A Uniform View of Backtracking

Fahiem Bacchus[1]

Department. of Computer Science, 6 Kings College Road, University Of Toronto, Toronto, Ontario, Canada, M5S 1A4, fbacchus@cs.toronto.edu *

Abstract. Backtracking search is a standard mechanism for solving constraint satisfaction problems (CSPs). Over the years a wide range of improvements of generic backtracking have been developed. These improvements have employed a seemingly wide range of insights, each accompanied by its own algorithmic techniques and data structures. In this paper we demonstrate that despite this seeming variety there is in fact a uniform way of viewing these improvements. In particular, we demonstrate that different backtracking algorithms can be categorized by the manner in which they discover, use, and store for future use, no-goods. This understanding can be used to provide a simplified presentation, a uniform implementation, and a simple theoretical framework for these algorithms. This unification also provides us with the appropriate conceptual apparatus to extend these algorithms in the non-binary case, and to identify new improvements.

1 Introduction

The chronological backtracking algorithm (BT) has a long history [1], and much research has been devoted to improving it. Backtracking algorithms are systematic and hence they can be used both to demonstrate that a CSP has no solution and as a basis for branch and bound optimization. For these applications and others backtracking algorithms are still among the best methods in practice. Hence, achieving a better understanding of backtracking so that, e.g., further improvements can be developed, remains important.

A vast array of improvements to BT have been developed in the literature. [2] made an important step towards systematizing these improvements by showing that their processing can be split into forward and backward phases. The forward phase consists of the processing performed when a new assignment is made, while the backward phase consists of the processing performed when backtracking occurs and assignments are undone. Furthermore, Prosser showed that the processing in these two phases was relatively independent. This allowed him to develop a new set of hybrid algorithms by mixing and matching the forward and backwards phases of different existing algorithms.

In this paper we go beyond this point of view, and show that the backwards and forward phases are not so distinct after all. In fact, both phases are doing exactly the same thing! They are simply discovering no-goods, albeit via different techniques. Furthermore, the backwards phase discovers no-goods using *exactly the same technique in all of these different algorithms*. This is in sharp contrast to Prosser's view where there

* This research was supported by the Canadian Government through their NSERC program.

were a range of more sophisticated backwards phases from simple chronological backtracking, to conflict directed backjumping. Rather we will show that the backjumping ability of an algorithm is completely determined by no-goods it discovers during by the backwards phase. And in turn, these no-goods are completely determined by the no-goods discovered during the previous forward and backwards phases, and by the manner in which these no-goods were stored.

This uniform view of the processing performed by backtracking algorithms yields a simple framework under which the various improvements can be viewed: most of these improvements are simply better ways of discovering no-goods or better ways of storing them for use in future backwards phases. [3] proved a number of important results about existing backtracking algorithms, showing for the first time that a number of popular algorithms are in fact sound and complete. However, their proofs required a number of algorithm specific techniques. Our framework yields a very systematic and simple method for proving these results. Furthermore, because this method is not algorithm specific it can easily be applied to prove the correctness of new backtracking algorithms.

The distinction between backward and forward phases has also lead to an unnecessary distinction in the field in the use of the no-goods discovered by these different phases. By recognizing that a no-good discovered in a backward phase is no different from a no-good discover in a forward phase, we can more fully utilize these no-goods. In particular, the no-goods discovered in the backward phase can be used to perform domain pruning as well as intelligent backtracking. This yields a very simple to implement improvement to a number of standard backtracking algorithms, and also allows new algorithms to be developed.

Finally, our framework is inherently non-binary. Many previous backtracking algorithms were developed specifically for binary CSP. The extension of these algorithms to general n-ary CSPs can sometimes be non-trivial: the algorithms often make subtle use of the binary restriction in their data structures or processing. In our framework these algorithms can be abstracted to the notions of discovering, storing, and using no-goods. This makes it much simpler to understand the issues involved in extending and implementing them for the n-ary case. In fact, our framework provides a uniform scheme for implementing a range of n-ary backtracking algorithms. [1]

In summary, we provide a simple a uniform view of backtracking algorithms. This view provides conceptual clarity to the range of improvements that have been developed in the literature. This is important for the task of communicating CSP technology and promoting its use in practical applications. On the theoretical side our view yields simple proof techniques for verifying the correctness of backtracking algorithms new and old. And on the practical side our view makes it easier to construct n-ary algorithms, immediately yields simple improvements to existing algorithms, and facilitates the development of new backtracking algorithms.

In the rest of the paper we will first give the background and notation we need. Then we present our framework and use it to explain the behavior of some existing backtracking algorithms. Simple proof techniques for verifying the correctness of backtracking

[1] Utilizing this uniform scheme we have developed a library of n-ary CSP algorithms that will soon be released for public use.

algorithms are presented next. And we close with a description of some of the improvements and new algorithms that our framework provides.

2 Background and Notation

A CSP consists of a set of variables $\{X_1, \ldots, X_n\}$ and a set of constraints $\{C_1, \ldots, C_m\}$. Each variable X has a finite domain of values *Dom*$[X]$, and can be assigned a value v, $X \leftarrow v$, if and only if $v \in$ *Dom*$[X]$. Let $\mathcal{A}$ be a set of assignments. No variable can be assigned more than one value, so $|\mathcal{A}| \leq n$ (i.e., the cardinality of this set is at most n). When $|\mathcal{A}| = n$ we call $\mathcal{A}$ a *complete* set of assignments. The set of variables assigned in $\mathcal{A}$ is *VarsOf*$[\mathcal{A}]$.

Each constraint C is over some set of variables *VarsOf*$[C]$, and has an arity equal to $|$*VarsOf*$[C]|$. A constraint is a set of sets of assignments: if the arity of C is k, then each element of C is a set of k assignments, one for each of the variables in *VarsOf*$[C]$. We say that a set of assignments $\mathcal{A}$ *satisfies* a constraint C if *VarsOf*$[C] \subseteq$ *VarsOf*$[\mathcal{A}]$ and there exists an element of C that is a subset of $\mathcal{A}$.

We say that $\mathcal{A}$ is *consistent* if it satisfies all constraints C such that *VarsOf*$[C] \subseteq$ *VarsOf*$[\mathcal{A}]$, i.e., it satisfies all constraints it fully instantiates. Further, $\mathcal{A}$ is a *consistent with* a *value* v (of some variable X) if $\mathcal{A} \cup \{X \leftarrow v\}$ is consistent. A *solution* to a CSP is a complete and consistent set of assignments.

Note that consistency is a local notion. It involves testing only those constraints a set of assignments fully instantiates. A *no-good* is a global notion. A set of assignments $\mathcal{A}$ is a *no-good* if there is no *unenumerated* solution containing $\mathcal{A}$. Any superset of a no-good must also be a no-good. No-goods depend on all of the constraints (they are global in nature). As a result although any inconsistent set of assignments must be a no-good, a consistency set might also be a no-good. Furthermore, there is no efficient complete decision procedure for detecting no-goods. For example, for an unsatisfiable CSP the empty set is a no-good, but proving this is co-NP complete.

A set of assignments $\mathcal{A}$ is said to be a *value-no-good* for a value v (of some variable $X \in$ *VarsOf*$[\mathcal{A}]$) if $\mathcal{A} \cup \{X \leftarrow v\}$ is a no-good. Note that a value-no-good need not be a no-good: all we know is that it will *become* a no-good if we add a particular assignment to it. However, if there is some variable $X \notin$ *VarsOf*$[\mathcal{A}]$ such that $\mathcal{A}$ is a value-no-good for every value $v \in$ *Dom*$[X]$, then we can conclude that $\mathcal{A}$ is a no-good. This follows immediately from the fact that every solution must contain an assignment $X \leftarrow v$ to X, but we already know that $\mathcal{A} \cup \{X \leftarrow v\}$ is not part of any unenumerated solution for every value v of X.

Two more consequences of our notation are worth noting. (1) If $\mathcal{A}$ is a no-good then for any assignment $X \leftarrow v \in \mathcal{A}$, we have that $\mathcal{A} - \{X \leftarrow v\}$ is a value-no-good for v. (2) We have defined no-goods in terms of the set of *unenumerated* solutions. Backtracking algorithms are systematic and are capable of enumerating all solutions. When all solutions are been enumerated it is convenient have a notion of no-good that depends only on the remaining unenumerated solutions. This will allow us to treat the two cases, searching for a single solution and searching for all solutions, identically. In the first case, there are no unenumerated solutions prior to the first one being found, and our definition of a no-good becomes identical with the standard definition. In the

second case, after each solution has been enumerated all sets contained in this and previous solutions only become no-goods.

2.1 Backtracking

A variable assignment tree is a tree in which every node is a set of assignments. The root is the empty set of assignments. At each node n a variable unassigned by n is selected. If that this variable is X with $\mathit{Dom}[X] = \{v_1, \ldots, v_k\}$, then n will have k children, with the i-th child being $n \cup \{X \leftarrow v_i\}$: the children of n extend n with all possible assignment to X. The terminal nodes are hence all possible complete sets of assignments.

Backtracking algorithms perform depth-first search in these variable assignment trees.[2] Using various techniques, like constraint checking, they avoid visiting various nodes (and the subtrees below them). We call the subtree actually explored by the algorithm the algorithm's backtracking search tree.

Since each node adds only one more assignment to its parent, each node in fact corresponds to an *ordered* set of assignments. Each assignment in the node will have an associated level at which it was made. We call the deepest assignment of a node n the assignment made by n.

The foundation of our unified framework is observation that complete backtracking algorithms will not visit a node n, or will backtrack from n, *if and only if* they are able to discover that n is a no-good.

Once n has been discovered to be a no-good it is known that no unenumerated solution extends it. Hence, there is no need to explore any of its descendants as they also are no-goods (they extend n). If some of them have already been explored, then there is no need to explore any more of them. So the algorithm can either immediately backtrack from n or never visit it in the first place. On the other hand if the algorithm never visits n or if it backtracks from n, it must have discovered that n is a no-good. If n is not known to be a no-good there might be a unenumerated solution extending n. By not visiting n or by backtracking from it the algorithm would miss this solution. Hence, for the algorithm to be complete[3] it can only avoid visiting n or backtrack from n when it has verified that n is a no-good.

3 No-goods

During search backtracking algorithms have various opportunities to discover no-goods. This fact has been exploited by a number of previous authors to develop alternative backtracking algorithms. In [2] no-goods, called conflicts, are used to generate more powerful backjumps in the conflict directed backjumping (CBJ) algorithm. In [4] no-goods, called eliminating explanations, are used to control a dynamic reordering of the branches in the search tree in the dynamic backtracking algorithm. In [5] no-goods

[2] In practice, the variable that is chosen to be instantiated next under a node n is determined heuristically by the algorithm.

[3] Completeness is the raison d'être of systematic backtracking algorithms.

are learned at deadends and used to improve the subsequent search. [6–9] also contain useful insights about the relation between no-goods and backtracking.

However, what has not previously been recognized is that, as demonstrated above, *every* backtracking algorithm uses no-goods to control its search, even if it manipulates these no-goods implicitly rather than explicitly. A unified understanding of backtracking algorithms can be achieved by categorizing the mechanisms they employ to discover, store, and use no-goods.

Discovering Standard algorithms use only a small number of different methods to discover no-goods during search.

Constraint checks. The algorithm can check if a node n is consistent when it visits it by checking that it satisfies all of the constraints it fully instantiates. If n is found to violate some constraint C, then the set of assignments to the variables of C is a no-good. This set is a subset of n. Constraint checks are utilized by algorithms like generic Backtracking (BT), backjumping (BJ), conflict-directed backjumping (CBJ), and dynamic backtracking (DBT).

Constraint propagation. When the algorithm is visiting node n it can enforce some level of local consistency among the unassigned (future) variable. By doing this it can discover that some $S \subseteq n$ is a value-no-good for a value v of an as yet unassigned variable X. Constraint propagation is utilized by algorithms like forward checking (FC), maintain arc consistency (MAC), and maintain generalized arc consistency (GAC). [4]

Detection of Solutions. If the algorithm is visiting n and it discovers that n is a solution, it can be enumerated. Once n has been enumerated, it becomes a no-good: no unenumerated solution can contain it.

Unioning value-no-goods. If a value-no-good has been discovered for every value v of a variable X, then the union of these value-no-goods, S, will be a value-no-good for every value of X. Hence S is a (new) no-good.

Unlike constraint checks and constraint propagation, successive unioning of value-no-goods is a complete method for discovering no-goods. Each union corresponds to a step in the construction of a resolution refutation (where the input clauses to the refutation are the no-goods discover at the leaves of the backtracking search tree) [8].

Using Once we have discovered a no-good we can use it to save work in the subsequent search. Current backtracking algorithms use no-goods consistencies in only two ways. Let S be a subset of the node n currently being visited by the backtracking algorithm.

Value pruning. If S is a value-no-good for a value v, then we can remove v from the domain of its variable and not return it until at least one assignment in S has been undone by backtracking. (That is, until we backtrack to the deepest assignment in S).

[4] Both constraint checks and constraint propagation also allow the algorithm to discover that various sets of assignments are consistent. This information is utilized in these algorithms to avoid redundant constraint checks. For example, in FC it is know that n's parent p is consistent with every unpruned value v of every future variable X. FC can maintain this condition at n without rechecking any constraint C with $\mathit{VarsOf}[C] \subseteq (\mathit{VarsOf}[p] \cup X)$. Our framework can be extended to account for these savings in constraint checks, but we will not do this for reasons of space.

v cannot appear in any unenumerated solution along with S. Hence, there is no need to attempt using it in any subtree extending S.

Backtracking If S is a no-good we can force backtracking. No unenumerated solutions extending S exist, so we can backtrack out of the subtree extending S. As noted above this is *always* what justifies backtracking. Say that $X \leftarrow v$ is the deepest assignment in S, and that n is the node that made this assignment. Not only does S justifies backtracking from n, but we also have that $S - \{X \leftarrow v\}$ is a value-no-good for v.

Note that in both cases if S is not a subset of the current node, then its assignments are not currently in force. Thus along the current path we cannot use S to justify backtracking or value pruning. However, some algorithms (e.g., [10]) will save S for future use in case the search ever visits a node extending S.

Storing The critical factor in most backtracking algorithms is the manner in which they store no-goods for future use. As we will demonstrate below, it is the quality of information that is stored about the value-no-goods discovered during search that *completely determines* the backjumping ability of the algorithm.

Backtracking algorithms can potentially discover an exponential number of no-goods during search. So various mechanisms must be used to manage the storage of these no-goods. In this paper we focus on storage schemes that utilize "automatic forgetting on backtrack."

In standard backtracking algorithms the (value) no-goods they discover are always subsets of the current node. Automatic forgetting on backtrack solves the storage problem by forgetting these no-goods as soon as they are no longer a subset of the current node, i.e., once we backtrack to a point where we undo one of the assignments in the no-good we forget the no-good. This ensures that at every stage of the search all of the stored no-goods will be subsets of the node the algorithm is currently visiting. One important practical implication of this is that instead of representing no-goods as sets of assignments we can represent them as sets of levels (the root is level 0): the assignments made along the current path at these levels is the no-good proper. [5]

Even with automatic forgetting on backtrack, backtracking algorithms additionally employ different compression schemes when they store the (value) no-goods they discover.

Existence. Instead of storing no-good we simply mark, in some way, that a value-no-good for various values exists. This "storage" scheme is used by generic backtracking (BT). The consequence of this is that later when we want to use the value-no-good again, all that we will be justified in assuming is that it consisted of the complete set of previous assignments.

Maximums. The level of the deepest assignment in the no-good can be stored. For example, the no-good $\{1, 4, 5\}$ (represented as a set of levels) we be stored as the number

[5] If the algorithms stores no-goods for use in future parts of the search it cannot employ automatic forgetting on backtrack nor can it store the no-goods as sets of levels (as the assignments made at these levels might change by the time the no-good is to be used again). Such algorithms must find some other scheme to avoid storing too many no-goods. Nevertheless, even in these algorithms, the no-goods are still discovered using the same mechanisms and can still be put to the same uses as described above.

5. The consequence of this is that later when we want to use the no-good again, all that we will be justified in assuming is that $\{1, 2, 3, 4, 5\}$ is a no-good. We have forgotten that levels 2 and 3 were not originally included in the no-good. BJ, FC, MAC, and GAC (maintain generalized arc consistency [11]) use this compression scheme when storing no-goods.

One value-no-good per variable. A common scheme to have one value-no-good set per variable, and to store in this set the running union of the value-no-goods discovered for the variable's values. CBJ, e.g., uses this scheme. BJ uses this scheme in conjunction with the previous one: it stores a single deepest level over all of the value-no-goods for the values for the variable. The alternative is to maintain each value-no-good in a separate set. This alternative is used by dynamic backtracking and by the backtracking algorithms developed in [12].

4 Using the framework

4.1 Understanding Previous Algorithms

BT, BJ and CBJ. A useful illustration of our framework is provided by the three algorithms, BT, BJ, and CBJ. These algorithms are *identical* in the way they discover no-goods (via constraint checks and unioning of value-no-goods) and in the way they use these no-goods (exclusively to perform backtracking). The only difference lies in how they store no-goods, which in turn determines their backtracking abilities. An example best illustrates the difference.

Let n be a consistent node at level 10 that is currently being visited. Say that at level i, $1 \leq i \leq 10$, the assignment $X_i \leftarrow v_i$ was made. Let variable X with $\mathit{Dom}[X] = \{a, b, c\}$ be the variable next instantiated by n's children, with $c_1 = n \cup \{X \leftarrow a\}$, $c_2 = n \cup \{X \leftarrow b\}$, and $c_3 = n \cup \{X \leftarrow n\}$ being n's three children. Finally, let C_1, C_2, and C_3 be constraints with $\mathit{VarsOf}[C_1] = \{X_1, X_3, X\}$, $\mathit{VarsOf}[C_2] = \{X_3, X\}$, and $\mathit{VarsOf}[C_3] = \{X_2, X_7, X\}$. Each of these algorithms will proceed to visit each of n's children and check whether or not they are consistent. Say that each child c_i violates constraint C_i. Then the sets $\{X_1 \leftarrow v_1, X_3 \leftarrow v_3, X \leftarrow a\}$, $\{X_3 \leftarrow v_3, X \leftarrow b\}$, and $\{X_2 \leftarrow v_2, X_7 \leftarrow v_7, X \leftarrow c\}$ will be discovered to be no-goods. Each of these no-goods yields a value-no-good for one of X's values.

CBJ unions these value-no-good into a single conflict set. So after visiting all of the children it has discovered that $\{X_1 \leftarrow v_1, X_2 \leftarrow v_2, X_3 \leftarrow v_3, X_7 \leftarrow v_7\}$ is a no-good. It then uses this no-good to immediately backtrack to level 7 where it undoes the assignment $X_7 \leftarrow v_7$. Furthermore, it has also discovered that $\{X_1 \leftarrow v_1, X_2 \leftarrow v_2, X_3 \leftarrow v_3\}$ is a value-no-good for the v_7 of X_7. Say that X_7 has no other values, then this single value-no-good covers all of X_7's values, and it must also be a no-good. CBJ uses this no-good to immediately backtrack to level 3 where it undoes the assignment $X_3 \leftarrow v_3$.

Under the same circumstances, BJ only stores the maximum level over the value-no-goods it has discovered for X's values, i.e., level 7. Hence after visiting all of the children it is only able to discover the weaker no-good $\{X_1 \leftarrow v_1, \ldots, X_6, \leftarrow v_6, X_7 \leftarrow v_7\}$. It then uses this no-good to immediately backtrack to level 7, and since X_7 has

no other values it can conclude that $\{X_1 \leftarrow v_1, \ldots, X_6 \leftarrow v_6\}$ is a no-good, and use this no-good to backtrack to level 6. BJ looses much of the backtracking ability of CBJ simply because it forgets information about the no-goods it discovers. In particular, the no-goods it uses to justify backtracking always include a complete sequence of levels from 1 to the maximum level it stored. Hence, any variable backtracked to will have at least one value whose value-no-good is the complete set of prior levels, and BJ cannot subsequently make a non-trivial backtrack from such variables. That is, BJ does not have the no-goods to make non-trivial backtracks at the internal nodes its search tree.

BT stores even less information. After visiting all of the children it only remembers that a value-nogood exists for all of X's values, and the only conclusion it can justify is that $\{X_1 \leftarrow v_1, \ldots, x_{10} \leftarrow v_{10}\}$ is a no-good. Hence it can only justify backtracking to level 10. In fact, it is not difficult to see that the full set of previous levels is the only no-good that BT is capable of discovering since it forgets so much information about the no-goods it learns at the leaf nodes. Hence BT can never justify anything stronger than a backtrack to the previous level.[6]

Constraint Propagation. Constraint propagating algorithms like FC, MAC and GAC discover no-goods involving values of future variables. For example, at level i GAC may discover that an assignment $X \leftarrow v$ has no supporting tuple in a constraint C where as it did have support in C at level $i-1$. Hence, the assignments made at levels 1 through i must contain a value-no-good for v, and this value-no-good must have i as its maximum level.

Various value-no-goods for v are easy to compute. The simplest is the full set of levels $\{1, \ldots, i\}$. This is the value-no-good utilized by the FC, MAC and GAC algorithms. They utilize this value-no-good to prune v from *Dom*$[X]$ at level i, restoring it only when the assignment at level i is undone. They store the level v is pruned, which corresponds to storing the maximum of the discovered value-no-good. A leaf node ℓ for these algorithms is a node at which constraint propagation deletes all remaining values from the domain of some future variable X, i.e., a value-no-good has been discovered for each of X's values. Since at least one value was deleted by the assignment at level ℓ, at least one of these value-no-goods has been stored as the full set of levels $\{1, \ldots, \ell\}$. Hence, when these value-no-goods are (implicitly) unioned the strongest conclusion is that the full set of prior assignments is a no-good, and these algorithms can only justify backtracking to the previous level. This also implies that recursively any variable backtracked to will have at least one value whose value-no-good is the complete set of prior levels. In other words, like BJ, FC, MAC and GAC can never justify non-trivial backtracks because of limited information they store about the value-no-goods they discover during their constraint propagation phase.

Better value-no-goods can be discovered and stored, and various methods have been developed for doing this (e.g., [11]). Algorithm that do value pruning pose the difficulty that a value-no-good for a value v of a variable X can be discovered at level i, in-

[6] Because BT and BJ discover such simple no-goods (complete sequences of levels) they do not need to maintain these no-goods explicitly. Thus in their specification one will not see any explicit manipulation of no-goods. Nevertheless, their processing is determined by the implicit no-goods they are discovering.

validated on backtrack, and then a different value-no-good for v discovered when we descend the search tree again. If only a running union of the value-no-goods is maintained, additional information will have to be maintained so that value-no-goods can be deleted from this union when they are invalidated by backtracking. In [2]'s version of FC-CBJ the levels at which the deleted values were pruned are unioned into the conflict set just prior to backtrack. This method only works for binary CSPs. If $v \in \mathit{Dom}[X]$ is pruned by FC at level i, the assignment $X \leftarrow v$ must have violated a binary constraint whose only other variable was assigned at level i. Hence, $\{i\}$ is the *complete* value-no-good for v, and no information is lost by storing only this pruning level. If v was pruned by a non-binary constraint, then there will be other assignments in its value-no-good, and the pruning level will be only the maximum of the value-no-good. Thus in the n-ary case if only the pruning level is stored, no non-trivial backtracks can be justified. [11] provides one method for avoiding this problem.

However, a much cleaner scheme is to store the value-no-goods for each value in a separate set. This requires more space, but is quite practical given the amount of RAM on todays machines. With separately stored value-no-goods we can update then on an individual basis, computing their union only when we backtrack. Another key advantage of separate value-no-goods is that they can be manipulated in more sophisticated ways to discover better no-goods. For example, say that we have a value-no-good for all values of the variable X, and that the maximum level in these sets is ℓ where the assignment $X_\ell \leftarrow v_\ell$ was made. Then we can backtrack to level ℓ and there set the union of X's value-no-goods minus ℓ to be a new value-no-good for v_ℓ. However, suppose that there is a binary constraint C between X_ℓ and X. Then it can easily be proved that a better value-no-good for v_ℓ is the union of the value-no-goods for all values of X that are consistent with v_ℓ under C. This is a union over a smaller set of value-no-goods, and thus it can be smaller. If it is smaller, it might support superior backjumping from level ℓ. This method and some others have been used in the backtracking algorithms presented in [12], but many other methods that can be developed.

4.2 Uniform proofs of completeness

Consider an algorithm that does a depth-first search of the entire variable assignment tree. It does not stop to do constraint checking, rather it proceeds directly to visit all complete assignments. At these leaf nodes it then checks whether or not the node is consistent. If it is, it reports it to be a solution. Clearly this algorithm is sound and complete, as it checks every possible complete assignment. That is, any solution it reports is a solution (sound) and it will report all solutions (complete).

Say that $S = \{X_1 \leftarrow v_1, \ldots, X_k \leftarrow v_k\}$ is a no-good. If we allow the above algorithm do either or both of (1) immediately backtrack from any node n extending S, and (2) at any node n that makes $k-1$ of the assignments in S prune the value used in the remaining assignment from the domain of its variable, then soundness and completeness is retained.

As we have pointed out all standard backtracking algorithms can be viewed as discovering no-goods and using them in these two ways. That is, at this level of abstraction all standard backtracking algorithms are exactly the above algorithm taking advantage of discovered no-goods to optimize their search.

Thus all we need to do to prove a backtracking algorithm sound and complete is to demonstrate that the no-goods it uses to justify value-pruning and backtracking are in fact sound no-goods. This means that the algorithm must use sound mechanisms to discover these no-goods, and if it subsequently compresses them, it must take proper account of the information lost.

4.3 Improving Backtracking Algorithms

One possible improvement to existing backtracking algorithms was described above: store separate value-no-goods and develop improved methods of manipulating them to discover new no-goods. More generally, external algorithms can also be used; if these algorithms return no-goods then they can easily be plugged into the backtracking process.

As pointed out above, each of the no-goods used for backtracking also yields a value-no-good for the value assigned at the level to which backtracking occurs. The existence of this new value-no-good has not previously been recognized in the literature, but once it is apparent that it exists it can be put to better use. This yields a simple to implement improvement for any backtracking algorithm that computes non-sequential no-goods for backtracking, e.g., any algorithm that does CBJ type backtracking. In particular, when we use the no-good S to justify backtracking to level ℓ, the deepest level in S, we can prune the value assigned at level ℓ back to the deepest level in $S - \{\ell\}$. For example, say that at level 10 the no-good $\{X_1 \leftarrow v_1, X_2 \leftarrow v_2, X_3 \leftarrow v_3, X_7 \leftarrow v_7\}$ is computed, where X_i is the variable assigned at level i. Then this no-good can be used to immediately backtrack to level 7, *and there prune value* v_7 *of variable* X_7 *back to level 3*: $\{X_1 \leftarrow v_1, X_2 \leftarrow v_2, X_3 \leftarrow v_3\}$ is the newly discovered value-no-good for v_7 justifying this pruning. Thus even if we once again attempt to assign a value to X_7 we will not need to consider v_7 until we backtrack to level 3. This could yield an exponential savings, as it might require exponential search to backtrack out of a subtree rooted by $X_7 \leftarrow v_7$ if this value was not pruned.

As a small test of the practical impact of this idea we altered an implementation of GAC-CBJ [13] so that it used the conflicts it computes for domain pruning as well as for backtracking. All of the apparatus for domain pruning and conflict set is already in place in GAC-CBJ, and thus the changes amounted to adding only 4 simple lines of code. To test the change we then ran the 6 hardest the logistic planning problems that came with their distribution. The results are given in following Table. This is only a small test, and the gains are moderate (the maximum being a 3 fold improvement for problem 27), and minor losses can occur. Nevertheless, considering the extra coding effort of 4 more lines, it is a worthwhile improvement inspired by the unified view of backtracking algorithms we have presented here.

Problem	GAC-CBJ	GAC-CBJ+P
6	12.38	8.22
15	133.22	101.29
18	811.17	809.13
20	31.53	25.81
25	14.48	14.47
27	64.82	22.02

The numbers in the table are CPU seconds require to find the shortest solution to the planning problem on a Pentium III 500MHz machine with 1GB RAM. GAC-CBJ+P is the version of GAC-CBJ that makes additional use of conflicts to pruning values backtracked to.

References

1. J. R. Bitner and E. Reingold. Backtracking programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.
2. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3), 1993.
3. Grzegorz Kondrak and Peter van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.
4. Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
5. Daniel Frost and Rina Dechter. Dead-end driven learning. In *Proceedings of the AAAI National Conference*, pages 294–300, 1994.
6. R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.
7. J. de Kleer. A comparison of ATMS and CSP techniques. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 290–296, Detroit, Mich., 1989.
8. David Mitchell. Hard prolems for csp algorithms. In *Proceedings of the AAAI National Conference*, pages 398–405, 1998.
9. A. B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, University of Oregon, 1995.
10. R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
11. X. Chen. *A Theoretical Comparison of Selected CSP Solving and Modeling Techniques*. PhD thesis, University of Alberta, 2000.
12. Fahiem Bacchus. Extending forward checking. Submitted to this conference, 2000.
13. P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the AAAI National Conference*, pages 585–590, Orlando, Florida, 1999.