_i = \emptyset$ *for each* $i \in I$*;* E_i *is the set of values that have been eliminated for the variable* i*.*
2. *If* $\overline{P} = I$*, so that* P *assigns a value to every element in* I*, it is a solution to the original problem. Return it. Otherwise, select a variable* $i \in I - \overline{P}$*. Set* $E_i = \widehat{\epsilon}(P, i)$*, the values that have been eliminated as possible choices for* i*.*
3. *Set* $S = V_i - E_i$*, the set of remaining possibilities for* i*. If* S *is nonempty, choose an element* $v \in S$*. Add* (i, v) *to* P*, thereby setting* i*'s value to* v*, and return to step 2.*
4. *If* S *is empty, let* (j, v_j) *be the last entry in* P*; if there is no such entry, return failure. Remove* (j, v_j) *from* P*, add* v_j *to* E_j*, set* $i = j$ *and return to step 3.*

We have written the algorithm so that it returns a single answer to the CSP; the modification to accumulate all such answers is straightforward.

The problem with Algorithm 2.5 is that it looks very little like conventional depth-first search, since instead of recording the unexpanded children of any particular node, we are keeping track of the *failed siblings* of that node. But we have the following:

Lemma 2.6 *At any point in the execution of Algorithm 2.5, if the last element of the partial solution* P *assigns a value to the variable* i*, then the unexplored siblings of the current node are those that assign to* i *the values in* $V_i - E_i$*.*

Proposition 2.7 *Algorithm 2.5 is equivalent to depth-first search and therefore complete.*

As we have remarked, the basic difference between Algorithm 2.5 and a more conventional description of depth-first search is the inclusion of the elimination sets E_i. The conventional description expects nodes to include pointers back to their parents; the siblings of a given node are found by examining the children of that node's parent. Since we will be reorganizing the space as we search, this is impractical in our framework.

It might seem that a more natural solution to this difficulty would be to record not the values that have been *eliminated* for a variable i, but those that remain to be considered. The technical reason that we have not done this is that it is much easier to maintain elimination information as the search progresses. To understand this at an intuitive level, note that when the search backtracks, the conclusion that has implicitly been drawn is that a particular node fails to expand to a solution, as opposed to a conclusion about the currently unexplored portion of the search space. It should be little surprise that the most efficient way to manipulate this information is by recording it in approximately this form.

3. Backjumping

How are we to describe dependency-directed backtracking or backjumping in this setting? In these cases, we have a partial solution and have been forced to backtrack; these more sophisticated backtracking mechanisms use information about the *reason* for the failure to identify backtrack points that might allow the problem to be addressed. As a start, we need to modify Algorithm 2.5 to maintain the explanations for the eliminated values:

Algorithm 3.1 *Given as inputs a constraint-satisfaction problem and an elimination mechanism* ϵ*:*

1. *Set* $P = E_i = \emptyset$ *for each* $i \in I$*.* E_i *is a set of eliminating explanations for* i*.*
2. *If* $\overline{P} = I$*, return* P*. Otherwise, select a variable* $i \in I - \overline{P}$*. Set* $E_i = \epsilon(P, i)$*.*
3. *Set* $S = V_i - \widehat{E}_i$*. If* S *is nonempty, choose an element* $v \in S$*. Add* (i, v) *to* P *and return to step 2.*
4. *If* S *is empty, let* (j, v_j) *be the last entry in* P*; if there is no such entry, return failure. Remove* (j, v_j) *from* P*. We must have* $\widehat{E}_i = V_i$*, so that every value for* i *has been eliminated; let* E *be the set of all variables appearing in the explanations for each eliminated value. Add* $(v_j, E - \{j\})$ *to* E_j*, set* $i = j$ *and return to step 3.*

Lemma 3.2 *Let* P *be a partial solution obtained during the execution of Algorithm 3.1, and let* $i \in \overline{P}$ *be a variable assigned a value by* P*. Now if* $P' \subseteq P$ *can be successfully extended to a complete solution after assigning* i *the value* v *but* $(v, E) \in E_i$*, we must have*

$$E \cap (\overline{P} - \overline{P'}) \neq \emptyset$$

In other words, the assignment of a value to some variable in $\overline{P} - \overline{P'}$ is correctly identified as the source of the problem.

Note that in step 4 of the algorithm, we could have added $(v_j, E \cap \overline{P})$ instead of $(v_j, E - \{j\})$ to E_j; either way, the idea is to remove from E any variables that are no longer assigned values by P.

In backjumping, we now simply change our backtrack method; instead of removing a single entry from P and returning to the variable assigned a value prior to the problematic variable i, we return to a variable that has actually had an impact on i. In other words, we return to some variable in the set E.

Algorithm 3.3 (Backjumping) *Given as inputs a constraint-satisfaction problem and an elimination mechanism* ϵ*:*

1. *Set* $P = E_i = \emptyset$ *for each* $i \in I$*.*
2. *If* $\overline{P} = I$*, return* P*. Otherwise, select a variable* $i \in I - \overline{P}$*. Set* $E_i = \epsilon(P, i)$*.*
3. *Set* $S = V_i - \widehat{E}_i$*. If* S *is nonempty, choose an element* $v \in S$*. Add* (i, v) *to* P *and return to step 2.*
4. *If* S *is empty, we must have* $\widehat{E}_i = V_i$*. Let* E *be the set of all variables appearing in the explanations for each eliminated value.*
5. *If* $E = \emptyset$*, return failure. Otherwise, let* (j, v_j) *be the last entry in* P *such that* $j \in E$*. Remove from* P *this entry and any entry following it. Add* $(v_j, E \cap \overline{P})$ *to* E_j*, set* $i = j$ *and return to step 3.*