Constraints

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

October 25, 2001

Chapter 1

Introduction

A constraint satisfaction problem (CSP), $\mathcal{P}$, consists of

1. A set of variables $\{V_1, \ldots, V_n\}$.
2. For each variable V_i a domain of possible values $Dom[V_i]$.
3. A set of constraints $\{C_1, \ldots, C_m\}$.

Each variable V can be assigned a value v, denoted by $V \leftarrow v$ if and only if $v \in Dom[V]$.

Intuitively, we want to find an assignment of values to the variables subject to the condition that the values *satisfy* all of the constraints. Each constraint is over some subset of the variables, and imposes a constraint on the values they can be simultaneously assigned. Even if it is easy to satisfy each individual constraint, it may be difficult to find an assignment that satisfies all of the constraints simultaneously.

More formally we make the following definitions.

A *feasible* set of assignments is a set of assignments $\{V_1 \leftarrow v_1, \ldots, V_k \leftarrow v_k\}$ such that each of the V_i is unique. That is, no variable can be assigned more than one value.

We will only be interested in feasible sets of assignments, so from here on we will assume assignments are feasible. Let $\mathcal{A}$ be a set of assignments, associated with $\mathcal{A}$ is a set $VarsOf[\mathcal{A}]$: the variables that have been assigned values in $\mathcal{A}$. Since a (feasible) set of assignments cannot assign distinct values to the same variable, it can have at most n elements. When it contains n assignments it is called a *complete* set of assignments.

Each constraint C_i is over some set of variables $VarsOf[C_i]$, and the arity of C_i is the cardinality of this set, $\| VarsOf[C_i] \|$. Every constraint C_i is a set of sets of assignments: if the arity of C_i is k then each element of C_i is a set of k assignments, one for each of the variables in $VarsOf[C_i]$. Thus our notation implies that for all A_1 and A_2 in C_i we must have $VarsOf[A_i] = VarsOf[A_2] = VarsOf[C_i]$.

We say that a set of assignments $\mathcal{A}$ *satisfies* a constraint C_i if $VarsOf[C_i] \subseteq VarsOf[\mathcal{A}]$ and there exists an element of C_i that is a subset of $\mathcal{A}$. Furthermore, we say that $\mathcal{A}$ is *consistent* (with

respect to a particular CSP), if it satisfies all constraints C such that $VarsOf[C] \subseteq VarsOf[\mathcal{A}]$. We denote this by $consistent(\mathcal{A})$.

A *solution* (a satisfying solution) to $\mathcal{P}$ is a complete and consistent set of assignments $\mathcal{A}$.

Observation Any subset of a solution is consistent.

In general we are interested in finding solutions to a CSP. More specifically, we might be interested in

1. Finding any solution (the satisfaction problem).
2. Enumerating or counting all solutions.
3. Finding a solution that optimizes some objective function.

1.1 The Constraint Domain

Within this general framework there are many specialized types of constraints, or constraint domains.

A constraint domain imposes certain restrictions on

1. The values the variables can take on (i.e., the $Dom[V_i]$ sets).
2. The type and representation of the constraints allowed.

1.1.1 The Constraint domain $\mathbb{R}$

Here the variable domain is the set of real numbers, and the constrains are expressions composed from real numbers, the functions $+$, $\times$, $-$, and $/$, the predicates $=$, $<$, $>$, $\neq$, $\leq$, $\geq$, and the logical connective $\wedge$ (conjunction).

Example 1 Say you have a loan for 2 years. We have the variables P, the principle of the loan, $B1$ the final balance after the first year, $B2$ the final balance after the second year, I the rate of interest, $R1$ the repayment after the first year, and $R2$ the repayment after the second year.

There are two constraints:

1. $B1 = P + P \times I - R1$
2. $B2 = B1 + B1 \times I - R2$

There are an infinite number of solutions to this CSP, one of them is

$$\{P \leftarrow 100, I \leftarrow 0.1, R1 \leftarrow 50, B1 \leftarrow 60, B2 \leftarrow 66, R2 \leftarrow 0\}.$$

A non solution is

$$\{P \leftarrow 100, I \leftarrow 0.1, R1 \leftarrow 0, B1 \leftarrow 60, B2 \leftarrow 66, R2 \leftarrow 0\}.$$

If we add the additional constraints:

- $B2 = 0$, $I = 0.1$, $R1 + R2 = 200$.

Then there are a whole set of solutions each with a different value of P and distribution between $R1$ and $R2$.

It is not difficult to see that if we allow a more general set of functions in our constrains the constraint domain $\mathbb{R}$ encompasses much of what has been studied in mathematics. Even without additional functions, this domain allows us to specify problems like finding the roots of polynomials:

Example 2 Let X and Y be the two variables, and the constrains be

- $Y = 100 \times X \times X \times X + 4 \times X + 6$
- $Y = 0$

There are at most three solutions—the three roots of this polynomial. Of course, not all of these roots need be solutions, as only the real roots lie in the domain of X.

1.1.2 The linear constraint domain

Here the domains of the variables is once again the reals. However, now we restrict the constraints to be linear equalities/in-equalities over the variables. For example the constraint could be the system of equations:

$$\begin{aligned} 1 + X &= 2Y + Z \\ Z - X &= 3 \\ X + y &= 5 + Z \end{aligned}$$

One nice feature of this constraint domain is that it has been well studied, and we can easily find a characterization of the complete set of solutions by simply solving this set of equations. Furthermore, linear programming algorithms can even find points in this set of solutions that optimize linear functions defined over the variables.

1.1.3 Linear constraints over Integer Domains

Here we have the same kind of linear equality/in-equality constraints as in the previous case, but we restrict some of the variables to take on integer values. This case has been well studied in operations research, it is the well known the integer linear programming problem.

1.1.4 The Finite Domain constraint domain

Here we restrict the domains of the variables to be finite sets. Once we do this we can include any kind of constraints. We can represent these constraints either

Intensionally The constraint is represented by a expression or program that evaluates to true or false dependent on whether or not the assignment satisfies the constraint.

Extensionally The constraint is represented explicitly as satisfying sets of assignments. (Note that since the variable domains are finite, the constraints must also be finite). This representation however can have size exponential in the number of variables the constrain is over.

Finite domain constraints have been the main subject of study in AI, and they will be our main focus in the course.

1.2 Example Constraint Problems

1.2.1 E-Commerce

Combinatorial Auctions are an important opportunity for e-commerce. Say Company A manufactures and sells a number of industrial products, products that are the essential inputs for a range of industrial processes.

Say that company B is engaged in a manufacturing process that requires as input a number of the products sold by company A. However, for company B these products are only useful together. If they do not get all of the inputs the remaining inputs are worthless to them.

Company A may sell to a number of other companies besides company B, and some of these other companies might also want to purchase some of the products that B wants, but they might not want all of them. How is company A to maximize its income?

One way is for company A to set up a combinatorial auction. The bidders make bids on bundles of goods, and the winners of these bids get their entire bundle. The question is how does company A decide who wins the auction?

This can be set up as a CSP as follows:

- Variables $\{B_1, \ldots, B_n\}$. One variable per bid.
- Domains, every domain is the set $\{0, 1\}$. $B_i \leftarrow 0$ means that B_i wins their bid and gets the product they requested.
- Constraints. We impose a constraint between every bid B_i, B_j whose bundle has a non-empty intersection (i.e., these two customers both want the same item). The constraint prohibits the joint assignment $\langle B_i \leftarrow 1, B_j \leftarrow 1 \rangle$.

Also we have an optimization criteria. Associated with each bid B_i is a price p_i, and the auctioneer wants to maximize

$$\sum_{i=1}^{n} p_i B_i,$$

(where by multiplying by B_i we mean that we are multiplying by the value of the variable).

1.2.2 The Golomb Ruler Problem

The Golomb ruler problem is the problem of finding the shortest ruler with marks at integer location such that every pair of distinct marks can measure a different distance.

Consider a set of N natural numbers (including 0) such that if i, j, k, l, are numbers in the set such that $i < j$, $k < l$, and $(i, j) \neq (k, l)$, then $(j - i) \neq (l - k)$. That is, the difference between any pair of numbers is distinct. These numbers are the marks on the ruler.

The Golomb ruler problem is to find, for various values of N, a set containing N marks satisfying the condition above such that we minimize the maximum number in the set. That is we want to find the shortest ruler.

The *decision* problem version is, given a fixed length L, and the number of marks N is there a layout of the marks within the range $0 - L$ that forms a golomb ruler. This can be set up as a CSP as follows:

- Variables, $\{M_1, \ldots, M_N\}$.
- $Dom[M_1] = 0$, $Dom[M_N] = \{1, \ldots, L\}$.
- Constraints, for every i, j, k, l, such that $(i, j) \neq (k, l)$ and $(j, i) \neq (k, l)$, we have the constraint $|j - i| \neq |k - l|$.

Now we can solve the optimization version of the problem by successively solving the problems with increasing values of L until we find a problem that has a solution.

Question 1 *What feature of the Golomb ruler problem allows us to solve the optimization problem by iterating over this sequence of decision problem?*

1.2.3 Satisfiability

The question as to whether or not a set of 3-SAT clauses are satisfiable is easily translated into a finite domain CSP. Say that we have the boolean variables $\{X_1, \ldots, X_n\}$ and the collection of 3-clauses $\{C_1, \ldots, C_m\}$ (an example 3-clause might be $\neg X_1 \vee X_{10} \vee \neg X_{11}$), we can construct the following CSP:

- Variables, $\{X_1, \ldots, X_n\}$—we use the same set of variables.
- Domains. Each variable has $\{\text{TRUE}, \text{FALSE}\}$ as its domain.

- Constraints. Each 3-clause becomes a constraint over the three variables that appear in the clause. The constraint prohibits the unsatisfying assignment. For example for the 3-clause $\neg X_1 \vee X_{10} \vee \neg X_{11}$) there would be a constraint between X_1, X_{10}, and X_{11} which prohibits the assignment $\{X_1 \leftarrow \text{TRUE}, X_{10} \leftarrow \text{FALSE}, X_{11} \leftarrow \text{TRUE}\}$.

1.3 Local Consistency/Constraint Propagation

For most CSPs, especially finite domain CSPs finding an assignment that satisfies a single constraint is not difficult. The difficulty lies in satisfying all of the constraints simultaneously.

The example of satisfiability shows that the question of whether or not a solution for a finite domain CSP exists is an NP-complete problem, so in general finding a simultaneous assignment to all of the constraints can be computationally difficult.

One useful way of reasoning with constraints is to apply local propagation to achieve local consistency. This process does not usually solve the CSP, but it can provide some useful information about it. Additionally, for some specialized problems achieving local consistency can solve the problem.

There are a number of simple examples of propagation:

- Bounds propagation when dealing with real or integer valued variables. For example, if we have that $0 \leq X \leq 3.3$, $0 \leq Y \leq 2.0$, and the constraint $X \leq Y$, then we can propagate this constraint to restrict the domain of X to the range $0 \leq 2.0$. Many more elaborate "interval" calculations can be identified, e.g., when $X < Y \times Z$ and we have bounds on Y and Z.

- Often constraint problems contain functionally determined values. For example, in modeling a combinatorial circuit as a constraint problem, we might have that the variable O is the output of the AND of I_1 and I_2. Thus if we know the value of I_1 and I_2 we can propagate these values through the constraint to determine the value of O.

For finite domain constraints we can treat propagation more systematically.

1.3.1 Local Consistency in Finite Domain CSPs

First we restrict ourselves to *binary* CSPs. Binary CSPs are CSPs where every constraint is over at most two variables. For binary constraints we can construct a *constraint network* from a constraint problem.

The constraint network is a graph formed by making each of the variables a node and making each binary constraint an edge between the two variables (nodes) it constraints.

On the constraint network 3 different types of local consistency can be defined.

- Node consistency.

- Arc consistency.

- Path consistency.

First some notation. If C is a constraint between variables V_1 and V_2 we write it as C_{V_1,V_2}, that is we use the variables it is over as a subscript. Furthermore if the assignment $\{V_1 \leftarrow x, V_2 \leftarrow y\}$ satisfies C_{V_1,V_2} (i.e., $\{V_1 \leftarrow x, V_2 \leftarrow y\} \in C_{V_1,V_2}$) we write $C_{V_1,V_2}(x, y)$. Note the order of the values agrees with the order of the variables in the subscript. Note also that when considered to be a set of set of assignments $C_{R,Q} = C_{R,Q}$, however due to our dependence on order in our notation we have $C_{R,Q}(x, y) \equiv C_{Q,R}(y, x)$.

Node Consistency

Node consistency is concerned with unary constraints only. It requires for every variable V, and every value $x \in Dom[V]$, that x satisfy all unary constraints on V.

For example, if $Dom[V] = \{0, 1, 2, 3, 4\}$ and we have the unary constraint $V \leq 3$, then the CSP would not be node consistent.

We can make a CSP node consistent by simply removing all inconsistent domain elements. In our example, we would simply remove the element 4 from the domain of V to obtain $Dom[V] = \{0, 1, 2, 3\}$.

Arc Consistency

Arc consistency is concerned with the local effects of each binary constraint. Say that C_{V_1,V_2} is a constraint (between variables V_1 and V_2 as per our notation). Then for every value $x \in Dom[V_1]$ there needs to be a value $y \in Dom[V_2]$ such that $C_{V_1,V_2}(x, y)$ (i.e, we must have that $\{V_1 \leftarrow x, V_2 \leftarrow y\} \in C_{V_1,V_2}$. If there is no such "supporting" value for x then it is clear that the assignment $V_1 \leftarrow x$ can never appear in any solution of the CSP. Thus we may remove it. This can be accomplished with the procedure *Revise*. Its inputs are two variables, V_1 and V_2 and the constraint between them C_{V_1,V_2}

```
Revise(V1,V2,C)
   DELETE := false;
   for x ∈ Dom[V1]
        SUPPORTED := false
        for y ∈ Dom[V2] while !SUPPORTED
             if C(x,y)
                  SUPPORTED := true
        if !SUPPORTED
             Dom[V1] := Dom[V1] - x
             DELETE := true
   return DELETE
```

We say that $x \in Dom[V]$ *supports* $y \in Dom[V']$ if $C_{V,V'}(x, y)$. Note that x supports y if and only if y supports x.

If we run this procedure on the arc between V_1 and V_2 we will achieve consistency in one direction only. That is every value in $Dom[V_1]$ will have a support in $Dom[V_2]$ but there may still be values $y \in Dom[V_2]$ for which there is no $x \in Dom[V_1]$ such that $C_{V_1,V_2}(x, y)$. Furthermore,

by deleting values from $Dom[V_1]$ we may now cause values in the domains of other variables to become unsupported.

To achieve arc consistency we must ensure that we achieve a stable state where every value has a support in every constraint. This can be achieved by an iterative process that converges to stability.

Consider what happens when we remove $x \in Dom[V_1]$. All variables that are constrained with V_1 may now contain values that are unsupported. However, say that we have just run *Revise(V_1,V_2)*, it is also clear that the removal of x from $Dom[V_1]$ will not affect any value in $Dom[V_2]$—x was removed because it had no supporting value in $Dom[V_2]$, hence no value in $Dom[V_2]$ depends on x for support. These observations allow us to restrict the amount of work we need to do to achieve arc-consistency, and they yield the algorithm AC-3:

```
AC-3()
   EnforceNodeConsistency()
   Queue := {(Vi,Vj) | there is an arc (vi,vj) (i ≠ j)
   while Queue is not empty
      (V1,V2) := remove an arc from Queue
      if Revise(V1,V2)
         Queue := Queue ∪ (vi,V1) | there is an arc (vi,V1),
                                    vi ≠ V1,
                                    vi ≠ V2
```

Note that in the algorithm Queue is viewed as being a set, an arc is only added to it if that arc was not already present.

Example 3 Consider the CSP:

- Variables: $\{V_1, \ldots, V_5\}$.
- Domains: $Dom[V_1] = Dom[V_2] = \{1, 2, 3\}$ and $Dom[V_3] = Dom[V_4] = Dom[V_5 = \{1, 2\}$.
- Constraints: $V_3 > V_1$, $V_3 > V_2$, $V_3 > V_4$, $V_3 > V_5$, and $V_5 > V_4$.

Question 2 *Draw the constraint network and achieve arc consistency in it.*

Observation If arc-consistency reduces the domain of some variable to the empty set, the CSP has no solution. If arc-consistency reduces the domain of each variable to a single value, these values are a solution. Otherwise the CSP might still have no, one, or many solutions.

AC-4 A number of improved algorithms for achieving arc-consistency have been developed. They revolve around the idea of using improved data structures that allow one to avoid having to scan and rescan all of the variable domains. AC-4 is a worst case optimal algorithm (although in practice it often does not perform as well as AC-3).

We use two data structures. First, **Supports[**V**,**x**]**, this is a set associated with each value of each variable. The set contains triples, $(V', y, C_{V,V'})$, specifying the values supported by x of V.

For each value y supported by x the triple notes the value, the variable whose domain contains the value, and the constraint between the variables. Second, **SupportCount[**V**,**x**,**$C_{V,V'}$**]**, a support count for each value of each variable. We maintain a separate support count for each constraint that V participates in. If any of these support counts drops to zero we know that x now has lost all of its support on the domain of V', and that we must now delete x.

```
AC-4()
   EnforceNodeConsistency()
   Set all Supports[V,x] = ∅
   Set all SupportCount[V,x,C] = 0
   ;;
   ;;Set up supporting values.
   ;;
   for all arcs (Vi,Vj) (in both directions)
      for x ∈ Dom[Vi]
         for y ∈ Dom[Vj]
            if Cij(x,y)
               SupportCount[Vi,x,Cij]++
               Supports[Vj,y] := Supports[Vj,y] ∪ (Vi,x,Cij)
            if SupportCount[Vi,x,Cij] == 0
                FailList := FailList ∪ (Vi,x)
                Dom[Vi] := Dom[Vi] - x
   ;;
   ;;Remove unsupported values
   ;;
   while FailList is not empty
      (V,x) := remove an unsupported value from FailList
      for (y,V',C) ∈ Supports[V,x]
          SupportCount[y,V',C]--
          if SupportCount[y,V',C] == 0
             FailList := FailList ∪ (V',y)
             Dom[V'] := Dom[V'] - y
```

As with AC-3, FailList is being treated as a set (i.e., no duplicates are allowed).

It can be noted by keeping a count we know when we have to trigger more extensive processing. AC-4 has been the basis for the algorithms AC-6 and AC-7 which embody certain optimizations over and above AC-4.

However, it should also be noted that the space requirements for the **Supports** sets is $O(ned)$ where n is the number of variables, d is the maximum domain size, and e are the number of edges in the constraint graph (number of constraints). For some CSPs this can be excessive.

AC-2000 and AC-2001 Recently simple modifications that make AC-3 asymptotically optimal (i.e., of the same order of complexity as AC-4) have been identified (Refining the Basic Constraint Propagation Algorithm, by Christian Bessiere and Jean-Charles Regin. IJCAI-2001 pages 309–315).

1.3.2 Generalized Arc Consistency GAC

The previous definition of arc consistency was specific to binary constraints. However, arc consistency is easily generalize to the non-binary case.

Definition 1.3.1 [Generalized Arc Consistent (GAC)]

> A value x of a variable V is generalized arc consistent in a constraint C if either $V \notin VarsOf[C]$, or $V \in VarsOf[C]$ and there exists an assignment for all of the other variables in C such that $V \leftarrow x$ along with these assignments is consistent.
>
> A value x of a variable V is generalized arc consistent if it is generalized arc consistent along every constraint.
>
> A CSP is generalized arc consistent if all of its values are generalized arc consistent.

It is not difficult to see that if the constraint C is binary, then this definition is identical to the previous defintion of binary arc consistency. GAC can be enforced by pruning all values that are not GAC using an algorithm very similar to AC3. However, in this case the time complexity of the algorithm grows exponentially with the arity of the constraint. An algorithms like AC4 can also be developed, but then space complexity (specifically the size of the support sets) grows exponentially with the arity of the constraint.

1.3.3 Path Consistency

Once again in the case of binary constraints we can define a higher form of arc consistency, called path consistency.

Observation If there is no constraint between V_1 and V_2 we can view the constraint as being the *universal* constraint. That is, we can view the "constraint" as permitting all possible values.

The next level of local consistency that has been studied in the literature, but that has not seen much practical application is path consistency. It is the condition that for every pair of values $x \in Dom[V_1]$ and $y \in Dom[V_2]$ such that $C_{V_1,V_2}(x, y)$, and third variable V_3 there must exist a value $z \in Dom[V_3]$ such that $C_{V_1,V_3}(x, z)$ and $C_{V_2,V_3}(y, z)$. Note that if there is no constraint between V_3 and either of V_1 or V_2 the condition is either vacuous or it reduces to simple arc-consistency.

However, to achieve path-consistency we must remove pairs of values (x, y) that do not have a path-support z. This cannot be done by removing values from the domains of variables, rather we must augment the binary constraints so that they rule these inconsistent pairs.

In practice path consistency has limited use since

- It is computationally cumbersome to augment constraints, whereas removing domain values can be done efficiently.
- The complexity of achieving path-consistency is cubic, rather than quadratic. This often makes it more expensive that is worthwhile.
- Augmenting constraints has less computational value than pruning domain values.

Note that this condition over triples of variables implies the following condition (which is where the name path consistency comes from).

A path of length m through the nodes $(V_0, V_1, \ldots, V_m)$ is path consistent if and only if for any values $x \in Dom[V_0]$ and $y \in Dom[V_m]$, such that $C_{V_0,V_m}(x, y)$ there exists a sequence of values $z_1 \in Dom[V_1], \ldots, z_{m-1} \in Dom[V_m]$ such that $C_{V_0,V_1}(x, z_1)$, $C_{V_1,V_2}(z_1, z_2)$, $\ldots$, $C_{V_{m-1},V_m}(z_{m-1}, y)$.

Question 3 *Prove that if the above condition on triples of variables holds then this notion of consistency along any path also holds. (Note that this only makes sense when dealing with binary constraints).*

1.3.4 (i,j) consistency

A generalization of node, arc, and path consistency is the notion of (i,j) consistency. It is defined quite simply:

> A CSP is (i,j) consistent if whenever we choose consistent values for any set of i variables, we can find values for any set of j additional values such that the values for all of the $i + j$ variables are consistent.

Remember that "consistent" means that every constraint all of whose variables are fully instantiated is satisfied. Note that this definition is not limited to binary constrains as were the previous notions of arc and path consistency.

- Binary Arc-Consistency is (1,1) consistency.
- Path-consistency is (2,1) consistency.
- (i,j) consistency for $i > 1$ requires us to augment the constraints in the CSP to achieve it.
- (1,i) which is called i inverse consistency can be achieved by pruning domain values only. This case will prove to be useful when we talk about backtracking search.

We will revisit i-j consistency when we discuss graphical properties of the constraint network.

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 $\{1, \ldots,$ `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 $\{1, \ldots,$ `level-1`$, V \leftarrow x_1\}$, ..., $\{1, \ldots,$ `level-1`$, V \leftarrow x_k\}$. Thus we can union these no-goods (Section 2.2.2) to obtain the new no-good $\{1, \ldots,$ `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 $\{1, \ldots,$ `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 $\{1, \ldots,$ `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 the variables values. As a no-good is found for each value this union is augmented. 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 algorithm 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 for 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.

```
CBJ(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         NoGoodSet[level-1] := {1,...,level-2}
         return level-1
      else
         return(0)

   V := pickNextVariable()
   BTLevel := level
   NoGoodSet[level] := ∅
   for x ∈ Dom[V]
      assign(V,x,level)
      if (NoGood := checkConsistentCBJ(V,x,level)) == ∅
         BTLevel := CBJ(level+1)
      else
         NoGoodSet[level] = NoGoodSet[level] ∪ (Nogood - {level})
      undo(V,x,level)
      if BTLevel < level
         return(BTLevel)

   BTLevel := max(NoGoodSet[level])
   NoGoodSet[BTLevel] := NoGoodSet[BTLevel] ∪ (NoGoodSet[level]-{BTLevel})
   return(BTLevel)


checkConsistencyCBJ(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(Levels VarsOf[C] were assigned)
   return(∅)
```

Figure 2.4: Conflict Directed Backjumping (CBJ)

2.4.6 Value Specific CBJ (vSCBJ)

One more improvement can be made to CBJ. CBJ maintains only one no-good for each variable, dynamically unioning together all of the no-goods it discovers for the variable's values. Value specific CBJ instead maintains an explicit no-good for each variable. Then, when it backtracks, if there exists a constraint between the variable we are backtracking to and the variable we are backtracking to, vSCBJ can apply constraint filtered unioning of the no-goods. Potentially, this can allow us to pass smaller (more powerful) no-goods back up the tree when backtracking. This in turn can lead to longer backjumps.

The algorithms is given Figure 2.5

```
vSCBJ(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         NoGoodSet[VarAt[level-1],ValAt[level-1]] :=  {1,...,level-2}
         return level-1
      else
         return(0)

   V := pickNextVariable()
   BTLevel := level
   for x ∈ Dom[V]
      assign(V,x,level)
      if (NoGood := checkConsistentCBJ(V,x,level)) == ∅
         BTLevel := vSCBJ(level+1)
      else
         NoGoodSet[V,x] = Nogood - {level}
      undo(V,x,level)
      if BTLevel < level
         return(BTLevel)

   BTLevel := max_{x∈Dom[V]}(NoGoodSet[V,x])
   C := Constraint between VarAt[BTLevel] and V
        //C is the universal constraint if there is no constraint
        //C, such that VarsOf[C] = {VarAT[BTLevel],V},
        //between these two variables exists.

   NoGoodSet[VarAt[BTLevel],ValAt[BTLevel] :=
         ∪_{x:x∈Dom[V] and C(ValAt[BTLevel],x)} NoGoodSet[V,x] - {BTLevel}
   return(BTLevel)
```

Figure 2.5: Value Specific CBJ (vSCBJ)

2.4.7 Adding Domain Pruning

Value specific Conflict directed backjumping makes an inefficiency that all of the previous algorithms possess apparent. Consider the example where at level 10 we have split on the variable V, and tried each of the values in its domain $Dom[V] = \{a, b, c\}$. Say that for each value we had found the following no-goods (perhaps via consistency checking, perhaps via search in the subtree below): for a, $Ng(a) = \{1, 2, 6\}$ (i.e., $\{1, 2, 6, V \leftarrow a\}$ is the no-good), $Ng(b) = \{1, 2, 3\}$, and $Ng(c) = \{1, 2\}$. **vSCBJ** would then decide to backtrack to level 6. Say that there was no constraint between the variable at level 6 and V, thus we pass back the union over all of the no-goods (minus the backtrack level), $\{1, 2, 3\}$ as the no-good for the value assigned at level 6. Now we try a different value at level 6, and search the subtree below that value.

In the new subtree below level 6, we might again try to split on variable V. And again we would attempt all of the values $\{a, b, c\}$. However, we already know that there is still a valid no-good for b and for c—we backed up far enough to make a a possibility again, but not far enough to make b or c valid again. The **vSCBJ** algorithm will ignore these no-goods, proceed to rediscover perhaps new no-goods for these values, and then overwrite the old no-goods. This is potentially wasted

work. Furthermore, it could have been that the previous no-good for b or c was discovered only from searching a large subtree below these values. Hence, the rediscovery of a no-good for these values could require a lot of computation.

We can take better advantage of the no-goods for specific values discovered during search by doing domain pruning. To accomplish this we need some simple data structures and subroutines.

1. `currentDom[V]`, for every variable we maintain a domain of current values. The original set of values is stored in the set Dom`[V]`.

2. `prunedVals[i]`, for every level of the search tree (from 0 through n where n is the number of variables of the CSP) we have a set of variable value pairs, (Var,val). A pair $(V, x) \in$ `prunedVals[i]` means that x has been pruned back to level i.

3. `prunedLevel[V,x]`, for every variable and value we set this array entry to be i if that value has been pruned to level i. If the value is unpruned the value of this array entry will not matter.

4. `prune(V,x,i)`. This subroutine removes `x` from `currentDom[V]`, places the pair `(V,x)` into `prunedVals[i]`, and sets `prunedLevel[V,x]` to be `i`.

5. `undo(V,x,level)`. Now this subroutine not only undoes the assignment $V \leftarrow x$ it also restores all of the values that are in the set `prunedVals[level]`, and for each of these values it empties their associated `NoGoodSet` entry. That is, for every pair `(V,x)` $\in$ `prunedVals[level]` it puts `x` back into `currentDom[V]` and sets `NoGoodSet[V,x]` $= \emptyset$.

2.4.8 Domain Pruning Algorithms

With these data structures in place we can give domain pruning versions of the previous algorithms. As we had noted before a value x of a variable V can be pruned to level i whenever i is the maximum of the no-good discovered for x.

Value specific CBJ with domain pruning is given in Figure 2.6 Note that in **vSCBJ+P** we iterate over the values in `currentDom[V]`. This occurs in all of the algorithms that do domain pruning.

It should be noted that once we prune every value back to the maximum of the no-good we discover for it, we can compute the backtrack level `BTLevel` by simply taking the maximum of these pruning levels. An alternate way of looking at this is that we need to backtrack to a level at which at least one value of `V` will be restored, and `BTLevel` is precisely this level.

Even though CBJ does not keep value specific no-goods we can still use the value no-goods as soon as we discover them. Once these no-goods have been unioned into the variable's no-good set, we lose the value specific information. The algorithm is specified in Figure 2.7

And finally we can even obtain some advantage from domain pruning in the Backjumping algorithm (Figure 2.8).

```
vSCBJ+P(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         NoGoodSet[VarAt[level-1],ValAt[level-1]] :=  {1,...,level-2}
         prune(VarAt[level-1],ValAt[level-1],level-2)
         return level-1
      else
         return(0)

   V := pickNextVariable()
   BTLevel := level
   for x ∈ Currentdom[V]
      assign(V,x,level)
      if (NoGood := checkConsistentCBJ(V,x,level)) == ∅
         BTLevel := vSCBJ+P(level+1)
      else
         NoGoodSet[V,x] = Nogood - {level}
         prune(V,x,max(NogoodSet[V,x]))
      undo(V,x,level)
      if BTLevel < level
         return(BTLevel)

   BTLevel := max_{x∈Dom[V]} prunedLevel[V,x]
   C := Constraint between VarAt[BTLevel] and V
        //C is the universal constraint if no constraint
        //exists between these two variables

   NoGoodSet[VarAt[BTLevel],ValAt[BTLevel] :=
         ∪_{x: x ∈Dom[V] and C(ValAt[BTLevel],x)} NoGoodSet[V,x] - {BTLevel}
   prune(VarAt[BTLevel],ValAt[BTlevel],
         max(NoGoodSet[VarAt[BTLevel],ValAt[BTLevel]]))
   return(BTLevel)
```

Figure 2.6: Value Specific CBJ with Pruning (vSCBJ+P)

Although the surface form of BJ+p is quite different, this algorithm is a version of what has been called BackMarking plus backjumping (BMJ).[6]

2.4.9 Algorithms that perform Constraint Propagation

One flaw in the backtracking algorithms presented above is that often when we instantiate a variable the reduced CSP may have an "obvious" flaw. For example, it might fail to be node consistent or arc consistent. However, because we never explicitly compute the reduced CSP the search might not notice this.

For example, it could be that once we assign a value to two variables, there is no value re-

[6] In these algorithms it can sometimes be possible to optimize when we prune values to the previous level. Often we need not do the actual pruning, rather it can be possible to keep track of these "single-level" pruned back values by simply using the state implicitly contained in the ***for*** loop that iterates over the variables domain. The counter for this loop tells us which values of the variable have already been processed.

```
CBJ+P(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         NoGoodSet[level-1] := {1,...,level-2}
         prune(VarAt[level-1],ValAt[level-1],level-2)
         return level-1
      else
         return(0)

   V := pickNextVariable()
   BTLevel := level
   NoGoodSet[level] := ∅
   for x ∈ currentDom[V]
      assign(V,x,level)
      if (NoGood := checkConsistentCBJ(V,x,level)) == ∅
         BTLevel := CBJ+P(level+1)
      else
         prune(V,x,max(Nogood-{level}))
         NoGoodSet[level] = NoGoodSet[level] ∪ (Nogood - {level})
      undo(V,x,level)
      if BTLevel < level
         return(BTLevel)

   BTLevel := max(NoGoodSet[Level]) //This is = max_{x∈Dom[V]} prunedLevel[V,x]
   NoGood := NoGoodSet[level]-{BTLevel}
   prune(VarAt[BTLevel],ValAt[BTLevel],max(NoGood))
   NoGoodSet[BTLevel] := NoGoodSet[BTLevel] ∪ NoGood
   return(BTLevel)
```

Figure 2.7: CBJ with Pruning (CBJ+P)

maining in the domain of a third variable that is consistent with these assignments. If the above algorithms were to choose this third variable as the next variable to split on, then they would immediately detect the inconsistency. This would immediately terminate the search below the first two assignments. However, the choice of which variable to instantiate next is made heuristically, and it could be that we choose some other variables first. Hence, the above algorithms could spend a lot of time searching a subtree in which there was an obvious failure. This work could be avoided if only they were clever enough to find the failed variable. Unfortunately, it turns out to be a computationally hard problem to choose the "best" next variable, so it is as hard to be clever in choosing the next variable to assign as it is to do the search with an non-optimal choice of next variable.

Algorithms that employ constraint propagation enforce a certain degree of local consistency on the reduced CSP (we never compute this reduced CSP explicitly). Hence, they attempt to detect obvious failures by performing certain checks on *all of the future variables*.[7] Sometimes, this checking is fruitful, in that it detects a failure, and sometimes this checking simply wastes time.

In fact, the enforcement of local consistency on the reduced CSP does not usually waste time (at least as long as we confine ourselves to enforcing a relatively low level of local consistency), as

[7] Thus even constraint propagation algorithms make no attempt to single out specifi c future variables.

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

   V := pickNextVariable()
   BTLevel := level
   maxBJLevel[level] := 0
   for x ∈ currentDom[V]
      assign(V,x,level)
      if (M := checkConsistentBJ(V,x,level)) == level
         BTLevel := BJ+P(level+1)
      else
         prune[V,x,M]
         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) // = BTLevel-1
   prune[VarAt[BTLevel],ValAt[BTLevel],BTLevel-1)
   return(BTLevel)
```

Figure 2.8: Backjumping with Pruning (BJ+P)

even if a failure is not detected, various values of the future variables can be pruned. This yields a simpler CSP to solve in the subtree below the current assignment. The varying sizes of the domains of the future variables also provides valuable information for variable ordering heuristics.

2.4.10 Forward Checking (FC)

Forward checking is the simplest of the constraint propagation algorithms. It involves enforcing node consistency (i.e., $(0, 1)$ consistency) at every node n on the sub-CSP below n (see Section 2.1 for a specification of the sub-CSP). The standard node consistency algorithm simply involves removing all domain values that are inconsistent with the unary constraints.

Consider how one would enforce node consistency on the sub-CSP in the sub-space below a node in the search tree. The unary constraints in this sub-CSP are those constraints that have all but one of their variables assigned by the current assignments. We do not want to generate an explicit representation of this unary constraint—we would have to discard the representation as soon as we backtrack. However, we can still enforce node-consistency by pruning the domain of the single uninstantiated variable of the constraint. In particular, we can determine for each value of the uninstantiated variable whether or not each value taken along with the set of current assignments satisfies the constraint. Checking the constraint using the current assignments to fix the

```
forwardCheck(V,level)
//Restore Node consistency given that V
//has just been assigned at level.

   forall (C,V_k) such that
          1. V_k ∈ VarsOf[C]
          2. All variables of C except for V_k have been assigned
          3. V ∈ VarsOf[C]
          //V_k is the sole unassigned variable of C.
      A := {V_1 ← x_1, ..., V_{k-1} ← x_{k-1}}
           where x_i is the current assignment of each assigned V_i ∈ VarsOf[C]
      for x ∈ CurrentDom[V_k]
           if A ∪ {V_k ← x} does not satisfy C
              prune(V_k, x, level)
      if CurrentDom[V_k] == {}
          return
          //There is no need to check any other constraints
          //the next level will instantiate V_k and will then
          //immediately backtrack to undo the current assignment to V.
```

Figure 2.9: Forward Check all New Unary Constraints

values of the other variables of the constraint is clearly equivalent to checking the unary constraint that arises from reducing the constraint by the current assignments. This process of enforcing node-consistency on the new unary constraints is called forward checking these constraints. The **forwardCheck** algorithm for forward checking all newly generated unary constraints at a node in the search tree is given in Figure 2.9.

With the **forwardCheck** subroutine, the rest of the FC algorithm is easy to specify. It is given in Figure 2.10. There are two things to note about FC.

1. FC does not check whether or not the new assignment $\mathtt{V} \leftarrow \mathtt{x}$ is consistent with the current set of assignments.

2. FC does not do any intelligent backtracking.

Question 4 *Let $\mathcal{A}$ denote the current set of assignments. $\mathcal{A}$ starts out being the empty set. Whenever FC executes the statement* `assign(V,x,level)` $\mathtt{V} \leftarrow \mathtt{x}$ *is added to $\mathcal{A}$, and whenever FC executes the statement* `undo(V,x,level)` $\mathtt{V} \leftarrow \mathtt{x}$ *is removed from $\mathcal{A}$.*

Prove that in FC $\mathcal{A}$ is always consistent. (Hence, FC need not explicitly check its consistency after making a new assignment).

That FC does not do any intelligent backtracking follows from the manner in which FC processes the no-goods it discovers during search. Every time FC detects that a value x of a future variable V fails to be node consistent in the sub-CSP, it has detected a no-good. The no-good consists of the assignment $V \leftarrow x$ along with all of the levels the other variables of the failed constraint were assigned. FC uses this no-good to prune x from the domains of V.

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

   V := pickNextVariable()
   for x ∈ currentDom[V]
      assign(V,x,level)
      forwardCheck(V,level) //This is propagateConstraints of TreeSearch.
      BTLevel := FC(level+1)
      undo(V,x,level)       //Undo will restore all pruned  values of
                            //the future variables.
      if BTLevel < level
         return(BTLevel)  //Only occurs when BTLevel == 0.

   return(level-1)
```

Figure 2.10: Forward Checking

As with the previous set of algorithms the no-goods that FC discovers can be stored in various ways, and the manner in which they are stored affects how they can be used later on in the search. For each value x of variable V that is pruned by constraint propagation, FC remembers only the maximum level of the no-good used to prune it. This maximum is stored in `prunedLevel[V,x]`, i.e., the level at which x has been pruned is this maximum, by the **forwardCheck** subroutine.

Looking at the FC code we can see that the (non-solution) leaf nodes in the search tree are those nodes where `pickNextVariable` has returned a variable with an empty `Currentdom`. Any variable with a non-empty `Currentdom` will cause FC to descent at least once to the next level. Given that `pickNextVariable` always returns a variable with an empty current domain if one exists, it must be the case that at the leaf node the variable returned has just had its current domain reduced to the empty set. That is, at least one value in its domain must have been pruned at the previous level.

Hence, if the current node is a leaf node at level ℓ, then, when FC computes a backtrack level by unioning together the no-goods of the values of the current variable, at least one of these no-goods must be the set $\{1, \ldots, \ell - 1\}$. (Since it only has the maximum it must assume that all of the previous levels are in the no-good. Furthermore, one of the maximums is $\ell - 1$.) Thus, FC must backstep to the previous level. Furthermore, the no-good that FC passes back for the value assigned at the previous level will be $\{1, \ldots, \ell - 2\}$, i.e., the complete set of previous levels. Hence, by induction at every non-leaf node there will be value whose no-good is the complete set of previous levels, and FC must always backstep rather than backjump. This also means that, like BT, FC need never keep explicit track of the no-goods it discovers during search, they all have the same structure. It need only keep track of the levels at which values have been pruned.

2.4.11 Forward Checking with CBJ

To allow FC to perform intelligent backtracking it becomes necessary to store more information about the no-goods it discovers. This can be done in two simple ways. First, like CBJ we can maintain a single conflict set for each variable, updating this set by unioning in the no-goods we discover for the values of the variable as we discover them. Or we can maintain separate no-goods for each value.

The first option, where we maintain a single conflict set for all of the values of a variable yields an algorithm called FC-CBJ.[8] In this algorithm we must alter the **forwardCheck** subroutine so that it updates the variable's conflict set with the full no-good it discovers for each pruned value. The new version of **forwardCheck** is given in Figure 2.11. The resulting new search algorithm **FCCBJ** is given in Figure 2.12.

There are a few things worth noting about the algorithm. First, we have the slight change that instead of indexing the data structure `NoGoodSet` by level (as was done in the previous CBJ algorithm) we must now index it by variable. This is necessary since forwardCheck updates the `NoGoodSet` of future variables; i.e., variables that have no assignment level associated with them. Second, the restoration process required when we backtrack is slightly more complex.

In particular, the subroutine `updateConflictSets(level)` must remove `level` from the `NoGoodSet` of all variables. Consider the variable V' whose value y has been pruned by the current assignment made at level ℓ. Together, $V' \leftarrow y$, the current assignment, and some set of assignments made at levels above ℓ, falsified some constraint C, thus causing y to be pruned to level ℓ. We are about to undo the current assignment, thus the no-good for $V' \leftarrow y$ is going to become invalid. This no-good must be removed from the set of no-goods associated with V', `NoGoodSet`$[V']$. More generally, the current level can no longer be a part of the reason any value of the variable was pruned. Removing the current level from these conflict sets is what `updateConflictSets(level)` does.

Note that for any particular level in the set `NoGoodSet`$[V']$ there is no record of the value(s) of V' that caused it to be added to `NoGoodSet`$[V']$. However, we do know that ℓ was the maximum level added to `NoGoodSet`$[V']$, thus we can remove this level. (Even if ℓ was added by multiple values of V', all of the no-goods of these values will be invalid when we undo the current assignment). The remaining levels to `NoGoodSet`$[V']$ that were added when we computed the no-good for y must remain—there is no record of which ones were added. However, after a no-good has been computed for every value of V', `NoGoodSet`$[V']$ will still be a valid no-good. It will in general contain many "garbage" elements left over from when values of V' were restored and then later new no-goods learned for them. But the superset of a no-good is itself always a no-good. On the other hand, these extra garbage elements can degrade the backjumping potential of the algorithm.

[8]The standard version of FC-CBJ does not prune values it backtracks to. That is, it does not use the no-goods discovered during by search to do domain pruning, it uses these only for intelligent backtracking. It uses only the no-goods it discovers during constraint propagation for domain pruning. So this version of FC-CBJ is a slight improvement.

```
forwardCheckCBJ(V,level)
   forall (C,V_k) such that
         1. V ∈ VarsOf[C]
         2. All variables of C except for V_k have been assigned
         3. V∈ VarsOf[C]
         //V_k is the sole unassigned variable of C.
      A := {V_1 ← x_1, ..., V_{k-1} ← x_{k-1}}
          where x_i is the current assignment of each assigned V_i ∈ VarsOf[C]
      for x ∈ CurrentDom[V_k]
          if A ∪ {V_k ← x} does not satisfy C
             NoGood := set of levels assignments in A were made
             NoGoodSet[V_k] := NoGoodSet[V_k] ∪ NoGood
             prune(V_k, x, level)
      if CurrentDom[V_k] == {}
         return
```

Figure 2.11: Forward Check all New Unary Constraints and Record Conflicts

```
FCCBJ(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         NoGoodSet[VarAT[level-1]] := {1,...,level-2}
         prune(VarAt[level-1],ValAt[level-1],level-2)
         return level-1
      else
         return(0)

   V := pickNextVariable()
   for x ∈ currentDom[V]
      assign(V,x,level)
      forwardCheckCBJ(V,level)
      BTLevel := FCCBJ(level+1)
      updateConflictSets(level)
      undo(V,x,level)
      if BTLevel < level
         return(BTLevel)

   BTLevel := max(NoGoodSet[V])
   NoGood := NoGoodSet[V]-{BTLevel}
   prune(VarAt[BTLevel],ValAt[BTLevel],max(NoGood))
      //Standard FC-CBJ does not do this pruning step.
   NoGoodSet[VarAT[BTLevel]] := NoGoodSet[VarAT[BTLevel]] ∪ NoGood
   return(BTLevel)
```

Figure 2.12: Forward Checking with CBJ

The second option, where we maintain a separate no-good for each value, does not suffer from this flaw (but it does require more storage to maintain these separate no-goods). These separate no-goods are emptied as soon as a backtrack to the maximum level they contain occurs (this maximum is also equal to the `pruneLevel` of the value). It yields an algorithm which we can call CFFC.[9] As before it requires an updated version of **forwardcheck**. This is given in Figure 2.13, and the new search algorithm CFFC is given in Figure 2.14.

2.4.12 MAC

Maintain Arc Consistency (MAC) is an algorithm that like FC does contraint propagation. However, it enforces a higher level of local consistency in the sub-problems it encounters during search. In particular, it enforces (1,1) consistency in the sub-problem. The "maintain" component of the algorithm comes from the fact that MAC first establishes (1,1) consistency and then continues to maintain (1,1) consistency as we instantiate variables.

To enforce (1,1) consistency we must (1) enforce node consistency over all constraints that have been reduced to unary as with FC, and (2) enforce arc consistency over all constraints that have been reduced to binary. The algorithm looks much like FC, the only difference being (1) we enforce (1,1) consistency prior to starting search, and (2) we perform different processing during the constraint propagation phase. The constraint propagation subroutine for MAC is given in Figure 2.15. This subroutine reestablishes arc consistency given that a new variable has been instantiated at `level`. Since the sub-CSP was arc consistency just before the new instantiation, we need first to prune the values of the future variables that have become node inconsistent. This is done using the same processing as forward checking. Then for all future variables that have had their domains reduced by this step, we need to check to see of the binary constraints they participate in generate any additional arc inconsistent values. We also have to check the newly created binary constraints to enforce arc consistency over them.

With **acCheck** the MAC algorithm becomes easy to specify—it is identical to FC. Figure 2.16 gives the algorithm.

Since MAC like FC remembers only the maximum of the no-goods it discovers during constraint propagation it is incapable of doing intelligent backtracking, for exactly the same reasons as FC.

2.4.13 MAC with CBJ

Hence, like FC-CBJ if we keep track of more information about the no-goods discovered we can implement intelligent backtracking with MAC. Again, we have two options (1) store all of the nogoods discovered for the values of a variable in one set, removing levels as we backtrack, or (2) store a separate nogood for each value and do constraint filtered unioning of nogoods on backtrack. The first option yields the algorithm MAC-CBJ, while the second yields the algorithm CFAC. They

[9] Called CFFC$^-$ in a previous publication.

```
forwardCheckCF(V,level)
   forall (C,V_k) such that
         1. V ∈ VarsOf[C]
         2. All variables of C except for V_k have been assigned
         3. V ∈ VarsOf[C]
         //V_k is the sole unassigned variable of C.
      A := {V_1 ← x_1, ..., V_{k-1} ← x_{k-1}}
          where x_i is the current assignment of each assigned V_i ∈ VarsOf[C]
      for x ∈ CurrentDom[V_k]
          if A ∪ {V_k ← x} does not satisfy C
             NoGood := set of levels assignments in A were made
             NoGoodSet[V_k,x] := NoGood //The only change from forwardCheckCBJ.
             prune(V_k, x, level)
      if CurrentDom[V_k] == {}
         return
```

Figure 2.13: Forward Check all New Unary Constraints and Record Value Specific Conflicts

```
CFFC(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         NoGoodSet[VarAt[level-1],ValAt[level-1]] := {1,...,level-2}
         prune(VarAt[level-1],ValAt[level-1],level-2)
         return level-1
      else
         return(0)

   V := pickNextVariable()
   for x ∈ currentDom[V]
      assign(V,x,level)
      forwardCheckCF(V,level)
      BTLevel := CFFC(level+1)
      undo(V,x,level) //Now undo must set NoGoodSet[V',y] = {}
                      //for all values y of variables V' it restores.
      if BTLevel < level
         return(BTLevel)

   BTLevel := max_{x∈Dom[V]} prunedLevel[V,x]
   C := Constraint between VarAt[BTLevel] and V
        //C is the universal constraint if there is no constraint
        //C, such that VarsOf[C] = {VarAT[BTLevel],V}

   NoGoodSet[VarAt[BTLevel],ValAt[BTLevel] :=
         ∪_{x: x ∈Dom[V] and C(ValAt[BTLevel],x)} NoGoodSet[V,x] - {BTLevel}
   prune(VarAt[BTLevel],ValAt[BTlevel],
         max(NoGoodSet[VarAt[BTLevel],ValAt[BTLevel]]))
   return(BTLevel)
```

Figure 2.14: Conflict Forward Checking CFFC

```
acCheck(V,level)
//Restore (1,1) consistency over the new binary constraints given that V
//has just been assigned at level.
   Queue := {}
   forall (C,V_k) such that
         1. V_k ∈ VarsOf[C]
         2. All variables of C except for V_k have been assigned
         3. V∈ VarsOf[C]
         //V_k is the sole unassigned variable of C.
      A := {V_1 ← x_1, ..., V_{k-1} ← x_{k-1}}
          where x_i is the current assignment of each assigned V_i ∈ VarsOf[C]
      for x ∈ CurrentDom[V_k]
          if A ∪ {V_k ← x} does not satisfy C
             prune(V_k, x, level)
             forall (V',V_k,C') such that
                   1. V' is a future variable.
                   2. V', and V_k are both constrained by C'
                   3. These are the only two unassigned variables of C'
                Queue := Queue ∪ {(V',V_k,C')}
      if CurrentDom[V_k] == {}
         return

   forall C such that
         1. C has only two unassigned variables remaining.
      V_1 := first unassigned variable of C
      V_2 := second unassgined variable of C
      Queue := Queue ∪ {(V_1, V_2, C)} ∪ {(V_2, V_1, C)}

   while Queue != {}
      (V',V_k,C') := remove element from Queue
      if Revise(V',V_k,C',level)
         forall (V'',V',C'') such that
                   1. V'' is a future variable.
                   2. V', and V'' are both constrained by C''
                   3. These are the only two unassigned variables of C''
                   4. V'' ≠ V_k
                Queue := Queue ∪ {(V'',V',C'')}

Revise(V,V',C,level)
   DELETE := false
   A := {V_1 ← x_1, ..., V_{k-1} ← x_{k-1}}
       where x_i is the current assignment of each assigned V_i ∈ VarsOf[C]
   for x ∈ CurrentDom[V]
      SUPPORTED := false
      for y ∈ CurrentDom[V'] while !SUPPORTED
         if ∪ {V ← x,V' ← y} satisfies C
            SUPPORTED := true
      if !SUPPORTED
         prune(V, x, level)
         DELETE := true
   return DELETE
```

Figure 2.15: Arc Check all New Binary Constraints

```
MAC(level)
   //We have established (1,1) consistency at level 0.
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         return(level-1)
      else
         return(0)

   V := pickNextVariable()
   for x ∈ currentDom[V]
      assign(V,x,level)
      acCheck(V,level)
      BTLevel := MAC(level+1)
      undo(V,x,level)
      if BTLevel < level
         return(BTLevel)
   return(level-1)
```

Figure 2.16: MAC

have structure as FC-CBJ and CFFC except that the no-goods discovered during constraint propagation are different because we are doing arc consistency rather than node consistency. Figure 2.17 gives an updated version of **acCheck** that stores the nogoods discovered during arc consistency processing.

With **acCheckCBJ** we can write the algorithm MAC-CBJ easily: it has the same structure as FCCBJ except that it utilizes **acCheckCBJ** rather than **forwardCheckCBJ**.

Notice that in **acCheckCBJ** when we prune a value x of V because it fails to have support on the domain of V' (given the settings of the other variables in the constraint C), the nogood we have discovered for x is the union of the set of nogoods of x's supports on V' (given the setting of the other variables in the constraint C) along with the levels the other variables in C were assigned. Since in MAC-CBJ we do not maintain the nogoods for the individual values of V' we must take all of the nogoods associated with V' as part of x's new nogood. (This set of nogoods is stored in `NoGoodSet[`V'`]`.

Using this observation we can implement option (2), where each value has its own no-good quite easily. The resulting algorithm CFAC has the same structure as CFFC, and a new version of the **acCheckCBJ** that updates value specific nogoods. In particular, in **acCheckCF** it becomes feasible to compute a better nogood for a value x that has been pruned because it has become arc inconsistent. The new subroutine **acCheckCF** is given in Figure 2.18.

2.4.14 N-ary Constraints and the GAC Algorithm

The FC algorithm waits until a constraint has all but one of its variables assigned until it checks the constraint. MAC waits until the constraint has all but two of its variables assigned. We can instead apply GAC (Generalized arc consistency).

```
acCheckCBJ(V,level)
   Queue := {}
   forall (C,V_k) such that
          1. V_k ∈ VarsOf[C]
          2. All variables of C except for V_k have been assigned
          3. V∈ VarsOf[C]
          //V_k is the sole unassigned variable of C.
      A := {V_1 ← x_1, ..., V_{k-1} ← x_{k-1}}
           where x_i is the current assignment of each assigned V_i ∈ VarsOf[C]
      for x ∈ CurrentDom[V_k]
           if A ∪ {V_k ← x} does not satisfy C
              NoGood := set of levels assignments in A were made
              NoGoodSet[V_k] := NoGoodSet[V_k] ∪ NoGood
              prune(V_k, x, level)
              forall (V',V_k,C') such that
                     1. V' is a future variable.
                     2. V', and V_k are both constrained by C'
                     3. These are the only two unassigned variables of C'
                 Queue := Queue ∪ {(V',V_k,C')}
      if CurrentDom[V_k] == {}
         return

   forall C such that
          1. C has only two unassigned variables remaining.
      V_1 := first unassigned variable of C
      V_2 := second unassgined variable of C
      Queue := Queue ∪ {(V_1, V_2, C)} ∪ {(V_2, V_1, C)}

   while Queue != {}
      (V',V_k,C') := remove element from Queue
      if ReviseCBJ(V',V_k,C',level)
         forall (V'',V',C'') such that
                     1. V'' is a future variable.
                     2. V', and V'' are both constrained by C''
                     3. These are the only two unassigned variables of C''
                     4. V'' ≠ V_k
                 Queue := Queue ∪ {(V'',V',C'')}

ReviseCBJ(V,V',C,level)
   DELETE := false
   A := {V_1 ← x_1, ..., V_{k-1} ← x_{k-1}}
        where x_i is the current assignment of each assigned V_i ∈ VarsOf[C]
   for x ∈ CurrentDom[V]
      SUPPORTED := false
      for y ∈ CurrentDom[V'] while !SUPPORTED
         if A ∪ {V ← x, V' ← y} satisfies C
            SUPPORTED := true
      if !SUPPORTED
         NoGood := set of levels assignments in A were made
         //The key difference between acCheck and acCheckCBJ
         NoGoodSet[V] := NoGoodSet[V] ∪ NoGood ∪ NoGoodSet[V']
         prune(V, x, level)
         DELETE := true
   return DELETE
```

Figure 2.17: Arc Check all New Binary Constraints and Set Conflicts

```
acCheckCF(V,level)
   Queue := {}
   forall (C,V_k) such that
          1. V_k ∈ VarsOf[C]
          2. All variables of C except for V_k have been assigned
          3. V∈ VarsOf[C]
          //V_k is the sole unassigned variable of C.
      A := {V_1 ← x_1, ..., V_{k-1} ← x_{k-1}}
           where x_i is the current assignment of each assigned V_i ∈ VarsOf[C]
      for x ∈ CurrentDom[V_k]
           if A ∪ {V_k ← x} does not satisfy C
              NoGoodSet[V_k,x] := set of levels assignments in A were made
              prune(V_k, x, level)
              forall (V', V_k, C') such that
                    1. V' is a future variable.
                    2. V', and V_k are both constrained by C'
                    3. These are the only two unassigned variables of C'
                 Queue := Queue ∪ {(V', V_k, C')}
      if CurrentDom[V_k] == {}
         return

   forall C such that
          1. C has only two unassigned variables remaining.
      V_1 := first unassigned variable of C
      V_2 := second unassgined variable of C
      Queue := Queue ∪ {(V_1, V_2, C)} ∪ {(V_2, V_1, C)}

   while Queue != {}
      (V', V_k, C') := remove element from Queue
      if ReviseCF(V', V_k, C', level)
         forall (V'', V', C'') such that
                    1. V'' is a future variable.
                    2. V', and V'' are both constrained by C''
                    3. These are the only two unassigned variables of C''
                    4. V'' ≠ V_k
                 Queue := Queue ∪ {(V'', V', C'')}

ReviseCF(V, V', C, level)
   DELETE := false
   A := {V_1 ← x_1, ..., V_{k-1} ← x_{k-1}}
        where x_i is the current assignment of each assigned V_i ∈ VarsOf[C]
   for x ∈ CurrentDom[V]
      SUPPORTED := false
      for y ∈ CurrentDom[V'] while !SUPPORTED
         if A ∪ {V ← x, V' ← y} satisfies C
            SUPPORTED := true
      if !SUPPORTED
         NoGoodSet[V,x] := set of levels assignments in A were made
         for y ∈ Dom[V']
            if A ∪ {V ← x, V' ← y} satisfies C
            NoGoodSet[V,x] := NoGoodSet[V,x] ∪ NoGoodSet[V',y]
         prune(V, x, level)
            //level must be equal to max(NoGoodSet[V,x]), since
            //all of x's supports on V' were pruned at higher levels.
         DELETE := true
   return DELETE
```

Figure 2.18: Arc Check all New Binary Constraints and Set Value Specific Conflicts

GAC can be enforced by pruning all values that are not GAC using an algorithm very similar to AC3. We can develop tree search algorithms that maintain GAC, which means that every constraint is checked after each assignment is made. In particular, it is possible to develop an algorithm MGAC, which maintains GAC but does no intelligent backtracking just like MAC; MGAC-CBJ, which maintains no-goods stored at the level of variables and does intelligent backtracking and pruning using the variable no-goods unioned together during search, just like MAC-CBJ; and CF-GAC, which maintains no-goods stored at the level of individual values and does even more powerful intelligent backtracking and pruning using the value nogoods unioned together with constraint filtering during search.

2.5 Notes

If we use the no-goods discovered during search (i.e., the no-goods passed up the search tree after we have exhausted all values of a variable) to prune domain values, then it is possible to prune values that are k-inverse inconsistent for arbitrary k.[10]

A good example of this phenomenon comes from the pigeon hole problem. Let V_1, V_2, V_3, V_4 be four variables each with domain $\{1, 2, 3\}$ and a not equal constraint between every pair (i.e., $V_i \neq V_j$, $i \neq j$, $i, j \in \{1, 2, 3\}$. Let $V_5, ...V_k$ be some set of additional variables with some other set of constraints between them.

First we can note that all values of $V_1, \ldots, V_4$ are (1,1) consistent, and also the are PIC (path inverse consistent) (1,2). But no value is (1,3) consistent.

Consider a node in the search tree where some set of previous variables have been assigned, we are at level 5, and the next four variables to be assigned are V_1, V_2, V_3 and V_4 in that order. If one examines the search tree using any of the algorithms that compute explicit no-goods and use them to prune values, we will see that an empty no-good will be computed for each of the values i of V_1 after we search the subtree below it. This empty no-good means that $V_1 \leftarrow i$ cannot participate in any solution with no extra conditions. That is, $V_1 \leftarrow i$ cannot participate in any solution, and we can prune i back to level 0. By pruning i to level 0 we remove it from consideration from the rest of the search. It can also be noted that an easy way of implementing the test for an empty no-good, is simply to add 0 to all no-good sets. The if we obtain a no-good with only 0 in it, then we can process it in the standard way by pruning the value involved to the maximum of its no-good set 0. (Or if we are using the no-good to backtrack, by backtracking to level 0 thus ending the search).

[10] k-inverse consistent is (1,k) consistency, as defined in Chapter 1.3.4. A value x of V is k-inverse consistent if $V \leftarrow x$ can be extended to a consistent assignment over any k additional variables.

Chapter 3

Dynamic Variable Ordering

Besides the various improvements one can make to generic backtracking, discussed in the previous chapter, the component that has the most dramatic effect on the efficiency of problem solving is variable ordering. That is, given that the search is at some node n, which of the future variables does `pickNextVar` select as the next variable to split on.

If there is some fixed ordering over the variables and `pickNextVar` always chooses the uninstantiated variable that is next in this ordering (ignoring for now the fact that `pickNextVar` is forced to choose variables with empty or singleton domain first), then there will be a fixed mapping from level of the search tree to the variable assigned at that level. (At level i the i variable in the ordering will always be chosen, unless there was a variable with a singleton domain that has displaced the order). In this case we say that a *static variable ordering* is being used. There was some early work on choosing good static variable orderings, but it is now acknowledged that *dynamic variable orderings* are much superior in practice to even the best static ordering.

In *dynamic variable ordering* the next variable chosen depends on the current set of assignments. Thus along each branch of the search tree the variables could be chosen in a different order. Given that we are going to employ dynamic variable ordering the question is: how do we choose the next variable. There is no computationally tractable way of determining the best next variable, so we must employ a heuristic to make the choice.

3.1 Increasing the Flexibility of Search Order

Dynamic ordering as defined above still has the property that the siblings of each node assign (different) values to the same variable. Thus there is not complete flexibility in choosing the variable ordering: once we choose a variable to split on we are committed to trying all of that variable's values.

Suppose we employ domain pruning in the tree search and furthermore we guarantee that each value x of a variable V that is attempted at level ℓ during the search is pruned back to at least level $\ell-1$ after the subtree under it is searched. (If we use the no-good passed back by the subtree search

```
FlexTreeSearch(level)
   if all variables are assigned
      current assignments is a solution, enumerate it
      if FINDALL
         {1,...,level-2} is a new no-good for assignment
            at level-1
         prune(VarAt[level-1],ValAt[level-1],level-2)
         return(level-1)
      else
         return(0)
NEXT:
   (V,x) := pickNextVarVal()
         //V is unassigned, and x is from CurrentDom[V].
         //Furthermore,
         //   1. If there is a variable with an empty CurrentDom
         //      return such a variable and NIL as value x.
         //   2. Otherwise if there is a variable with a singleton
         //      CurrentDom, then return such a variable with x
         //      set to its single remaining value.

   if(x != NIL)
      assign(V,x,level)
      BTLevel := level
      if checkConsistent(V,x,level)
         propagateConstraints(V,x,level)
         BTLevel := FlexTreeSearch(level+1)
      undo(V,x,level)
      if BTLevel < level
         return(BTLevel)
      goto NEXT
   //We drop out here only if we have a variable with an empty domain.
   NoGood := ComputeNoGoodFromExhaustedVariable(V)
   BTlevel := max(NoGood)
   setNoGoodOfAssignmentatBTLevel(BTLevel, NoGood-{BTLevel})
   prune(VarAT[BTLevel],ValAT[BTLevel],max(NoGood-{BTLevel}))
   return(BTLevel)
```

Figure 3.1: A more Dynamic TreeSearch Algorithmic Template

to prune x, then we will always satisfy this condition.) Then we can in fact use a more flexible scheme of variable ordering, a scheme where sibling nodes do not have to assign to the same next variable.

The pseudo code for a tree search algorithm achieving this flexibility is shown in Figure 3.1.

In this code the routine `pickNextVarVal` returns a variable value pair, rather than just a variable to split on. Thus in separate iterations of the loop it could assign values to different variables. That is, sibling nodes could assign to different variables. The key to the algorithm is that if one loop attempts, e.g., $V \leftarrow x$, x will then be pruned from the current domain of V (to at least the previous level) by the time we backtrack to execute another iteration of the loop. Hence, on the next loop if $V' \leftarrow y$ is attempted, in the subtree explored below $V' \leftarrow y$ various assignments to V might well be attempted, but $V \leftarrow x$ will never be: x is not in $CurDom[V]$ in this subtree. Thus

the search will never visit the same node twice (i.e., no two nodes in the search tree will ever have the same set of current assignments).

The reason for "switching in mid-stream" between splitting on V and V' might be that in the search of the subtree below $V \leftarrow x$ we end up pruning many values of V' to levels above the current level. If this occurs, it might be the case that when we backtrack to the current level we find that now V' has a smaller domain than V (even though we have removed x from $CurDom[V]$), and that now it is more sensible to split on V' than on V.

Once we have exhausted all of the values of some variable, we can, as before, use the no-goods discovered for that variable's values to compute a backtrack level and a new no-good to pass back to the backtrack level.

The empirical performance of this more flexible version of tree search needs to be tested.

3.2 Current Practice

The variable ordering heuristic that is most commonly used in current practice is based on the Brelaz heuristic first developed for graph coloring. In graph coloring we want to assign a color to each node so that no two adjacent nodes have the same color. The optimization version of the problem is to determine the graph's chromatic number—the minimum number of colors required to color the graph. The decision version of the problem would be to determine whether or not the graph could be colored using k colors for a specific value of k.

To encode a graph coloring problem as a CSP we would have a variable for each node, and the domain of values for each variable would be the set of k different colors. Finally, there would be an inequality constraint between every two variables whose corresponding nodes are connected.

Given a partial coloring of a graph, the *saturation* of a vertex is the number of different colored vertices connected to it. The Brelaz heuristic (Bz) first colors the vertex of maximum degree. Thereafter Bz selects an uncolored vertex of maximum saturation, and tie-breaks on the degree of the vertex in the uncolored subgraph.

In the context of the CSP, this corresponds to the heuristic where (1) we choose the variable with minimum remaining consistent values (i.e., values that are consistent with the assignments already made so far), (2) breaking ties by the number of constraints the variable has with the set of unassigned values. Note that these two steps will agree with Bz on the choice of the very first variable only if we start out with a set of variables all of which have the same domain size.

Except for some cases discussed below this heuristic works about as well as anything else that can be quickly computed.

3.3 Some Attempts to Understand the Choice of Heuristic

Current knowledge of the why this heuristic works, and more importantly how to design better heuristics, is still inadequate. But there have been a couple of studies on the subject. We describe two of them.

The first is due to Smith and Grant, and it revolves around the idea of *Fail First*.

3.3.1 The Fail First Principle

The fail first principle dates back the work of Haralick and Elliott in the late 70's. They used the principle

> The best search order is the one that minimizes the expected length or depth of each branch.

So one should always select next the variable that is most likely to fail—fail first. By doing this, and making an assumption that the failure of each variable is independent of the others, we minimize the expected length of each branch.

We can immediately detect a problem with this reasoning. Search effort depends on the number of nodes explored, and this is influenced by both the length of each branch and on the number of branches. Smith and Grant's study confirms that neglecting the branching factor and concentrating solely on failing early can lead to worst performance.

Now we turn to a description of the study they performed, providing some simple generalizations of their analytical results as we proceed.

We restrict our attention to Binary CSPs. Assume that there are n unassigned variables, $\{V_1, \ldots, V_n\}$, that the set of constraints are known, and that each variable has m_i values *consistent with the current set of assignments*. The heuristics studied require knowledge of these parameters at every point of the search. For algorithms that do no constraint propagation, e.g., CBJ, the m_i values are not known to the algorithm. To compute these values would require that we perform node consistency on the sub-CSP at every point of the search. If we go to the effort to do this, we may as well use this information to prune values of the future variables. That is, we may as well use an algorithm that does at least as much constraint propagation as FC. Hence, we will assume that we are in fact using such an algorithm, and thus for each future variable V_i, $CurDom[V_i]$ contains only values consistent with the current set of assignments. Thus we will have $m_i = |CurDom[V_i]|$.

Given a constraint between V_i and V_j, we can measure the *tightness* of the constraint.

Definition 3.3.1 [Tightness of a Constraint] Given that there are m values for the variable V_i and n values for V_j, then the tightness of the constraint $C_{i,j}$ between them is

$$1 - \frac{\left|\{(x,y) : x \in Dom[V_i] \text{ and } y \in Dom[V_j] \text{ and } C_{i,j}(x,y)\}\right|}{m \times n}$$

That is, the tightness is the proportion of pairs (of consistent values) that fail to satisfy the constraint.

Given that some set of assignments $\mathcal{A}$ have been made, we define the *current tightness* of a constraint C_{ij} that is between two future variables V_i and V_j, as being the tightness of C_{ij} with

respect to the current domains of the variables it constraints. That is, the proportion of pairs of unpruned values that satisfy C_{ij}:

$$1 - \frac{\left|\{(x,y) : x \in \mathit{CurDom}[V_i] \text{ and } y \in \mathit{CurDom}[V_j] \text{ and } C_{i,j}(x,y)\}\right|}{m_i \times m_j}$$

We use p_{ij} to denote the current tightness of C_{ij}. As we do the search, and make more assignments these numbers will change. In particular, there will be fewer consistent values so the values m_i are non-increasing. The current tightness of the different constraints might, however, increase or decrease, dependent on whether or not more consistent pairs have been pruned or more inconsistent pairs have been pruned.

The fail-first principle says that we should choose the unassigned variable that is most likely to produce an immediate failure after it is forward checked. We want to adopt a probabilistic model so that we can compute this as a probability of failure. Then if we assume that the failure of each variable is independent the length of each branch will be minimized by always splitting next on the variable that has highest failure probability.

Of course given any particular CSP, $\mathcal{P}$, and some point in the search space (i.e., some set of current assignments, $\mathcal{A}$) each future variable either will or will not produce a failure after forward checking, so we need an event space from which to compute the probabilities. For this purpose we build an ensemble of problems with a distribution over them, and we will assume that the sub-CSP generated from $\mathcal{P}$ by the current assignments, $\mathcal{P}|_{\mathcal{A}}$ was drawn randomly from that ensemble. Then the probability of failure for each future variable in this sub-CSP can be computed from the distribution over the ensemble (coupled with the assumption that $\mathcal{P}|_{\mathcal{A}}$ was drawn randomly).

For this to give answers useful for $\mathcal{P}|_{\mathcal{A}}$ the ensemble needs to contain problems that are as similar as possible to $\mathcal{P}|_{\mathcal{A}}$. To specify a distribution over the ensemble we use a generative probability model: the probability of any particular problem in the ensemble is the probability it was generated.

In particular, we generate random CSPs such that, (1) they have the same set of variables as $\mathcal{P}|_{\mathcal{A}}$ (i.e., the future variables), each with the same domain (i.e., the current domains of these future variables), and constraints over the same pairs of variables. However, each constraint is generated randomly. For the constraint C_{ij} we examine each pair of values $x \in \mathit{CurDom}[V_i]$ and $y \in \mathit{CurDom}[V_j]$ and include the set of assignments $\{V_i \leftarrow x, V_j \leftarrow y\}$ with probability $1 - p_{ij}$. Thus each constraint is constructed independently of the others, and each pair of assignments is allowed by the constraint independently of the other pairs.

It is not hard to see that in this distribution that for any two constraints C_{ij}, C_{kl}, and pairs of values (x,y) ($x \in \mathit{CurDom}[V_i]$ and $y \in \mathit{CurDom}[V_j]$) ($w$, z) ($w \in \mathit{CurDom}[V_k]$ and $z \in \mathit{CurDom}[V_l]$), $\mathcal{P}(C_{ij}(x,y)$ and $C_{kl}(w,z) = \mathcal{P}(C_{ij}(x,y)) \times \mathcal{P}(C_{kl}(w,z))$.[1]

[1]Remember our notation that if $C_{ij}(x,y)$ means that the constraint C_{ij} is satisfied by the set of assignments $\{V_i \leftarrow x, V_j \leftarrow y\}$.

Now we compute in this distribution the probability that a variable V_i will generate an immediate failure after its values have been forward checked.

$$\mathcal{P}(V_i \text{ fails}) = \mathcal{P}(\text{every } x \in CurDom[V_i] \text{ fails}) = \mathcal{P}(V \leftarrow x \text{ fails})^{m_i},$$

where $m_i = |CurDom[V_i]|$.

That is, due to the strong independence assumptions of our model, each value of V_i will fail independently, and will have an identical probability of failing.

Let V_j be another future variable, and let C_{ij} be the constraint between V_i and V_j. If there is no constraint between them then let C_{ij} be the universal constraint, in which case $p_{ij} = 0$: no tuple falsifies the constraint. Now consider the probability that $V_i \leftarrow x$ fails because it generates a domain wipe out of V_j (i.e., it is inconsistent with every value in V_j's current domain.

$$\mathcal{P}(V \leftarrow x \text{ is inconsistent with every value of } V_j) = p_{ij}^{m_j}$$

$V_i \leftarrow x$ fails under forward checking if it causes any of the other future variables to experience a domain wipe out.

$$\mathcal{P}(V_i \leftarrow x \text{ fails}) = 1 - \prod_{V_j \in FutureVars, j \neq i} (1 - p_{ij}^{m_j}) \tag{3.1}$$

Putting it all together. V_i will fails with probability

$$\mathcal{P}(V_i \text{ fails}) = \Big(1 - \prod_{V_j \in FutureVars, j \neq i} (1 - p_{ij}^{m_j})\Big)^{m_i} \tag{3.2}$$

Smith and Grant study four heuristics generated by different approximations to this probability. (Always choosing the variable with maximum failure probability).

1. FF. The first model is the fail first model of Haralick and Elliott. Choose next the variable with minimal current domain. This corresponds to assuming that term (3.1) is the same for every value of every variable. Thus (3.2) will be minimized by choosing the variable with minimum m_i.

2. FF2. In term (3.1) instead of using the current tightness p_{ij} of each constraint, they use the original tightness of the constraint, and the original domain sizes as an estimate for the current domain size of each future variable.

3. FF3. In term (3.1) instead of using the current tightness p_{ij} of each constraint, they use the original tightness of the constraint. But they do use the current domain sizes of the future variables.

4. FF4. They compute the current tightness p_{ij} of each constraint and use (3.2).

They find that contrary to the fail first principle FF4 does not perform better than FF. Rather it performs worse. It does have significantly shorter branches than FF, but sometimes it chooses to branch on a variable with a larger current domain, if that variable has tighter and or more constraints with the future variables. Thus in the search the tree tends to be wider albeit shorter, and the number of nodes explored with FF4 is higher than with FF.

3.3.2 Least Constrained Subproblem

In another study by Gent et al. they devised heuristics that yielded least constrained sub-CSPs. These heuristics also had the side effect that they tended to choose the most constrained variable next (as by instantiating the most constrained variable, the set of remaining future variables were least constrained). Thus their heuristics also have much in common with the fail first principle.

3.3.3 Open Questions

It is clear that to reduce the search cost one wants to reduce the size of search tree explored during backtracking. Thus failing early, i.e., having short branches is important. However, not having to many branches is also important, perhaps even more so.

This seems to be why the smallest current domain works so well, and why breaking ties by the number of constraints with future variables as a tie breaker is also valuable. The first influences the number of branches most strongly (and has some effect on depth of the branches, under certain probabilistic models of the CSP). The second influences the depth of the branches under the assumption that each constraint is equally tight.

In Gent et al. studies in which the constraints have different tightness were shown to fool the Bz heuristic, as in this case a simple count of the constraints can provide incorrect information about the probability of failure.

Some of the questions that remain in this area are

1. Obtaining a better formal model that can be used to develop heuristics that use some informed tradeoff between the number of branches, and length of these branches.

2. The depth and length of branches ignores the possibility of guiding the search to regions where it can learn powerful no-goods. For algorithms that use no-goods to do intelligent backtracking, it is possible that these powerful no-goods would allow skipping large parts of the search space.

3. Heuristics that can take advantage of the possibility of changing the variable being split on, as allowed in the more flexible version of tree search described above.

4. Heuristics that can work with algorithms that do not do constraint propagation, and thus do not know the number of consistent values in the domains of the future variables.

5. The choices made at the first levels of the search tree have the most impact on the ultimate size of the explored space. Yet at the top of the search tree we have the least information—fewer variables have been assigned and less constraint propagation has been performed. Hence, it would be useful to experiment with adaptive heuristics that do more work when they know less information.

With respect to item 4 there are at least two possibilities. First, in algorithms like CBJ+P that do domain pruning we do get varying domain sizes as search progresses. So we could use these domain sizes to compute heuristics. The drawback of this approach is during the early stages of the search we have very little information. Second, one could adapt the probabilistic analysis of Smith and Grant to predict the likelihood of failure of a new variable against the current set of assignments.

Chapter 4

Davis Putnam Procedures for SAT

A special type of CSP problem is to determine whether or not a propositional theory expressed as a set of clauses is satisfiable. Of course, satisfiability has a much longer history than constraint satisfaction, and specialized techniques have been developed for solving the satisfiability problem. Nevertheless, there is much in common between these techniques and the techniques used to solve CSPs.

Another point of interest for studying satisfiability is that there are many things that it can tell us about solving CSP, and vice versa.

First we define the satisfiability problem more formally. Let $\mathcal{L} = \{p_1, \ldots, p_n\}$ be a set of boolean variables (propositional symbols). These variables can take on the value true (1) or false (0). A truth assignment ϕ over $\mathcal{L}$ is a mapping from the p_i to $\{0, 1\}$.

Corresponding to each variable p_i is its negation $\neg p_i$ (we sometimes write the negation as $-p_i$). p_i and $\neg p_i$ are called *literals*. p_i is a positive literal while $\neg p_i$ is a negative literal. We extend the truth assignment over the p_i to a truth assignment over the literals by setting $\phi(\neg p_i) = 1 - \phi(p_i)$. We say that a literal l is true under ϕ if $\phi(l) = 1$.

A set of literals is called a *clause*, and a set of clauses is a *formula* in conjunctive normal form (CNF). A clause is interpreted to be the disjunction of the literals in it, and a formula is interpreted to be the conjunction of the clauses in it. A clause with only a single literal in it is called a *unit* clause. Thus, if C is a clause, $\phi(C) = 1$, (i.e., C is true under ϕ or C is *satisfied* by ϕ) if and only if $\phi(l) = 1$ for some literal $l \in C$; and if F is a formula, $\phi(F) = 1$ if and only if $\phi(C) = 1$ for all clauses in F. It is well know that any formula of a propositional logic defined over the symbols $\{p_1, \ldots, p_n\}$ is equivalent to a CNF formula, and there are systematic ways of converting arbitrary formulas into CNF.

Under this interpretation, if a clause contains both p_i and $\neg p_i$ for any variable p_i, then it will be true under any truth assignment. Hence, it will play no role in determining whether or not a formula F is satisfiable, and we may discard all such clauses from F.

Given a formula F the satisfiability question is "does there exists a truth assignment under which F is true." If there is then F is called *satisfiable*, otherwise it is *unsatisfiable*. This problem corresponds to finding an assignment of 0 or 1 to every boolean variable in F such that the assign-

```
DavidPutnam(F)
   for I = 1 to NumberOfVariablesIn(F)
      V := pickNextVariable()
      Resolvants := {}
      forall (C1,C2) such that
            1. C1 ∈ F and C2 ∈ F
            2. V ∈ C1
            3. ¬V ∈ C2
            Resolvants := Resolvants ∪ resolve(C1,C2)
               //Don't generate tautological resolvants.
      ClausesWithV := {C : C ∈ F and V ∈ C}
      F := F - ClausesWithV
      F := F ∪ Resolvants
         //V is not in any clause in Resolvants. So now V is not in F.
   if F == {}
      return(FALSE)
   else
      return(TRUE)
```

Figure 4.1: The original Davis Putnam Procedure

ment satisfies all of the constraints, where the constraints are specified by the fact that each clause in F must be satisfied. It is the problem first shown by Cook to be NP-Complete.

There are many restrictions one can impose on F, and these restrictions generate sub-classes of the satisfiability problem, some of which are easier than the general problem. If we restrict the clauses in F to contain no more than k literals we have the k-SAT problem. For $k = 2$ this problem can be solved in polynomial time (linear in fact), for $k \geq 3$ the problem is still NP-Complete. There are various other restrictions, e.g., Horn-Sat is the class where every clause in F contains at most one positive literal.

Given two clauses C_1, C_2, such that $p_i \in C_1$ and $\neg p_i \in C_2$, we can generate a new clause by *resolution*. This new clause is $C_1 \cup C_2 - \{p_i, \neg p_i\}$ and is called the *resolvant*.[1] It can be shown that F is unsatisfiable if and only if there is a series of resolutions (where each step may use the clauses produced by any of the previous steps or may simply resolve two clauses in F) that yields the empty clause.

The original algorithm for solving satisfiability problems was due to Davis and Putnam. The algorithm is shown in Figure 4.1. This algorithm is an example of a bucket elimination algorithm, which we will discuss later.

The procedure captures all of the ramifications of each variable by computing all of the resolvants that could be generated using that variable. Then all of the clauses with that variable can be discarded. The end result is an empty clause if and only if there are a sequence of resolution steps that could produce an empty clause. So if an empty clause results the procedure returns FALSE indicating that the original formula is unsatisfiable. (Otherwise the formula is satisfiable).

[1] Again if the new clause contains both the positive and negative instances of a literal we can discard it. So we will assume that resolution is restricted so that it is not allowed to operate on two input clauses that would generate a tautological resolvant.

```
DPLL(F)
   if F is empty
      return(TRUE)
      //The truth assignments made along this branch
      //satisfy the original formula F.
   if F contains an empty clause
      return(FALSE)
   if there is a pure literal l ∈ F
      return(DPPL(Reduce(F,l)))
   if there is unit clause {l} ∈ F
      return(DPPL(Reduce(F,l)))

   v := pickNextVariable()
   if DPPL(Reduce(F,v)) == TRUE
      return(TRUE)
   return(DPPL(Reduce(F,¬v)))

Reduce(F,l)
   F' := {}
   forall C ∈ F
      if {l,¬l} ∩ C == {}
         F' := F' ∪ {C}
      else if ¬l in C
         F' := F' ∪ {(C - ¬l)}
   return(F')
```

Figure 4.2: Davis Putnam Logemann Loveland Procedure

Unfortunately, the original Davis Putnam procedure has serious problems with the amount of space it consumes (although, to the best of my knowledge, there has no serious attempt to build and test a modern implementation). It could potentially generate a quadratic (quadratic in the number of variables in the current formula F) number of new clauses at every stage.[2]

This leads to a modified procedure due to Davis, Logemann, and Loveland, DLL.[3] This procedure is essentially the same as forward checking in CSPs. The only difference is that the algorithm utilizes special processing to check the constraints—these constraints all have a particular structure so one does not need to use the standard (generic) checking algorithms. Like FC it chooses a variable, splits on the two possible values of the variable, and explores the subproblem under each split. The algorithm is shown in Figure 4.2.

In the algorithm there are two special cases that are tested for prior to selecting the next variable to split on. First we test to see if there is a pure literal in the current formula F. A pure literal is a literal l such that $\neg l$ does not appear in any clause of F. We can immediately assign l the value 1, with no ramifications on the satisfiability of F. The other special case is that if we have a unit clause, we can immediately assign the literal it contains the value 1. This is the only assignment that will satisfy the unit clause. Finally, if neither of these special cases hold, we select a variable

[2]Nevertheless, recent studies have show that some degree of this style of "bottom up" resolutions are useful in quickly solving a satisfi ability problem.

[3]Although in the literature, DLL is commonly referred to as Davis Putnam.

to split on and then recursively examine the formula that arises from setting that variable to true (1) and to false (0).

When we assign a literal ℓ the value 1 the algorithm reduces the input formula by removing from it all clauses that have been made true by this assignment. This is the set of clauses that contain ℓ; if we can make the rest of clauses in the formula true, then these clauses will also be true. Furthermore, it reduces all the clauses that contain $\neg\ell$ by removing $\neg\ell$ from them. For these clauses setting $\neg\ell$ to be true is no longer an option for making these clauses true. The search must now find some other literal in the clause to set to 1.

4.1 The relationship with CSPs

The DPLL algorithm looks a bit different from backtracking CSP search. In particular, there is no consistency checking, nor any constraint propagation. However, these components do exist, they are simply disguised by the fact that DPLL is taking special advantage of the nature of the constraints, and the manner the constraints are represented.

The most obvious translation between a SAT problem and a CSP is to retain the set of boolean variables and their $\{0,1\}$ domains, and to translate the clauses into constraints. Each clause becomes a constraint over the variables in the constraint. Under this translation we can examine the processing DPLL performs and see what its analogue would be when solving the problem as a CSP.

First, when DPLL splits on a variable and searches the sub-Formula generated by the different values of the variable, this corresponds the same event in the CSP—splitting on the variable and recursing examine the consequences of each of the value assignments to that variable.

In backtracking search when we make a sequence of variable assignments we are conceptually reducing the original CSP to form a sub-CSP.[4] In generic CSP search, the sub-CSP generated by the assignments at a node is never explicitly represented. Rather the sub-CSP is accessed implicitly while searching the subtree below the node. For example, if C is a constraint over variables V_1, V_2, and V_3, then when we make the assignment $V_1 \leftarrow a$, C will be reduced in the sub-CSP generated to a "new" constraint C' over V_2 and V_3 such that $C'_{V_2,V_3}(x,y)$ is true if and only if $C_{V_1,V_2,V_3}(a,x,y)$ is true. A representation of C' is never explicitly generated. Rather we can simply test whether or not a pair of values (x,y) satisfies C' by simply testing whether or not the triple of values (a,x,y) satisfies the original constraint C.

When the constraints are clauses, however, it becomes easy to generate an explicit representation of the reduced constraints, and this is what DPLL does when it reduces the formula to generate a sub-formula.

If we make the assignment $V \leftarrow 1$, and we have a constraint that arose from the clause $(V, l_1, \ldots, l_n)$, then it will no longer matter what values we subsequently assign to $l_1, \ldots, l_n$; the constraint will always be satisfied in the subtree below this assignment (it has been reduced to the universal constraint). Thus we could legitimately remove this constraint from the sub-CSP, as

[4] See Chapter 2.1 for a definition of the sub-CSP.

does DPLL. However, the standard CSP algorithms do not remove constraints as they search the backtracking tree. DPLL has the advantage that due the special structure of clauses it can detect immediately that the constraint has been reduced to a universal constraint, and hence that it can be deleted.[5]

On the other hand, say that we have a constraint that arose from the clause $(\neg V, l_1, \ldots, l_n)$. It is not hard to see that the reduced constraint is simply the clause $(l_1, \ldots, l_n)$. So by rewriting the clause DPLL is simply generating the reduced constraint.

Finally, recall that forward checking is simply enforcing node-consistency in the sub-CSP. Thus it need only worry about newly generated unary constraints. These are precisely the unit clauses generated by the above clause reduction process. Each of these unit clauses, will under forward checking reduce the domain of the variable to a single value. It is always a good idea to instantiate variables with forced values, and this is what the special case for unit clauses accomplishes.

The only other case to consider it the special rule for pure literals. For this case there are a few things to notice

1. DPLL as specified simply tests for satisfiability. It is easy to modify it so that it enumerates all satisfying truth assignments. In this case the pure literal rule cannot be used, as it could remove some satisfying truth assignments from the search space.

2. Otherwise if we are only searching for the first solution, the pure literal rule again is simply an exploitation of the special nature of the constraints to detect a case where an assignment can be forced without effect on whether or not a solution can be found.

3. In practice the pure literal rule is usually not employed, as it generally it offers no significant savings.

4.1.1 Intelligent Backtracking

The core DPLL algorithm does not perform any intelligent backtracking. But it can be made to do so by importing ideas from CSPs. This was first done by Baryardo and Schrag. The idea is quite simple.

First, let us associate a level with every call to DPLL. So starting with level 1, every time we recurse we increment the level. Furthermore, let us remove the pure literal rule. Now we consider all the cases where no-goods (represented as sets of levels) are discovered by the algorithm.

- When splitting. Every time we split on variable at level ℓ, we first assign it one of its two values (in the pseudo code we always choose the value 1 first). This means that the value 0 for that variable has become invalid (a variable can only have one value). Thus ℓ is a no-good for the other value.

[5] Standard CSP algorithms do not have a representation of the constraint that permits a quick detection of the fact that the reduced constraint has become universal. This leads to an interesting research question: how to represent n-ary constraints so that when we assign variables of the constraint during search we can quickly update the representation and use it to detect when the constraint has become universal.

- When unit propagation is perform. Say that we have a clause reduced to an unit clause. Originally it may have contained other literals. Say that originally the clause was $(l_1, \ldots, l_n)$, and say that it has been reduced to the unit clause (l_n). Then by induction we can assume that each of the literals $l_1, \ldots, l_{n-1}$ has an associated no-good. But then the union of these no-goods is a no-good for $\neg l_n$.

  That is, the union of the no-goods for the other literals in the original clause is a reason why none of these literals could be true. Thus it is a reason why l_n must be true, and a reason (i.e., a no-good) why $\neg l_n$ cannot be be true.

- When we reach a dead end because we have an empty clause in F. If this clause was originally the set of literals $(l_1, \ldots, l_n)$ then by induction we will have a no-good associated with each of these literals. We can union these no-goods together, backtrack to the maximum level in the union, and set the remaining levels in the no-good as a new no-good for the value that was assigned at the backtrack level.

Putting these things together we obtain an intelligent backtracking version of DPLL (this is essentially the relsat algorithm developed by Bayardo and Schrag) shown in Figure 4.3. Note that, although the algorithm does not specify it, the nogood sets associated with each literal are, as normal, stored as sets of levels, and it is necessary to keep track and empty nogood sets that become invalid as we backtrack above their maximum level.[6]

Note that we could also quite easily add domain pruning to this algorithm,[7] and also constraint filtered no-good unioning. But no one has reported trying these ideas in the literature as yet.

4.1.2 Implementation

In practice the DPLL algorithm needs to be able to examine millions on nodes during a search, and perform 100s of millions of unit propagations. This leads to some practical requirements.

1. We cannot implement the reduction of the formula as actually making a new copy. This would take too much time and space.

2. We do not want to compute no-goods as specified in the Figure 4.3. Specifically we want to compute the unions only as we need them.

The basic processing of DPLL is very simple. Only two things occur. First, a literal is valued, either because it was choosen during backtracking, or because it was forced by a unit clause. When a literal is valued we need to detect whether or not a contradition has occured and whether or nto any new unit clauses have been generated. The first is easy, a contradition is generated if and only if the literal's negation was previously valued. The second is only sightly involved.

[6] The original algorithm contains some optimizations where by all unit propagations are performed prior to recursing to the next level.

[7] With binary variables, domain pruning generates singleton values. These need to be treated in the same way as unit clauses—we must force the algorithm to instantiate the remaining value before trying anything else.

```
RelSat(F,level)
   if F is empty
      The current assignments are a satisfying truth assignment
      return(0)

   if F contains an empty clause and the original clause is C
      NoGood := ∪ℓ ∈ C NoGoodSet[ℓ]
      BTLevel := max(NoGood)
      NoGoodSet[LiteralAt[BTLevel]] := NoGood - {BTLevel}
      return(BTLevel)

   if there is unit clause {l} ∈ F and the original clause was C
      NoGoodSet[¬l] := ∪ℓ ∈ C, ℓ ≠ l NoGoodSet[ℓ]
      BTLevel := DPPL(Reduce(F,l), level+1)
      if BTLevel == level
         NoGood := NoGoodSet[l] ∪ NoGoodSet[¬l]
         BTLevel := max(NoGood)
         NoGoodSet[LiteralAt[BTLevel]] := NoGood - {BTLevel}
      return(BTLevel)

   v := pickNextVariable()
   NoGoodSet[¬v] := {level}
   BTLevel := DPPL(Reduce(F,v), level+1)
   if BTLevel < level
      return(BTLevel)

   BTLevel := DPPL(Reduce(F,¬v), level+1)
   if BTLevel == level
      NoGood := NoGoodSet[l] ∪ NoGoodSet[¬l]
      BTLevel := max(NoGood)
      NoGoodSet[LiteralAt[BTLevel]] := NoGood - {BTLevel}
   return(BTLevel)
```

Figure 4.3: DPLL with intelligent Backtracking

A simple implementation for detecting new unit clauses would be to have a counter and a flag associated with every clause, and an index from each literal to the set of clauses it appears in. Initially all of the flags are false, and the counters are set to be equal to the clause length. When we value a literal, we mark the flag of every unmarked clause it appears in—these clauses are not satisfied. Then we examine all of the clauses that the literal's negation appears in. For every unmarked clause, we decrement the clause length. If the new length is one (1), we then search the clause to find the single unfalsified literal it contains, and unit propagate that literal. When we backtrack we must undo any changes we made to the clause length counters, and to the clause flags.

The new (and recent) standard in DPLL implementations is due to the work of Sharad Malik and his group at Princeton University. Using ideas from previous work (notably Zhang and Stickel) they have developed a very efficient SAT solver called ZCHAFF.

They have developed a much more efficient way of detecting unit clauses. Their method is choose two literals from each clause to link to the clause. None of the other literals index into the clause. These literals become the "watch" literals for the clause. We could instantiate other literals

in the clause, but this will not invoke any tests on the clause. The only time we check to see wether or not a clause has become unit is when one of its watch literals is falsified. At that time we look for another watch literal (distinct from the sole remaing watch literal). If during this search we detect a true literal, we do nothing—the clause is already satisfied, similarly if during the search we fail to find an new watch, we know that the other watch is to be forced by unit propagation.

Furthermore, on backtracking we need not modify the watch variables, even if we had moved them while descending down the current search path—any pair of watch variables is equally good.

This process reduces the work required to detect unit propagation by a factor equal to the length of the clause. Furthermore, there will often be many clauses that are never touched along any single path of the search tree.

The second part of a good implementation is a better way of manipulating no-goods. Rather than pre-computing the union of the no-goods that generated a unit propagation, it is easier to compute that union only when required. Thus when we find a contradition, which is a case where both a literal ℓ and its negation have been forced by unit propagation, we recursively compute the no-goods associated with each of the literals in clauses that forced them.

4.2 Branching Heuristics

Given the simplicity of the DPLL algorithm an interesting component of the work on building faster DPLL based satisfiability testers centers around the development of better variable selection heuristics. These branching heuristics invariably try to branch into the most constrained formula. In particular, they try to pick a variable that generates the most new clauses of shortest length (after unit propagation).

One system called satz, uses extensive one step look ahead to choose its next variable. In particular, it has a series of filters that it employs. Each filter lets more unassigned variables through. It finds the first filter that lets at least T variables through, where T is an empirically determined parameter.

Once it this sufficiently large set of filtered variables, it proceeds to test each one by successively assigning the variable true and then false. After each assignment it performs unit propagation, and then it counts the number of newly produced binary clauses. Let p be the number of new binary clauses produced by setting the variable to true, and n be the number of binary clauses produced by setting the variable to false. The heuristic score of the variable then becomes

$$p * n * 1024 + p + n$$

Note that this score penalizes those variables that generated good sub-formulas under one assignment but not under the other: it tends to promote balanced trees.[8] The procedure then branches on the variable with highest heuristic score.

[8]The success of this approach would seem to argue that a possible tie breaker for the simple minimum current domain heuristics used in CSPs, would be to estimate the effect of each of the possible values with a preference for balance between the values.

One other effect of this look ahead is that the procedure can detect *failed literals*. These are literals ℓ such that assigning true to them and then doing unit propagation yields a contradiction. For such variables, we can immediately assign them the value false. For example, in the formula

$$\Big\{(a, b, \neg c), (\neg a, \neg c), (\neg b, \neg c)\Big\}$$

c is a failed literal.

The ZCHAFF system uses a different heuristic, and it does clause learning as well and random restarts as well. These will be discussed later.

Recently, a good heuristic based on backbones has been developed (see the readings). Backbones will be discussed later.

4.3 DPLL and Resolution Proofs

Consider a formula that is unsatisfiable. DPLL will be able to prove that this formula is unsatisfiable. What is interesting is that the search tree explored by DPLL contains a resolution proof of unsatisfiability. If one starts at the leaf nodes and works back up the search tree, one can construct a resolution refutation of the formula.

A leaf node is one where we tried a variable and found that both of its values generate an empty clause. It could be that the value of the variable at the leaf node was set by a unit clause. But in that case, the other value, if it has been used, would have generated an empty clause. In particular, it would have cause the unit clause to become empty. In general, then there will be two empty clauses, one generated by each of the values of the variable at the leaf. These two empty clauses must contain complementary values of the variable (else, they would have appeared higher up in the tree and we would not have reached this leaf node). So we can resolve together each of these clauses. If we do this an pass the resolvant back to the parent, we can examine the other value of the parent, and by induction it also will have a resolved clause passed back to it if we work from the leafs of the search tree it generated. It can be shown that these two clauses must have complementary literals, and we can do another resolution and again pass the resolvant back up the tree.

In fact, one can show that this is exactly what the unioning of no-goods accomplishes. If one proceeds in this manner, one will obtain a tree structured collection of resolvants that derives the empty clause from clauses (at the terminal branches) in the original formula. Thus it is a resolution proof of the unsatisfiability of the original formula.

This has important ramifications about the complexity of DPLL, and thus of CSP backtracking search. In particular, the size of this resolution refutation is polynomial in the size of the explored search tree. Yet it is know that for many formulas there is no resolution refutation that is less than exponential in size. Note that this is a rather stronger result than NP-Completeness. It is a result that says that DPLL, and thus CSP backtracking must have exponential complexity in the worst case, irrespective of the question $P = NP$.