Chapter 2

Backtracking

There are various ways of finding solutions to CSPs, and these techniques fall into two categories: systematic, and non-systematic. *Systematic* techniques search for a solution systematically, thus once they have completed their search we know whether or not the CSP has a solution. That is, these techniques can be used to prove that a CSP is unsolvable. Similarly, systematic techniques can also be used to enumerate all solutions and as a basis for branch and bound search that can be used to find provably optimal solutions.

Non-systematic techniques, on the other hand employ search techniques that may or may not find solutions, and these solutions may or may not be optimal. Thus non-systematic techniques are solving an easier problem: they are not required to prove that their solution is optimal, and if they can't find a solution this does not constitute a proof that a solution does not exist. The advantage of non-systematic techniques is that since they are solving a much easier problem, they can often be much faster than systematic techniques.

Backtracking forms the basis of most systematic techniques. In this chapter we will describe backtracking on finite domain CSPs.

2.1 Backtracking—Tree Search over sets of Partial Assignments

The Search Tree. The basic idea of backtracking is quite simple. It involves searching in the space of sets of feasible variable assignments. This search space is explored by searching in a tree in which every node makes an assignment to a variable. In particular, starting at the root where no variables have been assigned, each node n makes a new variable assignment. The set of assignments make from the root to n defines a sub-space that must be searched in the sub-tree below n. Say that at the set of assignments made at node n is $\mathcal{S} = \{V_1 \leftarrow v_1, \ldots, V_k \leftarrow v_k\}$. Then the sub-space of assignments that must be explored below n are all the feasible sets of assignments that extend $\mathcal{S}$.

The children of n must partition the sub-space below n. Thus, if we find all of the solutions that exist in each of the sub-spaces defined by n's children, then the union of these sets of solutions

will be all of the solutions that exist in the sub-space below n. Hence, by searching below each child of the root, we can find all of the CSP's solutions. Of course, we can stop the search as soon as we find one solution, if that is all we want to find.

There are many ways of partitioning the search space below n. But the most common method is to pick an uninstantiated variable, and partition the space by branching on each possible assignment to that variable. Note that we must pick an uninstantiated variable, otherwise the set of assignments will no longer be feasible. Thus, for example, if V' is an uninstantiated variable, and $\{a, b, c\}$ are its possible values, we could define the children of n to be the nodes n_1, n_2, and n_3 where the assignments $V' \leftarrow a$, $V' \leftarrow b$, and $V' \leftarrow c$ respectively are made. Hence, below n_1, for example, we would explore the sub-space of feasible assignments that extend $\{V_1 \leftarrow v_1, \ldots, V_k \leftarrow v_k, V' \leftarrow a\} = \mathcal{S} \cup \{V' \leftarrow a\}$.

The sub-CSP There is another way of viewing the sub-space below each node of the search tree. Each node n has an associated set of assignments, the assignments made at n and along the path from the root to n. The sub-space below n can also be viewed as solving a sub-CSP.

Let $\mathcal{A} = \{V_1 \leftarrow v_1, \ldots, V_k \leftarrow v_k\}$ be any set of assignments. This set of assignments generates a sub-CSP that is a reduction of the original CSP. Say that the original problem was $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{V}$ is a set of variables $\{V_1, \ldots, V_n\}$, $\mathcal{D}$ is a set of domains for these variable $\{Dom[V_1], \ldots, Dom[V_n]\}$, and $\mathcal{C}$ is a set of constraints $\{C_1, \ldots, C_m\}$. Then the reduction of $\mathcal{P}$ by $\mathcal{A}$, $\mathcal{P}_\mathcal{A}$, is a new CSP $\mathcal{P}_\mathcal{A} = \langle \mathcal{V}_\mathcal{A}, \mathcal{D}_\mathcal{A}, \mathcal{C}_\mathcal{A} \rangle$ where:

1. $\mathcal{V}_\mathcal{A} = \mathcal{V} - \mathit{VarsOf}[\mathcal{A}]$, i.e., the unassigned variables.

2. $\mathcal{D}_\mathcal{A}$ the domains of the unassigned variables.

3. $\mathcal{C}_\mathcal{A}$: for each constraint C some of whose variables include assigned variables, i.e., C such that $\mathit{VarsOf}[C] \cap \mathit{VarsOf}[\mathcal{A}] \neq \emptyset$, we replace C with a reduction of C, and for each constraint none of whose variables have been assigned, we leave unchanged.

   The reduction of C by $\mathcal{A}$ is simply the constraint $C_\mathcal{A}$ which is over the unassigned variables of C only (thus we reduce the arity of the constraint by the number of its variables that have been assigned by $\mathcal{A}$). A set of assignments a (over the unassigned variables of C) is a member of the reduced constraint $C_\mathcal{A}$ if and only if $a \cup \mathcal{A}$ was a member of C.

   For example, say that $\mathit{VarsOf}[C] = \{V_1, V_2, V_3, V_4\}$ and that $\mathcal{A} = \{V_1 \leftarrow a, V_4 \leftarrow b\}$. Then $\mathit{VarsOf}[C_\mathcal{A}] = \{V_3, V_4\}$ and a set of assignments $\{V_2 \leftarrow x, V_3 \leftarrow y\} \in C_\mathcal{A}$ if and only if $\{V_1 \leftarrow a, V_2 \leftarrow x, V_3 \leftarrow y, V_4 \leftarrow b\} \in C$.

It should be noted that the constraints that are fully instantiated by $\mathcal{A}$ are removed (their arity is reduced to zero). Furthermore, many unary constraints may well be created in the sub-CSP.

Constraint Checking The key idea behind searching a tree of partial sets of assignments is that CSPs typically involve constraints over subsets of the variables. That is, although the CSP may

well contain a constraint C such that $VarsOf[C]$ is the entire set of variables (such constraints are sometimes called *global* constraints), it will usually also contain constraints C' where $VarsOf[C']$ is a much smaller set of variables.

We can take advantage of this during the tree search by checking constraints as soon as we have made an assignment to some or all of its variables. Suppose that at node n we have made the set of assignments $\{V_1 \leftarrow a, V_4 \leftarrow b, V_7 \leftarrow c\}$, and that there is a constraint C_{V_1,V_7} between V_1 and V_7 which does not contain the assignments $\{V_1 \leftarrow a, V_7 \leftarrow c\}$. Then we know that no extension of this set of assignments can possible satisfy all of the CSP's constraints—all such extensions will violate C_{V_1,V_7}. Thus we do not need to explicitly explore the sub-space below node n, and since this sub-space can have size exponential in the number of unassigned variables we can save a considerable amount of work in the search.

Checking constraints in order to avoid searching sub-trees is the essential component of backtracking search, and it makes backtracking much more efficient that the naive approach of systematically testing all possible complete sets of assignments (a method sometimes called "generate and test").

Optimizations Generic Backtracking (BT) is the simplest form of backtracking search. It employs the simple technique of checking constraints only when they have been fully instantiated. In particular, it checks whether or not the set of assignments at a node n is *consistent*.[1] However generic backtracking can easily be improved. Furthermore, these improvements yield such performance gains that generic backtracking is hardly ever employed in practice. The improvements to BT fall into two categories: constraint propagation, and intelligent backtracking.

Constraint Propagation Constraint propagation involves enforcing local consistency in the sub-CSP below a node. This can can reduce the size of the sub-CSP, and sometimes it can provide an immediate demonstration that the sub-CSP contains no-solutions thus allowing us to completely avoid searching the tree below a node.

Intelligent Backtracking During our search below a node n we can keep track of the "reasons" the search failed to find a solutions. Sometimes these reasons have nothing to do with the assignment made at n, and thus there is no need to search below n's siblings—the reason for failure remains valid in the sub-spaces below these siblings nodes as the only difference between them and n is the assignment made at n. Thus by tracking these reasons for failure we can sometimes backtrack to a level above n where the reason we discovered for failure is no longer valid.

Improvements to BT have been developed in the CSP literature over a long period of time. Similarly it has taken sometime to understand the structure of these improvements and the relationships between them. In this chapter we present a uniform way of looking at all of these improvements. However, this means that our presentation of some of these improved algorithms is often quite different from their original presentations.

[1] Remember that consistency is defined as satisfying all fully instantiated constraints, see Chapter 1.

2.2 No-Goods

The key notion used to unify these improvements is that of a *no-good*, or a *conflict*. A no-good is simply a set of assignments that is not contained in any solution. A CSP will have many no-goods, in general the number of distinct no-goods is exponential in the number of variables. Some of these no-goods are obvious, e.g., the complement of every constraint is a set of no-goods. However, some of these no-goods are very hard to find. The most obvious case is the question of whether or not the empty set is a no-good. It is a no-good if and only if the CSP has no solution. Since finding a solution to CSPs is an NP-complete problem, showing that a CSP has no solution is co-NP, and hence it is highly unlikely to be solvable in less that exponential time.

In finite domain CSPs we can also view no-goods to be propositional assertions. In particular, for any finite-domain CSP we can define a propositional language in which the atomic propositions are all the individual variable assignments. For example, "$V_1 \leftarrow a$" becomes an atomic proposition. The meaning of these propositions is obvious—"$V_1 \leftarrow a$" asserts that V_1 has the value a.

Once we have a symbol in the propositional language for each of the atomic propositions, we as usual, are then allowed to form sentences using conjunction $\wedge$, disjunction $\vee$, and negation $\neg$, and implication $\Rightarrow$. For simplicity of notation, however, we will denote the negated atomic proposition $\neg(V_i \leftarrow x)$ by the simpler $V_i \not\leftarrow x$.

2.2.1 Propositional Encoding of a Finite Domain CSP

With the propositional language so defined, we can then encode any finite domain CSP directly as a collection of propositional formulas. In particular, the CSP is encoded as the conjunction of three sets of formulas:

Primitive Constraints We convert each constraint of the CSP into a propositional formula. Let C be a constraint. We create a conjunction from each element of C. Let a be an element of C; a is a set of assignments, say $a = \{V_1 \leftarrow x_1^0, \ldots, V_k \leftarrow x_k^0\}$. From a we create the conjunction $(V_1 \leftarrow x_1^0) \wedge \cdots \wedge (V_k \leftarrow x_k^0)$. Then we disjoin together each of these conjunctions. Hence, each constraint of the CSP C is translated into a formula of the form

$$\begin{array}{ll} & (V_1 \leftarrow x_1^0) \wedge \cdots \wedge (V_k \leftarrow x_k^0) \\ \vee & (V_1 \leftarrow x_1^1) \wedge \cdots \wedge (V_k \leftarrow x_k^1) \\ \vdots & \\ \vee & (V_1 \leftarrow x_1^\ell) \wedge \cdots \wedge (V_k \leftarrow x_k^\ell) \end{array}$$

Exclusivity Each variable can only be assigned a single value. Let V be a variable of the CSP with $Dom[V] = \{x_1, \ldots, x_k\}$. Then for V we create k propositional formulas $V \leftarrow x_1 \Rightarrow (V \not\leftarrow x_2 \wedge \cdots \wedge V \not\leftarrow x_k)$, $V \leftarrow x_2 \Rightarrow (V \not\leftarrow x_1 \wedge V \not\leftarrow x_3 \cdots \wedge V \not\leftarrow x_k)$, ..., $V \leftarrow x_k \Rightarrow (V \not\leftarrow x_1 \wedge \cdots \wedge V \not\leftarrow x_{k-1})$.

Exhaustiveness In a solution each variable must be assigned a value. This yields a formula for each variable. For example, for the variable V used in the previous example we would obtain the formula $V \leftarrow x_1 \vee V \leftarrow x_2 \vee \cdots \vee V \leftarrow x_k$.

Let us call these three collections of formulas C, X and E respectively. Together these formulas capture all of the structure in a finite domain CSP, and the soundness of propositional reasoning can be used to observe a number of things.

First the set of solutions to the CSP is implied by these formulas. That is,

$$\begin{aligned}(C \wedge X \wedge E) \quad \Rightarrow \quad & (V_1 \leftarrow x_1^0 \wedge \cdots \wedge V_n \leftarrow x_n^0) \\ & \vee (V_1 \leftarrow x_1^1 \wedge \cdots \wedge V_n \leftarrow x_n^1) \\ & \cdots \\ & \vee (V_1 \leftarrow x_1^\ell \wedge \cdots \wedge V_n \leftarrow x_n^\ell)\end{aligned}$$

Where each clause represents one of the ℓ distinct solutions.

Second, the CSP has no solution if and only if $C \wedge X \wedge E$ is unsatisfiable.

And third, the negation of every no-good is implied by $C \wedge X \wedge E$. For example, $C \wedge X \wedge E \Rightarrow \neg(V_1 \leftarrow x_1, V_2 \leftarrow x_2)$ if and only if the set of assignments $\{V_1 \leftarrow x_1, V_2 \leftarrow x_2\}$ is a no-good.

Of course, it is no easier to reason with the propositional encoding that it is to reason directly with the original CSP encoding. The main purpose of the propositional encoding is that it can make various properties of CSPs easier to demonstrate. Among these properties are various ways in which no-goods can be combined to create new no-goods.

2.2.2 No-good Reasoning

Using this propositional view of the CSP we can make some simple observations about how no-goods can be manipulated. In particular, the propositional view shows that we can compute new no-goods by some simple reasoning steps.

Unioning a Set of No-goods Given a collection of no-goods which cover all values of a variable, we can resolve these with an exclusivity clause to produce a new no-good. This is best illustrated in an example.

Let V be a variable of the CSP with $Dom[V] = \{a, b, c, d\}$. And suppose we have the following four no-goods:

1. $\{V \leftarrow a, V_1 \leftarrow x_1\}$

2. $\{V \leftarrow b, V_2 \leftarrow x_2, V_4 \leftarrow x_4\}$

3. $\{V \leftarrow c, V_3 \leftarrow x_3\}$

4. $\{V \leftarrow d, V_3 \leftarrow x_3, V_4 \leftarrow x_4\}$

Since these no-goods cover all the possible assignments to V, we can union them together, removing all assignments to V, to obtain the new no-good

5. $\{V_1 \leftarrow x_1, V_2 \leftarrow x_2, V_3 \leftarrow x_3, V_4 \leftarrow x_4\}$.

This operation can be easily justified by the following propositional reasoning. The four no-goods yield the four formulas

1. $\neg(V \leftarrow a \wedge V_1 \leftarrow x_1)$

2. $\neg(V \leftarrow b \wedge V_2 \leftarrow x_2 \wedge V_4 \leftarrow x_4)$

3. $\neg(V \leftarrow c \wedge V_3 \leftarrow x_3)$.

4. $\neg(V \leftarrow d \wedge V_3 \leftarrow x_3 \wedge V_4 \leftarrow x_4)$

These simplify to the clauses

1. $V \not\leftarrow a \vee V_1 \not\leftarrow x_1$

2. $V \not\leftarrow b \vee V_2 \not\leftarrow x_2 \vee V_4 \not\leftarrow x_4$

3. $V \not\leftarrow c \vee V_3 \not\leftarrow x_3$.

4. $V \not\leftarrow d \vee V_3 \not\leftarrow x_3 \vee V_4 \not\leftarrow x_4$

The exhaustiveness clauses for V yields the formula

5. $V \leftarrow a \vee V \leftarrow b \vee V \leftarrow c \vee V \leftarrow d$

So we can resolve clause 5 against clauses 1-4[2] in a sequence of steps (including factoring out duplicate literals) to produce the new clause

6. $V_1 \not\leftarrow x_1 \vee V_2 \not\leftarrow x_2 \vee V_3 \not\leftarrow x_3 \vee V_4 \not\leftarrow x_4$

This is equivalent to the formula

6. $\neg(V_1 \leftarrow x_1 \vee V_2 \leftarrow x_2 \vee V_3 \leftarrow x_3 \vee V_4 \leftarrow x_4)$

And is clearly equivalent to the no-good produced by set union.

[2]Resolution in the propositional case is the sound rule of inference that combines two clauses together to produce a third. In particular, we can resolve the two clauses $A \vee B_1 \vee \cdots \vee B_n$ and $\neg A \vee C_1 \vee \cdots \vee C_m$, to produce the new clause $B_1 \vee \cdots \vee B_n \vee C_1 \vee \cdots \vee C_m$, where we have removed the conflicting proposition and disjoined the remaining propositions. In addition we can remove duplicate literals from the new clause in a process called factoring.

Constraint Filtered Unioning of no-goods We can refine the first case if we happen to have a constraint between two variables. Say that we have a constraint C_{V_1,V_2} between variables V_1 and V_2. Furthermore, say that the *supports* for $V_1 \leftarrow a$ on V_2 are $V_2 \leftarrow a$ and $V_2 \leftarrow b$ (i.e., only these assignments satisfy the constraint given that $V_1 \leftarrow a$). Then suppose we have the following collection of no-goods:

1. $\{V_2 \leftarrow a, V_3 \leftarrow x_3\}$

2. $\{V_2 \leftarrow b, V_3 \leftarrow x_3, V_4 \leftarrow x_4\}$

3. $\{V_2 \leftarrow c, V_5 \leftarrow x_5\}$

4. $\{V_2 \leftarrow d, V_5 \leftarrow x_5, V_6 \leftarrow x_6\}$

Then a new no-good is

5. $\{V_1 \leftarrow a, V_3 \leftarrow x_3, V_4 \leftarrow x_4\}$

In other words, if the set of no-goods cover the supports for an assignment, e.g., $V_1 \leftarrow a$ we can union these no-goods together, removing the assignments to the supporting variable, adding the supported assignment, to obtain a new no-good.

Again we can use the propositional encoding to justify the operation. In particular, we have from the constraint between V_1 and V_2 the formula $V_1 \leftarrow a \Rightarrow V_2 \leftarrow a \vee V_2 \leftarrow b$. In clause form this is $V_1 \not\leftarrow a \vee V_2 \leftarrow a \vee V_2 \leftarrow b$. We can resolve this against the two clauses produced by the first two no-goods:

1. $V_2 \not\leftarrow a \vee V_3 \not\leftarrow x_3$

2. $V_2 \not\leftarrow b \vee V_3 \not\leftarrow x_3 \vee V_4 \not\leftarrow x_4$

To obtain the no-good

1. $V_1 \not\leftarrow a \vee V_3 \not\leftarrow x_3 \vee V_4 \not\leftarrow x_4$.

This is the same as the no-good computed by set manipulations.

Other no-goods It is clearly possible to produce other no-goods as in general all no-goods can be generated by propositional reasoning. However, these no-goods will have to be produced during the tree search, and thus we can only utilize easily computed no-goods. Both types of no-goods mentioned here will be utilized in the tree search algorithms developed in this chapter. It is an open question as to whether on not other useful types of no-goods can be produced to aid tree search algorithms.

2.3 Depth-First Search

Typically the tree of partial assignments is searched depth-first. Breadth-first, best-first, and other types of search algorithms could be utilized. However, the best performing algorithms need to store a fair amount of data at every node (this extra data is used to optimize various computations), and the best-first algorithms can require storing a large number of nodes.

Depth-first search will only ever need to store a linear number of nodes (linear in the number of variables of the CSP), and thus have a clear space advantage. Furthermore, best-first and breadth-first search have their most significant advantage of depth-first search when the search space has many cycles. In such search spaces depth-first search may visit the same node an exponential number of times, significantly degrading its performance.[3] The search space of feasible partial sets of assignments, on the other hand, has no cycles. At each node we generate an partition of the remaining search-space (partitions are exclusive and exhaustive), thus the sub-space below each node is exclusive of all the other sub-subspaces in the search tree. It is perhaps for this reason that all of the backtracking algorithms developed have utilized depth-first search.

The Current Node During the search we visit various nodes in the tree, sometimes we descend to visit the children of a node, and sometimes we backtrack to return to visit one of the nodes ancestors or one of its siblings. We call the node that the search is currently visiting the *current node* and we call the set of assignments made at the current node, i.e., the assignments made along the path from the root the current node the *current assignments*. We call the set of uninstantiated variables at the current node the *future variables*, and the set of instantiated variables the *past variables*. Sometimes we call the variable assigned at the current node the *current variable*.

2.3.1 The No-goods encountered during Search

As we perform the depth-first search three different types of no-goods are frequently encountered.

1. During search the algorithms will check the consistency of the current assignments.[4]. If we find that this set of assignments is inconsistent because it fails to satisfy a constraint, say C, we have discovered a no-good that lies in the complement of C. For example, if C is over the variables V_1 and V_2, and the current set of assignments makes an assignment to V_1 and V_2 that violates C, then the this pair of current assignments is a no-good.

2. Search allows us to discover no-goods that cover every value of a variable. We can then union these no-goods together to obtain a new no-good as described above. Say that the children of a node n cover all assignments to the variable V (i.e., for each of V's possible values there is a child of n that assigns V that value). Then after we have completed the search of all of n's children we will have computed a set of no-goods that cover all of V's

[3] Depth-first search has no memory of the nodes it has previously visited.

[4] More generally, we may check the consistency of a set of assignments that extend the current assignments

possible values. We can union together these no-goods, discarding the assignments to V, to obtain a new no-good. This no-good can be viewed as having been produced by resolving together the no-goods produced by the search. If we have a constraint between two variables, we can also resolve together no-goods filtered by the constraint, as described above.

3. Many of the backtracking algorithms perform constraint propagation as they perform search. The idea here is that as we make new assignments a reduced CSP is produced. The search algorithm can then enforce some degree of local consistency in the reduced CSP. Enforcing local consistency in this way is called constraint propagation. As we have seen many forms of local consistency can be achieved by pruning values. In this case these values will be pruned from the domains of the future variables.

   There is an implicit no-good involving each of these pruned values. In particular, the value is pruned because given some subset of the current assignments the value cannot participate in any solutions. Hence, if we can identify the subset of the current assignments that were the "reason" for the value pruning, we will be able to construct a new no-good consisting of that subset and the pruned value.

The problem we encounter is that an exponential number of no-goods are encountered during search. Furthermore, many of these no-goods will not be useful in any other part of the search. However, many of the improvement to generic backtracking can be viewed as mechanisms for identifying and storing no-goods that can be use to optimize the remaining search.

Even with these mechanisms, however, we can still accumulate an exponential number of no-goods. Hence, some scheme is needed to delete many of these no-goods. A common technique used in conjunction with depth-first search is to remember no-goods that contain assignments from the current set of assignments (they might also contain a small number of assignments to the future variables). In this case we can employ automatic-forgetting on backtrack. That is, we automatically delete the no-good as soon as one of the current assignments it contains is undone by backtracking.

No-Goods stored as Sets of Levels The no-goods tracked by the algorithms we will discuss in this chapter contain at most one assignment to a future variable. The rest of their assignments are totally contained in the current assignments. Thus a common organizational scheme is to associate these no-goods with the value of the future variable they assign. That is, if the no-good contains the assignment $V \leftarrow a$ where V is a future variable, we will associate it with the value a of V (in the implementation this will mean that a of V will contain a pointer to this no-good). Then since the remaining assignments in the no-good are all contained in the current assignments, we will store these assignments as a set of integer levels. Each of these current assignments was made at a particular level along the current path (counting the root where no assignment has been made as level 0), thus the backtracking trail contains sufficient information to recover the particular assignment associated with each level.

We can union together these sets of levels in exactly the same fashion as unioning together sets of assignments—it is easy to map the operations on sets of levels to operations on sets of

assignments. We can also quite easily delete the no-good as soon as we backtrack to the maximum level it contain (once we backtrack to that level the search will undo the assignment at that level). Finally, this method of storing no-goods as sets of levels also gives us more flexibility when dealing with dynamic variable orderings (see Chapter 3).

2.3.2 No-goods over Unenumerated Solutions

In some cases we want to use depth-first search to enumerate all solutions. The search will uncover a solution when it finds a consistent complete set of assignments at depth n of the search tree (n is the number of variables in the CSP). Once a node containing a solution has been visited the algorithm can "enumerate" it—this might involve printing it out or performing some other operation on it. Since the depth-first search proceeds left to right in the tree, we will never find this solution again.

One way in which we can capture this is to redefine no-goods as being sets of assignments that cannot appear in any *unenumerated* solution. Once we enumerate a solution it becomes a no-good—it cannot appear in another unenumerated solution. By manipulating these no-goods in the same manner as the other no-goods (e.g., unioning collections of enumerated solution no-goods together) we can obtain a uniform treatment of the two cases where we want to find the first solution and where we want to find all solutions.

The algorithms described in this chapter do exactly this. They view the enumeration of a solution as the discovery of a new no-good. That no-good then plays exactly the same role as the other no-goods discovered during search (how these no-goods are used depends on the backtracking algorithm). Thus all of these algorithms can be used for both purposes.

2.4 Backtracking Algorithms

As mentioned above, tree search depends on being able to partition the sub-space below a node. There are many conceivable ways of partitioning this space, but the typical method is to split on all possible values of one of the future variables.

This is the method used in all of the algorithms we describe here, and it is the standard method used in the CSP literature. The choice of which future variable to split on is critical to performance, and we will discuss this choice further in Chapter 3.

2.4.1 Using No-goods

Above we described the different types of no-goods depth-first search encounters during search. Different backtracking algorithms are generated by different ways in which these no-goods are stored and used. The manner in which the no-good is stored is also critically related to how it can be subsequently used, as will become apparent later.

However, there are two generic ways in which the no-goods discovered during search can be used to improve backtracking.

Using No-goods for Intelligent Backtracking No-goods can be used to justify intelligent backtracking. Some backtracking algorithms always backtrack to the previous level. This is called chronological backtracking. However, by doing more explicit manipulation of no-goods other (non-chronological) backtracking algorithms are often able to backtrack many levels in one step.

Say that during search we are at level 11 of the search tree. We have completed searching all of the children of the current node, and by unioning together the no-goods we discovered in each of these sub-spaces we generate the new no-good (represented as a set of levels) $\{1, 3, 5\}$. This no-good means that no solution can contain the assignments made at levels 1, 3 and 5. Thus there is no need to continue searching at level 11. In fact, this no-good justifies backtracking all the way to level 5, and undoing the assignment made there.

Using No-goods for Domain Pruning The second important use of no-goods is for pruning domain values. Say that during search we have discovered the no-good $\{1, 2, 4, V \leftarrow a\}$, i.e., the assignments made at levels 1, 2, and 4 along with the assignment $V \leftarrow a$ to the future variable V is a no-good. Then clearly there is no need to try the assignment $V \leftarrow a$ below level 4 of the current path. By pruning this value from V's domain, and then restoring it once we backtrack to level 4 and retract the assignment made there, we can improve the efficiency of the search of the subtree below level 4. In particular, if we did not prune the value a of V we might have to consider this assignment many times in this subtree even though we already know it to be invalid given that the assignments at level 4 and above remain intact. It should also be noted that each time we consider this assignment we might have to do a considerable amount of work to demonstrate that no solution extends it.

When is it legitimate to prune a value of a variable to a particular level of the search tree? We make the following definition.

It is *sound* to prune a value a of variable V to level i if there exists no unenumerated solutions in the subtree below level i (of the current path) that contains $V \leftarrow a$.

Observation. It is not difficult to demonstrate that it is sound to prune value a of variable V to level i if and only if there exists a no-good containing only assignments made at level i or above along with $V \leftarrow a$.

2.4.2 Backtracking Algorithm Template

The backtracking algorithms can be viewed as particular instantiations of template shown in Figure 2.1.

The algorithm performs a DFS of the tree of partial assignments by invoking itself recursively. The search is initiated at level 1 by the call **TreeSearch(1)**. If all of the variables have been assigned the set of current assignments is a solution, and we can enumerate that solution.

```
TreeSearch(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         set {1,...,level-2} as a new no-good for assignment
            at level-1
         return(level-1)
      else
         return(0)

   V := pickNextVariable()
   BTLevel := level
   for x ∈ CurrentDom[V]
      assign(V,x,level)
      if checkConsistent(V,x,level)
         propagateConstraints(V,x,level)
         BTLevel := TreeSearch(level+1)
      undo(V,x,level)
      if BTLevel < level
         return(BTLevel)

   BTlevel := computeBackTrackLevel(V,level)
   setNoGoodOfAssignmentatBTLevel(BTLevel,V,level)
   return(BTLevel)
```

Figure 2.1: TreeSearch Algorithmic Template

The global variable `FINDALL` is set to true if we want to find all solutions. If we do then since we have just enumerated the solution, we know that no *unenumerated* solution can contain the set of assignments made at levels 1 through `level-2` along with the assignment made at `level-1`. Thus, the set $\{$`1,...,level-2`$\}$ is a new no-good for the assignment made at `level-1`.[5] That is, the last assignment made, at `level-1`, cannot be used again until at least one of the previous assignments made is undone. Otherwise we would simply obtain the same solution.

Otherwise, if we only want to find the first solution, we can backtrack to level 0, thus unwinding the recursion.

If there are unassigned future variables, we call the subroutine `pickNextVariable` to determine which future variable to split on next. `pickNextvariable` uses heuristics to decide which variable would be best to split on next, given the set of assignments that have already been made so far. There are, however, two special cases:

1. There is some variable with an empty `CurrentDom`. Then `pickNextVariable` must return one of these variables. The presence of such a variable indicates that the current path is a deadend—there is no compatible assignment that can be made to the variable with the empty domain. By returning the variable that has had a *Domain Wipeout*, **TreeSearch** can compute a suitable backtrack level as determined by the wipeout variable.

[5] By stating that a set is a no-good for an assignment $V \leftarrow a$, or for a particular value a of a variable V we mean that if we add $V \leftarrow a$ to this set we obtain a no-good.

2. If the first case does not hold, and there is a variable with a single value in its `Current-Dom`, then `pickNextVariable` should return one of these variables. This is simply an optimization—the assignment to this singleton value variable is forced, so we may as well make the assignment so that we can immediately propagate all of its consequences.

3. If neither of the first two cases hold, then `pickNextVariable` can decide which of the future variables to choose next using some heuristic ranking.

Once we have chosen the variable to split on, we then explore the subtree generated by each of its remaining values. For each value, we assign the variable that value and check the consistency of the newly augmented current set of assignments (`checkConsistent`). If the new set of current assignments is consistent we can then perform any necessary local propagation entailed by this new assignment (`propagateConstraints`) and then recurse to the next level.

After this we must undo the assignment and all of its consequences prior to trying the next value. For example, if the assignment caused us to pruned values from the domains of the future variables we must now restore those values.

If the value was consistent and we called **TreeSearch** recursively it could have been that in the subtree below, **TreeSearch** found a reason to backtrack above the current level. In this case `BTLevel` would have been set to a higher level, and we will prematurely terminate the examination of values of the current variable and return to the previous level (where we will continue to return until we reach the proper backtrack level).

Finally, if we ended up examining all values of the current variable the ***for*** loop would terminate normally. At this point in the search there would be some (perhaps implicit) no-good containing each of the values in the current variables domain. (Note that we would have no-goods for all of the values of the variable's original domain, not just for the values in the current domain). We will then use these no-goods to compute a level we can soundly backtrack to. That is, the highest level we can backtrack to while provably not missing any solutions. Once this backtrack level, `BTLevel` is computed by `computeBackTrackLevel`, we have a (perhaps implicit) no-good that contains the assignment made at `BTLevel`, so we record this no-good, as it is the no-good for one of the values of the variable that was split upon at `BTLevel`. Thus, once we try all of the values of the variable at `BTLevel` we will have a no-good for each of them, and we can recursively apply this same computation to backtrack from `BTLevel`.

The last thing **TreeSearch** does is return `BTLevel` as the backtracking level, which will cause the recursion to unwind up to this level.

2.4.3 Generic Backtracking (BT)

The earliest and simplest instantiation of **TreeSearch** is the generic backtracking algorithm BT. The BT algorithms is distinguished by the fact that it does perform any explicit storage of no-goods nor does it perform any constraint propagation. Rather it simply iterates over the variable domain with the implicit knowledge that *some* no-good exists for all of the values already iterated over. The algorithm is given in Figure 2.2. Once BT has chosen a variable V to split on it examines all

```
BT(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         return(level-1)
      else
         return(0)

   V := pickNextVariable()
   BTLevel := level
   for x ∈ Dom[V]
      assign(V,x,level)
      if checkConsistent(V,x,level)
         BTLevel := BT(level+1)
      undo(V,x,level)
      if BTLevel < level //Only occurs when BTLevel == 0
         return(BTLevel)

   return(level-1)
```

Figure 2.2: Generic Backtracking (BT)

of the values in V's original domain, $Dom[V]$. In particular, since it does not perform any domain pruning, `CurrentDom` will always be equal to the original domain. For each of these values x it checks the consistency of the current set of assignments when augmented by the assignment $V \leftarrow x$ (`checkConsistency`). If this augmented set of assignments is inconsistent then **BT** has "discovered" a no-good. However, since **BT** does not explicitly store this no-good, all that we know once `checkConsistency` returns is that the entire set $\{1, \ldots, \texttt{level-1}, V \leftarrow a\}$ is a no-good.

Suppose that at level ℓ we split on the variable V finding that every value of V is inconsistent with the prior assignments. We call such a node where the search does not recurse to a deeper level a *leaf* node. Suppose that $Dom[V] = x_1, \ldots, x_k$. At this point we know the no-goods $\{1, \ldots, \texttt{level-1}, V \leftarrow x_1\}$, ..., $\{1, \ldots, \texttt{level-1}, V \leftarrow x_k\}$. Thus we can union these no-goods (Section 2.2.2) to obtain the new no-good $\{1, \ldots, \texttt{level-1}\}$. This no-good means that we must (1) backtrack to the previous level, and (2) the no-good of the value assigned at that previous level must be $\{1, \ldots, \texttt{level-2}\}$.

We can then argue inductively that at every level ℓ of the search tree, once we have examined all of the values of the variable V assigned at level ℓ we will find one of two things

1. Some of the values x of V were inconsistent with the prior assignments, and thus we have the implicit no-good associated with them $\{1, \ldots, \ell - 1, V \leftarrow x\}$.

2. Some of the values y of V generated searchable subtrees but that once we backtracked from those subtrees the implicit no-good associated with them $\{1, \ldots, \ell - 1, V \leftarrow y\}$.

Hence, at the end of the ***for*** loop, the implicit union over these no-goods will the no-good $\{1, \ldots, \ell - 1\}$, and we must always backstep to the previous level.

In sum, BT must always backstep to the previous level due to the nature of the no-goods it discovers during search. Furthermore, since these no-goods are so uniform (they always contain all of the previous levels), there is no need to explicitly record or manipulate them.

2.4.4 Backjumping (BJ)

One early empirical observation is that BT often displays a behavior called *thrashing*. Thrashing is where BT explores multiple copies of a subtree in which it is doomed to discover the same flaw time and time again. This lead to early mechanisms for allowing BT to escape some instances of thrashing.

The Backjumping (BJ) algorithm was one early improvement over BT. The idea in BJ was to keep some additional information about the no-goods discovered during consistency checking. To provide the details of BJ we must examine more closely the manner in which consistency checking is performed.

We can check whether or not the set of assignments made at levels 1 through k, $\mathcal{A} = \{V_1 \leftarrow x_1, \ldots, V_k \leftarrow x_k\}$ is consistent by checking whether or not it satisfies each of the constraints it fully instantiates. Now consider what happens when, as in BT, we incrementally increase $\mathcal{A}$ by descending to a new level. Say that $\mathcal{A}$ is known to be consistent, and we add to it the new assignment $V' \leftarrow x'$. Clearly, we do not need to recheck all of the constraints that were over the previous variables $V_1, \ldots, V_k$. Rather, we simply need to check all those constraints C such that $V' \in VarsOf[C]$ and $VarsOf[C] \subseteq VarsOf[\mathcal{A}] \cup V'$. Furthermore, as soon as we find one violating constraint we can stop with the knowledge that the augmented $\mathcal{A} \cup V' \leftarrow x'$ is inconsistent.

BJ organizes these constraint checks in a different manner so that it can detect the earliest level at which $V' \leftarrow x'$ became inconsistent. In particular, it works level by level down the search tree, at each level ℓ checking the consistency of the set of constraints C such

$$V' \in VarsOf[C] \text{ and } VarsOf[C] \subseteq \{V_1, \ldots, V_\ell\}\,,$$

where V_i is the variable that was assigned at level i.

At the first level ℓ at which there exists a constraint C which $\mathcal{A} \cup \{V' \leftarrow x'\}$ fails to satisfy, BJ knows that $\{1, \ldots, \ell, V' \leftarrow x'\}$ is a no-good.

To track this information, BJ stores the maximum level over all of the values of each variable of the no-good discovered for that value. After it has examined all of the values of the variable it is able to backtrack to that maximum level, say M. However, since it has only stored the maximum level of all of its value no-goods, when it backtracks to level M the no-good it passes back for the value assigned at that level is $\{1, \ldots, M-1\}$. And, as we will see, this means that once BJ has made a backjump it must subsequently backstep to the previous level.

The algorithms is given in Figure 2.3.

Like BT, BJ does no domain pruning, so we examine all values in V's original domain, $Dom[V]$, and for each of these values it checks consistency of the current set of assignments when

```
BJ(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         maxBJLevel[level-1] = level-2
         return level-1
      else
         return(0)

   V := pickNextVariable()
   BTLevel := level
   maxBJLevel[level] := 0
   for x ∈ Dom[V]
      assign(V,x,level)
      if (M := checkConsistentBJ(V,x,level)) == level
         BTLevel := BJ(level+1)
      else
         maxBJLevel[level] = max(maxBJLevel[level], M)
      undo(V,x,level)
      if BTLevel < level
         return(BTLevel)

   BTLevel := maxBJLevel[level]
   maxBJLevel[BTLevel] := max(maxBJLevel[BTLevel],BTLevel-1)
         // Note that this max is equal to BTLevel-1.
   return(BTLevel)

checkConsistencyBJ(V,x,level)
   for i := 1 to level-1
      forall C such that
                1. V ∈ VarsOf[C]
                2. All other variables of C are assigned at level i or above
         if !checkConstraint(C) //Check agains current assignment.
            return(i)
   return(level)
```

Figure 2.3: BackJumping (BJ)

augmented by the assignment $V \leftarrow x$. However, unlike BT, BJ's consistency checking routine returns the minimum level at which it detects an unsatisfied constraint. BJ stores the maximum over all of its values of the no-good detected for these values.

Suppose that at level ℓ splitting on V yields a leaf node. Then `BTLevel = maxBJLevel[`ℓ`]` will be some level less than ℓ. Since each of V's values was discovered to be inconsistent, and `BTLevel` is the maximum level of all of these inconsistencies, we know that for every value $x \in Dom[V]$, $\{1, \ldots,$ `BTLevel`$, V \leftarrow x\}$ is no-good. Note that a shorter no-good might exist for some of these values x, but since BJ only stores the maximum, this is the only no-good that we know for certain applies to each of the values of V.

The union of these no-goods yields the new no-good $\{1, \ldots,$ `BTLevel`$\}$, and we can legitimately backtrack to level `BTLevel`. Furthermore, the no-good of the value assigned at `BTLevel` becomes $\{1, \ldots,$ `BTLevel` $- 1\}$, and this value must be considered when computing the maximum no-good level of the variable assigned at `BTLevel`. Hence, when we backtrack from

BTLevel, and we union together the no-goods of the values of the variable assigned at that level, we will obtain the new no-good $\{1, \ldots, \texttt{BTLevel} - 1\}$ and we will have to backstep from BTLevel.

Hence it is not hard to see that at any non-leaf node where we recurse on at least one value of the current variable, the maximum backtrack level will be set to be the previous level. In particular, it is only at leaf nodes that we can have non-trivial backjumps.

2.4.5 Conflict-Directed Backjumping (CBJ)

The reason that BJ can only backjump once lies in its representation of no-goods. It only stores the maximum level of the no-good. Thus, all that it is able to conclude is that all of the levels from level 1 down to this maximum is the no-good, and once that maximum has been used to generate a backjump, the rest of the levels in the no-good force it to subsequently backstep.

CBJ stores the no-good explicitly. In particular, it stores a dynamic updated union of the no-goods associated with each of its values. As a no-good is found for each value this union is augmented. The stored union does not include the assignments to the variable itself, as these assignments would be subsequently discarded anyway. Hence, once all of the values have been examined, the stored set is the new no-good, and CBJ can make direct use of it to guide backjumping and it can be passed on as a no-good for the value we backjump to.

CBJ gets some of its no-goods from checking constraints, and like BJ it organizes these constraint checks so as to detect the earliest level at which the current assignment became inconsistent. In particular, if the current assignment $V' \leftarrow x'$ fails to satisfy a constraint C, then the levels at which the other variables of C were instantiated along with $V' \leftarrow x'$ is a no-good. Thus we can union this no-good into the single no-good maintained by CBJ for the variable V'. The no-goods for the values that are consistent will be passed back by a lower level of the search tree when we backtrack to undo the assignments generated by these values.

The algorithms is given in Figure 2.4

Consider CBJ's behavior at a leaf node at level ℓ where it has split on the possible values of V. Each value of V will have a no-good returned for it by checkConsistentCBJ, and these no-goods will be unioned together into the set NoGoodSet[ℓ]. Note that level ℓ will be removed from each no-good added to this set. Hence, at the end of the ***for*** loop there will be some set of previous levels in NoGoodSet[ℓ], BTLevel which is the maximum of this set, will be above ℓ, and CBJ will backtrack to some prior level.

Suppose V' is the variable assigned at this backtrack level. Then just prior to backtracking CBJ will add the new no-good it computed into the set of no-goods V'. Again, however, it will delete BTLevel from this set prior to unioning it into the no-good set of V'. So recursively, once we have examined all of the values of V', CBJ will again be able to compute a no-good and backtrack again.

Since only a subset of the levels are passed back as a no-good to V', once we exhaust all of the values of V' it may well be that the no-good for V' allows once again for a non-trivial backjump. Hence, CBJ is able to perform non-trivial backtracks at non-leaf nodes as well as at leaf nodes.