Nogood processing in CSPs

by

George Katsirelos

A thesis submitted in conformity with the requirements for the degree of Phd
Graduate Department of Computer Science
University of Toronto

Copyright © 2008 by George Katsirelos
Abstract

Nogood processing in CSPs

George Katsirelos
Phd
Graduate Department of Computer Science
University of Toronto
2008

The constraint satisfaction problem is an NP-complete problem that provides a convenient framework for expressing many computationally hard problems. In addition, domain knowledge can be efficiently integrated into CSPs, providing a potentially exponential speedup in some cases.

The CSP is closely related to the satisfiability problem and many of the techniques developed for one have been transferred to the other. However, the recent dramatic improvements in SAT solvers that result from learning clauses during search have not been transferred successfully to CSP solvers. In this thesis we propose that this failure is due to a fundamental restriction of nogood learning, which is intended to be the analogous to clause learning in CSPs. This restriction means that nogood learning can exhibit a superpolynomial slowdown compared to clause learning in some cases. We show that the restriction can be lifted, delivering promising results.

Integration of nogood learning in a CSP solver, however, presents an additional challenge, as a large body of domain knowledge is typically encoded in the form of domain specific propagation algorithms called global constraints. Global constraints often completely eliminate the advantages of nogood learning. We demonstrate generic methods that partially alleviate the problem irrespective of the type of global constraint. We also show that more efficient methods can be integrated into specific global constraints and demonstrate the feasibility of this approach on several widely used global constraints.
Acknowledgements

I have been very fortunate to be advised by Fahiem Bacchus. He is an excellent researcher and teacher and our meetings always left me with more insight into the field and more motivation to explore further. Although I value his scientific advice greatly, I value even more the fact that he insisted on high quality research throughout. I can only hope that I will live up to his high standards in my own career. I would also like to thank the other – current and past – members of my advisory committee: Hector Levesque, Toniann Pitassi, Sheila McIlraith, Chris Beck and the external examiner Peter van Beek. I appreciated their detailed feedback on drafts of this thesis, which often challenged me to consider my work in a wider context. Their input greatly improved the quality of the final document.

Of course, I must also thank my family and friends for being there and having fun with me during all this. It took me a long time to finish, so there are too many names to list here. In the unlikely event any of you ever gets to read this, you know who you are. Whether we met in Toronto, Athens or Sydney, thanks. I will, however, mention Anastasia, who has been more important than anything else and has made this possible and worthwhile.
Contents

1 Introduction 1
  1.1 Backtracking search 3
  1.2 Our contributions 5

2 Formal Background 7
  2.1 Backtracking search for CSP 7
    2.1.1 Constraint Propagation 10
    2.1.2 Intelligent backtracking 16
    2.1.3 Variable Ordering 19
  2.2 SAT and Resolution 21
  2.3 DPLL 24
    2.3.1 Proof complexity of DPLL 26
    2.3.2 Clause learning 27
    2.3.3 State of the art in SAT solving 29
  2.4 Mapping CSPs to SAT 33
    2.4.1 Relation of FCCBJ to DPLL 34
    2.4.2 Adding learning to FCCBJ 35
    2.4.3 Maintaining generalized arc consistency 37

3 Generalized nogoods 39
  3.1 Generalizing nogoods 39
    3.1.1 Learning generalized nogoods 40
    3.1.2 Using g-nogoods during search 46
  3.2 Generalized nogoods are more powerful 52
    3.2.1 Expressive power 53
    3.2.2 Local pruning power 56
3.2.3 Global power .................................................. 59
3.3 Related work .................................................... 64
3.4 Empirical results ............................................... 65

4 Nogoods in constraint propagation 69
4.1 The problem .................................................... 69
4.2 Generic methods ............................................... 73
  4.2.1 Empirical evaluation ....................................... 74
  4.2.2 Improving on GAC-Generic-Nogood ..................... 79
  4.2.3 GAC-Generic-Nogood for bounds consistency propagators... 81
  4.2.4 A generic method for generating minimal labels .......... 82
  4.2.5 GAC schema ................................................. 83
4.3 Constraints with specialised propagators .................. 96
  4.3.1 Less-than .................................................. 96
  4.3.2 The cardinality constraint on binary variables ......... 97
  4.3.3 The lexicographic ordering constraint ................. 98
  4.3.4 Alldifferent constraint .................................. 103
  4.3.5 Global Cardinality Constraint .......................... 107
  4.3.6 Specifying counting and occurrence problems as combinations of the RANGE and ROOTS constraints .... 111

5 Logical combinations of constraints 118
5.1 The constraint language ...................................... 118
5.2 Computing nogoods .......................................... 121
  5.2.1 Entailment and disentailment ............................ 123
  5.2.2 Minimality ................................................. 124
5.3 GAC-Schema as a constraint expression .................... 125
5.4 Further generalization of nogoods .......................... 126
  5.4.1 Branching .................................................. 128

6 Conclusions ...................................................... 129

Bibliography ....................................................... 132
List of Figures

2.1 Plain backtracking search ........................................... 8
2.2 Forward checking...................................................... 11
2.3 Making a problem generalized arc consistent ......................... 13
2.4 Forward checking with conflict directed backjumping ................. 17
2.5 An example showing the width, induced width and tree decomposition of a constraint graph ........................................ 20
2.6 A resolution refutation as a DAG ..................................... 23
2.7 The DPLL algorithm .................................................... 25
2.8 Simplifying a formula by unit propagation ............................. 25
2.9 The correspondence between a DPLL refutation and a resolution refutation ........................................ 27
2.10 FCCBJ+S: FCCBJ with nogood learning ............................... 36

3.1 FCCBJ+S: FCCBJ with nogood learning .............................. 41
3.2 An example of an assignment trail and the corresponding nogood labels ........................................ 43
3.3 Computing a first-decision nogood after a domain wipeout .............. 44
3.4 An example of the operation of the 1st-Decision scheme. ............... 44
3.5 Forward checking with g-nogood learning ................................ 47
3.6 The specialization of propagate for forward checking ............... 48
3.7 A high level description of unit propagation .......................... 48
3.8 Resolving a single conflict using the first-decision scheme .......... 49
3.9 Making a non-assignment false ......................................... 49
3.10 Generic trail unwinding to resolve a conflict ......................... 50
3.11 Instantiation of resolve-unwind-trail to implement the 1-UIP scheme 50
3.12 Transforming a single g-nogood to a set of s-nogoods ................. 53
3.13 Proving unsatisfiability of MPH_a using g-nogood learning ............ 62
3.14 Comparison of FC based algorithms on crossword puzzle problems ... 66
4.21 The graphs generated for a GCC constraint .................................. 109
4.22 Enforcing HC on the RANGE constraint ........................................... 113

5.1 Computing $Valid(C, D)$ and $Inc(C, D)$ for constraint expressions ........ 120
5.2 Computing $Nogood(A, C, D)$ and $Good(A, C, D)$ for any assignment or non-assignment $A$ ................................................................. 122
5.3 Computing $Nogood(A, C, D)$ and $Good(A, C, D)$, using c-nogoods .......... 127
Chapter 1

Introduction

Many hard real world problems can be effectively modeled as finite domain constraint satisfaction problems. Informally, the problem seeks to make an assignment to $n$ finite domain variables, each with $d$ values in its domain, such that a set of constraints is satisfied. Common applications include scheduling, verification, design, configuration and games [Gent and Walsh, 1999].

A simple example that demonstrates how a problem can be modelled as a CSP is the $n$-queens problem. In it, the aim is to place $n$ queens on an $n \times n$ chessboard so that no queen can attack any other in one move. One CSP model is to use $n^2$ binary variables, each one representing a square of the chessboard. A variable is assigned the value 1 if a queen is placed on the corresponding square, 0 otherwise. Constraints are added for each row, column and diagonal so that no more than one queen can be placed on them. Another, more effective model, recognizes the fact that only a single queen can be placed on each column, so it uses only $n$ variables each with domain $\{1, \ldots, n\}$. Each variable represents a column of the board. Variable $V_i$ gets the value $j$ if the queen in the $i^{th}$ column is placed on the $j^{th}$ row. Constraints are then added for every pair of variables $V_i, V_j$ so that: the queens are on different rows, captured by the condition $i \neq j \Rightarrow V_i \neq V_j$; the queens are on different diagonals on the southwest-northeast direction, captured by the condition $i \neq j \Rightarrow V_i - i \neq V_j - j$; and the queens are on different diagonals on the northwest-southeast direction, captured by the condition $i \neq j \Rightarrow V_i + i \neq V_j + j$. This model achieves a smaller search space $n^n$, as opposed to $2^{n^2}$ for the first model, helping search algorithms to find solutions faster. Yet a third model utilizes the ability of CSPs to embed advanced propagation algorithms to improve their ability to dynamically prune the search space. In this case the model adds the
alldifferent constraint. An alldifferent constraint between \(k\) variables states that each of
these variables must have a different value. This third model augments the set of variables
used in the second model by adding variables \(V_{n+1} \ldots V_{3n}\). The constraints of this third
model are as follows: \(V_{n+i} = V_i + i\) for \(i = 1, \ldots, n\); \(V_{2n+i} = V_i - i\) for \(i = 1, \ldots, n\);
\(\text{alldifferent}(V_1, \ldots, V_n)\); \(\text{alldifferent}(V_{n+1}, \ldots, V_{2n})\); and \(\text{alldifferent}(V_{2n+1}, \ldots, V_{3n})\). We
note that because the new variables have values that are determined by the original
variables, during a backtracking search we need never explore a larger search space. We
also see that semantically the third model is the same as the second, but it allows us to
use advanced propagation algorithms known for the alldifferent constraint to solve the
problem faster.

Another interesting problem commonly modeled as a CSP is the quasigroup comple-
tion problem. In this problem, we are given an \(n \times n\) grid with some values filled in.
The objective is to fill all the squares in the grid with values in the range \(1 \ldots n\) so that
each value appears exactly once in each row and each column. A straightforward way to
implement this is to have one variable with domain \(\{1, \ldots, n\}\) for each grid square, so
that variable \(V_{i,j}\) has the value \(k\) if the grid square \(i, j\) has the value \(k\). We then fix the
value of the prefilled squares appropriately and post alldifferent constraints for each row
and each column.

The CSP problem is closely related to SAT. It can be seen as a generalization of
SAT to multi-valued variables and constraints more general than propositional clauses.
As such, it is trivially NP-hard. This generalization, however, makes CSP a much more
expressive framework for modeling practical problems. Even though formally the two
frameworks are equivalent, the features of CSPs that make them attractive are:

- Multivalued variables, as opposed to binary variables, can more naturally represent
  real world concepts, such as the type of shift assigned to an employee and the
  number of employees that must be assigned to a shift in scheduling, the identity of
  a node of a graph, the warehouse where an item is located in a logistics planning
  problem and so on.

- The ability to have non-clausal constraints allows us to express complex relations-
  ships between some sets of variables as a single constraint. Examples include arith-
  metic constraints, the alldifferent constraint mentioned earlier, and others. An
  instructive example is the \textit{parity} constraint, which states that an odd number of
  variables from the vector \(x\) can be made true. Translating this to a SAT representa-
tion directly would require an exponential number of clauses in the size of the vector $\mathbf{x}$ [Darwiche and Marquis, 2002]. In this case, there exists a linear size binary decision diagram (BDD) that represents this constraint [Darwiche and Marquis, 2002], which can then be converted to CNF [Eén and Sörensson, 2006], giving a linear transformation. In any case, a polytime computable constraint can be transformed to CNF from the corresponding Turing Machine [Bacchus, 2007]. However, it is more natural to express the constraint directly.

- It is easy to integrate specialized or domain specific knowledge in CSP solvers in the form of specialized constraint propagators. For example, there exists a polynomial time propagator for the alldifferent constraint mentioned earlier [Régin, 1994]. In SAT, with clausal representations, we are mainly limited to resolution as the inference mechanism. Since the pigeon hole problem is easily encoded by a single alldifferent constraint, resolution would require exponential time [Haken, 1985] to infer what can be inferred in polynomial time by the specialized alldifferent constraint.

## 1.1 Backtracking search

There are three main classes of algorithms for solving CSPs: backtracking search, local search and dynamic programming. In this thesis, we deal only with backtracking search algorithms, although some of the caching techniques utilized in backtracking are closely related to dynamic programming methods [Bacchus et al., 2003].

Plain backtracking search is by itself not very useful. Modern CSP solvers always implement some form of constraint propagation. Constraint propagation involves preforming additional reasoning at each node of the backtracking search tree to eliminate impossible values for the as yet unassigned variables. The different methods of propagation vary in terms of the complexity of the inference they preform in order to detect these impossible values. Commonly used forms of constraint propagation include forward checking [Haralick and Elliott, 1980] (FC) and maintaining generalized arc consistency [Mackworth, 1977a] (MAC). In addition, features like intelligent backtracking [Prosser, 1993] and learning [Dechter, 1990, Frost and Dechter, 1994, Ginsberg, 1993, Bayardo Jr and Miranker, 1996] have been proposed, but are not used as frequently. These approaches have often been presented using an ad hoc approach, which might
obscure their relationship and make it hard to prove their correctness. Moreover, it is not always easy to combine them [Prosser, 1995]. It has been observed that in effect, all these algorithms discover and use nogoods [Bacchus, 2000]. Traditionally, nogoods are defined as sets of assignments that cannot be part of any solution. In [Bacchus, 2000] it was shown that any algorithm that discovers correct nogoods and uses them correctly is complete. We will present this view of backtracking in more detail in chapter 2.

Backtracking search is also the basic algorithm for most modern SAT solvers. In particular, these solvers are based on DPLL [Davis et al., 1962], which is a backtracking search solver utilizing a basic level of constraint propagation called unit propagation. Modern SAT solvers extend DPLL with conflict resolution and clause learning techniques [Marques-Silva and Sakallah, 1999, Moskewicz et al., 2001], yielding a family of algorithms called CDCL (conflict driven clause learning), that have proven very successful in solving many hard real world instances. Noting that nogoods are equivalent to clauses, one would expect that nogood learning should produce equally impressive results when solving CSPs as clause learning has produced when solving SAT, but this has not been the case.

In this thesis, we investigate the reasons for this. First, we note in chapter 2 that both CSP backtracking search and DPLL (and its derivative CDCL) can be p-simulated by resolution. This means that every CSP (DPLL) search that proves an instance is unsatisfiable can be transformed into a resolution refutation for the instance that is at most polynomially larger than the search tree. In fact, non-learning backtracking search and DPLL can be p-simulated by tree-like resolution, a limited form of resolution that may only use a clause once along each path to the empty clause. There exists an exponential separation between unrestricted resolution and tree-like resolution, which means that there exist infinite families of instances which have polynomial sized unrestricted resolution refutations, but for which the shortest tree-like resolution refutation is exponential in size. Since non-learning CSP backtracking search and DPLL can be p-simulated by tree-like resolution, these instances will also be hard for them. CDCL cannot be p-simulated by tree-like resolution. In fact, it has been shown that CDCL cannot be p-simulated by regular resolution [Beame et al., 2004], which is significantly stronger than tree resolution. CSP backtracking search with nogood learning does not share all of the advantages of CDCL, however. It has been shown that because of the traditional definition of nogoods, backtracking with nogood learning can be p-simulated by negative resolution [Mitchell, 2003], a restriction of resolution that requires that one of the input clauses
of each resolution step contains exclusively negative literals. Unrestricted resolution is exponentially separated from negative resolution [Buresh-Oppenheim and Pitassi, 2003]. This means that although nogood learning for CSPs moves backtracking beyond tree-like resolution, it does not move it beyond negative resolution.

Despite these problems, the integration of specialized propagation algorithms for many constraints makes CSP backtracking more powerful than resolution in some aspects. Some of these algorithms can in polynomial time infer impossible values (prunings) which would need an exponential number of resolution steps to derive. Not every problem which is hard for resolution can become easy with the use of these propagators, but sometimes this is the case. For example, the pigeon hole problem can be solved in polynomial time using the propagator for the alldifferent constraint. Even then, the presence of the pigeon hole problem needs to be explicitly identified and modelled in the CSP by an alldifferent constraint. This means that should an instance of this problem arise dynamically while solving some instance, the solver might not be able to identify it and thus would not be able to apply its faster specialized inference procedure.

1.2 Our contributions

In this thesis, we explore the connection between CSPs and resolution more deeply. First, we show how to circumvent the negative resolution limitation of CSP solvers by introducing generalized nogoods\footnote{Generalized nogoods were also introduced independently in [Mitchell, 2003].} in chapter 3. These are nogoods that may contain non-assignments (or prunings) in addition to assignments. We show how they can be discovered and used in CSP solvers. We discuss first the intuitive aspect of why it helps to discover generalized nogoods. We then study how backtracking algorithms can be adapted to work correctly in the presence of generalized nogoods. Finally, we show that generalized nogoods are both locally more powerful, in that each one can prune more values than the corresponding standard nogood, and that they are also globally more powerful. Specifically, from the backtracking algorithm $A$ that does no nogood learning, we can derive two algorithms, $A_S$ that performs standard nogood learning and $A_G$ which learns generalized nogoods. We show that $A_G$ is super-polynomially separated from $A_S$.

Further, in chapter 4, we investigate how generalized nogoods can be learned from the specialized algorithms used to propagate various constraints. First, we show that
standard techniques for learning nogoods from constraint propagation can often produce
useless nogoods. We then show how to discover generalized nogoods from any constraint,
which by itself can improve performance significantly. Still, it is important to discover
smaller or minimal nogoods whenever possible. We present methods for discovering
smaller nogoods both for general, extensionally represented constraints, as well as for
several widely used global constraints. This is important, as it allows us to integrate the
specialized non-resolution local reasoning that takes place within these algorithms with
the global power of resolution.

We also present empirical results that show that learning generalized nogoods can
be worthwhile in practice. We present a range of results covering several algorithms as
well as a range of problem domains. In some cases, the resulting algorithms can be up
to three orders of magnitude faster than the previous state of the art, but sometimes
the gain in reduction of the search space is offset by the overhead of learning generalized
nogoods.

In chapter 5 we examine how generalized nogoods can be learned from constraint ex-
pressions which use logical operators to combine constraints [Bacchus and Walsh, 2005].
We also further extend nogoods to include literals that represent the truth of a con-
straint. This generalization provably improves the power of the nogoods discovered from
constraint expressions. In addition, it provides us with information which can be used to
extend the branching strategy so that we can branch on constraints as well as variables.
Even though this was possible before, it presented a problem in that there was no infor-
mation to use to choose one of the constraints. Extending the branching strategy in this
way also makes the solver more powerful as demonstrated in [Järvisto et al., 2005].

Finally, we conclude in chapter 6 with a recapitulation of the contributions of this
thesis and the opportunities it presents for further research.
2.1 Backtracking search for CSP

A constraint satisfaction problem $\mathcal{P}$ is a tuple $\langle V, D, C \rangle$, where $V$ is a set of variables, $D$ is the domain of values for these variables\(^1\) and $C$ is a set of constraints, whose size is polynomial. Given $V \in V$ and $x \in D$, we say that $V \leftarrow x$ is an assignment, it assigns to $V$ the value $x$. The goal is to find a set of assignments that satisfy the constraints, assigning exactly one value to each variable.

An assignment set is a set of assignments $\mathcal{A} = \{X_1 \leftarrow a_1, \ldots, X_k \leftarrow a_k\}$ such that no variable is assigned more than one value. We use $\text{scope}(\mathcal{A})$ to denote the set of variables assigned values in $\mathcal{A}$. A constraint $C$ consists of a set of variables called its scope, $\text{scope}(C)$, and a polynomial time computable function, which we also denote by $C$, such that $C : D^{\text{scope}(C)} \rightarrow \{0, 1\}$. We say that an assignment set $\mathcal{A}$ to the variables in the scope of $C$ satisfies $C$ if $C(\mathcal{A}) = 1$ and falsifies it otherwise. We also use $C$ as an indicator function for its set of satisfying assignments, so we write $\mathcal{A} \in C$ if $C(\mathcal{A}) = 1$. In general, if $\text{scope}(C) \subseteq \text{scope}(\mathcal{A})$, $\mathcal{A}$ satisfies $C$ if $\exists \mathcal{A}' | \mathcal{A}' \subseteq \mathcal{A} \land \mathcal{A}' \in C$.

An assignment set falsifies (or violates) $C$ if it does not satisfy it. Further, we say that an assignment set $\mathcal{A}$ is consistent if it satisfies all constraints it covers: $\forall C \text{ scope}(C) \subseteq \text{scope}(\mathcal{A}) \Rightarrow \exists \mathcal{A}' | \mathcal{A}' \subseteq \mathcal{A} \land \mathcal{A}' \in C$. A solution to the CSP is a consistent assignment set containing all of the variables of the CSP. Since each constraint is checkable in poly-time, we see that finding a solution to the CSP is in NP. As noted in Chapter 1, SAT is an instance of a CSP, thus finding a solution to a CSP is an NP-Complete problem.

---

\(^1\)Different domains for different variables are allowed, but it is sufficient for this discussion to consider the notationally simpler case where all the variables have the same domain of values.
Associated with every CSP $\mathcal{P} = < V, D, C >$ is a constraint hypergraph $G_\mathcal{P}$. Each variable of $V$ is a vertex in the graph, and each constraint $C \in C$ is a hyperedge containing the variables in $\text{scope}(C)$. If the arity of each constraint in $C$ is at most 2, the CSP is called a binary CSP. The hypergraph associated with a binary CSP is an ordinary graph. Furthermore, the hypergraph associated with non-binary CSPs can be converted to an ordinary graph by replacing each hyperedge by a clique of edges between the variables in the hyperedge.

Given a set of assignments $\mathcal{A}$, the reduction of a constraint $C$ by $\mathcal{A}$ is a new constraint $C_{\mathcal{A}}$, such that $\text{scope}(C_{\mathcal{A}}) = \text{scope}(C) - \text{scope}(\mathcal{A})$ and its satisfying assignments are $C_{\mathcal{A}} = \{ \mathcal{A}' | \mathcal{A}' \cup \mathcal{A} \text{ satisfies } C \}$. The reduction $C_{\mathcal{A}}$ of a set of constraints $\mathcal{C}$ by $\mathcal{A}$ is the result of reducing each constraint by $\mathcal{A}$. The reduction $\mathcal{P}_{\mathcal{A}}$ of a problem $\mathcal{P} = < V, D, C >$ induced by the set of assignments $\mathcal{A}$ is the tuple $< V - \text{scope}(\mathcal{A}), D, C_{\mathcal{A}} >$.

Complete CSP solvers typically use refinements of backtracking search. An outline of the plain backtracking search algorithm is shown in figure 2.1.

**Figure 2.1 Plain backtracking search**

```
BT(l)
1. if All variables are instantiated
2. return success
3. Choose an uninstantiated variable $V$ % $V$ is the current variable
4. foreach value $x \in \text{Dom}(V)$
5. $V \leftarrow x$
6. $\text{conflict} = \text{false}$
7. foreach constraint $C$ s.t. $V \in \text{scope}(C)$
8. if all variables $\in \text{scope}(C)$ are instantiated % $C$ is completely instantiated
9. if $C$ is violated
10. $\text{conflict} = \text{true}$
11. break
12. if $\text{conflict} \equiv \text{false}$
13. if ( BT(l+1) $\equiv$ success )
14. undo+return success
15. undo $V \leftarrow x$
16. undo+return fail
```

Plain backtracking starts with the empty assignment and proceeds incrementally to build a solution. In line 3, it chooses a variable, called the *current variable* and tries to assign a value to it in line 5. Variables that have already been assigned are called *past variables* and unassigned variables are called *future variables*. The assignment effectively extends the *current assignment*, which is the set of assignments to past variables along
with the assignment to the current variable. After making the assignment, it finds in lines 7 and 8 all constraints $C$ such that $V \in \text{scope}(C)$ and an assignment has been made to all of its variables. Such a constraint is said to be completely instantiated. If all completely instantiated constraints are satisfied, the search goes on to recursively assign a value to a future variable in line 13, making it the current one. If no variables remain unassigned, a solution has been found (line 2). If the assignment to the current and past variables cannot be extended to a solution (determined either at line 10 or by an unsuccessful return from the recursive call at line 13), the assignment is undone at line 15 and another value from the domain of the current variable is chosen and the procedure is repeated. If all values have been tried, the search retracts the assignment and backtracks by exiting the loop of lines 4–15 and returning to the previous recursive call where it tries another value for the previously instantiated variable. Always backtracking to try a new value for the previously instantiated variable is called chronological backtracking. Thus an assignment may fail either because it is itself inconsistent or because the recursive calls below it failed to find a solution. The problem is unsatisfiable if the initial recursive call returns fail, and satisfiable if it returns success.

In the rest of this thesis, we discuss algorithms that are variations of this backtracking search algorithm. For brevity, we always assume that an assignment as well as all of its consequences are retracted when a recursive call returns or before a different value is assigned to a variable.

The backtracking algorithm implicitly searches a tree: the root of the tree corresponds to the empty assignment. At each node of the tree an assignment is made and the subtree rooted at that node is explored. The current assignment $A$ is found by following the path from the current node to the root of the tree. This is also called the prefix of the current node. We see that at a node where the current assignment is $A$, backtracking search essentially solves the reduction $P_A$ of the problem $P$. The distance of a node from the root of the tree is called the level of the node. Nodes closer to the root are shallower while those that are further are called deeper nodes. The same terminology is used for levels (shallow level and deep level).

Nogoods are the formalization of the concept that some subtrees of the search tree can be shown to be empty of solutions without search, so we need not visit them.

**Definition 2.1.1** (Nogood [Dechter, 1990]). A nogood $Ng$ for a problem $P$ is a set of assignments that cannot be part of any solution of $P$. 

Observation 2.1.2. If \( N_g \) is a nogood then any set \( N_g' \supseteq N_g \) is also a nogood.

If \( N_g \) is a nogood, then as soon as backtracking search visits a node \( v \) whose prefix contains \( N_g \), we know that no solution can be found in the subtree rooted at \( v \). This is because any extension of the prefix of \( v \) must necessarily contain \( N_g \), but \( N_g \) is not part of any solution, hence the prefix cannot be extended to a solution. The backtracking search can then immediately backtrack.

This means that nogoods allow us to prune the explored search space. The question then becomes how to identify that a set is a nogood.

Proposition 2.1.3. Determining whether a set of assignments \( A \) is a nogood for a problem \( P \) is coNP-hard.

Proof. Let \( P_A \) be the reduction of \( P \) induced by \( A \). Showing that \( A \) is a nogood is equivalent to showing that \( P_A \) is unsatisfiable, which is coNP-hard.

On the other hand, some nogoods can be tractably identified. We see that since a solution must satisfy all constraints, any set of assignments that directly violates a constraint is a nogood. This provides an alternate justification for backtracking when a constraint is violated.

Improvements on generic backtracking focus on three areas: constraint propagation, intelligent backtracking and variable ordering heuristics.

2.1.1 Constraint Propagation

By propagating the effects of past assignments the size of the explored search space can be reduced. The fundamental technique is domain pruning, which removes some values from the domains of the future variables. As soon as an assignment is made, a constraint propagation algorithm can be called to compute the consequences that the past assignments have on the domains of future variables. Values that are removed from future domains are linked to the current assignment, so that when the search backtracks past this assignment, these values can be restored. If the assignment is made at level \( l \), then the value removals associated to it are said to be pruned to level \( l \) and we write \( \text{prune}_l(V \leftarrow a) = l \) for each value \( V \leftarrow a \) that has been pruned to level \( l \).

At any point, the set of values that may be considered for assignment to a future variable \( V \) is called its current domain, \( \text{CurDom}(V) \). If the entire domain of a future
variable is wiped out (a domain wipeout – DWO), i.e., removed by constraint propagation, we have discovered an inconsistency—we cannot assign that future variable any of its values—so we can immediately backtrack. Note that an assignment to a variable can be regarded as reducing its current domain to just the assigned value.

**Figure 2.2** Forward checking.

```plaintext
int FC(l)
1. If all variables assigned return success
2. choose next variable V. Always choose a variable that has had all of its values pruned if one exists.
3. foreach d∈CurDom(V)
4. V ← d
5. forwardCheck()
6. if (FC(l+1) ≡ success)
7. return success
8. return fail
```

**forwardCheck()**

```plaintext
1. Let A be the current set of assignments
2. foreach constraint C s.t. ∃ exactly one V_f ∈ scope(C) with |CurDom(V_f)| > 1
3. foreach d ∈ CurDom(V_f)
4. if A ∪ {V_f ← d} violates C
5. CurDom(V_f) = CurDom(V_f) − {d}
6. if CurDom(V_f) = ∅
7. return
8. if ∃ V s.t. CurDom(V) = {d}
9. V ← d
10. forwardcheck()
```

**Forward Checking.** The simplest type of constraint propagation is *forward checking* [Haralick and Elliott, 1980] and adding forward checking propagation to backtracking results in the forward checking algorithm, denoted by FC and shown in figure 2.2. FC proceeds similarly to plain backtracking, except that in FC after an assignment is made at line 4, the *forwardCheck()* procedure is called at line 5. Whenever all but one of the variables of a constraint C are instantiated, *forwardCheck* checks each of the values in the current domain of the sole future variable against the current assignment to the other variables of C in the loop at lines 3–7. Any values that are found to be inconsistent (i.e., violate the constraint) are removed from the current domain in line 5. These value removals may cause some future variables to have a current domain of size 1, in which case the variable V_f is *forced*, we can commit to that remaining value as an assignment in
line 9, and apply forward checking again in line 10. This is done until an inconsistency is found in line 6 or no more inconsistent or forced values remain. Then the search continues at the next level. If an inconsistency was found by calling \texttt{forwardCheck}, the recursive call will select the variable whose domain was wiped out and immediately return. Unlike plain backtracking search, FC does not have to ensure that the completely instantiated constraints are satisfied. Assignments that would completely instantiate but not satisfy a constraint must have been removed by \texttt{forwardCheck} at some previous level.

**Maintaining Arc Consistency.** A more powerful way of checking the values of future variables is to apply arc consistency, a notion introduced by Mackworth [Mackworth, 1977a]. The algorithm that extends backtracking by ensuring that the values of future variables are arc consistent is called \texttt{MAC}, or maintaining arc consistency. In \texttt{MAC}, we are assured that we can extend any assignment we might make to a future variable by one more variable. The definition of \texttt{MAC} is based on the assumption of a binary CSP, where the arity of all constraints is at most 2. Its generalization to \texttt{n-ary} CSPs yields the notion of generalized arc consistency (GAC) and the algorithm \texttt{MGAC} that maintains generalized arc consistency [Mackworth, 1977b, Mohr and Masini, 1988]. \texttt{MGAC} ensures that for any constraint, we can extend the current assignment by some values for the unassigned variables in the constraint so that the constraint will be satisfied.\footnote{GAC views each constraint in isolation.}

Formally, we define GAC using the notion of supports.

**Definition 2.1.4 (Support).** A satisfying tuple of a constraint \(C\) is a set of assignments to all the variables in scope\((C)\) such that \(C\) is satisfied. A tuple is valid if none of its values has been removed by constraint propagation. A valid satisfying tuple that includes the assignment \(V \leftarrow x\) is called a supporting tuple or a support for \(V \leftarrow x\) in \(C\).

We can define generalized arc consistency for value, constraints and problem using supports.

**Definition 2.1.5 (Generalized arc consistency).** A value \(V \leftarrow x\) is GAC in a constraint \(C\) if there a support for \(V \leftarrow x\) in \(C\). A constraint \(C\) is GAC if all values in the current domains of its variables are GAC. A problem \(P\) is GAC if all of its constraints are GAC.

A generic algorithm that enforces GAC on any constraint is based on trying to identify a support for each value. We can make a constraint GAC by examining each of its values
and removing those that are not GAC. We can make a problem GAC by making all its constraints GAC. Since making a constraint GAC may affect the current domains of its variables, it might cause other constraints to no longer be GAC. Such constraints will have to be reprocessed to again make them GAC. Hence, the process of making a problem GAC is an iterative one that terminates when no more domain values are pruned and all of the constraints have become GAC or until an inconsistency is found.

Figure 2.3 Making a problem generalized arc consistent

```
enforce-gac
1.  CQ = update-CQ(PQ) % PQ is the queue of pruned values
2.  while CQ ≠ {} % CQ is the queue of constraints to be updated
3.    C = choose(CQ) % Choose a constraint C and remove it from CQ
4.    if enforce-gac-constraint(C) ≡ FALSE
5.      return
6.    CQ = update-CQ(PQ)
```

```
enforce-gac-constraint(C)
1.  foreach V ∈ scope(C)
2.    foreach x ∈ CurDom(V)
3.      if ¬∃ supporting tuple for V ← x in C
4.        CurDom(V) = CurDom(V) − {x}
5.      if CurDom(V) ≡ {} 
6.        return FALSE
7.    PQ = PQ ∪ {(V ← x, C)}
8.    return TRUE
```

```
update-CQ(PQ)
1.  CQ = {}
2.  foreach (V ← x, C) ∈ PQ
3.    foreach C′ : V ∈ scope(C) ∧ C′ ≠ C
4.      CQ = CQ ∪ {C′}
5.    PQ = {}
6.    return CQ
```

An outline of the algorithm that makes a problem GAC is shown in figure 2.3. The entry point to the algorithm is the procedure enforce-gac. This procedure assumes the existence of a pruning queue PQ which contains all the prunings made since it was last called, and maintains a constraint queue CQ which contains all constraints on which GAC has to be established. It uses PQ to insert into CQ all constraints that may no
longer be GAC, by calling **update-CQ** in line 1. It then iterates in lines 2–6 over **CQ** and calls the procedure **enforce-gac-constraints** to ensure that each of the constraints in **CQ** is GAC. The fact that making a constraint GAC may make other constraints no longer be GAC is reflected in the fact that **CQ** is updated after each call to **enforce-gac-constraint** in line 6.

The outline for **enforce-gac-constraint** is intentionally left vague, because an extensive literature has been developed for enforcing GAC on a general constraint without any assumptions about its structure (for binary constraints AC-3 [Mackworth, 1977a], AC-4 [Mohr and Henderson, 1986], AC-5 [Van Hentenryck et al., 1992], AC-6 [Bessière, 1994], AC-7 [Bessière et al., 1995], AC-2000 [Bessière and Régis, 2001], AC-2001 [Zhang and Yap, 2001], AC-* [Régis, 2005a] and for non-binary constraints **GAC-Schema** [Bessière and Régis, 1997]). Moreover, as we mentioned in chapter 1, for many constraints such as the alldifferent constraint, there exist specialized propagation algorithms that can enforce GAC in polynomial time, whereas any generic procedure for enforcing GAC (including **enforce-gac-constraint**) would take time exponential in the arity of the constraint. These alternate propagation algorithms can be invoked in place of **enforce-gac-constraint** when they exist for a constraint **C**.

Many specialized propagation algorithms can be made **incremental**. That is, they maintain internal state which can be updated as values are pruned from the domains of the variables of the constraint. This allows them to maintain GAC faster than if the state had to be created from scratch. Thus, the total work that the propagation algorithm has to perform over an entire branch is amortized over the length of the branch. This is in contrast to examining the constraint from scratch at each invocation of the algorithm. Clearly, for some constraints the latter approach is the only one that is possible. Consider however the example of the constraint **BOOLEANCARDINALITY**(x, d) over a set **x** of **n** binary variables, which states that at least **d** of them get the value 1. We can design an incremental algorithm and a non-incremental propagation algorithm to maintain GAC for this constraint. While both algorithms have a time complexity of **O(n)** when considered for a single node, the cost of the incremental algorithm can be spread over an entire branch, whereas the cost of the non-incremental algorithm is applied at every node, yielding a worst case complexity of **O(n^2)** over an entire branch. We will study this constraint in more detail in chapter 4, where we will present the incremental algorithm.
Bounds consistency. Another type of consistency often encountered in practice is bounds consistency (BC) [Van Hentenryck et al., 1998, Dechter, 2003]. Unlike other forms of consistency mentioned here, the definition of bounds consistency assumes a total ordering of the values of each variable of the constraint.

Definition 2.1.6 (Bounds support). Let $C$ be a constraint such that $\text{scope}(C) = \{V_1, \ldots, V_n\}$ and there exists a total ordering of the values in the domains of $V_1, \ldots, V_n$. Let $x_k \in \{\min(\text{CurDom}(V_k)), \max(\text{CurDom}(V_k))\}$ for some $k \in \{1, \ldots, n\}$. A bounds support for $V_k \leftarrow x_k$ in $C$ is a set of assignments $A = \{V_1 \leftarrow x_1, \ldots, V_k \leftarrow x_k, \ldots, V_n \leftarrow x_n\}$ that satisfies $C$ and $\min(\text{CurDom}(V_i)) \leq x_i \leq \max(\text{CurDom}(V_i))$ for all $i \in \{1, \ldots, n\} - \{k\}$.

Bounds consistency is defined in terms of bounds supports, in the same way that generalized arc consistency is defined in terms of supports, but only the bounds of each variable have to be supported.

Definition 2.1.7 (Bounds consistency). A constraint $C$ is bounds consistent if for every $V \in \text{scope}(C)$, $x \in \{\min(\text{CurDom}(V)), \max(\text{CurDom}(V))\}$, there exists a bounds support for $V \leftarrow x$ in $C$.

Bounds consistency is weaker than generalized arc consistency in two respects: first, it only requires that the minimum and maximum of the current domain of a variable are supported; second, the values in a bounds support need not be in the current domains of the variables, they only need lie in the range $[\min(\text{CurDom}(V)), \max(\text{CurDom}(V))]$.

Bounds consistency is useful because it is often much cheaper to enforce it rather than GAC. For many constraints it is NP-hard to enforce GAC. For example, the sum constraint over a vector of variables $x$ asserts that the values of its variables add up to a specific number. We can see that determining whether each value is GAC is an instance of the subset sum problem, therefore maintaining GAC is NP-hard. On the other hand, there exist polynomial time algorithms than can maintain bounds consistency. Maintaining bounds consistency may also be significantly cheaper than maintaining GAC even when GAC can be achieved in polynomial time. For example, the best known algorithm for maintaining GAC on the alldifferent constraint is $O(n^{2.5})$ [Régis, 1994], while $O(n \log n)$ algorithms are known that maintain bounds consistency [Puget, 1998, Lopez-Ortiz et al., 2003]. In some cases, the search space reduction achieved by using the more expensive algorithm may not translate in an overall runtime reduction. There
are even cases where the more expensive algorithm will not result in any further search space reduction [Schulte and Stuckey, 2005].

Note that simply plugging in a bounds consistency algorithm for a constraint in place of \texttt{enforce-gac-constraint} would be wasteful, because the bounds consistency algorithm would be called every time there was any change in the domain of the constrained variables. Instead, we would need to modify \texttt{update-CQ} to know for which constraints only bounds consistency is maintained. Such constraints are not inserted into the propagation queue unless the bounds on the domains of the constrained variables have changed.

### 2.1.2 Intelligent backtracking

Intelligent backtracking is a term that collectively refers to a variety of techniques which allow a solver to detect that a specific dead-end is not related to some assignments. Specifically, if a dead-end is discovered at level \( l \) and the algorithm can detect that the assignments made at levels \( j \in \{k+1, \ldots, l\} \) are not related to the dead-end, it can \textit{jump back} to level \( k \), because trying different assignments to those variables would still lead to a dead-end. That is, instead of chronologically backtracking to level \( l - 1 \), it avoids the extra work of trying different values for the variables at these levels, instead backtracking directly to try another value for the variable at level \( k \).

We examine here the algorithm FCCBJ [Prosser, 1993]\(^3\), shown in figure 2.4. The structure of FCCBJ is similar to that of FC. Whenever FC prunes the value \( V \leftarrow x \) of a future variable, we can identify a nogood responsible for that pruning. That nogood contains the assignments already made to the variables of the constraint and the future value that has just been pruned. In FCCBJ, we use this nogood to label the pruning referring to its nogood label with the notation \textit{Reason}(\( V \leftarrow x \)). If, during forward checking, we wipe out the domain of a variable \( V \), we can deduce a provably valid new nogood \( N_g = \bigcup_{i \in \{1, \ldots, d\}} (\text{Reason}(V \leftarrow i) - \{V \leftarrow i\}) \), which is done in line 10.

**Proposition 2.1.8.** Given a set of nogoods \( \text{Reason}(V \leftarrow i) \) \( \forall \; i \in \text{Dom}(V) \), the set \( N = \bigcup_{i \in \text{Dom}(V)} (\text{Reason}(V \leftarrow i) - \{V \leftarrow i\}) \) is also a nogood.

**Proof.** We cannot extend \( N \) by any assignment to \( V \) without obtaining a nogood. Indeed, \( N \cup \{V \leftarrow i\} \) is a nogood for any \( i \), as it is a superset of \( \text{Reason}(V \leftarrow i) \). But any solution

\(^3\)We actually examine FCCBJ with some improvements suggested in [Bacchus, 2000] which make it easier to discuss in the current setting.
Figure 2.4 Forward checking with conflict directed backjumping.

```plaintext
int FCCBJ(l)
1. If all variables assigned return (success, 0)
2. choose next variable V. Always choose a variable that has had all of its values pruned if one exists.
3. foreach d∈CurDom(V)
4. V ← d
5. forwardCheck()
6. (status, k) = FCCBJ(l+1)
7. undo(V ← d)
8. if (k < l)
9. return (status, k)
10. NewNG = \bigcup_{d∈Dom(V)}\{Reason(V ← d) − \{V ← d\}\}
11. X ← a = deepest assignment in NewNG
12. Reason(X ← a) = NewNG % Prune the assignment and label it using NewNG
13. k = lvl(X ← a) % level X ← a was made.
14. return(fail, k)
```

forwardCheck()
```plaintext
1. set A to the current assignment
2. foreach constraint C s.t. \exists exactly one \ V_f ∈ scope(C) with |CurDom(V_f)| > 1
3. foreach d ∈ CurDom(V_f)
4. if A ∪ \{V_f ← d\} violates C
5. Reason(V_f ← d) = \{V ← x | V ∈ scope(C) ∧ V ← x ∈ A \} ∪ \{V_f ← d\}
6. if \exists V s.t. CurDom(V) = \{d\}
7. V ← d
8. forwardcheck()
```

must contain an assignment to V, therefore N cannot be extended to a solution, so it is also a nogood.

Now suppose that the deepest assignment in Ng is made at level k, while the domain wipeout occurred at level l > k. We know that there is no reason to try any alternatives at levels k + 1 . . . l − 1, so we can jump back to level k directly, where the assignment X ← a was made. The value of k is determined in lines 11–13, while the backjumping part is handled in lines 8–9. If k = 0 then the problem is unsatisfiable (line 14). In line 12, we prune the assignment X ← a to level j < k, where j is the level of the second deepest assignment in Ng and use Ng to label that pruning. This label has the same syntactic form as the labels produced by forward checking. If all alternatives to X fail, we can again deduce a new nogood and potentially jump further back instead of chronologically backtracking. We handle both cases uniformly by ignoring a domain wipeout when it occurs, instead we always choose a variable with an empty domain if one
exists. This way, a variable whose domain has been wiped out by constraint propagation is treated as if the subtree under each of its assignments was found to have no solution.

Throughout FCCBJ, three invariants are maintained with respect to the labels $Reason(V \leftarrow a)$.

1. The nogood $Reason(V \leftarrow a)$ always contains $V \leftarrow a$ itself.

2. Every assignment in $Reason(V \leftarrow a) - \{V \leftarrow a\}$ is true and made earlier than $V \leftarrow a$. This stems from the use of $Reason(V \leftarrow a)$ as a justification for the pruning. If an assignment in it is made later than $V \leftarrow a$, then it cannot possibly have had anything to do with this pruning, so it cannot appear in $Reason(V \leftarrow a)$.

3. Third, if $V \leftarrow a$ is pruned at level $l$, then the deepest assignment in $Reason(V \leftarrow a) - \{V \leftarrow a\}$ is made true at level $l$. Let $W \leftarrow b$ be the deepest assignment in $Reason(V \leftarrow a) - \{V \leftarrow a\}$, and say it was made at level $k$ while $V \leftarrow a$ was made at level $l$. It would be a violation of the second invariant if $k > l$. It would not be incorrect if $k < l$, but it would be suboptimal, since $V \leftarrow a$ would still lead to failure if made at levels $k \ldots l - 1$, so it should be pruned at $k$.

These invariants are enforced both by $forwardCheck$ when it labels values that it prunes and also in lines 11–12 of FCCBJ. Invariants 2 and 3 are in fact maintained for the nogood labels by all algorithms that we study in this thesis.

One complication not shown in figure 2.4 is that forward checking may force an assignment to be made by reducing a variable’s domain size to 1. Because such an assignment is forced and not a decision, we cannot allow it to appear in nogoods that we discover. The reason is that FCCBJ is designed so that when a new nogood is discovered, it jumps back to the level where it made the deepest decision assignment that appears in that nogood. Assume we discover a nogood of the form $Ng = A \cup \{X \leftarrow 1, Y \leftarrow 1\}$, so that $A$ is a set of decision assignments made at levels earlier than $l$, $X \leftarrow 1$ is the decision assignment made at $l$ and $Y \leftarrow 1$ is an assignment that was forced at level $l$. FCCBJ can determine that it needs to jump back to level $l$. However, the new nogood cannot be used to prune and label the decision assignment, because it contains $Y \leftarrow 1$ which is in fact a consequence of $X \leftarrow 1$ and the resulting nogood would violate the second invariant mentioned above. The assignment $Y \leftarrow 1$ can be pruned and labeled using $Ng$, but when the rest of the assignments to $X$ have been explored no new nogood will be discovered because $X \leftarrow 1$ does not have a label. Therefore, non-decision assignments
cannot appear in the nogoods that FCCBJ discovers. This is accomplished by modifying the forward checking labeling procedure, so that when a non-decision $Y \leftarrow k$ assignment appears in a label, it is replaced by the set $\bigcup_{i \in \text{Dom}(Y) - \{k\}} (\text{Reason}(Y \leftarrow i) - \{Y \leftarrow i\})$.

Reasoning as we did in the proof of proposition 2.1.8, we see that this set of assignments implies that $Y \leftarrow k$ must be forced, so the nogood labels remain valid.

Other forms of intelligent backtracking can be explained using variations of this scheme. An interesting variation is dynamic backtracking (DBT, [Ginsberg, 1993]). In DBT, after a new nogood is discovered and used to jump back to level $k$, the variable ordering heuristic is modified so that after a new value is assigned at level $k$, the assignments made at levels $k + 1, \ldots, l$ are the same ones as those that were just retracted.

We have not discussed here how intelligent backtracking techniques can be combined with other forms of constraint propagation than forward checking. This has been addressed before [Prosser, 1995, Jussien et al., 2000], but we will delay this discussion to chapter 4.

2.1.3 Variable Ordering

In a backtracking algorithm, the choice of which variable to assign next is called the variable ordering heuristic. In the description of the algorithm that we gave in section 2.1, an arbitrary choice can be made. Variable ordering is very important in the performance of any backtracking-based solver and merits further examination. However, we will not present any results in this thesis on variable ordering heuristics, therefore the presentation in this section is not as detailed as for other topics. A more detailed discussion can be found in [Dechter, 2003].

There are two categories of variable ordering heuristics: static and dynamic. A static heuristic is computed before the search is run. Given any instantiation, it will return the next variable in the static ordering that needs to be assigned, except if a variable with an empty domain exists. A dynamic variable ordering will examine various properties of the state of the solver to determine which variable to choose. Two properties often examined when computing a dynamic ordering include the size of the current domain of the variable and the number of future variables it is constrained with (degree). From these metrics, we can construct DVO (dynamic variable ordering) heuristics that choose the variable with minimum sized current domain (dom); choose the one with maximum current degree (deg); choose the one with minimum domain and use maximum degree as a
tie breaker \( (\text{dom+deg}) \) [Brélaz, 1979]; or choose the one with the lowest ratio of domain size to degree \( (\text{dom/deg}) \) [Bessière and Régin, 1996]. More recently, the DVO \( \text{dom/udeg} \) has been proposed [Lecoutre et al., 2004], which is a weighted version of \( \text{dom/deg} \). Initially all constraints have weight 1, but every time a constraint \( C \) is violated (this can mean that propagation of \( C \) causes a domain wipeout), its weight is increased by 1. \( \text{dom/udeg} \) for variable \( V \) is then the size of \( V \)'s current domain of values divided by the sum of the weights of the constraints it participates in. Clearly, other possibilities exist, for example by adding randomness to any of these DVOs or by examining other aspects of the state of the solver.

**Figure 2.5** An example showing the width, induced width and tree decomposition of a constraint graph. (a) The constraint graph (b) The width of the graph is 3 given the ordering \( V_1, V_2, V_3, V_4, V_5, V_6 \) (c) The width is 2 given the ordering \( V_5, V_3, V_6, V_1, V_2, V_4 \). (d) The induced widths for the orderings in (c) and (d), are 3 and 2, respectively. The additional edges are dashed. (f) The tree decomposition of the graph in (e).
An effect of a static variable ordering heuristic is that it defines the width of the constraint graph. Consider a CSP $\mathcal{P}$ and its constraint graph $G$. For any variable $V$, the corresponding vertex in $G$ is $v_V$. Given an ordering and a variable $V$, if $V'$ precedes $V$ in the ordering and there exists an edge between $v_{V'}$ and $v_V$, then $V'$ is an ancestor of $V$. The maximum number of ancestors over all variables is the width $w$ of the given ordering. We can also construct the induced graph $G^*$ given an ordering. We traverse the graph in reverse order of the ordering. For each variable, we join its ancestors with an edge. The induced width $w^*$ of a graph is the minimum width of its induced graph for all orderings [Dechter, 1992]. Figure 2.5(a) shows a constraint graph; 2.5(b) and (c) shows this graph rearranged according to an ordering to demonstrate how the width of the ordering is computed; and 2.5(d) and (e) shows the induced graph for the same ordering.

Among other features related to the induced width, solving the problem with nogood learning using this ordering requires only $O(ndw^*+1)$ time [Bayardo Jr and Miranker, 1996]. This is an improvement over the $2^{O(n)}$ bound that would normally apply. However, finding an ordering of minimum induced width is an NP-hard problem.

We can also see that the induced width of the constraint graph is equal to its treewidth [Bodlaender, 1993]. Given an ordering $d$ that produces the minimum induced width, we construct the corresponding tree decomposition $(X, T = (I, F))$ as follows. There exists a node $i_V$ in $T$ for each vertex $v_V$ in $G$. We traverse the induced graph $G^*$ in the order $d$. The parent of $i_V$ is $i_{V'}$ so that $v_{V'}$ is the most recent ancestor of $v_V$ according to $d$. The set $X_V$ associated with $i_V$ contains all the ancestors of $v_V$ according to $d$. For example, in 2.5(f) we show the tree decomposition corresponding to the graph and ordering in 2.5(f).

## 2.2 SAT and Resolution

Here, we introduce the formalism of proof complexity, which provides a framework to compare the relative power of different algorithms for unsatisfiable problems. Proof complexity compares proof systems based on the size of the shortest proofs possible within those systems. The restriction to unsatisfiable problems does not make the comparisons less meaningful. Informally, the reason for this is that when solving a satisfiable instance, a solver is likely to make incorrect guesses about which path to follow (which assignments to make). Thus, before finding a solution, it will first have to construct refutations of
unsatisfiable subproblems, so that the efficiency of the solver on satisfiable instances
depends on its efficiency in unsatisfiable instances.

In the next few sections, we examine CSP solving techniques as proof systems and
their relationship to resolution, which is in practice the most widely used proof system
for satisfiability. Constraint satisfaction is a generalization of satisfiability. As such there
exist close connections between the two.

The SAT problem is that of determining whether or not a propositional formula
expressed in conjunctive normal form (CNF) is satisfiable. A CNF formula \( \phi \) consists
of a conjunction of disjunctions of propositional literals. Each disjunction of literals is
called a clause. For convenience, we write \((x, \overline{y}, z)\) instead of \((x \lor \overline{y} \lor z)\) for a clause and
\(\{(x, \overline{y}, z), (x, w, \overline{z})\}\) instead of \((x \lor \overline{y} \lor z) \land (x \lor w \lor \overline{z})\) for a formula.

Resolution [Robinson, 1965] is a proof system that is complete for unsatisfiable CNF
formulas. A resolution step is an application of the rule that given two clauses \((A, y)\)
and \((B, \overline{y})\) we can deduce a new clause \((A, B)\). We write \((A, B) = R((A, y), (B, \overline{y}))\) for
the resolvent and \(y = r((A, y), (B, \overline{y}))\) for the resolved variable. If the empty clause can
be derived starting with the original clauses of a formula \(\phi\) using resolution steps, then
\(\phi\) is unsatisfiable. More formally an unsatisfiability proof, or resolution refutation, of
the original formula is a sequence of clauses \(C_1 \ldots C_m\) such that each \(C_i\) is either (a) an
original clause or (b) derived from resolving two clauses \(C_a C_b\) such that \(a, b < i\). The size
of the proof is the number of resolution steps needed to arrive at the empty clause. We
can construct a DAG from a resolution refutation as follows. There exists one vertex for
each clause used in the proof and an edge from the vertex \(C_j\) to \(C_i\) if and only if \(C_j\) is
one of the two clauses that were resolved together to yield \(C_i\). Each edge is labeled with
the name of the variable that was resolved away to produce the new clause. This graph
is a directed acyclic graph (DAG). An example of a resolution refutation of the formula
\[\{(\overline{y}), (w, \overline{v}, \overline{z}), (w, y, \overline{z}), (v, z), (\overline{v}, z), (\overline{z}, x, q), (\overline{q}, x)\}\] is shown in figure 2.6.

Of the systems that we examine here, unrestricted, or general, resolution is the most
powerful. However, the space of possible resolutions available at each step is too large
for a heuristic to handle, so in practice general resolution is not used that much. Various
refinements of resolution have been proposed to overcome this problem. Tree-like reso-
lution [Davis et al., 1962, Tseitin, 1983] uses each clause exactly once on each path from
the clauses of \(\phi\) to the empty clause. If a clause needs to be used more than once, it has to
be rederived. We see that tree-like resolution produces a DAG that is a tree\(^4\), hence the name. Note that the refutation in figure 2.6 is not a tree-like resolution since the derived clause \((z)\) is used twice. Negative resolution [Goerdt, 1992] has the restriction that one of the input clauses in each resolution step must be a negative clause, i.e., a clause that contains only negative literals. Ordered resolution [Davis and Putnam, 1960, Tseitin, 1983]\(^5\) requires that the variables resolved away along every path of the DAG respect a fixed ordering of the variables. Regular resolution [Tseitin, 1983] requires that on each path from the empty clause to a clause in the original formula, a variable appears at most once as an edge label. Linear resolution [Beame et al., 2004] requires that the resolvent of one step is one of the clauses being resolved on in the next step. Linear resolution is complete for Horn clauses. Trivial resolution [Beame et al., 2004] requires that each clause \(C_i\) in a sequence \((C_1, \ldots, C_n)\) is either an initial clause or the resolvent of \(C_{i-1}\) and an initial clause. Trivial resolution is ordered, regular, and linear (and therefore tree-like) and we will see later that it is the form of resolution used to derive conflict clauses in clause-learning SAT solvers.

**Figure 2.6** A resolution refutation as a DAG. Each vertex is labeled by the clause it represents. An edge to \(C_i\) is labeled with the variable that was resolved away to produce \(C_i\).

\[
\begin{array}{c}
(x, z) \\
|\hspace{1cm}|
\end{array}
\]

In order to compare the relative power of different proof systems, we first need a few definitions. A system \(A\) can \(p\)-simulate a system \(B\) if any refutation of a formula

\(^4\)The input clauses may be used more than once, so the DAG is only a tree if we remove the vertices that correspond to the input clauses.

\(^5\)The DP and DLL algorithms were proposed before resolution was introduced by Robinson in [Robinson, 1965], but actually implement ordered and tree-like resolution, respectively.
Chapter 2. Formal Background

\( \phi \) using \( B \) can be transformed into a refutation using \( A \) that is at most larger by a polynomial\(^6\) factor. \( A \) and \( B \) are polynomially equivalent if they can \( p \)-simulate each other. If a system \( A \) can \( p \)-simulate a system \( B \), but the reverse does not hold, there exists a separation between the systems. If there exists an infinite family of formulas \( \phi_n \) such that the shortest \( B \)-refutation of \( \phi_n \) is superpolynomially larger than the shortest \( a \)-refutation of \( \phi_n \), then we say that \( A \) is superpolynomially separated from \( B \). If the shortest \( B \)-refutation of \( \phi_n \) is exponentially larger in size, we say that \( A \) is exponentially separated from \( B \). Two systems are incomparable if neither can \( p \)-simulate the other.

The relationship of the various resolution refinements has been studied in [Buresh-Oppenheim and Pitassi, 2003] among others. Clearly, unrestricted resolution can \( p \)-simulate all the refinements mentioned, since proofs in these systems are already proofs in the system of unrestricted resolution. In addition, it is exponentially separated from tree resolution, negative resolution and ordered resolution. Negative resolution is exponentially separated from tree-like resolution and ordered resolution is incomparable to both tree-like resolution and negative resolution.

2.3 DPLL

The DPLL algorithm [Davis et al., 1962] is a backtracking search algorithm that solves SAT problems. An outline of the algorithm is presented in figure 2.7. Its operation depends on unit propagation, outlined in figure 2.8. When a literal \( l \) is set to true (say, by choosing to instantiate its variable at a decision point), the formula is simplified by removing all clauses that contain that literal, since they are made true and no longer affect the problem, and by removing the literal \( \overline{l} \) from all clauses that contain it. As a result of this simplification some clauses may become unit; that is, contain a single literal. In that case, that literal is forced to true. This may cause a further simplification of the formula, possibly causing more clauses to become unit. Eventually, either there exists an empty clause and DPLL backtracks, or no more unit clauses exist and the search continues by choosing another variable to branch on.

We do not address in detail the heuristic for choosing a variable to branch on in line 5 of figure 2.7. This is in fact one of the most important elements in determining the performance of a modern DPLL-based solver. In [Davis et al., 1962] the heuristic

---

\(^6\)This means polynomial in the size of the proof.
Figure 2.7 The DPLL algorithm

\textbf{DPLL(} \phi \textbf{)}
1. \textbf{if } \phi \equiv \emptyset \textbf{ then}
2. \textbf{return TRUE}
3. \textbf{if } \emptyset \in \phi \textbf{ then}
4. \textbf{return FALSE}
5. Choose a variable \textbf{l}
6. \textbf{if } \textbf{DPLL(simplify(} \phi \cup \{\{l\}\}) \textbf{) then}
7. \textbf{return TRUE}
8. \textbf{return DPLL(simplify(} \phi \cup \{\{l\}\}))

Figure 2.8 Simplifying a formula by unit propagation

\textbf{simplify(} \phi \textbf{)}
1. \textbf{if } \exists c \in \phi \textbf{ s.t. } c \equiv \{l\} \textbf{ then}
2. \textbf{foreach } c' \in \phi , \textbf{s.t. } l \in c
3. \phi = \phi - \{c'\}
4. \textbf{foreach } c' \in \phi , \textbf{s.t. } \bar{l} \in c
5. \phi = \phi - \{c'\} \cup \{c' - \{l\}\}
6. \textbf{return simplify(} \phi \textbf{)}
7. \textbf{else}
8. \textbf{return } \phi

proposed was to choose a variable from a minimum-length clause. This will tend to simplify the formula quickly, as smaller clauses can become unit clauses more easily.

For unsatisfiable formulas, we can construct a resolution refutation of \phi from the search tree of DPLL. First, we need to observe that at every leaf node, where the empty clause is part of the formula, we can find the clause in the original formula that has been reduced to the empty clause via unit propagation. Moreover, the series of unit propagation that lead to the falsification of the clause can be retraced as resolution steps. So, if the clause that was made false was \((A \lor l)\) and \(l\) was made false because of unit propagation of the clause \((B \lor \bar{l})\), we can resolve these two clauses to get \((A \lor B)\), which is also false. We can follow this chain of unit propagations back to the decision literal of the node to get a clause that is false and contains the decision literal at that node. In that way, we can associate each leaf node with a clause. Consider now the parent of two nodes. Assume the parent node branched on a variable \textbf{x}. Each of the two child nodes will have falsified clauses \textbf{A} and \textbf{B} respectively. By following the chain of unit propagations from \textbf{A} to \textbf{x} and from \textbf{B} to \textbf{\bar{x}}, we get two clauses \(C \lor x\) and \(D \lor \bar{x}\),
which we can resolve to get a new clause $C \lor D$, which we associate with the parent node. This new clause will also be falsified, since all literals in the clauses that were used to derive it were false to begin with. By applying this procedure to every node in the search tree, we derive a series of resolution steps. The clause associated with the root node of the search tree will be the empty clause, since this is the only clause falsified by the empty assignment. Therefore, the series of resolution steps derived constitute a resolution refutation of $\phi$.

To clarify this construction, consider the following example. Let $\phi = \{(\neg y), (w, \neg z), (\neg w, y, \neg z), (v, z), (\neg v, z, x), (\neg q, x, q), (\neg q, x)\}$, which we also used for the example in figure 2.6. A DPLL refutation of the formula is shown in figure 2.9. For simplicity and to keep the correspondence with figure 2.6 clear, unit propagation is not used in this refutation. In it, every node is marked by the corresponding clause that is discovered in the resolution refutation. An internal node, marked by $\cdot$ indicates that the associated clause is the result of resolution, while a leaf node, marked by $\times$ indicates that the associated clause is one that exists in the original formula. At each node, the search branches on a variable being true or false. Each node is connected to its children by edges, which are marked by the literal that is being branched on. We can refer to each node by the unique sequence of decisions made on the path from the root to it. Therefore, the leftmost leaf node can be identified by $xzy$, while the rightmost is $\neg xzv$. Note that not all literals on the path from the root are involved in the clause associated with a node. This DPLL refutation is not minimal, as branching on $w$ in node $\neg w z$ leads to searching the same subtree twice in the next level. As a result, the construction of the resolution proof described above does not apply to this DPLL tree. The clauses associated with the nodes $\neg z w$ and $\neg z \neg w$ are not resolvable. We can work around this problem by simply modifying the construction so that if the clauses associated with the child nodes are not resolvable, we choose one of them to label the parent node as well, which is what we did in figure 2.9.

### 2.3.1 Proof complexity of DPLL

By the construction of the resolution refutation from the search tree, we see that every proof of unsatisfiability by DPLL can be transformed to a tree-like resolution refutation. Moreover, the size of the tree-resolution refutation is linear in the size of the DPLL refutation (there are as many resolvants as nodes explored), so DPLL is $p$-simulated by tree resolution.
Even compared with tree resolution, we can see that DPLL may perform unneeded work. Going back to the example of section 2.3, we see that the implied resolution steps in the subtree rooted at the node $xz\overline{w}$ are unnecessary, because the clause $(x, \overline{z})$ associated with the node $xzw$ does not refer to the variable $w$. Still, DPLL as presented cannot detect and exploit this fact, because it does not actually keep track of the clauses that are discovered. We can work around this limitation of DPLL by computing at each node the clause that was learned and then jumping back over the assignments that do not appear in the clause to the deepest one that does [Bayardo Jr. and Schrag, 1996]. In this example, the algorithm would be able to deduce that visiting the node $xz\overline{w}$ could not possibly lead to a solution, thus would jump back to node $xz$ and try the alternate instantiation of $z$.

Even so, it should be noted that it is not easy to find a resolution proof of minimum length [Alekhnovich et al., 2001], so for any given instance, even if a short tree resolution proof exists, we may not be able to find a short proof with DPLL.

### 2.3.2 Clause learning

The limitation of DPLL to tree resolution can be lifted by learning clauses during search. For example, at each node visited, the clause of the corresponding resolution proof discussed above can be added to the theory. This is the strategy used in [Bayardo Jr. and Schrag, 1996]. But this is not the only choice. To demonstrate, we first introduce the implication graph [Marques-Silva and Sakallah, 1999]. The impli-
cation graph is a DAG, in which we have a vertex $v$ for each literal $l$ that is true, either because the algorithm branched on its variable or because it was made true via unit propagation. In the latter case, it means there exists either a clause in the original theory or a learned clause that was made unit forcing this literal. We call this clause $c$. Each literal $l' \in c, l' \neq l$, is false, therefore there exists a vertex $v_{\neg \sigma}$ for $\neg l$. We add an edge from $v_{\neg \sigma}$ to $v$ for each $l' \in c, l' \neq l$. For example, if the literal $l$ was made true by unit propagation of the clause $(x, y, z, l)$, we add edges to the vertex for $l$ from each of the vertices for $\bar{x}$, $y$ and $z$. Vertices corresponding to decision literals have no incoming edges. For brevity, when talking about the implication graph, we will use the term literal to refer to a vertex associated with that literal.

Let us now consider the case of a conflict. A conflict occurs when there exists a variable $x$ such that both $x$ and $\bar{x}$ have been asserted. In [Marques-Silva and Sakallah, 1999] it was suggested that we can partition the graph in two, calling one side the conflict side and one the reason side. The conflict side must always contain at least the vertices that correspond to the conflicting literals. We can construct a new clause as follows: we start with the clause $(x, \bar{x})$. If any literal in it is in the conflict side, we resolve with the clause that forced it. We repeat this until there are no literals in the conflict side. The clause derived at the end of this procedure, as well as clause derived at each resolution step can be added to the theory.

The authors of [Marques-Silva and Sakallah, 1999] further identified the usefulness of so-called unique implication point (UIP) clauses. In the implication graph, we can identify articulation vertices; i.e., vertices such that all paths from the most recent decision to any literal in the conflict have to pass through that vertex. We then construct a partition such that every vertex on a path between the articulation vertex and the conflict, including the conflict, is in the conflict side, while everything else is in the reason side. The clause constructed from this kind of partition is called a UIP clause. There may potentially be many UIP vertices, including the decision literal vertex itself. UIP clauses have the interesting property that they contain exactly one literal from the current decision level. Thus, when we backtrack we can immediately flip the literal. Some experimental results [Moskewicz et al., 2001] suggest that the best choice for SAT solvers is the 1-UIP clause, which corresponds to the articulation vertex nearest to the conflict. Most modern SAT solvers use 1-UIP clause learning.

Notice also that the resolution derivation of clauses based on the implication graph is a trivial resolution [Beame et al., 2004]. Each variable is only resolved away once, since
a variable may appear in the implication graph once, either as $l$ or $\overline{l}$. At any point during the iteration, we maintain a tentative clause $c$. If $c$ contains a literal $x$ from the conflict side, we resolve $c$ with an existing clause $c_x$ which contains $\overline{x}$ and the resolvent becomes the new tentative clause. Therefore, the resolution steps constitute a linear resolution derivation. Since variables are only resolved away once, this means that the derivation is a trivial resolution derivation.

The family of algorithms that learn clauses during their operation is collectively called CDCL (conflict-directed clause learning). It was shown [Beame et al., 2004] that in terms of proof complexity, CDCL is stronger than tree-like resolution and that it cannot be $p$-simulated by regular resolution.

### 2.3.3 State of the art in SAT solving

Modern systematic SAT solvers are typically based on CDCL. Here we describe other parts that have proved crucial in implementing an efficient SAT solver.

**Unit propagation.** A SAT solver spends a significant part of its time performing unit propagation [Moskewicz et al., 2001]. The obvious method of performing unit propagation is to maintain a flag and a counter for each clause. The flag indicates whether or not the clause is satisfied. If the clause is not satisfied, the counter indicates how many of its literals are non-false. Both of these variables are updated whenever a literal appearing in the clause is set. When a literal appearing in the clause is made true the flag is set to true, and when a literal in the clause is made false the counter is decreased. When the counter reaches one, the clause is unit, so the remaining non-false literal is forced. On backtracking, these variables need to be restored to their previous values.

When performing clause learning, the clause database may contain a large number of large clauses. The work that needs to be done by the obvious method for each literal that is set rises dramatically. In [Moskewicz et al., 2001], an alternative method of performing unit propagation is proposed. For each clause, we maintain two *watch literals*. The watch literals need to be non-false. If one of the watched literals becomes false, we try to find a non-false replacement for it in the clause. If none is found, the clause has become unit and the other watch literal is forced. If both watch literals are already false and no replacement can be found for either, the clause has become empty and conflict resolution is performed. Nothing needs to be done on backtracking, because two watch literals that
are valid at a deeper level are also valid at shallower levels. In addition, nothing needs to be done when any of the non-watch literals in the clause are set.

The watch literal technique is not strictly better than the counting technique. Using watch literals, we have to examine each clause much less often, especially as the size of the clause increases, but the operation performed when a clause is examined is linear in the size of the clause as opposed to constant in the case of the counting technique. On the other hand, the fact that nothing needs to be done on backtracking is a clear advantage over the counting technique, both in time and space complexity. Although no detailed studies have been published comparing the two methods, practically all SAT solvers that need to perform unit propagation use the watch literal technique for large clauses. For binary and ternary clauses, other data structures have been proposed that are closer to the counting technique, but with better constants [Ryan, 2004].

**Deletion of learned clauses.** Recording new clauses indefinitely would eventually cause unit propagation to consume such a large proportion of the solver’s time that search progress would practically stop. On the other hand, techniques typically used to limit the size of the clause database, such as length-bounded or relevance-bounded learning often remove too many clauses for them to be useful. Therefore, modern SAT solvers record all clauses and periodically scan the database to remove some of them until the database has been reduced to a certain size. The criteria used for removal typically include the number of literals that have been forced by each clause (i.e., the clause’s activity) as well as its size.

**Branching heuristic.** The branching heuristic, which decides which literal to make true next plays a great role in the performance of a SAT solver. Most SAT solvers use a variation of the VSIDS (Variable State Independent Decaying Sum) heuristic, described in [Moskewicz et al., 2001], which works as follows. A counter is associated with each literal. Whenever a new clause is added to the database the counter of each literal appearing in the clause is incremented. Periodically, all such counters are divided by some small constant, so that more recently learned clauses weigh more heavily in these counters. The branching heuristic chooses the literal with the highest counter value as the next decision.

This heuristic will follow paths that overlap with recently learned clauses. Then, when a new clause is learned, it will also overlap with other recently learned clauses.
Chapter 2. Formal Background

The result is that clusters of similar clauses will be learned, which can then be resolved together.

Based on this idea, the heuristics VMTF (Variable Move To Front, used in the solver Siege) [Ryan, 2004] and Berkmin (used in the solver Berkmin) [Goldberg and Novikov, 2002] try to make this purpose more explicit. VMTF maintains a list of variables. When a new clause is learned, a small number of randomly selected variables that appear in the clause are moved to the front of the list. The heuristic chooses the first unset variable in the list. Berkmin uses VSIDS, but only applies it to the literals that appear in the most recent unsatisfied clause. Both of these heuristics place more emphasis on choosing literals from recent clauses than choosing literals that appear on many clauses. Among other advantages, the authors of [Goldberg and Novikov, 2002] claim that this emphasis on recent clauses allows the heuristic to adapt much more quickly to changing conditions in the search space. For example, by flipping a variable, different clauses might become relevant, making different variables the best choice. VSIDS will be slow adapting to this, as recent clauses that were relevant before the variable was flipped will still weigh heavily on the decision. In VMTF and Berkmin, the variables that appear in the new clauses will immediately be given preference.

Another aspect of SAT branching heuristics that seems especially important is that they are very fast in making their decision. The VSIDS heuristic can be implemented by putting the variables in a heap, so making a decision is an $O(\log n)$ operation. Still, computing the heuristic takes 10% of the runtime of the solver Chaff. Computing either VMTF or the Berkmin heuristic is an amortized $O(1)$ operation.

Finally, most SAT solvers, including the ones we just discussed often employ some form of randomization in the branching heuristic. This is typically in the form of using randomness as a tie breaker, even though other techniques such as a random renaming of variables are sometimes used. The purpose of randomization is to increase the robustness of the heuristic. The intuition is that among the set of variables which are indistinguishable by a branching heuristic, some are actually good choices while other are not. In this context, a good choice is one that leads the solver to explore a subtree that contains a solution or for which a refutation can be discovered quickly. Any deterministic heuristic may consistently choose a bad candidate for some inputs. However, when randomness is
added, no input can lead the heuristic to always make a bad decision.\footnote{This argument is similar to the argument used to show that quicksort exhibits $O(n \log n)$ behaviour when randomness is used to choose the guard [Cormen \textit{et al.}, 2001].} This argument of course depends on the heuristic generating a set of candidates that contains good choices.

**Restarting.** Most modern SAT solvers use a restarting scheme. Restarting involves periodically undoing all current assignments and making a new choice at the root of the search tree. Each period between restarts is called a run. Restarting does not affect the completeness of the algorithm, because the clauses that have been learned are still in the clause database, so the same part of the search space will not be explored.\footnote{This is not true when the solver also performs clause deletion, which is most often the case. A scenario can be conceived where a solver periodically deletes some clauses, restarts, then discovers the same clauses and deletes them again. In such a scenario the solver is no longer complete. In practice, however, this has proved to be an unlikely scenario, so we ignore it in this section.} The intuition for using restarting is the same as that for using randomization in the branching heuristic. Any branching heuristic may make choices that make it hard to discover that a subtree contains no solution. Restarting can give the solver a chance to stop searching that subtree but use the knowledge captured in the learned clauses to quickly explore other subtrees, thus improving its robustness. In addition, it has been shown [Beame \textit{et al.}, 2004] that adding restarting to CDCL makes it as powerful as unrestricted resolution, in the sense that it can $p$-simulate it.

Even when not performing clause learning, it has been shown that restarting can provide a significant speedup for solving certain classes of problems. These are classes for which the distribution of the sizes of the search trees, even though they are finite, can be modeled by a heavy tailed distribution; i.e., a distribution which has infinite variance [Gomes \textit{et al.}, 1997]. For these problems, restarting can improve the speed of the solver. Completeness is not lost if more time is allowed for each run as the number of runs increases. Note that this does not mean that time allowed for each run increases monotonically. Indeed, if each run of the algorithm is independent of the previous runs, the optimal restart strategy is Luby’s [Luby \textit{et al.}, 1993], which is not monotonic.

There are three restarting strategies currently used [Huang, 2007]: fixed restarting, where each run contains exactly $n$ branches, where $n$ is determined experimentally for each solver; geometric restarting, with parameters $n$ and $p > 1$, where the $i^{th}$ run contains $n \ast p^{i-1}$ branches and Luby’s [Luby \textit{et al.}, 1993] strategy with parameter $p$, where the number of branches explored for each run follows the pattern
It was shown in [Huang, 2007] that the restart strategy can have a significant effect on the effectiveness of a solver, even though no strategy has proved clearly better than all others.

### 2.4 Mapping CSPs to SAT

A CSP can be translated to a SAT problem in several ways [Walsh, 2000]. The most useful one for our purposes is the direct mapping. Given a CSP \( P = \langle V, D, C \rangle \), we create a SAT instance as follows. We have \(|V| \times |D|\) variables \( x_{i,j} \). The variables \( x_{i,j} \) are intended to be true iff \( V_i \) has been assigned the value \( j \). For each variable, we add clauses \((x_{i,j} \lor x_{i,k})\) for all pairs of values of the same variable, to ensure that a variable cannot be assigned two different values. We add a clause \((x_{i,0} \lor x_{i,1} \lor \ldots \lor x_{i,d})\) to ensure that a variable is assigned at least one value. We view each constraint as representing a set of nogoods; i.e., assignments to the variables of the constraint that violate the constraint. A nogood asserts that not all of its assignments can be made simultaneously, thus it is the negation of a conjunction of assignments, which is a disjunction of the negation of these assignments. So, we can translate each nogood into a clause, which we add to the theory. For example, the nogood \( \{V_1 \leftarrow 1, V_2 \leftarrow 2\} \) becomes the clause \((\overline{x}_{1,1} \lor \overline{x}_{2,2})\). In a similar way, we can talk about a mapping between assignments. An assignment \( \{V_1 \leftarrow 1, V_2 \leftarrow 2, \ldots\} \) in the CSP maps to the assignment \( \{x_{1,1}, x_{2,2}, \ldots\} \) in the SAT problem. The pruning of a value \( V_i \leftarrow j \) in the CSP corresponds to forcing the literal \( \overline{x}_{i,j} \). Under this mapping we get a bijection between models of the SAT theory and solutions of the CSP. We also get a bijection between nogoods of the CSP instance and implied negative clauses, i.e., clauses that contain negative literals only, of the SAT theory.

The direct encoding handles binary CSP variables suboptimally. It is enough to use one Boolean proposition to represent a binary CSP variable, but the direct encoding will introduce two, which are mutually exclusive, along with clauses to make them mutually exclusive. If only one propositional variable is used for each binary CSP variable then the nogoods generated from the constraints will not always be negative clauses. However, typical CSP solvers treat binary variables in the same way as variables with \( k > 2 \) values. Thus the direct encoding is a more useful tool for understanding the behaviour of CSP.

---

9For simplicity, we assume that all variables of the CSP have the same domain size, but this restriction can be easily lifted.
algorithms in terms of a corresponding SAT theory.

### 2.4.1 Relation of FCCBJ to DPLL

It was also shown in [Walsh, 2000] that unit propagation on the direct mapping to SAT is equivalent to forward checking on the original CSP in terms of propagation power. Consider an assignment $\mathcal{A}$ in the CSP and the equivalent assignment $\mathcal{A}'$ in the SAT problem. Forward checking the assignment $\mathcal{A}$ in the CSP will produce a set of pruned values $S$. Applying unit propagation on the SAT problem after making the assignments $\mathcal{A}'$ will similarly force a set of literals $S'$ to true. The result of [Walsh, 2000] states that the value $V_i \leftarrow j$ is in $S$ iff the literal $x_{i,j}$ is in $S'$.

Although unit propagation and forward checking yield the same amount of constraint propagation, this does not mean that DPLL run on the direct SAT encoding will necessarily be identical to FC run on the CSP.

A critical difference is that DPLL employs a more general branching strategy than is typically used to describe FC. In particular, FC choses a variable then tries all of that variable’s values one after the other. DPLL on the other hand can branch on the values of a variable independently. For example it might pursue a path $\bar{x}_{i,j} \bar{x}_{k,m} \bar{x}_{h,a}$ which corresponds to pruning the values $V_i \leftarrow j$ and $V_k \leftarrow m$ then setting $V_h \leftarrow a$.

We can give FC this flexibility by employing a 2-way branching scheme (as opposed to the standard $d$-way branching). In 2-way branching we allow FC to assign a variable a value $V_i \leftarrow j$, and then if that assignment fails to yield a solution, prune $V_i \leftarrow j$ and then continue on to another branching decision.

Consider now adding intelligent backtracking to both algorithms. It was shown in [Chen and van Beek, 2001] that in the absence of learning, any proof of unsatisfiability in FCCBJ can be converted to one without any backjumps, by changing the variable and value ordering so as to remove assignments that would have to be backjumped over. The same argument holds for DPLL without learning as well, so FCCBJ without learning is not in principle any more powerful than FC, neither is DPLL with backjumping more powerful than DPLL. Still, in practice we often see that CBJ does help because the branching heuristic does not make optimal decisions. So, we compare the backjumping technique used in both algorithms.

We saw that DPLL in effect computes a clause at each node and can use that clause to jump back over decision levels that branch on literals that do not appear in the clause.
and whose unit propagants also do not appear in the clause. Since every literal that appears in that clause is false, we can associate a level with each literal \( x \) that is true in the current branch so that \( \text{level}(x) = k \) iff DPLL branched on \( \overline{x} \) at level \( k \) or \( \overline{x} \) is one of the unit propagants at level \( k \). We can produce a set of levels \( L \) from each clause \( c \) so that \( L(c) = \{ k \in \{1 \ldots l\} \mid \exists x \in c \text{ s.t. level}(x) = k \} \). Then, at a node with branches \( x \) and \( \overline{x} \), we can jump back to the deepest level in \( L(x) \cup L(\overline{x}) \). This is exactly the same technique that is employed in FCCBJ, thus DPLL with CBJ will visit the same nodes as FCCBJ with 2-way branching.

### 2.4.2 Adding learning to FCCBJ

Just as the limitation of DPLL to tree like resolution can be lifted by adding clause learning, so can the limitation of FCCBJ. It is straightforward to add nogood learning to FCCBJ, as seen by comparing FCCBJ in figure 2.4 to FCCBJ with learning in figure 2.10.

Recall that in FCCBJ, whenever a value is pruned by forward checking because of a constraint \( C \), it is labeled with a nogood. This nogood is the set of assignments that violate \( C \), one of which is the pruned value and the rest are all past assignments. This happens in the `forwardCheck()` call in line 5. Nogoods discovered by forward checking are called base nogoods as they are implicitly contained in the problem’s constraints. Whenever all the values of a variable \( V \) have been found impossible, a new nogood is deduced in line 10, that is the union of the nogoods for each of the values of \( V \) minus these values. This is used both to jump back as well as to prune and label the assignment made at the level we are jumping back to. This nogood has the same syntactic form as the nogoods learned from forward checking, that it is a set of assignments. This means that the first time that line 10 is executed, its input will be nogoods discovered by forward checking, but after that the input can be any mix of base nogoods and nogoods discovered in line 10.

In FCCBJ+S, we record the nogood discovered in line 10. The algorithm then has to make sure that no branch is ever visited that contains this nogood. As in DPLL, this is managed by performing unit propagation of the recorded nogoods, by calling `NoGoodProp` in line 5. However, nogoods are sets of assignments, all of which cannot be simultaneously true. Thus, unlike DPLL where unit propagation ensures that all literals in a clause do not become false at the same time, the purpose of unit propagation in FCCBJ+S is to ensure that not all assignments in the nogood become true at the same
time. When all but one assignments are true, the remaining one is made false. This means that the corresponding value is pruned. Like forwardCheck, NogoodProp labels the values that it prunes with a nogood. The label is the nogood that became unit.

**Figure 2.10** FCCBJ+S: FCCBJ with nogood learning

```c
int FCCBJ+S(l)
1. If all variables assigned return(success, 0)
2. choose next variable V. Always choose a variable that has had all of its values pruned if one exists.
3. foreach d∈CurDom(V)
4. V ← d
5. NoGoodProp() and forwardCheck() : forwardCheck() is not shown here.
6. (status, k) = FCCBJ+S(l+1)
7. undo(V ← d)
8. if k < l
9. return(status, k)
10. NewNG = ∪d∈Dom(V)(NG[d, V] − {V ← d})
11. storeNoGood(NewNG)
12. X ← a = deepest assignment in NewNG
13. Reason(X ← a) = NewNG % Prune the assignment and label it using NewNG
14. k = lvl(X ← a) % level X ← a was made.
15. return(fail, k)
```

It was shown in [Mitchell, 2003] that even though FCCBJ+S cannot be simulated by tree like resolution, it can be simulated by negative resolution. Recall that negative resolution is a refinement in which we are only allowed to resolve together clauses of the form \(A ∨ x\) and \(B ∨ \neg x\) where \(B\) contains only negative literals. We can see why FCCBJ+S can be simulated by negative resolution. Consider line 10 of figure 2.10. This step is in fact a series of negative resolution steps. We start with the positive clause \((x_i, 1, \ldots, x_i, d)\) which states that the variable \(V_i\) must have a value. Then, for each value \(d\) in the domain of \(V_i\), we resolve with the clause responsible for the pruning of \(x_{i,d}\). Recall that FC and NoGoodProp associate with each pruned value the nogood that pruned it. But nogoods are negative clauses and as a result these steps are negative resolution steps. In addition, the final clause generated is itself a negative clause, therefore a nogood. Thus, the algorithm can still be p-simulated by negative resolution.

There exists an exponential separation between general resolution and negative resolution [Buresh-Oppenheim and Pitassi, 2003], which means that there exist infinite families of formulas that are refutable in a polynomial number of general resolution steps, but require a exponential number of negative resolution steps.
DPLL augmented with learning cannot be p-simulated by negative resolution. This is because of the branching strategy of DPLL which tries both $x$ and $\bar{x}$ at each node. As a result, the clauses learned may contain both positive and negative literals. This holds as well for other schemes such as the 1-UIP scheme which learn clauses that may contain non-decision literals.

It was shown [Mitchell, 2003, Hwang and Mitchell, 2005] that the separation between negative resolution and general resolution also carries over to the algorithms. That is, there exists an infinite family of problems such that an algorithm that uses an appropriate learning and 2-way branching scheme will show an instance of this family is unsatisfiable in polynomial time, but the same algorithm with any $d$-way branching scheme is bound by negative resolution and will thus need exponential time to solve the instance.

The key to Mitchell’s result is that it learns subsets of the prefix that caused the contradiction. With 2-way branching these prefixes involve both assignments and non-assignments making the learned subsets (learned nogoods) more general than standard CSP nogoods. That is, the algorithm uses the notion of generalized nogoods, which we introduced independently in [Katsirelos and Bacchus, 2003, Katsirelos and Bacchus, 2005]. Generalized nogoods are as general as clauses and are the reason that the algorithm can show this improvement. In the next chapter, we will explore the properties and meaning of generalized nogoods further and show other ways of learning them.

2.4.3 Maintaining generalized arc consistency

Before we proceed, we need to address algorithms that maintain generalized arc consistency during search. So far, we have only talked about algorithms that perform forward checking during the constraint propagation phase. This does not affect our observations significantly. Maintaining GAC using a generic GAC algorithm requires time exponential in the arity of the constraint. Each pruning caused by GAC is implicitly associated with a clause explaining the pruning. In general, the number of resolution steps needed to discover this clause is exponential in the arity of the constraint, so an equivalency is maintained.

There are however complications. As we mentioned in chapter 1, one of the most attractive aspects of CSPs in practice is the availability of a large number of specialized propagators for large arity constraints. These propagators achieve some degree of consistency (sometimes GAC, often only bounds consistency) in polynomial time, whereas
a generic algorithm would require exponential time. If we try to reconstruct a resolution refutation out of a tree search refutation that uses these propagators, we could potentially need an exponential (in the arity of the constraint) number of resolution steps to explain the prunings. This effect is hard to characterize. There are instances that are hard without the use of propagators and easy with them, but there are also instances that are hard either way.

Rather than trying to quantify the effect of propagators on the complexity properties of the algorithms, we will deal in chapter 4 with the practical issue of integrating constraint propagators with nogood learning techniques and efficiently discovering nogoods for the prunings generated by the propagators.
Chapter 3

Generalized nogoods

We show here that the notion of nogood as previously defined in the CSP literature is limited in comparison to the equivalent notion of a propositional clause. We then show how to generalize nogoods (to create \textit{g-nogoods}) so as to remove this limitation and make them equivalent to propositional clauses, and we also provide an intuitive view of their meaning. We then show that g-nogoods are more powerful than standard nogoods (\textit{s-nogoods}) both locally (each g-nogood prunes more branches than the corresponding s-nogood) and globally (using g-nogoods instead of s-nogoods yields more powerful algorithms).

3.1 Generalizing nogoods

Recall first the standard definition of nogoods from chapter 2. A nogood is a set of assignments that cannot be extended to a solution. From here on, we will refer to nogoods that adhere to this definition as \textit{standard nogoods} or \textit{s-nogoods}.

Consider the constraint $X + Y < Z$ with $\text{Dom}(X) = \text{Dom}(Y) = \{0, 1, 2\}$, and $\text{Dom}(Z) = \{1, 2, 3\}$. This constraint is violated by the assignment set $A = \{X \leftarrow 0, Y \leftarrow 1, Z \leftarrow 1\}$, thus $A$ is a nogood. It is also impossible to satisfy $C$ if we assign $Z \leftarrow 1$ and at the same time prune 0 from the domain of $X$. However, this condition cannot be expressed as a standard nogood, since there is no single set of assignments representing this condition.

We can generalize nogoods so that they can express such conditions by allowing nogoods to contain either assignments or non-assignments. Non-assignments correspond to pruned values—once the value $a$ is pruned from the domain of $V$ we can no longer make the assignment $V \leftarrow a$, thus $V \leftarrow a$ becomes false. With generalized nogoods we
can then express the above condition with the set \( \{ X \neq 0, Z \leftarrow 1 \} \).

We now proceed to a formalization of generalized nogoods. A partial assignment to a
CSP assigns a unique value to some variables, and thus each time a variable is assigned
a value all other values in the variable’s domain are implicitly pruned.

**Definition 3.1.1** (Expanded partial assignment). An **expanded partial assignment**
to a CSP is a partial assignment to which all implicitly true non-assignments (prunings) have been added.

**Definition 3.1.2** (Expanded solution). An **expanded solution** to a CSP is an expanded
partial assignment that is also a solution.

For example, if the CSP contains only the variables \( X \) and \( Y \) each with the domain
of values \( \{ 1, 2, 3 \} \), then the partial assignment \( \{ X \leftarrow 1 \} \) corresponds to the expanded
partial assignment \( \{ X \leftarrow 1, X \neq 2, X \neq 3 \} \), while the solution \( \{ X \leftarrow 1, Y \leftarrow 2 \} \)
corresponds to the expanded solution \( \{ X \leftarrow 1, X \neq 2, X \neq 3, Y \neq 1, Y \leftarrow 2, Y \neq 3 \} \).
We can now define a generalized nogood.

**Definition 3.1.3.** A **generalized nogood** (or g-nogood) is any set of assignments and/or non-assignments that is not contained in any expanded solution.

Under this definition, every standard nogood is also a generalized nogood.

Recall our observation from the previous chapter that standard nogoods are equivalent
to negative clauses. We see that generalized nogoods are not restricted in this way, and are
in fact equivalent to propositional clauses. For example, the g-nogood \( \{ X \neq a, Y \leftarrow b \} \)
corresponds to the clause \( (X = a \lor Y \neq b) \). This allows us to talk about resolution in the
context of g-nogoods. In the rest of this thesis, we will reuse the notation \( R(A, B) \) which
was introduced in the previous chapter to denote the resolvent of two clauses. When \( A \)
and \( B \) are g-nogoods instead of clauses, \( R(A, B) \) will be used to denote the resolvent of
the two g-nogoods. If \( A \) contains \( X \neq a \) and \( B \) contains \( X \leftarrow a \), then \( R(A, B) = A \cup B - \{ X \leftarrow a, X \neq a \} \). Similarly, we reuse the notation \( r(A, B) \) to denote the assignment
that is used in the resolution step, so that in the previous example \( r(A, B) = X \leftarrow a \).

### 3.1.1 Learning generalized nogoods

Backtracking search can be adapted so that it learns new generalized nogoods (g-nogoods)
during search. To make this precise we first make a few observations. (a) Any expanded
solution must contain one of $X \leftarrow a$ or $X \not\leftarrow a$: if the expanded solution does not contain $X \leftarrow a$ it must contain some other assignment to $X$ and thus must contain $X \not\leftarrow a$. (b) If $Reason_1$ and $Reason_2$ are two g-nogoods containing $X \leftarrow a \in Reason_1$ and $X \not\leftarrow a \in Reason_2$, then $Reason_3 = (Reason_1 \cup Reason_2) - \{X \leftarrow a, X \not\leftarrow a\}$ is a new g-nogood: if $S$ is an expanded solution extending $Reason_3$ then since $S$ must contain one of $X \leftarrow a$ or $X \not\leftarrow a$ it must also extend one of $Reason_1$ or $Reason_2$ contradicting the fact that these sets are g-nogoods. This point is also a consequence of the fact that g-nogoods are clauses.

Consider now the algorithm FCCBJ+S, shown in figure 2.10 and repeated for convenience in 3.1. Recall that in order for the algorithm to perform intelligent backtracking, forward checking needs to label each pruned value with a nogood. This nogood is the result of a direct constraint violation. If $V \leftarrow x$ was pruned because of constraint $C$, the nogood consists of the assignments to the other variables of $C$ together with $V \leftarrow x$. Since this nogood contains assignments only, it is an s-nogood. When all values of a variable $V$ are pruned, we use their s-nogood labels to compute a new nogood in line 10. The new nogood is the union of the labels of all values of $V$ minus the values themselves. We call the scheme used to compute a nogood in line 10 of this figure the CBJ scheme. We showed in chapter 2 that line 10 simulates a number of negative resolution steps. The new nogood is stored and used as a new constraint, so that when all but one assignments in it are true, the remaining one is made false by pruning the corresponding value.

---

### Figure 3.1 FCCBJ+S: FCCBJ with nogood learning

```c
int FCCBJ+S(l)

1. If all variables assigned return (success, 0)
2. choose next variable $V$. Always choose a variable that has had all of its values pruned if one exists.
3. foreach $d \in CurDom(V)$
4. $V \leftarrow d$
5. NoGoodProp() and forwardCheck()
6. (status, k) = FCCBJ+S(l+1)
7. undo($V \leftarrow d$)
8. if $k < l$
9. return (status, k)
10. $NewNG = \bigcup_{d \in Dom(V)} \left\{ NG[d, V] - \{V \leftarrow d\}\right\}$
11. storeNoGood($NewNG$)
12. $X \leftarrow a =$ deepest assignment in $NewNG$
13. $Reason(X \leftarrow a) = NewNG$ % Prune the assignment and label it using $NewNG$
14. $k = lvl(X \leftarrow a)$ % level $X \leftarrow a$ was made.
15. return (fail, k)
```
In order to discover g-nogoods, we first have to modify FCCBJ+S so that as it explores a branch, it creates a trail of assignments and prunings, i.e., non-assignments. Each decision is immediately placed on the trail. When forward checking prunes a value, it places the corresponding non-assignment on the trail, labeling it with a nogood reason. Similarly, when nogood propagation prunes a value, it places the non-assignment on the trail and labels it with the nogood that became unit. When the domain of a variable is reduced to a single value, the sole remaining value is placed on the trail as a forced assignment. The nogood used to label this assignment is the “must-have-a-value” nogood. This trail is implicitly generated by FCCBJ+S, so we only make this explicit here.

We also need to slightly modify the notion of depth, which we discussed in chapter 2, so that it reflects not only the level where a (non-)assignment was made, but also its position within the trail. We say that the depth of the assignment $V \leftarrow d$ that occurs at line 4 of FCCBJ+S (figure 3.1) is $l.0$. The other (non)-assignments forced by forwardCheck and NoGoodProp occur in a sequence after this assignment, and the depth of each of these is $l.j$, so that if $k$ (non)-assignments have been made at level $L$ other than the decision, $l.1$ is the earliest of those and $l.k$ is the latest.

We say that (non)-assignment $A$ with $lvl(A) = H.i$ is deeper than (non)-assignment $B$ with $lvl(B) = l.j$, if $H > L$ or if $H = L \land i > j$. Also all assignments with depth $H.0$ for some $H$ are called decision assignments, and all other (non)-assignments are called forced. All forced (non)-assignments are labeled by an associated g-nogood that contains their negation. The labeling g-nogood for forced (non)-assignments is set by forwardCheck (in which case it is just an s-nogood) or by NoGoodProp. We also define $prunelvl(A) = lvl(\neg A)$. For example, if we have pruned the value $a$ from the domain of variable $V$ at the position $l.j$ in the trail, then $lvl(V \not\leftarrow a) = l.j$ and $prunelvl(V \leftarrow a) = l.j$. Clearly the notations are interchangable, so we will use the one that is more informative in a given context.

**Example 1.** Let $P = (V, D, C)$ be a constraint problem with $V = \{V_1, V_2, V_3\}$, $D(V_1) = D(V_2) = D(V_3) = \{1, 2, 3, 4\}$ and $C = \{C_1, C_2\}$, where $C_1 = V_1 < V_2$, and $C_2 = V_1 < V_3$. In addition, let the nogood store be $N = \{N_1\}$ where $N_1 = \{V_1 \not\leftarrow 4, V_2 \not\leftarrow 3\}$. At the first level we make the assignment $V_1 \leftarrow 2$. This causes the values $V_1 \leftarrow 1, V_1 \leftarrow 3, V_1 \leftarrow 4$ to be pruned by the implied “at-most-one-value” nogood, then the values $V_2 \leftarrow 1, V_2 \leftarrow 2, V_3 \leftarrow 1, V_3 \leftarrow 2$ are pruned by forward checking the constraints $C_1$ and $C_2$. Then, unit propagation of $N_1$ will make $V_2 \not\leftarrow 3$ false,
which means it will make the assignment \( V_2 \leftarrow 3 \). Then, the “at-most-one-value” nogood will prune the value \( V_2 \leftarrow 4 \). Following these events, the depths are as follows:

\[
\begin{align*}
\text{lvl}(V_1 \leftarrow 2) & = 1.0, \text{lvl}(V_1 \not\leftarrow 1) = 1.1, \text{lvl}(V_1 \not\leftarrow 3) = 1.2, \text{lvl}(V_1 \not\leftarrow 4) = 1.3, \\
\text{lvl}(V_2 \not\leftarrow 2) & = 1.5, \text{lvl}(V_3 \not\leftarrow 1) = 1.6, \text{lvl}(V_3 \not\leftarrow 2) = 1.7, \text{lvl}(V_2 \leftarrow 3) = 1.8, \text{lvl}(V_2 \not\leftarrow 4) = 1.9.
\end{align*}
\]

The trail and the associated depths and labels for each (non-)assignment are shown in figure 3.2. Note that the depth of each (non-)assignment depends not only on the problem but also some implementation details. For example, if forward checking is applied on \( C_2 \) before \( C_1 \) instead of the other way around, then the order of the prunings of \( V_2 \leftarrow 1, V_2 \leftarrow 2, V_3 \leftarrow 1, V_3 \leftarrow 2 \) would change and therefore so would the depths assigned to them. Similarly, if unit propagation is applied before forward checking, \( V_2 \leftarrow 4 \) would be pruned before other values.

![Figure 3.2 An example of an assignment trail and the corresponding nogood labels.](image)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Assignment</th>
<th>Reason</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>( V_1 \leftarrow 2 )</td>
<td>Decision</td>
<td>( { V_1 \leftarrow 2 } )</td>
</tr>
<tr>
<td>1.1</td>
<td>( V_1 \not\leftarrow 1 )</td>
<td></td>
<td>( { V_1 \leftarrow 2, V_1 \leftarrow 1 } )</td>
</tr>
<tr>
<td>1.2</td>
<td>( V_1 \not\leftarrow 3 )</td>
<td>“at-most-one-value” nogood</td>
<td>( { V_1 \leftarrow 2, V_1 \leftarrow 3 } )</td>
</tr>
<tr>
<td>1.3</td>
<td>( V_1 \not\leftarrow 4 )</td>
<td></td>
<td>( { V_1 \leftarrow 2, V_1 \leftarrow 4 } )</td>
</tr>
<tr>
<td>1.4</td>
<td>( V_2 \not\leftarrow 1 )</td>
<td>( C_1 )</td>
<td>( { V_1 \leftarrow 2, V_2 \leftarrow 1 } )</td>
</tr>
<tr>
<td>1.5</td>
<td>( V_2 \not\leftarrow 2 )</td>
<td></td>
<td>( { V_1 \leftarrow 2, V_2 \leftarrow 2 } )</td>
</tr>
<tr>
<td>1.6</td>
<td>( V_3 \not\leftarrow 1 )</td>
<td>( C_2 )</td>
<td>( { V_1 \leftarrow 2, V_3 \leftarrow 1 } )</td>
</tr>
<tr>
<td>1.7</td>
<td>( V_3 \not\leftarrow 2 )</td>
<td></td>
<td>( { V_1 \leftarrow 2, V_3 \leftarrow 2 } )</td>
</tr>
<tr>
<td>1.8</td>
<td>( V_2 \leftarrow 3 )</td>
<td>( N_1 )</td>
<td></td>
</tr>
<tr>
<td>1.9</td>
<td>( V_2 \not\leftarrow 4 )</td>
<td>“at-most-one-value” nogood</td>
<td>( { V_2 \leftarrow 3, V_2 \leftarrow 4 } )</td>
</tr>
</tbody>
</table>

When a domain wipeout occurs a new g-nogood can be computed by replacing line 10 of FCCBJ+S with a call to the algorithm in figure 3.3. We use \( \equiv \) to denote \( \leftarrow \) or \( \not\leftarrow \).

We call this scheme for learning a new g-nogood a 1st-Decision scheme. It iteratively replaces the deepest (non-)assignments in the implicit “variable must have a value” nogood until it arrives at a g-nogood whose deepest (non-)assignment is a decision assignment. This iterative replacement generally leaves a number of the original non-assignments \( V \not\leftarrow d \) (line 1) in \( \text{NewNG} \), i.e., the scheme learns a g-nogood rather than an s-nogood. This new g-nogood can then be used just as before to backtrack and to label the assignment we undo on backtrack (lines 11-15). We show that \( \text{NewNG} \) is a valid g-nogood at each iteration and that the process must terminate with \( \text{NewNG} \) containing a single decision assignment.
Figure 3.3 Computing a first-decision nogood after a domain wipeout

\[
\text{compute-first-decision-nogood}(V) \quad \% \text{Domain of } V \text{ has been wiped out}
\]

1. \( \text{NewNG}_0 = \{V \not\in d \mid d \in \text{Dom}(V)\} \)
2. \( i=0 \)
3. \( \text{do} \)
4. \( X \leftarrow a = \text{deepest (non)-assignment in } \text{NewNG} \)
5. \( \text{if } X \leftarrow a \text{ is not a decision assignment} \)
6. \( \text{NewNG}_{i+1} = \text{NewNG}_i \cup \text{Reason}(X \leftarrow a) - \{X = a, X \leftarrow a\} \)
7. \( i \leftarrow i + 1 \)
8. \( \text{until } X \leftarrow a \text{ is a decision assignment} \)
9. \( \text{NewNG} = \text{NewNG}_i \)

Example 2. Let \( P = (\mathcal{V}, \mathcal{D}, \mathcal{C}) \) be a constraint problem with \( \mathcal{V} = \{V_1, V_2, V_3, V_4\} \), \( D(V_1) = D(V_2) = D(V_3) = D(V_4) = \{1, 2, 3, 4\} \) and \( \mathcal{C} = \{C_1, C_2, C_3\} \), where \( C_1 = V_1 < V_2, C_2 = V_1 < V_3, C_3 = V_4 \neq V_2 \). In addition, let the nogood store be \( \mathcal{N} = \{N_1, N_2\} \) where \( N_1 = \{V_1 \not\in 4, V_2 \leftarrow 4\} \) and \( N_2 = \{V_2 \leftarrow 3, V_3 \not\in 2\} \). At the first level we make the assignment \( V_4 \leftarrow 1 \) and at the second level, we make the assignment \( V_1 \leftarrow 2 \). At the end of the second level, the domain of \( V_2 \) is wiped out, so we perform conflict resolution using the 1\textsuperscript{st}-Decision scheme. The assignment trail, depths and nogood labels after propagation in both levels is shown in figure 3.4.

Figure 3.4 An example of the operation of the 1\textsuperscript{st}-Decision scheme.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Assignment</th>
<th>Label</th>
<th>Iteration</th>
<th>Tentative nogood</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>( V_4 \leftarrow 2 )</td>
<td>{( V_4 \leftarrow 2 )}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>( V_4 \not\in 1 )</td>
<td>{( V_4 \leftarrow 2, V_4 \leftarrow 1 )}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>( V_4 \not\in 3 )</td>
<td>{( V_4 \leftarrow 2, V_4 \leftarrow 3 )}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>( V_4 \not\in 4 )</td>
<td>{( V_4 \leftarrow 2, V_4 \leftarrow 4 )}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>( V_2 \not\in 2 )</td>
<td>{( V_4 \leftarrow 2, V_2 \leftarrow 2 )}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td>( V_1 \leftarrow 2 )</td>
<td>{( V_1 \leftarrow 2 )}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>( V_1 \not\in 1 )</td>
<td>{( V_1 \leftarrow 2, V_1 \leftarrow 1 )}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>( V_1 \not\in 3 )</td>
<td>{( V_1 \leftarrow 2, V_1 \leftarrow 3 )}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>( V_1 \not\in 4 )</td>
<td>{( V_1 \leftarrow 2, V_1 \leftarrow 4 )}</td>
<td>5</td>
<td>{( V_2 \not\in 2, V_1 \leftarrow 2 )} \text{- Stop}</td>
</tr>
<tr>
<td>2.4</td>
<td>( V_2 \not\in 1 )</td>
<td>{( V_2 \leftarrow 2, V_2 \leftarrow 1 )}</td>
<td>4</td>
<td>{( V_2 \not\in 2, V_1 \not\in 4, V_1 \leftarrow 2 )}</td>
</tr>
<tr>
<td>2.5</td>
<td>( V_3 \not\in 1 )</td>
<td>{( V_1 \leftarrow 2, V_3 \leftarrow 1 )}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>( V_3 \not\in 2 )</td>
<td>{( V_1 \leftarrow 2, V_3 \leftarrow 2 )}</td>
<td>3</td>
<td>{( V_2 \not\in 1, V_2 \not\in 2, V_1 \not\in 4, V_1 \leftarrow 2 )}</td>
</tr>
<tr>
<td>2.7</td>
<td>( V_2 \not\in 4 )</td>
<td>( N_1 )</td>
<td>2</td>
<td>{( V_2 \not\in 1, V_2 \not\in 2, V_3 \not\in 2, V_1 \not\in 4 )}</td>
</tr>
<tr>
<td>2.8</td>
<td>( V_2 \not\in 3 )</td>
<td>( N_2 )</td>
<td>1</td>
<td>{( V_2 \not\in 1, V_2 \not\in 2, V_2 \not\in 4, V_3 \not\in 2 )}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>{( V_2 \not\in 1, V_2 \not\in 2, V_2 \not\in 3, V_2 \not\in 4 )}</td>
</tr>
</tbody>
</table>

In this figure, propagation proceeds downwards, while conflict resolution to get the
new 1st-Decision nogood proceeds from the bottom up. The procedure starts with the “must-have-a-value” nogood for $V_2$ and iteratively resolves away the deepest assignment. At the $i^{th}$ iteration of 1st-Decision, we resolve the label of iteration $i$ with the tentative nogood of iteration $i - 1$ to get the new tentative nogood of iteration $i$. At the fifth iteration, the tentative nogood is $\{V_2 \neq 2, V_1 \leftarrow 2\}$. The non-assignment $V_2 \neq 2$ has been made at level 1, while the assignment $V_1 \leftarrow 2$ is the decision at level 2. Since $V_1 \leftarrow 2$ is the deepest of the two and is a decision, the process stops. Unlike the CBJ scheme, this process left the non-assignment $V_2 \neq 2$ in the resulting nogood.

**Proposition 3.1.4.** The procedure in figure 3.3 (a) produces a valid nogood at each iteration; (b) terminates and the deepest assignment in the resulting nogood is the decision assignment at the level where the domain wipeout occurred.

**Proof.** (a) At line 1 $NewNG_0$ is initialized to the “must-have-a-value” nogood which is by definition valid. At each iteration $i$ the deepest assignment in $NewNG_i$, $X \leftarrow a$ is labeled by a nogood that contains $\overline{X} \leftarrow \overline{a}$. This means that $NewNG_i \cup Reason(X \leftarrow a) - \{X \leftarrow a, X \leftarrow \overline{a}\}$ which is computed in line 6 is the resolvent of $NewNG$ and $Reason(X \leftarrow a)$, thus it is a valid nogood.

(b) Let the level where the DWO (domain wipeout) occurs be $l$. We show first that at every step $i$ the deepest assignment in the trail that appears in $NewNG_i$ is made no earlier than $l.0$. Assume there exists an $i$ where the contrary holds. Consider the sequence of resolution steps that was used in line 6 to derive $NewNG_i$. These can be mapped to a sequence of nogood and forward check propagation steps (triggered by either learned nogoods or by constraints), which lead to the DWO. This means that a DWO can be derived from $NewNG_i$ by propagation. Since the deepest assignment in $NewNG_i$ was made before $l.0$, we should have derived the DWO before making the decision at $l.0$, which is a contradiction.

Observe that for each $NewNG_i$, $NewNG_{i+1}$ is created in line 6 by replacing a (non-)assignment $X \leftarrow a$ by the nogood that was used to label its complement $\overline{X} \leftarrow \overline{a}$. This label necessarily contains (non-)assignments that appear earlier than $X \leftarrow a$ on the assignment trail. Thus, at each iteration, the algorithm proceeds on the assignment trail backwards from the DWO. But we know that a finite number of assignments are on the trail between the most recent decision and the DWO and that it cannot reach a point on the trail before $l.0$. Thus the algorithm will terminate. Moreover, since no decisions were made between $l.0$ and the DWO, it will terminate when the deepest assignment in
Chapter 3. Generalized nogoods

$\text{NewNG}_i$ is the decision made at $l.0$. □

Notice that compute-first-decision-gnogood does not change the invariants mentioned in chapter 2, page 18. Specifically, a (non-)assignment $A$ that is made false at level $l$ is labeled with a nogood $Reason(A)$ that contains $A$; the rest of the assignments in $Reason(A)$ are made true earlier than $A$ was falsified, the deepest assignment was made true at level $l$. As discussed in chapter 2, these invariants are important for both correctness (the first and second) and efficiency (the third). In fact, compute-first-decision-gnogood also depends on these invariants for correctness. For example, if the first invariant did not hold, the resolution step of line 6 would not be possible.

3.1.2 Using g-nogoods during search

In order to construct an algorithm that learns g-nogoods, it is not sufficient to simply modify line 10 of FCCBJ+S. There is an underlying assumption (see section 2.4.2) in the algorithm FCCBJ+S that decisions can only make nogoods unit, even after constraint propagation. Indeed, s-nogoods can never become completely true, generating a contradiction, during constraint propagation and unit propagation, because at each level only a single assignment is made.

When we learn g-nogoods, however, this is no longer the case. A g-nogood may contain non-assignments as well as assignments and we make many assignments at each level. So we must handle the case when a learned g-nogood has become true (indicating a contradiction).

Definition 3.1.5. A nogood $Ng$ whose assignments are all true at node $n$ is a conflict at node $n$.

Example 3. Consider the set of nogoods $N_1 = \{V_1 \leftarrow 1, V_2 \leftarrow 2\}, N_2 = \{V_1 \leftarrow 1, V_3 \leftarrow 3\}$ and $N_3 = \{V_2 \not\leftarrow 2, V_3 \not\leftarrow 3\}$. If we make the assignment $V_1 \leftarrow 1$, unit propagation on $N_1$ will cause $V_2 \leftarrow 2$ to be pruned and unit propagation on $N_2$ will cause $V_3 \leftarrow 3$ to be pruned. At this point, all the assignments of $N_3$ will be true and therefore $N_3$ is a conflict. □

In order to handle conflicts, we need to modify the main loop of the algorithm as well. The revised algorithm is shown in figure 3.5. In fact, this algorithm has no parts that are specific to forward checking, other than calling propagate-forward-checking, which is in turn shown in figure 3.6.
This algorithm looks similar to FCCBJ+S. However, where FCCBJ+S assumes that the only possible reason for a deadend in the search tree is that the domain of a variable has been wiped out, either by constraint propagation or by search, FCCBJ+G handles the possibility that any learned nogood may give rise to a conflict. This is done by handling domain wipeouts and other conflicts in a uniform manner. Before the search begins, the "must-have-a-value" nogood for each variable $V$ is added to the nogood database explicitly and denoted $Reason[V]$. It then becomes unnecessary to explicitly handle domain wipeouts. If the domain of variable $V$ is wiped out, unit propagation on $Reason[V]$ will detect that this nogood has become true and handle it as such.

**Figure 3.5** Forward checking with g-nogood learning

```c
int FCCBJ+G(l)
1. If all variables assigned return(success, 0)
2. choose next variable $V$.
3. foreach $d \in CurDom(V)$
4. $V \leftarrow d$
5. propagate-forward-checking($V$)
6. if conflict
7. (status, k) = (fail, resolve-conflict(conflict))
8. else
9. (status, k) = FCCBJ+G(l+1)
10. undo($V \leftarrow d$)
11. if k < l
12. return (status, k)
13. return (fail, resolve-conflict(conflict))
```

FCCBJ+G calls `propagate-forward-checking` in line 5 and expects it to either succeed without detecting any conflict or to set the global variable `conflict` to point to the nogood that has become true. In the former case, it goes on to search the subtree in line 9. In the latter case, it calls `resolve-conflict` in line 7 in order to discover new nogoods and jump back as needed. Note that `propagate-forward-checking` simply calls `forward-check` and `unit-propagate` (shown in figure 3.7), which handles the low level details of unit propagation as well as updating the variable `conflict`.

The procedure `resolve-conflict` (figure 3.8) implements the first-decision scheme we also showed in figure 3.3. The difference is that this procedure works with any nogood that has become true and takes care of recording the nogood.

**Unit propagation.** We have already seen that unit propagation on s-nogoods simply detects when all but one assignments have been made true and then prunes the value
that corresponds to the final assignment. In g-nogoods, this is not sufficient, as the final assignment may in fact be a non-assignment. Let \( X \not\leftarrow a \) be that non-assignment. To make \( X \not\leftarrow a \) false we must assign \( X \) the value \( a \), i.e., make \( X \leftarrow a \) true. We do this with the procedure \texttt{make-non-assignment-false}, shown in figure 3.9.

**Far backtracking.** Many modern SAT solvers use a technique called *far backtracking* when dealing with conflicts. The main difference is that line 11 of \texttt{resolve-conflict} (figure 3.8) becomes:

11. \textbf{return} \( \texttt{lvl} \) (second deepest (non)-assignment in \textit{NewNG})

The algorithm then backtracks to that level, performs unit and constraint propagation to account for the new nogood, makes a new choice at that level and continues the search.

Note that this does not affect the completeness of the solver. The solver backtracks to an earlier level but does not assume that the part of the search tree that was skipped does not have any solutions. Instead, it repeats the search, possibly making a different branching choice, because new nogoods have been learned since the branching choice was originally made. The intuition is that this will lead the solver to make a different choice, possibly discovering a shorter refutation of the reduced problem. It is hoped that this
**Figure 3.8** Resolving a single conflict using the first-decision scheme

```c
int resolve-conflict(Ng)
{
    NewNG₀ = Ng
    i = 0
    do
        X ⇔ a = deepest (non)-assignment in NewNG
        if X ⇔ a is not a decision assignment
            NewNGᵢ₊₁ = NewNGᵢ ∪ Reason(X ⇔ a) − {X ⇔ a, X ⇔ a}
            i ← i + 1
        until X ⇔ a is a decision assignment
    NewNG = NewNGᵢ
    record(NewNG)
    return lvl(deepest (non)-assignment in NewNG)
}
```

**Figure 3.9** Making a non-assignment false

```c
make-non-assignment-false(X ⇔ a, Ng)
{
    Push X ⇔ a on the assignment stack and label it with Ng
    foreach d ∈ CurDom(X) − {a}
        Prune X ⇔ d and label it with {X ⇔ a, X ⇔ d}
}
```

will increase the robustness of the solver. In that respect, far backtracking is not unlike restarting.

**Different nogood learning schemes.** There are many possible schemes for learning generalized nogoods other than the first-decision scheme. We have covered already the CBJ scheme (line 10 of figure 3.1) and the first-decision scheme (figure 3.8). We also mentioned in chapter 2 that the 1-UIP [Zhang et al., 2001] scheme has proven very successful in modern SAT solvers. The first-decision and 1-UIP schemes are actually instances of the generic `resolve-unwind-trail` scheme shown in figure 3.10, with different conditions. In figure 3.11 we show how `resolve-unwind-trail` can be instantiated to implement the 1-UIP scheme.

The condition used by the 1-UIP scheme is to terminate when `NewNGᵢ` contains a single (non-)assignment at the decision level. A UIP nogood is useful because it allows the search to assert the complement of the deepest (non-)assignment when it backtracks. Note that a first-decision nogood is also a UIP nogood, thus it shares this advantage. A non-UIP nogood would allow the solver to backtrack, but it would not be unit after backtracking, thus no propagation would occur. Further, a 1-UIP nogood is the one that
is chronologically “closer” to the conflict.\footnote{1-UIP nogoods are also topologically closer to the conflict in the implication graph, as covered in chapter 2.} No formal argument has been presented on why this property should be desirable, but it has been successful in practice and most modern SAT solvers use this scheme.

Finally, note that even though the CBJ scheme cannot be expressed as an instantiation of resolve-unwind-trail, $NG_{CBJ}$ can be derived from any first-decision or 1-UIP nogood by continuing to resolve away non-decision (non-)assignments until only decision assignments are left in the nogood.

When learning 1-UIP nogoods, we encounter the problem that this nogood cannot necessarily be used to label the decision made at the backtrack level and therefore cannot be used with FCCBJ+$G$ as it is presented. A simple way to work around this problem is to compute the 1-UIP nogood as well as the first-decision nogood (if they are different) and record both. This can potentially be inefficient, however.

In practice, it is very hard to implement g-nogood learning without also using far backtracking. The reason is that when we do not perform far backtracking, unit propagation of the newly learned nogood requires that we insert the new pruning in the middle of the assignment trail and consequently update the rest of the trail so that it is consistent with the inserted pruning. When we perform far backtracking, the new pruning is
placed at the end of the assignment trail and we do not need to do any more to ensure consistency. Therefore, in our empirical evaluation of these algorithms, we always use far backtracking.

**Other improvements.** In every CSP, a set of nogoods always exists. Specifically, we know that for every variable $V$ with domain size $d$ there exists one “must-have-a-value” nogood $\{ V \not\leftarrow 1, \ldots, V \not\leftarrow d \}$ and $O(d^2)$ “at-most-one-value” nogoods $\{ V \leftarrow i, V \leftarrow j \}, 1 \leq i < j \leq d$. We can use the presence of these nogoods to further improve performance.

**Choice of watch assignments.** During unit propagation, when we try to find a replacement for a watch that is no longer non-true, we stop as soon as we discover a replacement watch. If instead we keep searching for a better replacement, we might be able to save more work later. We define an ordering $<_w$ between two potential watches $w_1$ and $w_2$ so that $w_1 <_w w_2$ if $w_1$ is false but $w_2$ is not; or $w_1$ was made false at level $k$ and $w_2$ was made false at level $k' > k$. This requires that we always examine the entire nogood, as opposed to stopping when a replacement watch is found. This carries a runtime overhead, but this overhead might be offset by needing to perform fewer unit propagations.

Next, we discuss the application of hyperresolution to the known nogoods. Hyperresolution is a resolution step that involves more than two clauses. It has been suggested that the application of hyperresolution in SAT solvers [Bacchus, 2002] can lead to improved performance in some cases. In CSPs, given that the “must-have-a-value” and “at-most-one-value” nogoods are always present, we can apply some simple hyperresolution rules at almost no runtime cost.

**Domain clause subsumption.** Assume that the nogood $A \cup \{ V \leftarrow x, V \not\leftarrow y, V \not\leftarrow z \}$ is discovered. The presence of the “at-most-one-value” nogoods allows us to remove the literals $V \not\leftarrow y, V \not\leftarrow z$ from the nogood by implicitly resolving the new nogood with $\{ V \leftarrow x, V \leftarrow y \}$ and $\{ V \leftarrow x, V \leftarrow z \}$. The new nogood is shorter, which helps both in the strength (it will cause more prunings) and in the efficiency of propagation (it will have lower overhead).
Detection of single-variable nogoods. By applying some extra care in the choice of watch literals during unit propagation, we can detect when a nogood has been reduced to a single-variable nogood. A single-variable nogood has the form \( N_g = A \cup \{ V \not\leftarrow x_1, \ldots, V \not\leftarrow x_k \} \) and every assignment in \( A \) is true. For each value \( i \notin \{ x_1, \ldots, x_k \} \), we can resolve \( N_g \) with each of the nogoods \( \{ V \leftarrow i, V \leftarrow j \} \forall j \in \{ x_1, \ldots, x_k \} \). We end up with the set of nogoods \( A \cup \{ V \leftarrow i \} \), with every assignment in \( A \) being true. This allows us to prune \( V \leftarrow i \) for every \( i \notin \{ x_1, \ldots, x_k \} \), even though \( N_g \) itself is not unit. To detect this, we amend the \( \langle w \rangle \) ordering as follows: given a watch \( w \), which will not be replaced, and two potential watches \( w_1 \) and \( w_2 \), \( w_1 \prec_w w_2 \) if \( w \) and \( w_2 \) are non-assignments to the same variable but \( w_1 \) is not; or \( w_1 \) is false and \( w_2 \) is not; or \( w_1 \) was made false at level \( k \) and \( w_2 \) was made false at level \( k' > k \). If during unit propagation the two watches \( w_1 \) and \( w_2 \) of a nogood \( N_g \) are non-assignments to the same variable, we know that \( N_g \) has been reduced to a single-variable nogood and we can perform the pruning described earlier.

Notice that using single variable nogoods in this way violates the first of the invariants we described for using nogoods to label pruned values. Specifically, if \( N_g = A \cup \{ V \not\leftarrow x_1, \ldots, V \not\leftarrow x_k \} \) is used to prune a value \( V \leftarrow i \) with \( i \notin \{ x_1, \ldots, x_k \} \), then \( \text{Reason}(V \leftarrow i) = N_g \), but \( N_g \) does not contain \( V \leftarrow i \), a fact on which \text{resolve-single-conflict} \, \text{and resolve-single-conflict-uip} \, \text{depend on for correctness}. This can be easily corrected. As described above, it is easy to generate a nogood that does contain \( V \leftarrow i \), which is \( A \cup \{ V \leftarrow i \} \). Moreover, in an implementation this nogood does not have to be generated explicitly so that there is no overhead involved.

3.2 Generalized nogoods are more powerful

We described briefly in the introduction to this chapter that g-nogoods can compactly express conditions that cannot be expressed by any single s-nogood. As an example, we used the constraint \( C = X + Y < Z \), where \( X, Y \) and \( Z \) have only non-negative values in their domains. In that case, the g-nogood \( \{ X \not\leftarrow 0, Z \leftarrow 1 \} \) captures the condition that if we prune 0 from the domain of \( X \) and assign 1 to \( Z \), \( C \) cannot be satisfied. This condition cannot be captured by any single s-nogood.

In this section, we attempt to formalize the notion that g-nogood learning is more powerful than s-nogood learning in the following three ways: first, we show in section 3.2.1 that g-nogoods can compactly encode large numbers of s-nogoods, and prune paths not
pruned by any of these s-nogoods; second, we show in section 3.2.2 that, given a conflict, the 1st-Decision g-nogood that can be learned from this conflict prunes more branches than the CBJ s-nogood; finally and more interestingly, we prove in section 3.2.3 a concrete result about the power g-nogoods add to backtracking search, namely, that there exists a superpolynomial separation between algorithms that perform g-nogood learning and those that perform s-nogood learning only.

### 3.2.1 Expressive power

First, we show that g-nogoods can compactly represent a possibly exponentially sized set of standard nogoods. For example, consider the g-nogood \( \{X \not\leftarrow 0, Z \leftarrow 1\} \) of the earlier example, with \( \text{Dom}(X) = \{0, 1, 2\} \). This compactly captures the nogoods \( \{X \leftarrow 1, Z \leftarrow 1\} \) and \( \{X \leftarrow 2, Z \leftarrow 1\} \). When the g-nogood contains non-assignments to multiple variables, the set of s-nogoods it covers grows exponentially. For example, if all of the variables \( V_1, \ldots, V_{10} \) have domain \( \{0, 1, 2, 3, 4, 5\} \), then the g-nogood \( \{X \leftarrow 1, V_1 \not\leftarrow 0, \ldots V_{10} \not\leftarrow 0\} \) captures \( 5^{10} \) different s-nogoods. A procedure that transforms a g-nogood to its equivalent set of s-nogoods is shown in figure 3.12.

**Figure 3.12** Transforming a single g-nogood to a set of s-nogoods

```
gnogood-to-snogoods(Ng)
1. S = {}  
2. foreach variable X s.t. X \not\leftarrow a \in Ng for some a  
3.     foreach X \leftarrow b s.t. X \not\leftarrow b \notin Ng  
4.     Ng_{1,X} = \{Y = d s.t. Y = d \in Ng \land Y \neq X\}  
5.     S = S \cup \text{gnogood-to-snogoods}({X \leftarrow b} \cup Ng_{1,X})  
6. if S \equiv \emptyset  
7.     return \{Ng\}  
8.     return S
```

The procedure works as follows. In the loop of lines 2–5, for each variable \( X \) which appears in \( k_X \) non-assignments in \( Ng \), it creates \( |\text{Dom}(X)| - k_X \) nogoods. Each of these nogoods contains none of the \( k_X \) non-assignments (the intermediate variable \( Ng_{1,X} \) in line 4) but contains an assignment \( X \leftarrow b \) so that \( X \not\leftarrow b \) is not in \( Ng \). It then proceeds in line 5 to eliminate other non-assignments from each of the new nogoods. The correctness of each recursive call of this procedure is shown next. We also show that the resulting set of s-nogoods is equivalent to the original g-nogood.
Proposition 3.2.1. The procedure gnogood-to-snogoods transforms a valid g-nogood into a set of valid s-nogoods at each recursive call. It terminates with a set of s-nogoods. The resulting set of s-nogoods rules out the same set of expanded partial assignments as the original g-nogood.

Proof. Consider each of the nogoods \( \{X \leftarrow b\} \cup Ng_{\downarrow X} \) generated in line 5. We can generate such a nogood by resolving \( Ng \) with the “at-most-one-value” nogood \( \{X \leftarrow b, X \leftarrow d\} \) for each \( d \) such that \( X \not\leftarrow d \) appears in \( Ng \). This will remove all \( X \not\leftarrow d \) from \( Ng \) (which is done at line 4) and add \( X \leftarrow b \) (line 5). Thus the nogoods generated at each step are valid.

The procedure will terminate, because each recursive call completely eliminates all the non-assignments involving one of the variables. So if non-assignments from \( n \) variables appear in \( Ng \), a tree of recursive calls of depth at most \( n \) will be generated. Since non-assignments from all variables are eliminated, \( S \) contains s-nogoods only.

We show by induction on the number \( n \) of variables in the g-nogood that every partial assignment ruled out by \( Ng \) is also ruled out by at least one s-nogood in \( S \). For \( n = 0 \), it is obvious, as the only such g-nogood is \( \{\} \), which is expanded to the set \( \{\{\}\} \) of s-nogoods, both of which rule out all partial assignments.

Assume that the hypothesis holds for \( n = k - 1 \). We show that it holds for \( n = k \). Let \( X \) be the \( k^{th} \) variable. If \( X \) is assigned \( X \leftarrow a \) in the partial assignment, the g-nogood contains either \( X \leftarrow a \) or \( X \not\leftarrow b, b \neq a \). In the former case, all generated s-nogoods contain \( X \leftarrow a \). In the latter case, a subset \( S_X \) of the generated s-nogoods contain least one s-nogood that contains \( X \leftarrow a \), since the g-nogood contains \( X \not\leftarrow b \) but not \( X \not\leftarrow a \). This then means that \( Ng - \{X \leftarrow d \mid d \in Dom(X)\} \), which has \( k - 1 \) variables, rules out a different set of assignments from \( S_X \), which violates the inductive hypothesis.

Conversely, let an expanded partial assignment be ruled out by at least one s-nogood in \( S \) but not \( Ng \). Let \( N_s \) be that s-nogood. Then, for every variable \( X \) in \( N_s \), if \( X \) is assigned \( X \leftarrow a \) in the partial assignment, \( N_s \) must also contain \( X \leftarrow a \). Then, either \( Ng \) contains \( X \leftarrow a \), or it contains a set of non-assignments \( \{X \not\leftarrow d_1, \ldots, X \not\leftarrow d_k\} \), with \( a \neq d_i \) for all \( i \). \( Ng \) subsumes \( N_s \) for all variables, thus is also rules out the partial assignment, which is a contradiction.

Note that if a nogood \( Ng \) contains non-assignments from \( d \) different variables \( V_1, \ldots, V_d \) with \( V_i \) appearing \( n_{V_i} \) times, then the number of s-nogoods that are generated by gnogood-to-snogoods is \( \prod_{1 \leq i \leq d}(|Dom(V_i)| - n_{V_i}) \). Thus, the fewer non-assignments
Chapter 3. Generalized nogoods

that appear from each variable, the larger the number of s-nogoods that are generated. When $|\text{Dom}(V_i)| = 2$ for all $i$ then $\prod_{1 \leq i \leq d} (|\text{Dom}(V_i)| - n_{V_i}) = 1$.

Propagation with a g-nogood. A generalized nogood can prune paths in the search tree that the set of standard nogoods it captures cannot. For example, \{\(X \leftarrow 1, V_1 \not\leftarrow 0, \ldots V_{10} \not\leftarrow 0\)\} with \(D(V_1) = \ldots = D(V_{10}) = \{0, 1, 2, 3, 4, 5\}\) can prune branches where none of the \(V_i\) have yet been assigned, while none of the \(5^{10}\) standard nogoods it captures can prune any branches without all but one of the variables \(X, V_1, \ldots, V_{10}\) having been assigned. For instance, in any branch where all the assignments \(V_1 \leftarrow 0 \ldots V_{10} \leftarrow 0\) have been pruned, but \(V_1, \ldots, V_{10}\) are unassigned, unit propagation on the g-nogood would cause \(X \leftarrow 1\) to also be pruned. On the other hand, none of the \(5^{10}\) s-nogoods will be unit.

This can translate to potentially exponential savings in terms of the search space explored. If an algorithm were to discover this g-nogood during search, it would be able to prune a number of branches from the search tree. An algorithm that is only able to discover s-nogoods would have to discover each of the \(5^{10}\) equivalent nogoods individually and thus would have to search an exponentially larger search tree.

Interaction among g-nogoods. Consider the behavior of unit propagation on s-nogoods and g-nogoods. With s-nogoods, when all assignments are true except one, unit propagation ensures that the remaining one is false by pruning one or more values. This means that unit propagation can only prune values. But since s-nogoods contain only positive assignments, pruning a value can only satisfy an s-nogood, never make it unit. Thus unit propagation never “propagates” over a set of s-nogoods. In contrast, g-nogoods may contain non-assignments, which become true when the corresponding value gets pruned. This occurs much more often, thus g-nogoods are more likely to become unit as a result of pruning a value. For example, if (1) \(\{Y \leftarrow a, Z \leftarrow b\}\), (2) \(\{X \leftarrow a, Z \not\leftarrow b\}\), (3) \(\{W \leftarrow b, X \not\leftarrow a\}\) are g-nogoods in the store, and we make the assignment \(Y \leftarrow a\), unit propagation will force the pruning \(Z \not\leftarrow b\) using nogood 1, then \(X \not\leftarrow a\) using nogood 2 and then \(W \not\leftarrow b\) using nogood 3.

Similarly, the interaction between unit propagation and constraint propagation is largely one way with s-nogoods: prunings generated by unit propagation may lead constraint propagation to deduce more impossible values, but values pruned during constraint propagation cannot make s-nogoods unit. With g-nogoods this interaction is
two-way: g-nogoods becoming unit can cause more constraint propagation and values pruned by constraint propagation can make g-nogoods unit.

3.2.2 Local pruning power

We compare the pruning power of the learning schemes CBJ and first-decision. We prove that the nogood learned under the first-decision scheme will prune at least as many branches as the CBJ nogood. The following result covers a more general case, which is then easily applied to the specific comparison of CBJ and first-decision.

**Proposition 3.2.2.** Consider a CSP $\mathcal{P} = \langle V, D, C, N \rangle$, where $N$ is a database of nogoods entailed by $C$, and a partial assignment $A$ that causes a conflict after constraint propagation and unit propagation, while no subset of $A$ satisfies this condition. Consider now two nogoods $NG_1$ and $NG_2$ such that $NG_1$ can be derived from the conflict by trivial resolution (defined in section 2.2) and $NG_2$ can be derived from the same conflict by applying further trivial resolution steps to $NG_1$. Let $\mathcal{P}_1 = \langle V, D, C, N \cup \{NG_1\} \rangle$ and $\mathcal{P}_2 = \langle V, D, C, N \cup \{NG_2\} \rangle$. Then

- Every branch that is a conflict for $\mathcal{P}_2$, is also a conflict for $\mathcal{P}_1$ after constraint and unit propagation, but there exist branches where the converse is not true.
- $\mathcal{P}_1$ and $\mathcal{P}_2$ are incomparable with respect to unit propagation. There exist branches where some values are pruned in $\mathcal{P}_1$ but not in $\mathcal{P}_2$ and branches where some values are pruned in $\mathcal{P}_2$ but not in $\mathcal{P}_1$.

**Proof.** We note that the only difference between $\mathcal{P}_1$ and $\mathcal{P}_2$ is in the addition of the nogoods $NG_1$ and $NG_2$, respectively. Therefore, in any branch any difference in propagation between $\mathcal{P}_1$ and $\mathcal{P}_2$ can be attributed to a difference in unit propagation caused by $NG_1$ and $NG_2$. In the rest of this proof, we talk about $NG_1$ and $NG_2$ instead of $P_1$ and $P_2$.

First we show that for any branch where $NG_2$ is true, $NG_1$ is also true. We prove this by induction on the number $i$ of trivial resolution steps required to derive $NG_2$ from $NG_1$. The base case where $i = 0$ is trivially true, since it means that $NG_2 = NG_1$. Assume that it is true for $i = n$. We write $R(A, B)$ to denote the nogood that is the resolvent of the nogoods $A$ and $B$ and $r(A, B)$ to denote the variable that was used in this resolution step. For convenience, we also write $R(C_1, \ldots, C_n) = R(C_1, R(\ldots, R(C_{n-1}, C_n)))$. Then
we can write $NG_2 = R(C_{n+1}, R_n)$, where $R_n = R(C_n, R(\ldots, R(C_1, NG_1)))$, for some $C_1, \ldots, C_{n+1} \in \mathcal{N}$. Since it is the resolvent of $C_{n+1}$ and $R_n$, $NG_2$ contains all but one literals from each of these nogoods, so they are unit. Let $r(C_{n+1}, R_n) = x$, $C_{n+1} \setminus NG_2 = \{x\}$ and $R_n \setminus NG_2 = \{\overline{x}\}$. Since $C_{n+1} \in \mathcal{N}$ and $C_{n+1}$ is unit, unit propagation on $C_{n+1}$ will force the literal $x$ to false. Therefore, all the literals in $R_n = R(C_n, \ldots, R(C_1, NG_1))$ are true and $R_n$ is true, therefore $NG_1$ is true because of the inductive hypothesis.

We show now that there exists a branch where $NG_1$ is true but $NG_2$ is not. Let $C_1 = \{X_1 \not\in a, X_2 \not\in b, X_3 \not\in c, X_4 \not\in d\}$, $C_2 = \{X_1 \leftarrow a, X_5 \not\in e, X_3 \not\in c\}$, $C_3 = \{X_3 \leftarrow c, X_6 \not\in f, X_7 \not\in h\}$, $C_4 = \{X_2 \leftarrow b, X_6 \not\in f, X_7 \not\in h\}$. Let $X_4 \leftarrow d$, $X_5 \leftarrow e$, and $X_7 \leftarrow h$ be pruned at earlier decision levels and the decision at the current level is $X_6 \leftarrow f$.

After this decision, $C_4, C_3$ and $C_2$ are unit and force $X_2 \not\in b$, $X_3 \not\in c$ and $X_1 \not\in a$, respectively. These unit propagations make $C_1$ true, making it a conflict. The trail at the decision level at the time of the conflict is $\{X_6 \leftarrow f, X_2 \not\in b, X_3 \not\in c, X_1 \not\in a\}$.

Now, let us use resolve-unwind-trail to derive the nogoods $NG_1$ and $NG_2$. The conflict nogood is $N_g = C_1$. We stop after 1 iteration of the loop to get $NG_1$ and after 3 iterations to get $NG_2$. Thus, $NG_1 = R(C_2, N_g) = \{X_2 \not\in b, X_3 \not\in c, X_4 \not\in d, X_5 \not\in e\}$ and $NG_2 = R(C_4, R(C_3, NG_1))) = \{X_4 \not\in d, X_5 \not\in e, X_6 \leftarrow f, X_7 \not\in h\}$. Trivially, the branch $A_1 = \{X_2 \not\in b, X_3 \not\in c, X_4 \not\in d, X_5 \not\in e\}$ makes $NG_1$ true, but not $NG_2$. We also verify that $NG_2$ is not made true after unit propagation. Unit propagation of $C_1$ forces the assignment $X_1 \leftarrow a$, while unit propagation of $C_2$ forces $X_1 \not\in a$. $C_3$ is false because it contains $X_3 \leftarrow c$, while the branch contains $X_3 \not\in c$ and $C_4$ is false because it contains $X_2 \leftarrow b$ while the branch contains $X_2 \not\in b$. $NG_2$ remains non-true and non-unit, as the assignments $X_6 \leftarrow f$ and $X_7 \leftarrow h$ are not set either way. Note that, inevitably, by the first item of this theorem, any branch that makes $NG_1$ true will also make $C_1$ true, which is the case in this example.

Finally, we show the incomparability of unit propagation on $NG_1$ and $NG_2$. We use the same example. The branch $A_1 - \{X_2 \not\in b\}$ clearly makes $NG_1$ unit, while $NG_2$ is not unit and does not become unit after unit propagation. On the other hand, the branch $A_2 = \{X_4 \not\in d, X_5 \not\in e, X_6 \leftarrow f\}$ makes $NG_2$ unit but not $NG_1$. After unit propagation, the assignments $X_7 \leftarrow h$ and $X_1 \not\in a$ are forced because of $NG_2$ and $C_2$, respectively, but none of the other nogoods become unit.

This proposition allows us to deduce the following.

**Corollary 3.2.3.** Consider a CSP $\mathcal{P} = \langle V, D, C, \mathcal{N}\rangle$, where $\mathcal{N}$ is a database of nogoods
entailed by $C$, and a partial assignment $A$ that causes a conflict after constraint propagation and unit propagation, while no subset of $A$ satisfies this condition. Consider now two nogoods $NG_{CBJ}$ and $NG_{1D}$ such that $NG_{CBJ}$ is derived from the conflict by the CBJ scheme and $NG_{1D}$ is derived from the conflict by the first-decision scheme. Let $P_{CBJ} = \langle V, D, C, N \cup \{NG_{CBJ}\} \rangle$ and $P_{1D} = \langle V, D, C, N \cup \{NG_{1D}\} \rangle$. Then

- Every branch that is a conflict for $P_{CBJ}$, is also a conflict for $P_{1D}$ after constraint and unit propagation, but there exist branches where the converse is not true.

- $P_{CBJ}$ and $P_{1D}$ are incomparable with respect to propagation. There exist branches where some values are pruned in $P_{CBJ}$ but not in $P_{1D}$ and branches where some values are pruned in $P_{1D}$ but not in $P_{CBJ}$.

This is a direct application of proposition 3.2.2 by observing that $NG_{CBJ}$ can be derived by applying more resolution steps after deriving $NG_{1D}$ so that no non-assignments are left. Since $NG_{CBJ}$ learns s-nogoods only, this shows that g-nogoods are locally more powerful than s-nogoods.

Note that this result only deals with a static context and examines how many branches can potentially be pruned by each nogood. One of the consequences of this is that the nogood database $N$ is fixed. Within the context of a backtracking search algorithm, however, the nogood database is not fixed. In order to use this result in this context, we need to assume that no nogood is ever forgotten. This is not a realistic assumption, as in practice it is likely that for hard problems the available system memory will eventually be exhausted if we never forget learned nogoods. Besides, the cost of unit propagating these nogoods will become so great that the search will stop making significant progress. Still, empirical results (section 3.4) suggest that learning first-decision or 1-UIP g-nogoods is more effective than learning s-nogoods.

Finally, even though this establishes an ordering among the nogoods that we may learn from a conflict with respect to their pruning power, we cannot deduce that the ordering also applies to their usefulness in practice. Their incomparability with regard to unit propagation leads us to believe that it is a matter of empirical study to determine which nogood learning scheme is better in practice. However, we will show in the next section that learning generalized nogoods is more effective than learning standard nogoods, regardless of the amount of pruning generated by each of the the individual learned nogoods.
3.2.3 Global power

First consider the family of CSP backtracking algorithms $\mathcal{F}$ that perform any combination of the standard techniques for constraint propagation (e.g., maintaining arc consistency), intelligent backtracking (e.g., conflict directed backjumping), dynamic variable ordering, and dynamic value ordering. We require that an algorithm in $\mathcal{F}$ performs at least forward checking. $\mathcal{F}$ includes such standard algorithms as MAC, FC, FC-CBJ, etc., with any strategy for variable and value ordering. On the weaker side of propagation, $\mathcal{F}$ does not include standard backtracking search without propagation. On the stronger side of propagation, $\mathcal{F}$ does not include the use of special purpose propagators, since without restrictions propagators can add arbitrary power to the algorithm.  

Now add to $\mathcal{F}$ the ability to discover, store, and unit propagate standard nogoods; call this new family of algorithms $\mathcal{F}^{NG}$s. In contrast, consider adding to $\mathcal{F}$ the ability to discover, store and unit propagate generalized nogoods; call this new family $\mathcal{F}^{NG}$g.

Observe first that algorithms in $\mathcal{F}^{NG}$s and $\mathcal{F}^{NG}$g can produce non-tree resolution refutations, while that is impossible for any algorithm in $\mathcal{F}$. This suggests we can observe an exponential separation between $\mathcal{F}^{NG}$s and $\mathcal{F}$ [Hwang and Mitchell, 2005]. This is confirmed in the empirical results in section 3.4. In addition, we can separate $\mathcal{F}^{NG}$g from $\mathcal{F}^{NG}$s, even for a relatively weak algorithm in $\mathcal{F}^{NG}$g in terms of constraint propagation.

Proposition 3.2.4. There exists an infinite family of CSP problems of increasing size $n$ on which any algorithm in $\mathcal{F}^{NG}$s takes super-polynomial time (specifically $n^{\Omega(\log n)}$). In contrast, there exists an algorithm $A_{FC}^{NG}$s $\in \mathcal{F}^{NG}$s, which performs forward checking, with a particular variable and value ordering and a simple deterministic strategy for learning g-nogoods, and can solve problems in this family in time $O(n^3)$.

Proof. There exists an infinite family of SAT instances $\text{MPH}_n$ (for modified pigeonhole principle), first used in [Goerdt, 1992] to prove the superpolynomial separation of general resolution from negative resolution. The same family was then changed into an equivalent CSP family in [Mitchell, 2003] and used to show the superpolynomial separation of 2-way branching from d-way branching. Here, we will use $\text{MPH}_n$ to refer to the family of CSP instances. There are $n$ variables, $V_0$ to $V_{n-1}$, each with domains $D(V_i) = \{1, \ldots, m\}$. $\text{MPH}_n$ is only defined for $n$ such that $m = \lfloor \log n \rfloor$ is even. Define the constraint

\footnote{Specifically, we disallow propagators that can determine in time polynomial in the arity of a constraint $C$ that $C$ cannot be satisfied, when the shortest resolution proof of unsatisfiability of $C$ is exponential in the arity of $C$.}
Chapter 3. Generalized nogoods

\[ R_r(X, Y) = X \neq r \lor Y \neq r. \] The constraints of MPH\(_n\) are

\[
C_{\text{MPH}_n} = \bigcup_{r \in \{1, \ldots, m\}} \left\{ \begin{array}{l}
R_r(V_i, V_j) : \\
k \in \{b2^r : 0 \leq b \leq n/(2^r)\}, \\
k \leq i < k + 2^{r-1}, \\
k + 2^{r-1} \leq j < k + 2^r
\end{array} \right\}
\] (3.1)

These constraints state that for each pair of variables \(\{V_0, V_1\}, \ldots, \{V_{n-2}, V_{n-1}\}\), only one can take the value 0. For each quadruple \(\{V_0, V_1, V_2, V_3\}, \ldots, \{V_{n-4}, V_{n-3}, V_{n-2}, V_{n-1}\}\), if any of \(V_0, V_1\) gets the value 1, then 1 cannot be assigned to either \(V_2\) or \(V_3\) and vice versa. This pattern is repeated \(m\) times, with the last one being that if any of \(V_0, \ldots, V_{n/2-1}\) is assigned the value \(m\), then \(m\) cannot be assigned to any of \(V_{n/2}, \ldots, V_{n-1}\). We can see that any instance of MPH\(_n\) is unsatisfiable. Moreover, \(\text{MPH}_n(V_0, \ldots, V_{n-1}) = \text{MPH}_{n/2}(V_0, \ldots, V_{n/2-1}) \cup \text{MPH}_{n/2}(V_{n/2}, \ldots, V_{n-1}) \cup \{R_m(V_i, V_j) : 0 \leq i \leq n/2 - 1, n/2 \leq j \leq n - 1\}\).

In [Mitchell, 2003], it is shown that any algorithm in \(F\) that only performs s-nogood learning can be \(p\)-simulated by negative resolution. That is, the search tree produced by a backtracking search algorithm that learns s-nogoods can be easily transformed into a negative-resolution proof that is at most polynomially bigger than the search tree. Since the size of the shortest negative resolution refutation of MPH\(_n\) is superpolynomial in \(n\) (shown in [Goerdt, 1992]), it follows that backtracking search with s-nogood learning also needs at least \(n^{\Omega(\log n)}\) time to solve these instances.

It is also shown [Mitchell, 2003] that MPH\(_n\) can be solved with 2-way branching in polynomial time using a simple branching and learning strategy. That proof makes use of g-nogoods. We show here that the key to Mitchell’s result is not 2-way branching, but rather the ability to learn g-nogoods. Thus, we are able to show the stronger result that there exists a g-nogood learning scheme and k-way branching strategy which solves instances of MPH\(_n\) in polynomial time. We use the following learning scheme: we discover all nogoods but record only those that consist entirely of non-assignments or that have size 1. That is, we run a modified version of the procedure resolve-unwind-trail. This procedure always runs until all assignments in NewNG are decision assignments. However, at each iteration, we record the intermediate nogood NewNG if it consists entirely of non-assignments or has size 1. We use the following branching strategy. If a variable \(V\) has a value \(d\) in its domain that is unconstrained (it would not cause any domain reduc-
tions), we ignore $V$ until all remaining variables are unconstrained in this way. Among the rest of the variables, we identify a maximally constrained variable/value pair $V \leftarrow x$ (i.e., one that would cause the maximum number of value prunings when assigned) and choose $V$ as the next variable. Ties between two variables are broken by considering the second most constrained value and so on, then lexicographically if all values of both variables are equally constrained. The values of $V$ are tried in order of decreasing constrainedness, with ties between values broken by choosing the lexicographically greater one.

We can now verify that the algorithm will prove the unsatisfiability of the instance in polynomial time. The search is shown in figure 3.13. At the root, all values of all variables are equally constrained, so the branching heuristic will lexicographically choose $V_0$, and instantiate it to $m$. Forward checking will prune the value $m$ from the variables $V_{n/2} \ldots V_{n-1}$ thus creating the instance $\text{MPH}_{n/2}(V_{n/2} \ldots V_{n-1})$, which is unsatisfiable. The variables $V_1, \ldots, V_{n/2-1}$ will be unconstrained, since they can all be assigned the value $m$. We can see that the nogood $N_1 = \{V_{n/2} \not= m, \ldots, V_{n-1} \not= m\}$ will be discovered after proving the unsatisfiability of the instance $\text{MPH}_{n/2}(V_{n/2} \ldots V_{n-1})$. To see why this is true, consider the set of non-assignments that are made after the assignment $V_0 \leftarrow m$. They are $V_0 \not= 1, \ldots, V_0 \not= m-1$ and $V_{n/2} \not= m, \ldots, V_{n-1} \not= m$. Any nogood must contain the assignments $V_{n/2} \not= m, \ldots, V_{n-1} \not= m$, otherwise the instance $\text{MPH}_{n/2}(V_{n/2} \ldots V_{n-1})$ would not be created and the subtree would contain a solution. None of the non-assignments $V_0 \not= 1, \ldots, V_0 \not= m-1$ participate in any constraint propagation either at the root or in deeper levels, thus they do not appear as labels of any other prunings and the mechanism of resolve-unwind-trail will not examine them. Thus, the nogood $\{V_{n/2} \not= m, \ldots, V_{n-1} \not= m\}$ must be discovered.

Now the algorithm will try $V_0 \leftarrow m-1$, which is the next most constrained value of $V_0$. This will prune $m-1$ from the domains of $V_{n/4}, \ldots, V_{n/2-1}$. Now the value $m$ of variables $V_1 \ldots V_{n/2-1}$ is the most constrained. However, the value $m-1$ of variables $V_1, \ldots, V_{n/4-1}$ is unconstrained, so these variables are ignored. The heuristic choice among $V_{n/4}, \ldots, V_{n/2-1}$ will be decided by the lexicographic ordering of these variables, so the next variable chosen will be $V_{n/4}$. When the assignment $V_{n/4} \leftarrow m$ is made, the constraints $V_{n/4} \not= m \lor V_i \not= m, n/2 \leq i \leq n-1$ will make the non-assignments $V_{n/2} \not= m, \ldots, V_{n-1} \not= m$, so the nogood $\{V_{n/2} \not= m, \ldots, V_{n-1} \not= m\}$ will be-

---

3If all remaining variables have an unconstrained value, it would imply that we have found a solution, so this never actually happens.
Figure 3.13 Proving unsatisfiability of $\text{MPH}_n$ using g-nogood learning. Gray nodes are branching nodes, while white nodes are leaves, labeled with the nogood that they make true. Gray nodes are labeled with the nogood that is discovered after proving unsatisfiability of the subproblem. Edges between nodes are labeled with the corresponding decision assignment. The edges off the node $\text{MPH}(V_{n/4}, \ldots, V_{n/2-1})$ indicate that the conflicts resulting from $V_{n/4} \leftarrow m, \ldots, V_{n/2-1} \leftarrow m$ are discovered using $N_1$ in parallel and independently from the refutation of $\text{MPH}(V_{n/4}, \ldots, V_{n/2-1})$.

The discovery of the nogoods $\{V_{n/4} \leftarrow m\}, \ldots, \{V_{n/2-1} \leftarrow m\}$ happens in parallel with the solving of $\text{MPH}_{n/4}(V_{n/4}, \ldots, V_{n/2-1})$, but it does not interfere with it. The reason is first that the discovery of these nogoods does not require exploring of a subtree, thus the cost of this discovery is bounded by the cost of constraint propagation. Second, all the values $V_{n/4} \leftarrow m \ldots V_{n/2-1} \leftarrow m$ are equally constrained, so they do not affect the branching heuristic.

During the proof of unsatisfiability of $\text{MPH}_{n/4}(V_{n/4}, \ldots, V_{n/2-1})$, we can see that the nogood $N_2 = \{V_{n/4} \neq m-1, \ldots, V_{n/2-1} \neq m-1\}$ will be discovered. In general, after the assignment $V_1 \leftarrow m-i+1$ is made, the nogood $N_i = \{V_{n/2^i} \neq m-i+1, \ldots, V_{n/2^i-1} \neq m-i+1\}$ will be discovered. Then, when the assignment $V_1 \leftarrow m-i+2$ is made, $N_i$ will be used to discover the nogoods $\{V_i \leftarrow m-i+1\}, 1 \leq i \leq n/2^i-1$ and so permanently remove the corresponding values.
from consideration. This means that a smaller instance of MPH is created, leading to
the discovery of \( N_{i+1} \). Eventually, the values of \( V_1 \) will be exhausted and the proof of
unsatisfiability concludes.

Let us examine the complexity of the algorithm. We saw that to solve an MPH\(_n\)
instance, the algorithm solves an MPH\(_{n/2}\) instance, an MPH\(_{n/4}\) instance and so on.
Accounting for the computation of the branching heuristic, which is \( O(nm) < O(n^2)\),
and forward checking \( n^2 \) binary constraints (in constant time for each of these simple
constraints, therefore \( O(n^2) \) in total), the time complexity of the algorithm is
\( S(n) = S(n/2) + S(n/4) + \ldots + S(1) + O(n^2) \leq O(n^2) \). \( \square \)

Note that despite its being less flexible, d-way branching solves the instances with
no worse complexity than 2-way branching. The question of whether there exist in-
stances where the additional flexibility afforded by 2-way branching leads to improved
performance remains open.

It has been shown that unrestricted resolution is actually exponentially separated
from negative resolution [Buresh-Oppenheim and Pitassi, 2003]. However, it is not clear
whether this stronger separation can be translated to a separation between s-nogood
learning and g-nogood learning.

It is known that nogood learning can yield significant savings in terms of the size of the
explored search tree (for example, [Bayardo Jr and Miranker, 1996, Dechter, 1986]). This
result shows that learning g-nogoods can yield these gains in more cases than s-nogoods,
showing a super-polynomial improvement over standard nogoods (since s-nogoods are a
special form of g-nogoods, g-nogoods can always simulate the performance of s-nogoods).

It should be pointed out, however, that there exists a paradox of sorts when com-
paring two nogood learning algorithms in practice (including algorithms that only use
nogoods for backjumping). This has to do with the interaction between nogood learn-
ing and variable and value ordering heuristics. Let \( A_1 \) and \( A_2 \) be two nogood learning
algorithms, with \( A_1 \) being more powerful than \( A_2 \) both locally and globally, in the sense
of propositions 3.2.2 and 3.2.4. Assume that we use \( A_1 \) and \( A_2 \) to solve a problem \( P \).
Even though \( A_2 \) is weaker than \( A_1 \), it may still end up exploring a smaller search tree
than \( A_1 \) to solve \( P \). We present here an informal argument for why this may happen.
Assume that at some point during search, these algorithms visit the same part of the
search space. \( A_1 \) is more powerful, so it may quickly discover nogoods that allow it to
stop exploring that part of the search space. \( A_2 \) on the other hand keeps exploring and
may find a nogood that allows it to stop exploring an even bigger part of the search space. If this pattern occurs often enough, $A_2$ ends up solving the problem faster (in terms of the size of the search tree) than $A_1$.

### 3.3 Related work

Nogoods were first introduced in truth maintenance systems [Stallman and Sussman, 1977], in a somewhat ad-hoc manner. The term was defined more formally in ATMSs (Assumption-based Truth Maintenance Systems) (see, for example, [Forbus and de Kleer, 1993]) and the connection was made to clauses. The usage of the term nogood in ATMS is only used to separate the original formula clauses from clauses learned later on. Therefore ATMS nogoods are equivalent to clauses. However, ATMS operate on problems defined in terms of Boolean variables only. It is when we have multi-valued variables that there exists a real distinction between clauses (g-nogoods) and nogoods (s-nogoods).

Nogoods were introduced in the CSP literature by Dechter [Dechter, 1986]. Dechter’s definition is what we refer to as standard nogoods in this thesis. Their relationship to generalized nogoods was discussed extensively in this chapter.

The generalization proposed here is equivalent to the one proposed in [Focacci and Milano, 2001], even though they are syntactically distinct. The authors of that paper investigated generalized nogoods only in the context of symmetry breaking. The results we obtain here are disjoint from their previous results.

Mitchell [Mitchell, 2003] also proposed the same generalization independently and showed the first theoretical results regarding this generalization. However, that work was limited to 2-way branching and contained no empirical results.

Finally, Hawkins and Stuckey [Hawkins and Stuckey, 2006] have investigated integrating a SAT solver with a BDD based constraint solver. Although they investigate the empirical aspect of integrating a constraint solver with the nogood reasoning performed by a SAT solver, they provide no theoretical results and, more importantly, their study is limited to constraints represented as BDDs. An extension of this work and of the published results of this thesis proposes an alternate encoding of variable domains such that the bounds of a variable can be captured in a clause using two literals only (one for the lower bound and one for the upper bound), resulting in potentially much smaller clauses in, for example, problem domains that use arithmetic constraints [Ohrimenko et al., 2007].
3.4 Empirical results

We present now the results of an empirical evaluation of the algorithms described above. All experiments were performed using EFC, the solver that was developed for this thesis [Katsirelos, 2004], on a 2.4GHz Pentium 4 system with 6GB of RAM. All instances were run for a maximum of 20000 seconds. If the solver has not reached a decision within that time limit, the instance is reported as unsolved.

In the following discussion, we use the following terminology to describe algorithms:

- **FCCBJ**: forward checking with conflict directed backjumping, as shown in figure 2.4.
- **FCCBJ+S**: forward checking with standard nogood learning, as shown in figure 2.10.
- **FCCBJ+G**: forward checking with generalized nogood learning (figure 3.5), using the first-decision learning scheme (figure 3.8).
- **FCUIP**: forward checking with generalized nogood learning (figure 3.5), using the 1-UIP learning scheme (figure 3.11).

We performed experiments on hard crossword problems. We chose crossword puzzles, as this is a constraint problem that has been used in the past [Beacham et al., 2001, Ginsberg et al., 1990]. It is hoped that the performance of the g-nogood learning on these problems is indicative of its performance in other domains. However, since forward checking is not used in practice [Bessière and Régin, 1996], we defer a more extensive evaluation until the next chapter, when we study integration of g-nogood learning with algorithms that enforce a stronger level of consistency.

The authors of [Beacham et al., 2001] performed an extensive study of models, algorithms and DVO heuristics on a set of 50 crossword puzzles, using two different dictionaries: “UK” and “words”, resulting in 100 instances. We compare the four algorithms described above using the combination of model and DVO heuristic which proved the most effective for forward-checking based algorithms in that study. The model uses one variable for each letter in the puzzle and posts a constraint for each slot of the puzzle that the letters form a word in the corresponding dictionary. For any two slots of the same length, a constraint is posted that requires that the slots get different words. The DVO used is dom+deg.

We first present in figure 3.14 the CPU time and nodes visited to solve some interesting instances. Each row in this table contains results for a specific instance. The first column
Figure 3.14 CPU time and nodes visited to solve crossword puzzle instances. A - indicates that the timeout of 20000 seconds expired before the solver produced an answer.

<table>
<thead>
<tr>
<th>Instance</th>
<th>FCCBJ+S</th>
<th>FCCBJ+G</th>
<th>FCUIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Nodes</td>
<td>Time</td>
</tr>
<tr>
<td>UK-21.01</td>
<td>1.6</td>
<td>11664</td>
<td>15.27</td>
</tr>
<tr>
<td>UK-21.04</td>
<td>1491.22</td>
<td>1170356</td>
<td>-</td>
</tr>
<tr>
<td>UK-21.10</td>
<td>278.42</td>
<td>463057</td>
<td>367.41</td>
</tr>
<tr>
<td>UK-23.01</td>
<td>7.75</td>
<td>51375</td>
<td>34.15</td>
</tr>
<tr>
<td>UK-23.03</td>
<td>0.56</td>
<td>3718</td>
<td>0.68</td>
</tr>
<tr>
<td>UK-23.06</td>
<td>5203.07</td>
<td>2994762</td>
<td>-</td>
</tr>
<tr>
<td>UK-23.10</td>
<td>134.59</td>
<td>415718</td>
<td>2199.28</td>
</tr>
<tr>
<td>words-15.01</td>
<td>5395.93</td>
<td>2423058</td>
<td>-</td>
</tr>
<tr>
<td>words-15.04</td>
<td>276.52</td>
<td>797739</td>
<td>1781.08</td>
</tr>
<tr>
<td>words-15.06</td>
<td>43</td>
<td>229426</td>
<td>138.99</td>
</tr>
<tr>
<td>words-15.07</td>
<td>100.68</td>
<td>429882</td>
<td>603.5</td>
</tr>
<tr>
<td>words-15.10</td>
<td>5752.77</td>
<td>3702947</td>
<td>-</td>
</tr>
<tr>
<td>words-19.03</td>
<td>156.56</td>
<td>526797</td>
<td>-</td>
</tr>
<tr>
<td>words-19.04</td>
<td>15.46</td>
<td>118325</td>
<td>43.47</td>
</tr>
<tr>
<td>words-21.01</td>
<td>3921.84</td>
<td>5056685</td>
<td>-</td>
</tr>
<tr>
<td>words-21.06</td>
<td>45.11</td>
<td>168548</td>
<td>350.27</td>
</tr>
<tr>
<td>words-21.07</td>
<td>32.69</td>
<td>149151</td>
<td>173.98</td>
</tr>
<tr>
<td>words-23.03</td>
<td>144.57</td>
<td>672178</td>
<td>5006</td>
</tr>
<tr>
<td>words-23.04</td>
<td>14715.87</td>
<td>6194196</td>
<td>-</td>
</tr>
<tr>
<td>words-23.07</td>
<td>99.26</td>
<td>293758</td>
<td>1075.12</td>
</tr>
<tr>
<td>words-23.08</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

contains the name of the instance, which contains the name of the dictionary used (“UK” or “words”), the size of the crossword layout (5,10,15,21 or 23 on each side) and the name of the layout of that specific size (a number from 1 to 10). The 2nd, 4th and 6th columns show the CPU time required to solve this instance using FCCBJ+S, FCCBJ+G and FCUIP, respectively. The time of the algorithm which solved the instances faster is emboldened. Similarly, the 3rd, 5th and 7th columns show the nodes visited by the same algorithms. We do not show results for FCCBJ, as it was not at all competitive in these instances.

We note first that these results confirm our earlier observations. In most cases, learning generalized nogoods pays off in terms of the size of the explored search tree. In most of the instances, learning generalized nogoods leads to a smaller search tree. This is most evident in the case of FCUIP. In many cases, however, FCCBJ+S explores
a smaller search tree, a result that seems to contradict our observations. This can be attributed to two heuristic factors. The first is simply the fact that all algorithms use a dynamic variable ordering. Differences in the amount of propagation between s-nogood learning and g-nogood learning can lead the DVO heuristic to make different choices, which may guide the search towards a solution sooner or towards a subtree that contains useful nogoods. The other factor is that even when the DVO does not affect the size of the search tree significantly, the added local power of g-nogoods may make the solver avoid searching a subtree which contains useful nogoods. Thus, by being less powerful locally, FCCBJ+S may visit parts of the search space which allow it to learn more useful nogoods.

Figure 3.15 Number of Nogoods learned and average number of times a nogood became Unit for a few crossword puzzle instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>FCCBJ+S</th>
<th>FCCBJ+G</th>
<th>FCUIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N  U</td>
<td>N  U</td>
<td>N  U</td>
</tr>
<tr>
<td>UK-21.04</td>
<td>5950 5.04</td>
<td>5603 25.7</td>
<td>3823 31.3</td>
</tr>
<tr>
<td>UK-23.06</td>
<td>1600 2.96</td>
<td>1002 18.6</td>
<td>3742 16.9</td>
</tr>
<tr>
<td>words-15.01</td>
<td>1353 4.05</td>
<td>7143 24.3</td>
<td>2624 26.7</td>
</tr>
<tr>
<td>words-19.03</td>
<td>2783 1.83</td>
<td>1006 19.3</td>
<td>2113 20.2</td>
</tr>
<tr>
<td>words-21.01</td>
<td>2412 1.57</td>
<td>1664 15.8</td>
<td>4489 18.7</td>
</tr>
</tbody>
</table>

We can also verify our claims regarding the local power of nogoods. In figure 3.15 we show the number of recorded nogoods and the average number of times a nogood became unit or true over all the nogoods learned during the execution of the algorithm for a specific instance. We see that in general fewer g-nogoods are discovered, but each of them becomes unit more often. Thus, g-nogoods are responsible for more propagation than s-nogoods.

On the other hand, the results do not show improvement with regard to the CPU time needed to solve these instances. Even in the instances where these exists a clear reduction in the size of the search tree, FCCBJ+G and FCUIP are sometimes slower than FCCBJ+S in terms of nodes visited per second. For example, in the instance UK-21.04 FCUIP visits approximately 30% fewer nodes than FCCBJ+S but needs 3.5 times as much CPU time to find a solution. Thus, the reduction of the search tree has to be significant for g-nogood learning to pay off in CPU time, which still happens is some cases. The

---

We use the term useful nogood in an informal sense, meaning a nogood whose presence helps the solver explore a smaller search space.
reason for the slowdown is that g-nogoods tend to be longer than s-nogoods. This is illustrated in figure 3.16. In it, we show the average nogood size, the average number of times the watched literals of a nogood were made true and the number of nodes visited per second for each of the three algorithms. We see that there exists a clear inverse correlation between the average size of nogoods and the number of nodes visited per second. The explanation for this lies in the second metric presented, which is the average number of times that a nogood has to be examined by the unit propagation algorithm, i.e., a watched assignment has become true. Although the performance penalty paid for storing larger nogoods is much smaller than it would be without using the watched literal technique, it is still measurable.

Figure 3.16 Average nogood size, number of nogood Checks per nogood and Nodes visited per second for a few crossword puzzle instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>FCCBJ+S A</th>
<th>FCCBJ+S C</th>
<th>FCCBJ+S N</th>
<th>FCCBJ+G A</th>
<th>FCCBJ+G C</th>
<th>FCCBJ+G N</th>
<th>FCUIP A</th>
<th>FCUIP C</th>
<th>FCUIP N</th>
</tr>
</thead>
<tbody>
<tr>
<td>UK-21.04</td>
<td>20.94</td>
<td>3476.55</td>
<td>784</td>
<td>101.68</td>
<td>4912.23</td>
<td>63</td>
<td>79.36</td>
<td>1845.01</td>
<td>155</td>
</tr>
<tr>
<td>UK-23.06</td>
<td>16.4</td>
<td>795.15</td>
<td>575</td>
<td>104.39</td>
<td>2892.85</td>
<td>106</td>
<td>87.29</td>
<td>3496.44</td>
<td>812</td>
</tr>
<tr>
<td>words-15.01</td>
<td>49.94</td>
<td>3619.44</td>
<td>449</td>
<td>128.51</td>
<td>2328.78</td>
<td>80</td>
<td>44.58</td>
<td>93.27</td>
<td>584</td>
</tr>
<tr>
<td>words-19.03</td>
<td>38.4</td>
<td>1537.45</td>
<td>3376</td>
<td>105.69</td>
<td>1114.19</td>
<td>104</td>
<td>58.26</td>
<td>611.02</td>
<td>984</td>
</tr>
<tr>
<td>words-21.01</td>
<td>24.94</td>
<td>408.07</td>
<td>1289</td>
<td>117.10</td>
<td>1910.97</td>
<td>202</td>
<td>105.61</td>
<td>3081.86</td>
<td>953</td>
</tr>
</tbody>
</table>
Chapter 4

Nogoods in constraint propagation

Global constraints, i.e., constraints of large arity, are common in real CSPs. To deal with such constraints the field has developed the powerful notion of propagators. These are special purpose algorithms for efficiently enforcing local consistency, typically GAC, on global constraints. We show here that integrating nogood learning with these special purpose algorithms presents some challenges and we present ways to address these challenges.

4.1 The problem

Recall that in the algorithm FCCBJ (figure 2.4), forward checking must label each value $V \leftarrow x$ that it prunes with a nogood which justifies the pruning. Forward checking generates base nogoods that directly violate a constraint. When all values of a variable $V$ have been pruned at level $l$, we can jump back to a previous level $k$. The reason that we are able to jump back is captured in a new nogood which is derived by unioning together the nogoods that have been used as labels for the values of $V$ minus the values of $V$. This nogood is also used to prune and label the value assigned at level $k$. The algorithm depends on forward checking labeling each pruned value $V \leftarrow x$ with a nogood, otherwise it cannot compute a reason why the search can backtrack.

Algorithms that employ more powerful propagation methods cannot label pruned values in the same way that forward checking can. The reason is that in most cases no direct violation of a constraint can be singled out as the reason for that pruning. Consider the example of maintaining generalized arc consistency (GAC). Maintaining GAC works as follows: all constraints are initially put in a queue. For each constraint $C$, all values
of the constrained variables are examined. If a value \( V \leftarrow x \) is not supported, that is if no unpruned tuple exists that satisfies \( C \) and includes \( V \leftarrow x \), it is removed from the domain of \( V \) and then all constraints that involve \( V \), except \( C \), are placed on the queue. The process continues until the queue is empty. When the search branches on a new assignment \( V \leftarrow y \), all other values of \( V \) are pruned so that constraints are placed on the queue. Note that values can be removed without there being a direct constraint violation. For example, consider the constraint \( C = X < Y \), where the domain of \( X \) is \( \{0, \ldots , 10\} \) and the domain of \( Y \) is \( \{0, \ldots , 5\} \). Enforcing GAC on this constraint will remove the values \( 5, \ldots , 10 \) from the domain of \( X \), even when no assignment has been made to \( Y \). Thus, no base nogood exists that directly violates \( C \) to justify these prunings.

In order to integrate methods such as maintaining GAC with nogood learning techniques (including CBJ), we need to develop techniques that can label every value \( V \leftarrow x \) with a nogood containing \( V \leftarrow x \). We write \( \text{Reason}(V \leftarrow x, C) \) for this nogood, but only write \( \text{Reason}(V \leftarrow x) \) when we do not care which constraint generated it or if it is clear from the context.

The standard technique for learning nogoods from GAC learns nogoods containing only assignments (s-nogoods). It works as follows. GAC is established before the search begins. Each value \( V \leftarrow x \) that is pruned during this phase is inconsistent with the problem and the nogood used to label it is \( \{V \leftarrow x\} \).\(^1\) After that, whenever a new assignment \( X \leftarrow a \) is made, GAC is established again for all constraints that involve \( X \). When a value \( Y \leftarrow b \) is pruned while establishing GAC for constraint \( C \), the nogood used to label \( Y \leftarrow b \) is computed by the algorithm in figure 4.1. We write \( \text{Reason}_d(Y \leftarrow b, C) \) for the nogood produced by this algorithm. The algorithm starts with the set \( \{Y \leftarrow b\} \). It then unions in the set of assignments made to other variables in the scope of the constraint in lines 2–3. Finally, in lines 4–6, for each pruned value \( Z \leftarrow i \) such that \( Z \neq Y \) is an unassigned variable in the scope of the constraint, it unions in \( \text{Reason}(Z \leftarrow i) - \{Z \leftarrow i\} \). Note that when all but one of the variables of the constraint are assigned, the behaviour of GAC is identical to the behaviour of FC and the nogoods computed by the algorithm in figure 4.1 are base nogoods and are in fact the same ones that would be computed by forward checking.

To illustrate the algorithm, consider the constraint \( C = X + Y < Z \), with the domains of \( X, Y \) and \( Z \) all being non-negative. Now suppose that the values \( Y \leftarrow 0 \)

\(^1\)In fact these values are simply removed from their domains and not labeled at all, as these labels are not used anywhere.
Figure 4.1 Computing s-nogood labels for values pruned by GAC

```plaintext
int GAC-CBJ-Nogood(C, Y ← b)
1. \( N_g = \{ Y ← b \} \)
2. foreach \( Z \in \text{scope}(C) \) s.t. \( Z \) is assigned value \( i \) : \( N_g = N_g \cup \{ Z ← i \} \)
3. foreach \( Z \in \text{scope}(C) − \{ Y \} \) s.t. \( Z \) is unassigned, \( i ∈ \text{Dom}(Z) − \text{CurDom}(Z) \):
4. \( N_g = N_g \cup (\text{Reason}(Z ← i) − \{ Z ← i \}) \)
7. return \( N_g \)
```

and \( Y ← 1 \) have been pruned and labeled with the nogoods \( \{ V_1 ← 1, V_2 ← 2, Y ← 0 \} \) and \( \{ V_3 ← 1, V_4 ← 2, Y ← 1 \} \), respectively, and that the search branches on the assignment \( X ← 0 \). When GAC is established for \( C \), the values 0, 1 and 2 will be pruned from the domain of \( Z \). Let us now construct the nogood used to label \( Z ← 0 \). It will consist of the assignment \( X ← 0 \) unioned with the pruned value \( Z ← 0 \) unioned with the nogoods used to label the prunings of the values of \( Y \) minus these values themselves: \( \{ V_1 ← 1, V_2 ← 2, V_3 ← 1, V_4 ← 2 \} \). The final nogood will then be \( \{ V_1 ← 1, V_2 ← 2, V_3 ← 1, V_4 ← 2, X ← 0, Z ← 0 \} \). This technique has been proposed by [Chen, 2000] and it was shown to generate nogoods, that is sets of assignments that cannot be part of a solution.

These nogoods, just like the constraint violating nogoods used in FCCBJ+S, are by themselves not useful. They cannot prune any search paths that would not be pruned by GAC. The initial nogood labeling a GAC pruning simply captures the conditions under which GAC pruned the value: under the same conditions GAC would perform the same pruning. The power of nogoods comes from learning new nogoods. These new nogoods move us beyond the power of the base level of constraint propagation by capturing subsets of the prefix that were not directly pruned by propagation but instead required some search to refute.

The nogoods generated by \textbf{GAC-CBJ-Nogood} are the result of unioning together several other nogoods. For values that are pruned in shallow levels of the search tree, the number of nogoods that are unioned together is likely to be small, simply as a result of the fact that few values have been pruned. However, as the search reaches deeper levels of the search tree, more values are pruned, so the loop in lines 4–6 of \textbf{GAC-CBJ-Nogood} unions together more nogoods. Because each pruned value of the other variables may be pruned by different constraints, this process may introduce many variables in \( N_g \) that
are unrelated to \( C \), thus the nogoods generated in deeper levels tend to be larger.

Nogoods that contain all of the current decision assignments are called saturated. Saturated nogoods are not useful either for backtracking or for learning new nogoods. To see this, we need to once again examine the CBJ algorithm and more specifically the part of it that executes on backtracking from the current node. We reproduce the relevant lines of the algorithm from figure 2.10:

10. \( \text{NewNG} = \bigcup_{d \in \text{Dom}(V)} \{ \text{NG}(V \leftarrow d) - \{ V \leftarrow d \} \} \)
11. \( \text{storeNoGood(} \text{NewNG} \text{)} \)
12. \( X \leftarrow a = \text{deepest assignment in } \text{NewNG} \)
13. \( \text{Reason(} X \leftarrow a \text{)} = \text{NewNG} \)
14. \( k = \text{level } X \leftarrow a \text{ was made.} \)
15. \( \text{return(} \text{fail,} k \text{)} \)

In line 10, a new nogood is derived from the nogoods used to label the prunings of the values of \( V \). If one of these nogoods is saturated, the new one will also be saturated, since it will necessarily contain all the decision assignments in the current branch. As a result, the search will step back to the previous level, so the nogood is useless for backjumping. This new nogood is also useless for pruning future paths, as the exact same setting of decision assignments will not be examined again by backtracking search, unless restarting is used. Moreover, the new saturated nogood will be used to label the assignment \( X \leftarrow a \) made at the previous decision level. Thus, the nogood discovered at that node will also be saturated, as will all nogoods learned on this branch. Once a saturated nogood has been used to label a pruned value \( X \leftarrow a \), the search will not escape this series of backsteps unless a nogood is discovered in one of the subtrees rooted in another assignment to \( X \) that allows the search to jump back over the assignment to \( X \).

Generalized nogoods allow us to learn better nogoods from GAC propagators. As with s-nogoods, the initial g-nogood labeling a pruned value is not in itself useful, but these g-nogoods are more useful inputs for learning new nogoods.

In the next sections we will first introduce generic methods for computing g-nogoods from constraint propagators. When the generic methods are insufficient, we can also generate g-nogoods from GAC by exploiting the special structure of the constraint. As mentioned, an extensive literature has appeared developing specialized algorithms that propagate structured constraints. These existing propagators can be augmented to also generate g-nogood labels. It turns out that the paradigm of thinking about the set of
pruned values that caused another value to be pruned can often lead to natural g-nogoods from propagators. We will illustrate with several widely used constraints.

4.2 Generic methods

The simplest method to generate a g-nogood label for a value $V \leftarrow x$ pruned while enforcing GAC for constraint $C$ is to exploit the simple observation that any GAC algorithm prunes a value when that value has lost all its supports in that constraint. A g-nogood that captures this condition is appropriate as a label for a $V \leftarrow x$. To construct this g-nogood we have to identify a set of pruned values that cover all the supports of $V \leftarrow x$. Assuming $S$ is such a set, we construct an appropriate nogood label for $V \leftarrow x$ by unioning \{ $V \leftarrow x$ \} with \{ $X \not\leftarrow a \mid a \in S$ \}, which is a set which contains the non-assignments that correspond to the values that appear in $S$. Clearly, this is a nogood: if we make all the non-assignments in that g-nogood, GAC will again prune $V \leftarrow x$, because $x$ will have lost all its supports, thus no assignment that contains both the non-assignments and $V \leftarrow x$ can satisfy $C$.

One set that covers the supports of $V \leftarrow x$ in $C$, is the one that contains all pruned values of variables in the scope of $C$. This set is sufficient, otherwise the value would not be pruned in the first place. Using this set, the g-nogood computed for label $V \leftarrow x$ is $Reason_g(V \leftarrow x, C) = \{ V \leftarrow x \} \cup \{ X \not\leftarrow a \mid a \in Dom(X) - CurDom(X) \land X \in scope(C) - \{ V \} \}$. **GAC-Generic-Nogood** is the procedure that generates this nogood given $V \leftarrow x$ and $C$ and assuming that $V \leftarrow x$ is generalized arc inconsistent in $C$. It is shown in figure 4.2 This nogood is inexpensive to compute, and it outperforms the standard s-nogood that would be generated for the same pruning, as we will show next.

**Figure 4.2** Computing a generic g-nogood label for a value pruned by GAC

```c
int GAC-Generic-Nogood(C, Y \leftarrow b) {
    Ng = \{ Y \leftarrow b \}
    foreach Z \in scope(C) s.t. Z is assigned value i
        Ng = Ng \cup \{ Z \leftarrow i \}
    foreach Z \in scope(C) - \{ Y \} s.t. Z is unassigned
        foreach i \in Dom(Z) - CurDom(Z)
            Ng = Ng \cup \{ Z \not\leftarrow i \}
    return Ng
}
```
Compared to the s-nogoods that are computed using the standard technique we can make the following observations. First, the s-nogood can be generated starting with the g-nogood if we resolve $\textit{Reason}_g(V \leftarrow x)$ with the nogood $\textit{Reason}(X \leftarrow a)$ for each $X \neq a$ that appears in $\textit{Reason}_g(V \leftarrow x)$. An immediate result of this is that theorem 3.2.2 applies to $\textit{Reason}_g(V \leftarrow x)$ and $\textit{Reason}_s(V \leftarrow x)$, thus the former is more powerful locally than the latter. Since neither nogood expresses more information than what we can already get from the GAC algorithm, this means that the g-nogood better captures the conditions that caused GAC to prune $V \leftarrow x$.

More significantly, g-nogoods are local to the constraint, i.e., they contain (non-)assignments only to variables in the scope of the constraint. In contrast, as we described earlier, s-nogoods are global, which means that they potentially include assignments to many variables unrelated to the constraint at hand, even generating saturated nogoods in some cases.

### 4.2.1 Empirical evaluation

In the next few sections, we show empirical results that cover some of the algorithms resulting from a combination of some of the following features, with respect to how nogoods are used:

- No nogoods are discovered.
- Nogoods are used for backjumping only and then discarded.
- Nogoods are used for backjumping and stored and unit propagated.

If nogoods are discovered, we have the following options

- Nogoods used to label prunings by GAC (base nogoods) may be either s-nogoods generated by $\textit{GAC-CBJ-Nogood}$ or g-nogoods generated by $\textit{GAC-Generic-Nogood}$.

- When a conflict is encountered, we may learn a g-nogood using the 1$^{st}$-Decision scheme (figure 3.3) or an s-nogood using the CBJ scheme (figure 2.10).

Finally, we may employ restarting with a fixed interval in order to improve the robustness of the algorithm.
We report first on results from the instances of the 2005 CSP Solver competition [van Dongen et al., 2005]. We only examine the algorithms GAC, GACCBJ+G and GACCBJ+G+R.

- **GAC**: discovers no nogoods, i.e., only maintains GAC at each node and chronologically backtracks on conflicts.

- **GACCBJ+G**: stores nogoods for later use, labels prunings using g-nogoods generated by GAC-Generic-Nogood and uses the 1st-Decision scheme to learn g-nogoods at conflicts.

- **GACCBJ+G+R**: The same algorithm as GACCBJ+G, but with restarting every 12 conflicts, which is intended to increase robustness.\(^2\)

All algorithms use the \texttt{dom/wdeq} variable ordering heuristic, while values are assigned lexicographically.

We separate random and non-random problems and further separate families that contain only binary instances and those that may also contain non-binary instances. We do not show data about families where all 3 algorithms solved exactly the same number of instances.

The results agree with what we expect from experience from SAT solvers in the random category. Clause learning SAT solvers do not perform well in random problems and neither do the nogood learning algorithms in this case. In non-random problems, the results are mixed. GACCBJ+G+R demonstrates the increased robustness expected by adding restarting to the algorithm, solving at least as many instances as GACCBJ+G in all but 4 cases. Compared to GAC, g-nogood learning sometimes pays off, as happens most notably in the quasigroup completion families “QCP” and “QWH”. In other cases, the overhead of discovering and unit propagating the learned nogoods outweighs the benefits in terms of reduction of the size of the explored search space. This can be expected in families such as “pigeons”, which is as hard for unrestricted resolution as for tree-like resolution, thus nogood learning cannot help.

\(^2\)We experimented with different restarting schemes and different intervals, but saw no significant improvement in the robustness of the algorithms.
Figure 4.3 Results with the instances of the 2005 CSP Solver competition.

<table>
<thead>
<tr>
<th>Family name</th>
<th># Instances</th>
<th>GAC</th>
<th>GACCBJ+G</th>
<th>GACCBJ+G+R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frb</td>
<td>40</td>
<td>15</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>random</td>
<td>980</td>
<td>607</td>
<td>441</td>
<td>497</td>
</tr>
<tr>
<td>Non random binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QCP</td>
<td>120</td>
<td>77</td>
<td>88</td>
<td>90</td>
</tr>
<tr>
<td>QWH</td>
<td>395</td>
<td>328</td>
<td>349</td>
<td>349</td>
</tr>
<tr>
<td>e.ddr</td>
<td>46</td>
<td>35</td>
<td>35</td>
<td>41</td>
</tr>
<tr>
<td>fapp01</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>fapp02</td>
<td>11</td>
<td>3</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>geo</td>
<td>120</td>
<td>94</td>
<td>87</td>
<td>94</td>
</tr>
<tr>
<td>graph</td>
<td>26</td>
<td>24</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>pigeons</td>
<td>20</td>
<td>7</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>qa</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>qk</td>
<td>18</td>
<td>12</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>Non random non binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSP</td>
<td>61</td>
<td>47</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>Golomb</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>cc</td>
<td>14</td>
<td>10</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>cril</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>gr</td>
<td>10</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>lemma</td>
<td>10</td>
<td>8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>ramsey</td>
<td>16</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

More structured domains.

We now go on to examine in more detail a family of structured problems, where nogood learning is expected to help. We use instances from the logistics planning domain, using the CSP models produced by CPlan [van Beek and Chen, 1999]. This domain seems amenable to nogood learning. In the study where these instances were presented, it was observed that GAC was ineffective, while GACCBJ performed much better.

Here, we compare the following algorithms.

- **GACCBJ**: only uses nogoods for backjumping.
- **GACCBJ+S**: stores nogoods for later use, labels prunings using s-nogoods and uses
the CBJ scheme to learn s-nogoods at conflicts.

- **GACCBJ+G**: stores nogoods for later use, labels prunings using g-nogoods generated by **GAC-Generic-Nogood** and uses the 1st-Decision scheme to learn g-nogoods at conflicts.

- **GACCBJ+G_s**: stores nogoods for later use, labels prunings using s-nogoods and uses the 1st-Decision scheme to learn g-nogoods at conflicts.

The variable ordering heuristic is based on the standard **dom+deg** heuristic but a third metric which is domain specific is used as a final tiebreaker.

**Figure 4.4** CPU time and nodes visited to solve instances with various algorithms that maintain GAC.

<table>
<thead>
<tr>
<th>Instance</th>
<th>GACCBJ</th>
<th>GACCBJ+S</th>
<th>GACCBJ+G_s</th>
<th>GACCBJ+G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>nodes</td>
<td>time</td>
<td>nodes</td>
</tr>
<tr>
<td>10-11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15-15</td>
<td>52.4</td>
<td>89524</td>
<td>11.46</td>
<td>5089</td>
</tr>
<tr>
<td>18-11</td>
<td>497.34</td>
<td>155662</td>
<td>494.04</td>
<td>94504</td>
</tr>
<tr>
<td>22-11</td>
<td>85.17</td>
<td>10519</td>
<td>212.98</td>
<td>9618</td>
</tr>
<tr>
<td>26-12</td>
<td>678.55</td>
<td>99220</td>
<td>26.6</td>
<td>3202</td>
</tr>
<tr>
<td>26-13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>28-12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

We show the results in figure 4.4. We only show data for interesting instances, that is instances that were solvable by at least one but not all algorithms or the difference between the fastest and slowest algorithms is at least a factor of 5. Each row contains results for one instance. The first column contains the name of the instance. After that, there exist two columns for each algorithm. The first column shows the CPU time needed to solve that instance, in seconds, while the second one shows the number of nodes visited. A '-' in both columns indicates that the timeout of 20000 seconds expired before the solver produced an answer.

We see that the algorithms **GACCBJ+S** and **GACCBJ+G_s** provide little appreciable improvement in CPU time over **GACCBJ**. Some of the solvable instances are solved faster, even more than an order of magnitude faster in the case of instance 26-12. However, no instances that are unsolvable within the timeout with **GACCBJ** are solvable with **GACCBJ+S** or **GACCBJ+G_s**. Moreover, this can be attributed to the ineffectiveness of the learned nogoods, not just to any overhead of performing unit propagation on the learned nogoods.
This can be confirmed by observing that the reduction achieved in the number of nodes visited with $\text{GACCBJ+S}$ and $\text{GACCBJ+G}_s$ is not significant. In contrast, $\text{GACCBJ+G}$ provides a significant improvement both in the number of nodes visited and the actual CPU time used, even solving well within the timeout 3 instances that timeout with all 3 other algorithms. These results support our previous conjecture that when the propagation labels pruned values using s-nogoods, these s-nogoods are of limited use for learning g-nogoods later.

**Nogood saturation.**

We now report on some more measurements that further support the conjecture that labeling prunings with s-nogoods is ineffective. In figure 4.5 we show a measure of how saturated a nogood is, specifically we compute the ratio of $n$, the number of variables the nogood involves, to $b$, the depth of the current node. For each of the instances shown in figure 4.4 that were solved by all algorithms, we report the ratio $n/b$ for base nogoods, i.e., nogood labels produced from GAC, and for conflict nogoods, i.e., nogoods that were discovered by conflict resolution.

**Figure 4.5 Measure of saturation for s-nogoods and g-nogoods.**

<table>
<thead>
<tr>
<th>Instance</th>
<th>$\text{GACCBJ+S}$</th>
<th>$\text{GACCBJ+G}_s$</th>
<th>$\text{GACCBJ+G}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>base conflict</td>
<td>base conflict</td>
<td>base conflict</td>
</tr>
<tr>
<td>15-15</td>
<td>0.22 0.25</td>
<td>0.41 0.38</td>
<td>0.01 0.11</td>
</tr>
<tr>
<td>18-11</td>
<td>0.18 0.29</td>
<td>0.16 0.26</td>
<td>0.01 0.11</td>
</tr>
<tr>
<td>22-11</td>
<td>0.03 0.03</td>
<td>0.04 0.01</td>
<td>0.01 0.01</td>
</tr>
<tr>
<td>26-12</td>
<td>0.22 0.27</td>
<td>0.26 0.23</td>
<td>0.01 0.09</td>
</tr>
</tbody>
</table>

The number of variables involved in s-nogoods can be as high as one fifth of the size of the current branch, for base nogoods discovered from constraint propagation. The problem persists when multiple base nogoods are combined to discover a new nogood. $\text{GACCBJ+G}_s$ suffers from this problem as much as $\text{GACCBJ+S}$, despite the fact that fewer base s-nogoods are combined to produce new nogoods. The behavior of $\text{GACCBJ+G}$ is at the other extreme. Since base g-nogoods contain at most as many variables as the arity of the constraint, the saturation measure of these nogoods is extremely small. The saturation measure of learned nogoods more accurately portrays the advantage of $\text{GACCBJ+G}$ over $\text{GACCBJ+G}_s$. It is important to note that this measure does not completely characterise the relative effectiveness of either the base or the learned nogoods discovered. As noted
in the previous chapter, a g-nogood over \( k \) variables may correspond to an exponential in \( k \) number of s-nogoods over \( k \) variables.

### 4.2.2 Improving on GAC-Generic-Nogood

Despite these positive results, for some global constraints \textbf{GAC-Generic-Nogood} still tends to generate overly long g-nogood labels that are not very useful for deriving new nogoods. The extreme effects that occur with s-nogoods are usually avoided, but the benefits of using g-nogoods are diminished or even disappear completely under the overhead of learning and unit propagating the learned g-nogoods. These effects might be avoided if an effort is made to discover minimal nogoods to label values pruned by GAC. This is hard to do efficiently for general constraints, but may be easy to do if we know more about the structure of the constraint. Thus, in the next few sections we will see how we can modify existing constraint propagators to also label the values they prune with g-nogoods that are minimal in size. We will examine two aspects of the labeling procedures: minimality of the generated labels and complexity of the labeling algorithm.

**Minimality.** We are interested in determining whether the g-nogood labels that are generated by a labeling procedure are minimal with respect to GAC on \( C \), according to the following definition.

**Definition 4.2.1** (Minimal nogoods for a propagator). Let \( P \) be a constraint propagation algorithm for a class of constraints \( C \). Let \( L(P) \) be a labeling procedure for values pruned by \( P \). A nogood label \( Ng \) produced by \( L(P) \) for a value \( V \leftarrow x \) pruned by \( P \) because of a constraint \( C \) is minimal for \( P \) iff for all (non-)assignments \( X \leftarrow a \), making the assignments in \( Ng \) − \{\( X \leftarrow a \), \( V \leftarrow x \)\} and executing \( P \) on \( C \) fails to prune \( V \leftarrow x \).

It may not always be possible to efficiently identify minimal labels, but getting labels that are shorter than those generated by \textbf{GAC-Generic-Nogood} may still be worthwhile.

Also note that definition 4.2.1 does not imply that a nogood is minimal for all propagation algorithms of a constraint, as an algorithm may not enforce GAC on the constraint.

**Definition 4.2.2** (Minimal nogoods for a constraint). Let \( C \) be a class of constraints and \( P \) a (potentially not polynomial time) propagation algorithm that enforces GAC on an instance of \( C \). Let \( L(C) \) be a labeling procedure for values pruned by \( P \). A nogood
Chapter 4. Nogoods in constraint propagation

label $Ng$ produced by $L(P)$ for a value $V \leftarrow x$ pruned by $P$ because of a constraint $C$ is minimal for $C$ if it is minimal for $P$.

**Complexity.** For a labeling procedure, we need to examine what overhead it imposes with respect to the complexity of the propagation algorithm. It is always possible to generate minimal labels using dynamic programming techniques, but the cost of this is in general prohibitive (see section 4.2.4).

In general, the complexity of a propagation algorithm may be examined in the following contexts:

- Within a single invocation of the propagator. A list of pruned values is given to the propagator as input and a list of values to be pruned are returned. Interaction with other constraints is not an issue.

- Over a single branch in a backtracking search algorithm. This context involves multiple invocations of the propagator. Between invocations, the solver may visit more nodes along the same branch, which means that it makes assignments and invokes propagators for other constraints in the problem. The invocations of other propagators may cause more values to be pruned. The propagator is informed of the changes in the domains of the variables constrained. This may be different from the single invocation behaviour of the propagator, as it may be able to incrementally update its data structures. An example of this is the lexicographic ordering propagator (section 4.3.3) which runs in linear time in a single invocation, but the total cost of all invocations of this propagator over an entire branch is still linear.

- Over multiple branches in backtracking search. This also involves multiple invocations of the propagator. Between invocations, the solver may make more assignments and invoke other propagators and it may also backtrack to earlier levels. The propagator is informed which values are restored to their domains when backtracking. This is relevant because in some cases we may be able to achieve better performance over many branches at the expense of having to do more work on backtracking [Régin, 2005b].

When we augment an existing propagator, we need to examine how the complexity of the propagator is affected in each of these contexts.
In the rest of this chapter, when discussing complexity of a propagation or labeling algorithm for a constraint $C$, we let $n$ be the arity of the constraint, $n = |\text{scope}(C)|$, and $d$ be the number of all the values in the domains of the constrained variables, $d = |\bigcup_{V \in \text{scope}(C)} \text{Dom}(V)|$.

4.2.3 GAC-Generic-Nogood for bounds consistency propagators.

One piece of information that is provided by propagators in most constraint programming systems and not exploited by GAC-Generic-Nogood is the level of consistency that they maintain. We will address here specifically bounds consistency. Let us first recap the relevant definitions.

**Definition 4.2.3 (Bounds support).** Let $C$ be a constraint such that $\text{scope}(C) = \{V_1, \ldots, V_n\}$ and there exists a total ordering of the values in the domains of $V_1, \ldots, V_n$. Let $x_k \in \{\min(\text{CurDom}(V_k)), \max(\text{CurDom}(V_k))\}$ for some $k \in \{1, \ldots, n\}$. A *bounds support* for $V_k \leftarrow x_k$ in $C$ is a set of assignments $A = \{V_1 \leftarrow x_1, \ldots, V_k \leftarrow x_k, \ldots, V_n \leftarrow x_n\}$ that satisfies $C$ and $\min(\text{CurDom}(V_i)) \leq x_i \leq \max(\text{CurDom}(V_i))$ for all $i \in \{1, \ldots, n\} - \{k\}$.

Bounds consistency is defined in terms of bounds supports, in the same way that generalized arc consistency is defined in terms of supports, but whereas GAC requires that there exists a support for every value in the domain of a variable, bounds support only requires that the bounds of each variable are supported.

**Definition 4.2.4 (Bounds consistency).** A constraint $C$ is bounds consistent if for every $V \in \text{scope}(C)$, $x \in \{\min(\text{CurDom}(V)), \max(\text{CurDom}(V))\}$, there exists a bounds support for $V \leftarrow x$ in $C$.

Let $C$ be a constraint for which we only enforce bounds consistency (BC). From the definition of bounds support, we see that values pruned in the middle of a domain do not affect the behaviour of a bounds consistency propagator. Thus, we can produce a version of GAC-Generic-Nogood that produces smaller labels for prunings generated from a BC propagator. The procedure BC-Generic-Nogood produces the label $\text{Reason}_{BC}(V \leftarrow x, C) = \{V \leftarrow x\} \cup \{X \not\leftarrow a \mid a \in \text{Dom}(X) - \text{CurDom}(X) \land (a < \min(\text{CurDom}(X)) \lor a > \max(\text{CurDom}(X))) \land X \in \text{scope}(C) - \{V\}\}$. The difference between the label produced by BC-Generic-Nogood and that produced by
**GAC-Generic-Nogood** is that **BC-Generic-Nogood** only places in the label a non-assignment \( X \not\leftarrow a \) if \( a \) is outside the bounds of the current domain of \( X \).

We can similarly refine **BC-Generic-Nogood** if we know that a specific propagator only examines the lower (upper) bounds of the constrained variables. This is the case for the less-than constraint \( X < Y \), for example. The propagator for this constraint only examines the lower bound of \( X \) and the upper bound of \( Y \). Thus, when producing a label for \( X \leftarrow a \), only pruned values that are greater than the upper bound of \( Y \) need to be included in the label. Note that this improved technique does not rely on any augmentation of the propagation algorithm for \( X < Y \), but uses the already available information that the propagator only needs to be invoked when the lower bound of \( X \) or the upper bound of \( Y \) changes. The labels produced this way are still not minimal, but can be smaller than those produced by **GAC-Generic-Nogood**.

### 4.2.4 A generic method for generating minimal labels.

It is possible to generate provably minimal nogood labels for each pruning using an algorithm such as **ReplayXPlain** [Junker, 2001]. **ReplayXPlain** can generate a minimal nogood starting with a potentially non-minimal one by repeated invocations of the constraint propagator, using a dynamic programming approach. The algorithm is shown in figure 4.6.

---

**Figure 4.6** The **ReplayXPlain** algorithm

```
ReplayXPlain(C, N)
1. M = {}
2. NM = {}
3. while (M does not falsify C)
4.   T = M
5.   foreach i ∈ N – M – NM
6.     T = T ∪ {i}
7.   if T falsifies C
8.     M = M ∪ {i}
9.     NM = N – T
10.   break
11. endfor
12. return M
```

The algorithm gets a constraint \( C \) and a nogood \( N \) as input. It is based on the principle that if \( X \) does not falsify \( C \) but \( X ∪ \{x\} \) does, then \( x \) is necessarily in a minimal
nogood. Moreover the assignments in \( N - (X \cup \{x\}) \) cannot be part of this minimal nogood, since the constraint is falsified without these assignments being made. Based on this observation, ReplayXPlain works as follows. It tries to determine the sets \( M \), which is the set of assignments known to be in the minimal nogood and \( NM \) which is the set of assignments known not to be in it. In each iteration of the loop in lines 3–11, a single assignment is added to \( M \) and any number of assignments, including none, may be added to \( NM \). The loop starts with a temporary set \( T \) set to \( M \). Each assignment in \( N - M - NM \) is added to a \( T \), one at a time, in the loop in lines 5–11. When \( T \) falsifies \( C \), the last assignment added to it is known to be part of a minimal nogood and is thus added to \( M \), while the assignments in \( N - T \) are added to \( NM \). The loop in lines 3–11 terminates when \( M \) falsifies \( C \), in which case it is a minimal nogood. Note that there exist potentially many minimal nogoods, which are not all necessarily minimum, and this algorithm only discovers one of them. The one that it discovers depends on the order in which assignments are added to \( T \).

An example of the operation of ReplayXplain is shown in figure 4.7. Suppose we start with the non-minimal nogood \{a, b, c, d, e, f, g, h, i\} but the constraint admits a minimal nogood \{a, b, e, g, i\}. For each step of the algorithm, the first column shows the input that ReplayXplain passes to the propagator, the second shows the result of the propagator, which is either conflict or non-conflict, the third shows the set of assignments that are known by ReplayXplain to be part of a minimal nogood and the fourth column shows which assignments are known not to be part of the minimal nogood.

The problem with ReplayXplain is that it needs to perform \( O(n^2) \) invocations of the constraint propagator, where \( n \) is the size of the nogood produced by GAC-Generic-Nogood. In [Junker, 2001], different compromises are explored in the form of variations of ReplayXPlain, but all require at least \( O(n) \) calls of the constraint propagator with a different number of assignments to process each time. This cost is prohibitive for use in practice, as the overhead of finding minimal nogoods overwhelms the reduction in the size of the explored search space. Moreover, since it treats the propagator as a black box, it cannot take advantage of any incrementality that the propagator itself may take advantage of.

### 4.2.5 GAC schema

GAC-Schema [Bessière and Régis, 1997] is an algorithm for enforcing generalized arc
Figure 4.7 An example illustrating the operation of ReplayXplain

<table>
<thead>
<tr>
<th>Input</th>
<th>Conflict</th>
<th>$M$</th>
<th>$NM$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${a, b, c, d, e, f, g, h, i}$</td>
<td>yes</td>
<td>${i}$</td>
<td>${}$</td>
</tr>
<tr>
<td>${i}$</td>
<td>no</td>
<td>${i}$</td>
<td>${}$</td>
</tr>
<tr>
<td>${i, a}$</td>
<td>no</td>
<td>${i}$</td>
<td>${}$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>${i, a, b, c, d, e, f}$</td>
<td>no</td>
<td>${i}$</td>
<td>${}$</td>
</tr>
<tr>
<td>${i, a, b, c, d, e, f, g}$</td>
<td>yes</td>
<td>${g, i}$</td>
<td>${h}$</td>
</tr>
<tr>
<td>${g, i}$</td>
<td>no</td>
<td>${g, i}$</td>
<td>${h}$</td>
</tr>
<tr>
<td>${g, i, a}$</td>
<td>no</td>
<td>${g, i}$</td>
<td>${h}$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>${g, i, a, b, c, d}$</td>
<td>no</td>
<td>${g, i}$</td>
<td>${h}$</td>
</tr>
<tr>
<td>${g, i, a, b, c, d, e}$</td>
<td>yes</td>
<td>${e, g, i}$</td>
<td>${f, h}$</td>
</tr>
<tr>
<td>${e, g, i}$</td>
<td>no</td>
<td>${e, g, i}$</td>
<td>${f, h}$</td>
</tr>
<tr>
<td>${e, g, i, a}$</td>
<td>no</td>
<td>${e, g, i}$</td>
<td>${f, h}$</td>
</tr>
<tr>
<td>${e, g, i, a, b}$</td>
<td>yes</td>
<td>${b, e, g, i}$</td>
<td>${c, d, f, h}$</td>
</tr>
<tr>
<td>${b, e, g, i}$</td>
<td>no</td>
<td>${b, e, g, i}$</td>
<td>${c, d, f, h}$</td>
</tr>
<tr>
<td>${b, e, g, i, a}$</td>
<td>yes</td>
<td>${a, b, e, g, i}$</td>
<td>${c, d, f, h}$</td>
</tr>
</tbody>
</table>

consistency on arbitrary constraints. The constraints can be expressed in one of three ways: extensionally, as a set of allowed tuples (GAC-Schema-Allowed); extensionally, as a set of conflicting tuples (GAC-Schema-Conflicts); or intentionally with a predicate (GAC-Schema-Predicate).

**GAC-Schema-Allowed** exhibits polynomial complexity with respect to the number of tuples needed to represent the constraint. This complexity is still exponential with respect to the arity of the constraint. **GAC-Schema-Conflicts** runs in the worst case in time exponential in the arity of the constraint, but in practice performs much better. Despite this worst case complexity, these generic propagation algorithms can be used when a constraint can be represented using only a polynomial number of tuples, so they are useful in practice.

We present here methods for discovering better nogoods from **GAC-Schema-Allowed** and **GAC-Schema-Conflicts**. We call our nogood labeling versions **Nogood-GAC-Schema-Allowed** and **Nogood-GAC-Schema-Conflicts** for **GAC-Schema-Allowed** and **GAC-Schema-Conflicts**, respectively.
GAC-Schema-Allowed

GAC-Schema-Allowed is used when a constraint can be described by a small (i.e., of polynomial size in the arity of the constraint) set of satisfying tuples \( T \). A rough outline of it is given in figure 4.8. We omit details about the implementation of line 1, as it is not pertinent to our discussion. The satisfying tuples that contain \( X \leftarrow a \) are the subset \( T[X \leftarrow a] \), while each individual tuple can be accessed by \( T[X \leftarrow a][0], \ldots, T[X \leftarrow a][k_X \leftarrow a - 1] \), where \( k_X \leftarrow a = |T[X \leftarrow a]| \). The algorithm operates by identifying for each \( X \leftarrow a \) a valid tuple in \( T[X \leftarrow a] \), i.e., a tuple such that the assignments it contains are in the current domains of their corresponding variables. The index of this tuple is stored in \( \text{current}[X \leftarrow a] \). The algorithm takes advantage of monotonicity along a branch, which is the fact that if tuples are always examined in the order of their indices along a branch and the tuple with index \( \text{current}[X \leftarrow a] \) is no longer valid, then no tuple with index less than \( \text{current}[X \leftarrow a] \) is valid. Thus, when a value in \( T[\text{current}[X \leftarrow a]] \) is pruned, it only needs to examine tuples with index greater than \( \text{current}[X \leftarrow a] \) to find a new support. When no support can be found, the value is pruned. In order for the monotonicity to be maintained for every branch, the values stored in \( \text{current} \) must be restored to their previous value when backtracking.

Figure 4.8 An outline of the GAC-Schema-Allowed algorithm. \( C \) is a constraint, \( T \) is the set of supporting tuples of \( C \). We write \( T[X \leftarrow a] \) for the set of supporting tuples of \( X \leftarrow a \), while \( \text{current}[X \leftarrow a] \) is an index into \( T[X \leftarrow a] \). The values of \( \text{current} \) are restored by the solver on backtracking.

GAC-Schema-Allowed\((C, T, \text{current})\)
1. \( \text{foreach } X \leftarrow a \text{ whose support was pruned} \)
2. \( \text{foundsupport} = \text{FALSE} \)
3. \( \text{for } i \in (\text{current}[X \leftarrow a] + 1, k_X \leftarrow a) \)
4. \( \text{if } T[X \leftarrow a][i] \text{ is valid} \)
5. \( \text{current}[X \leftarrow a] = i \)
6. \( \text{foundsupport} = \text{TRUE} \)
7. \( \text{break} \)
8. \( \text{if foundsupport} \equiv \text{FALSE} \)
9. \( \text{prune } X \leftarrow a \)

When GAC-Schema-Allowed decides to prune a value \( X \leftarrow a \) while propagating constraint \( C \), it means that all of its supporting tuples have become invalid, i.e., at least one value has been pruned from each of them: \( \forall t \in T[X \leftarrow a] \exists X' \leftarrow b \in \)
Chapter 4. Nogoods in constraint propagation

86

t s.t. $X' \leftarrow b$ is pruned. So, we need to find a set of pruned values that cover the supports of the $X \leftarrow a$. This set together with $X \leftarrow a$ forms a valid nogood. Finding the minimum such set of values is an instance of the minimum set cover problem, which is NP hard, but can be approximated within $1 + \log |T|$. The approximation algorithm, shown in figure 4.9, runs in time $O(|T|^2 n)$, where $n$ is the arity of the constraint and $d$ is the domain size of the variables. Alternately, an $O(|T|d)$ algorithm, shown in figure 4.10, approximates the minimum set cover within at most $n - 1$. We also show a naive algorithm in figure 4.11, which has no approximation guarantees.

Figure 4.9 Approximating the minimum nogood for a pruning in GAC-Schema-Allowed within $1 + \log |S|$. $C$ is a constraint, $T$ is the set of supporting tuples of $C$ and $X \leftarrow a$ is the pruned value for which we generate the nogood.

Nogood-GAC-Schema-Allowed-log-approx($C$, $T$, $X \leftarrow a$)
1. $Reason(X \leftarrow a) = \{X \leftarrow a\}$
2. Initialize $freq[V \leftarrow i] = 0$ for all $V$ and $i$
3. while TRUE
4. foreach support $S \in T[X \leftarrow a]$
5. if $\exists V \not\leftarrow i \in Reason(X \leftarrow a) : V \leftarrow i \in S$
6. continue
7. foreach $V \leftarrow i \in S : i \notin CurDom(V)$
8. $freq[V \leftarrow i] = freq[V \leftarrow i] + 1$
9. $A_{max} = V \not\leftarrow i : V \in scope(C) \land i \notin CurDom(V), freq(V \leftarrow i)$ is maximum
10. if $freq[A_{max}] \equiv 0$
11. return $Reason(X \leftarrow a)$
12. $Reason(X \leftarrow a) = Reason(X \leftarrow a) \cup A_{max}$

The minimum-set-cover problem is defined as follows:

Definition 4.2.5. minimum-set-cover. Given a set $S$ and a collection of sets $P$, such that $S \subseteq \bigcup_{p \in P} p$, find a minimum cardinality subset $P'$ of $P$ so that for every element $s \in S$ there exists a set $p \in P'$, $s \in p$.

Finding a minimum nogood label is an instance of the minimum-set-cover problem, where the set to be covered is the set of supports of $X \leftarrow a$. For each pruned value, we have a set that contains the supports it appears in. The objective is to find a minimum set of values that cover all supports.

The algorithm Nogood-GAC-Schema-Allowed-log-approx is based on the approximation algorithm for the minimum-set-cover problem in [Johnson, 1974]. It starts with $\{X \leftarrow a\}$ as the tentative nogood. It then picks the assignment that appears in
the most supports of $X \leftarrow a$ and adds it as a non-assignment to the tentative nogood and removes from consideration the supports that are covered by this assignment. It continues until all supports are covered.

We can also view the problem as an instance of the \textit{minimum-hyper-vertex-cover} problem.

\textbf{Definition 4.2.6.} Given a hyper graph $G = (V, E)$, find a minimum cardinality set of vertices $X$, such that for every hyperedge $e \in E$, there exists $v \in X$ such that $v \in e$.

To solve the minimum nogood problem, we construct a graph that has a vertex for each pruned value and a hyperedge for each support that contains all vertices whose values appear in that support.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.10.png}
\caption{Approximating the minimum nogood for a pruning in GAC-Schema-Allowed within $n - 1$. $C$ is a constraint, $T$ is the set of supporting tuples of $C$ and $X \leftarrow a$ is the pruned value for which we generate the nogood.}
\end{figure}

\begin{algorithm}
1. $\text{Reason}(X \leftarrow a) = \{X \leftarrow a\}$
2. \textbf{foreach} support $S \in T[X \leftarrow a]$
3. \hspace{1em} \textbf{if} $\exists V \not\leftarrow i \in \text{Reason}(X \leftarrow a)$ : $V \leftarrow i \in S$
4. \hspace{2em} \textbf{continue}
5. \hspace{1em} \textbf{foreach} $V \leftarrow i \in S$ : $i \not\in \text{CurDom}(V)$
6. \hspace{1em} $\text{Reason}(X \leftarrow a) = \text{Reason}(X \leftarrow a) \cup \{V \not\leftarrow i\}$
7. \hspace{1em} \textbf{return} $\text{Reason}(X \leftarrow a)$
\end{algorithm}

The algorithm \textbf{Nogood-GAC-Schema-Allowed-k-approx} is based on the standard approximation algorithm for this problem. For each lost support, it adds all pruned values in it as non-assignments to the nogood. If a support is already covered by assignments that were previously added to the nogood, it is ignored. The approximation bound for this algorithm is derived for the case where all hyperedges have the same cardinality, which does not hold here. However, only the part of the argument about the tightness of the bound depends on this condition, so the algorithm will still discover a nogood that is at most $n - 1$ times larger than the minimum nogood.

The naive algorithm picks one assignment from each lost support. If a support contains an assignment that has already been added to the nogood, it is ignored. The naive algorithm provides no approximation guarantees, as can be seen with a simple example. Consider the constraint $C(X, Y, Z)$, which contains the following supports
for $X \leftarrow a$: $\{(a, a, a), (a, b, a), (a, c, a), (a, d, a)\}$. Assume \textbf{GAC-Schema-Allowed} determines that $X \leftarrow a$ must be pruned. A reasonable implementation of the algorithm might pick the first pruned value for each support at line 5. The nogood used to label $X \leftarrow a$ in this case would be $Reason(X \leftarrow a) = \{X \leftarrow a, Y \neq a, Y \neq b, Y \neq c, Y \neq d\}$. Clearly, an example that exhibits worst case behaviour can be constructed for any implementation of the “pick” verb in line 5 of the algorithm. In contrast, the algorithm \textbf{Nogood-GAC-Schema-Allowed-k-approx} would compute the nogood $\{X \leftarrow a, Y \neq a, Z \leftarrow a\}$, while \textbf{Nogood-GAC-Schema-Allowed-log-approx} would compute the nogood $\{X \leftarrow a, Z \neq a\}$.

\textbf{Figure 4.11} A naive algorithm to compute a non-maximal nogood for a pruning in \textbf{GAC-Schema-Allowed}. $C$ is a constraint, $T$ is the set of supporting tuples of $C$ and $X \leftarrow a$ is the pruned value for which we generate the nogood.

\textbf{Nogood-GAC-Schema-Allowed-naive}($C$, $X \leftarrow a$)
1. $Reason(X \leftarrow a) = \{X \leftarrow a\}$
2. \textbf{foreach} support $S \in T[X \leftarrow a]$
3. \textbf{if }\exists V \neq i \in Reason(X \leftarrow a) : V \leftarrow i \in S
4. \textbf{continue}
5. \textbf{Pick an }V \leftarrow i \in S : i \notin \text{CurDom}(V)
6. $Reason(X \leftarrow a) = Reason(X \leftarrow a) \cup \{V \neq i\}$
7. \textbf{return }Reason(X \leftarrow a)

Note that for binary constraints all the previous algorithms are equivalent.


\textit{Proof.} When we prune $X \leftarrow a$ in a constraint $C(X, Y)$, the nogood used to label this pruning must contain one value of $Y$ from each tuple that contains $X \leftarrow a$. But each supporting tuple of $X \leftarrow a$ contains exactly one value of $Y$ and each value of $Y$ may appear in at most one supporting tuple of $X \leftarrow a$. Therefore, if there exist $l$ supporting tuples, the minimum nogood will contain $l$ values from $Y$. Moreover, all the above algorithms will discover the same (minimum) nogood, because all of them choose values from supporting tuples only, so they cannot discover a larger nogood. \hfill $\square$
Complexity. In order to determine that a value $X \leftarrow a$ should be pruned, the GAC-Schema-Allowed algorithm needs to iterate over all supporting tuples of $X \leftarrow a$ exactly once. The algorithms Nogood-GAC-Schema-Allowed-naive and Nogood-GAC-Schema-Allowed-k-approx also require one iteration over all supporting tuples of $X \leftarrow a$ to generate a nogood label, so the complexity of GAC-Schema-Allowed is the same as the combination of GAC-Schema-Allowed and Nogood-GAC-Schema-Allowed-naive or Nogood-GAC-Schema-Allowed-k-approx. On the other hand, Nogood-GAC-Schema-Allowed-log-approx is quadratic in the number of supporting tuples, so the combination of GAC-Schema-Allowed with Nogood-GAC-Schema-Allowed-log-approx to generate labels is more expensive than GAC-Schema-Allowed alone.

This analysis is based on an execution of GAC-Schema-Allowed and the labeling algorithms in a static context. The results are the same along a single branch, as a value is only pruned once on each branch and therefore the labeling algorithm is only called once. On the other hand, over multiple branches the situation changes. Consider two branches that share nodes until level $l$ and diverge from $l+1$ on and that the value $X \leftarrow a$ is pruned on both branches in level $l+1$, while the value of current[$X \leftarrow a$] at level $l$ is $k_X \leftarrow a - 2$. The total number of tuples examined by GAC-Schema-Allowed for these two branches is the work common to both branches $k_X \leftarrow a - 1$ plus 1 more to prune $X \leftarrow a$ at each branch, for a total of $k_X \leftarrow a + 1$. On the other hand, Nogood-GAC-Schema-Allowed-k-approx would examine $2k_X \leftarrow a$ tuples, as all work is performed when a value is pruned, not earlier. The labeling algorithm could be modified to also incrementally construct a label. However, this would mean that its state would also have to be restored on backtracking. The state of the propagation algorithm is a single index per value, so for a constraint $C$ the size of the state is $nd$, with $n = |\text{scope}(C)|$ and $d = \max(|\text{Dom}(V).V \in \text{scope}(C)|)$. The labeling algorithm has more complex state, since it keeps a (partially constructed) label for each value. Each label may be as large as $nd$, so that the total size of the state of the labeling algorithm is $(nd)^2$. Therefore, making the labeling algorithm incremental would still make the combination of propagation and labeling more expensive than simple propagation over many branches. This analysis also applies to Nogood-GAC-Schema-Allowed-log-approx and Nogood-GAC-Schema-Allowed-naive, even though it is not obvious how Nogood-GAC-Schema-Allowed-log-approx could be made incremental.
Empirical results We compare now the above implementations of the Nogood-GAC-Schema-Allowed algorithm.

Figure 4.12 CPU time and nodes visited to solve instances that use GAC-Schema-Allowed, using different implementations of the Nogood-GAC-Schema-Allowed algorithm.

<table>
<thead>
<tr>
<th>Instance</th>
<th>generic</th>
<th>k-approx</th>
<th>naive</th>
<th>naive-random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>nodes</td>
<td>time</td>
<td>nodes</td>
</tr>
<tr>
<td>10-sat</td>
<td>77.22</td>
<td>3979</td>
<td>86.10</td>
<td>3972</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>85.62</td>
<td>3983</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>85.45</td>
<td>3975</td>
</tr>
<tr>
<td>15-9-mod</td>
<td>-</td>
<td>-</td>
<td>568.96</td>
<td>139782</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>567.45</td>
<td>139782</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23-3</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>0.67</td>
<td>2762</td>
<td>0.66</td>
<td>2807</td>
</tr>
<tr>
<td></td>
<td>0.66</td>
<td>2807</td>
<td>0.66</td>
<td>2807</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>23</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>24-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.66</td>
<td>2762</td>
<td>0.66</td>
<td>2807</td>
</tr>
<tr>
<td></td>
<td>0.66</td>
<td>2807</td>
<td>0.66</td>
<td>2807</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7-unsat</td>
<td>1.37</td>
<td>528</td>
<td>1.47</td>
<td>522</td>
</tr>
<tr>
<td></td>
<td>1.37</td>
<td>496</td>
<td>1.44</td>
<td>522</td>
</tr>
<tr>
<td>8-sat</td>
<td>0.78</td>
<td>153</td>
<td>0.84</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>0.80</td>
<td>153</td>
<td>0.81</td>
<td>153</td>
</tr>
<tr>
<td>8-unsat</td>
<td>26.48</td>
<td>4777</td>
<td>29.57</td>
<td>4987</td>
</tr>
<tr>
<td></td>
<td>29.06</td>
<td>5003</td>
<td>28.25</td>
<td>4683</td>
</tr>
<tr>
<td>9-sat</td>
<td>9.61</td>
<td>950</td>
<td>10.4</td>
<td>949</td>
</tr>
<tr>
<td></td>
<td>9.95</td>
<td>949</td>
<td>10.11</td>
<td>948</td>
</tr>
<tr>
<td>9-unsat</td>
<td>233.25</td>
<td>20376</td>
<td>248.56</td>
<td>19680</td>
</tr>
<tr>
<td></td>
<td>257.32</td>
<td>19679</td>
<td>251.97</td>
<td>20181</td>
</tr>
<tr>
<td>TSP-10</td>
<td>3.16</td>
<td>82</td>
<td>3.10</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>2.99</td>
<td>82</td>
<td>3.06</td>
<td>82</td>
</tr>
<tr>
<td>TSP-15</td>
<td>180.17</td>
<td>1132</td>
<td>171.65</td>
<td>1084</td>
</tr>
<tr>
<td></td>
<td>168.56</td>
<td>1101</td>
<td>170.11</td>
<td>1097</td>
</tr>
<tr>
<td>merged</td>
<td>0.16</td>
<td>102</td>
<td>0.17</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>0.17</td>
<td>102</td>
<td>0.18</td>
<td>102</td>
</tr>
</tbody>
</table>

We compared the algorithms on non-binary instances from the 2005 CSP competition [van Dongen et al., 2005]. The results are shown in figure 4.12. In it, we show the CPU time in seconds and number of nodes visited to solve some of the instances using GAC-Generic-Nogood, Nogood-GAC-Schema-Allowed-k-approx, Nogood-GAC-Schema-Allowed-naive and Nogood-GAC-Schema-Allowed-naive-random, in order. We do not show data for Nogood-GAC-Schema-Allowed-log-approx, as its quadratic complexity rendered it much slower that all other algorithms. Each algorithm was given a maximum of 10 minutes to solve each instance. If it failed to solve the instance within that limit, a timeout was reported and a ‘-’ placed on the table.

The results are mixed. In most cases, each of the algorithms we presented outperforms the generic algorithm in terms of number of nodes visited to solve a problem. There exist some exceptions (for example, the instances 24-3 and 8-unsat), which can be attributed to the nogood paradox that we also saw earlier: discovering smaller (thus locally more powerful) nogoods may end up causing the solver to explore a larger part of the search space, because it does not visit parts of the search space where it may discover even more...
powerful nogoods. In the rest of the instances, the algorithm that generates nogoods using **GAC-Generic-Nogood** visits at least the same number of nodes as the rest of the algorithms. However, the gains in terms of the size of the explored search space are modest and in most cases are not enough to overcome the overhead of discovering shorter nogoods. On the other hand, the slowdown exhibited is also modest, and there exists an instance where using either the $k$-approximation algorithm or the naive algorithm allows us to solve the instance within the timeout, where using the generic algorithm or naive-random times out. So it appears that using the $k$-approximation algorithm or the naive algorithm is almost always preferable in this set of benchmarks.

**GAC-Schema-Conflicts**

**GAC-Schema-Conflicts** is used when a constraint $C$ can be described by a small (i.e., of polynomial size in the arity of the constraint) set of conflicting tuples $T(C)$, each of which is an s-nogood. When **GAC-Schema-Conflicts** on constraint $C$ prunes a value $V \leftarrow x$, it means that all possible complete instantiations over the current domains of $scope(C)$ that assign $x$ to $V$ are conflicts. To produce a smaller nogood label than the one generated by **GAC-Generic-Nogood**, $Ng$, we first transform $Ng$ to a set of s-nogoods that are a subset of the set of conflicts of $C$, as described in section 3.2.1. Recall that a g-nogood $Ng$ that contains a set of non-assignments $\{X \not\leftarrow d_1, \ldots, X \not\leftarrow d_k\}$ can be transformed into $|Dom(X) - k|$ nogoods each of which contains one of $\{X \leftarrow x | x \in Dom(X) - \{d_1, \ldots, d_k\}\}$ and none of $\{X \not\leftarrow d_1, \ldots, X \not\leftarrow d_k\}$. If $n_X$, non-assignments from each of $X_1, \ldots, X_d$ appear in $Ng$, then $\prod_{1 \leq i \leq d} (|Dom(X_i)| - n_X)$ s-nogoods can be produced in this way. The function **gnogood-to-snogoods**, reproduced in figure 4.13 from the previous chapter performs this transformation.

**Figure 4.13** Transforming a single g-nogood to a set of s-nogoods

```plaintext
gnogood-to-snogoods(Ng)
1. $S = \{\}$
2. foreach variable $X$ s.t. $X \not\leftarrow a \in Ng$ for some $a$
3.     foreach $X \leftarrow b$ s.t. $X \not\leftarrow b \notin Ng$
4.     $Ng|_X = \{Y \leftarrow d \; s.t. \; Y \leftarrow d \in Ng \land Y \neq X\}$
5.     $S = S \cup gnogood-to-snogoods(X \leftarrow b) \cup Ng|_X$
6.     if $S \equiv \emptyset$
7.         return $\{Ng\}$
8.     return $S$
```
Let $Ng$ be the nogood produced by GAC-Generic-Nogood to label the pruning of a value $V \leftarrow x$ and $S(Ng) = \text{gnogood-to-snogoods}(Ng)$. To create a smaller label than $Ng$, we must remove non-assignments from it. Let $R$ be a set of non-assignments. As the size of $R$ grows, so does the size of the set $S(Ng - R)$. If $S(Ng - R)$ is not a subset of the set $T(C)$ of conflicting tuples of $C$, then $Ng - R$ is not a nogood,\(^3\) as it represents a constraint that is tighter than $C$.

The goal of finding a minimal nogood label can then be seen as maximizing $R$, which in turn means maximizing $S(Ng - R)$ so that $S(Ng - R) \subseteq T(C)$. In other words, we have to find a maximal set of conflicts that agrees with the current set of instantiations and can be represented as a single g-nogood.

The algorithm in figure 4.14 shows how we can remove some non-assignments from the default nogood $Ng$ calculated for the removal of $V \leftarrow x$. The procedure Nogood-GAC-Schema-Conflicts generates in line 1 the generic nogood $Ng$ that GAC-Generic-Nogood would generate and then simply calls the helper Nogood-GAC-Schema-Conflicts-inner in lines 3–6 until no assignments can be removed from $Ng$.

We turn to Nogood-GAC-Schema-Conflicts-inner before we explain why this iterative call is necessary. In the loop in lines 1–4, we calculate $|(S(Ng - \{X \not\leftarrow a\}) \cap T(C))|$.

\(^3\)At least, it is not a nogood that can be derived from $C$ alone.
for every non-assignment \(X \not\leftarrow a\) in \(Ng\). This is the number of s-nogoods in \(S(Ng - \{X \not\leftarrow a\})\) that are also present in \(T(C)\). To compute this, we count the number of s-nogoods \(c\) in \(T(C)\) that are also in \(S(Ng - \{X \not\leftarrow a\})\) for each non-assignment \(X \not\leftarrow a \in Ng\).

Then, in lines 5–6 we choose a variable \(X\) and remove from \(Ng\) all non-assignments \(X \not\leftarrow a\) which can safely be removed. This is determined by checking that \(|(S(Ng - \{X \not\leftarrow a\}) \cap T(C))| = |S(Ng - \{X \not\leftarrow a\})|\). Clearly, \(|(S(Ng - \{X \not\leftarrow a\}) \cap T(C))| \leq |(S(Ng - \{X \not\leftarrow a\})|\). If \(|(S(Ng - \{X \not\leftarrow a\}) \cap T(C))| < |(S(Ng - \{X \not\leftarrow a\})|\), then \(S(Ng - \{X \not\leftarrow a\}) \not\subseteq T(C)\), therefore it is not a nogood. Otherwise \(X \not\leftarrow a\) can safely be removed from \(Ng\).

We will show next that we can only remove multiple non-assignments from \(Ng\) if they belong to the same variable. This explains that in order to get a minimal nogood, we need to call the function repeatedly until no more non-assignments can be removed. In the context of \texttt{Nogood-GAC-Schema-Conflicts-inner}, this means we have to potentially choose which variable that will be. We can choose the variable either to minimize the size of \(Ng\) or to maximize the size of \(S(Ng)\). At most \(|\text{scope}(C)|\) calls will be required to get a minimal nogood. However, the choice made by the heuristic in line 5 will affect which of several minimal nogoods we end up with.

In general we cannot remove non-assignments for more than one variable from \(Ng\) in a single iteration. For example, consider the constraint \(C\) with \(\text{scope}(C) = \{X, Y, Z\}\) and \(T(C) = \{\{1, 1, 2\}, \{1, 1, 3\}, \{1, 2, 1\}, \{1, 3, 1\}, \{1, 2, 2\}, \{1, 3, 2\}, \{1, 3, 3\}\}\), with \(\text{Dom}(X) = \text{Dom}(Y) = \text{Dom}(Z) = \{1, 2, 3\}\). Let the values \(Y \leftarrow 1\) and \(Z \leftarrow 1\) be pruned. \texttt{GAC-Schema-Conflicts} will prune \(X \leftarrow 1\). The nogood label produced by \texttt{GAC-Generic-Nogood} is \(Ng = \{X \leftarrow 1, Y \not\leftarrow 1, Z \not\leftarrow 1\}\). For both \(Y \not\leftarrow 1\) and \(Z \not\leftarrow 1\), we see that they can safely be removed from \(Ng\), so \(\{X \leftarrow 1, Y \not\leftarrow 1\}\) and \(\{X \leftarrow 1, Z \not\leftarrow 1\}\) are both nogoods. However, we cannot remove both. That would leave us with \(\{X \leftarrow 1\}\), which is not a nogood, as the tuple \(\{X \leftarrow 1, Y \leftarrow 1, Z \leftarrow 1\}\) satisfies \(C\).

On the other hand, we can remove multiple non-assignments that belong to the same variable \(X\). If we have determined that \(X \not\leftarrow a\) and \(X \not\leftarrow b\) are both safe to remove from \(Ng\), we can tell that \(S(Ng - \{X \not\leftarrow a, X \not\leftarrow b\}) = S(Ng - \{X \not\leftarrow a\}) \cup S(Ng - \{X \not\leftarrow b\})\). This stems from the fact that line 5 of \texttt{gnogood-to-snogoods} produces \(S(Ng - \{X \not\leftarrow a, X \not\leftarrow b\})\) simply by unioning together \(S(Ng - \{X \not\leftarrow a\})\) and \(S(Ng - \{X \not\leftarrow b\})\). Because we can safely remove either of \(X \not\leftarrow a\) and \(X \not\leftarrow b\), we have \(S(Ng - \{X \not\leftarrow a, X \not\leftarrow b\}) \cap T(C) = (S(Ng - \{X \not\leftarrow a\}) \cap T(C)) \cup (S(Ng -
\( \{X \not\leftarrow b\} \cap T(C) = S(Ng - \{X \not\leftarrow a\}) \cup S(Ng - \{X \not\leftarrow b\}) = S(Ng - \{X \not\leftarrow a, X \not\leftarrow b\}) \), so \( S(Ng - \{X \not\leftarrow a, X \not\leftarrow b\}) \) is also a nogood.

**Complexity.** We examine in detail first the complexity of Nogood-GAC-Schema-Conflicts-inner and then that of Nogood-GAC-Schema-Conflicts itself and finally in conjunction with GAC-Schema-Conflicts.

The first observation we need to make is that even though Nogood-GAC-Schema-Conflicts reasons about the size of \( S(Ng) \), we never explicitly generate this set. This is important, as the size of \( S(Ng) \) is exponential in \(|Ng|\) and potentially bigger than \(|T(C)|\). To see this, we examine gnogood-to-snogoods again. In lines 3–5, gnogood-to-snogoods generates an s-nogood for each value \( X \leftarrow a \) of \( X \) that does not appear as a non-assignment \( X \not\leftarrow a \) in \( Ng \). Conversely, a conflict \( c \) cannot be in \( S(Ng) \) if \( c \) contains an assignment \( X \leftarrow a \) whose complement \( X \not\leftarrow a \) appears in \( Ng \). Since the size of \( c \) is always \( n = |\text{scope}(C)| \), the membership check of line 4 costs \( O(n) \).

For example, consider the conflict \( c = \{X_1 \leftarrow a, X_2 \leftarrow b, X_3 \leftarrow c\} \) and the g-nogood \( Ng = \{X_1 \not\leftarrow a, X_1 \not\leftarrow b, X_2 \not\leftarrow a, X_2 \not\leftarrow c, X_3 \not\leftarrow a, X_3 \not\leftarrow b\} \), with the domains of all variables being \( \text{Dom}(X_1) = \text{Dom}(X_2) = \text{Dom}(X_3) = \{a, b, c\} \). We see that \( c \) is not in \( S(Ng) = \{\{X_1 \leftarrow c, X_2 \leftarrow b, X_3 \leftarrow c\}\} \), since it contains \( X_1 \leftarrow a \), while \( Ng \) contains \( X_1 \not\leftarrow a \). On the other hand, we can see that it is in \( S(Ng - \{X_1 \not\leftarrow a\}) = \{\{X_1 \leftarrow c, X_2 \leftarrow b, X_3 \leftarrow c\}, \{X_1 \leftarrow a, X_2 \leftarrow b, X_3 \leftarrow c\}\} \), because \( Ng - \{X_1 \not\leftarrow a\} \) does not contain \( X_1 \not\leftarrow a \).

This observation also allows us to efficiently decide if there exists a single non-assignment \( X \not\leftarrow a \) which can be removed from \( Ng \) so that \( c \in S(Ng - \{X \not\leftarrow a\}) \). Specifically, if \( c \not\in S(Ng) \), then there exists a non-assignment \( X \not\leftarrow a \in Ng \) such that \( c \in S(Ng - \{X \not\leftarrow a\}) \) if and only if \( X \leftarrow a \in c \) and \( c \) contains no other assignment \( X' \leftarrow b \), \( X' \neq X \) such that \( X' \not\leftarrow b \in Ng \). This condition can also be determined in \( O(n) \) time.

We see that the loop in lines 1–4 of Nogood-GAC-Schema-Conflicts-inner is executed \(|Ng||T(C)|\) times. Since the cost of the membership check is \( O(n) \), the total cost of this loop is \( O(n|Ng||T(C)|) \). The heuristic choice of line 5 can be made by examining each non-assignment in \( Ng \), so is executed in time \( O(|Ng|) \) and the removal of non-assignments at line 6 is made in time \( O(d) \), where \( d \) is the maximum domain size. In total, the cost of Nogood-GAC-Schema-Conflicts-inner is \( O(max(n|Ng||T(C)|, |Ng|, d)) \).

In practice, \(|T(C)|\) will be much larger than either \( d \) or \(|Ng|\), so the loop dominates the
runtime and the cost will be \( O(n |N_g| |T(C)|) \).

Figure 4.15 More efficient Nogood-GAC-Schema-Conflicts-inner

```
Nogood-GAC-Schema-Conflicts-inner(C, V ← x, N_g)
1. foreach conflict c ∈ T(C)
2.     if c ∉ S(N_g)
3.         if ∃ X ← a ∈ c s.t. c ∈ S(N_g − {X ← a})
4.             #ng[X ← a] = #ng[X ← a] + 1
5.     Choose X ∈ scope(C) − {V}
6.     N_g = N_g − {X ← a s.t. #ng[X ← a] = |S(N_g − {X ← a})| − |S(N_g)|}
7. return N_g ∪ {V ← x}
```

However, the loop in line 1–4 can be made more efficient if we make the observation
that we only need to examine how many more s-nogoods are in \( S(N_g − \{X ← a\}) \) as
compared to \( S(N_g) \). Thus, the total cost of the loop can be reduced to
\( O(n |T(C)|) \), as shown in figure 4.15.

As we already discussed, Nogood-GAC-Schema-Conflicts-inner will be called
at most \( n \) times, therefore the cost of Nogood-GAC-Schema-Conflicts is in
\( O(n^2 |T(C)|) \).

As was the case with GAC-Schema-Allowed, GAC-Schema-Conflicts is also
an incremental algorithm, which needs \( O(n |T(C)|) \) time to prune a value on a branch.
Thus, Nogood-GAC-Schema-Conflicts presents an overhead over labeling pruned
values with GAC-Generic-Nogood. Alternatively, we can compromise between
generating a minimal nogood and GAC-Generic-Nogood, by only changing Nogood-
GAC-Schema-Conflicts to call Nogood-GAC-Schema-Conflicts-inner at most \( k \)
times, for some fixed \( k \), before it returns. In that case, labelling values with Nogood-
GAC-Schema-Conflicts does not alter the complexity of GAC-Schema-Conflicts
on a single branch. It does however change it when examining the behaviour over many
branches. Since Nogood-GAC-Schema-Conflicts is not incremental, the work it performs
cannot be shared among many branches. Trying to make it incremental would
require keeping \( O(nd) \) state for each value that may be pruned, or \( O(n^2d^2) \) total state
which would have to be restored on backtracking. This means that we cannot avoid some
overhead in order to generate smaller nogoods than those computed by GAC-Generic-
Nogood.

We do not evaluate this method empirically, because constraints defined in this way
were not used extensively in the CSP competition from which we drew these benchmarks.
4.3 Constraints with specialised propagators

We now turn our attention to some constraints for which there exist specialized propagators which are faster than the generic GAC-Schema propagator. These constraints appear in many domains, so it is worthwhile to explore methods to generate small nogoods from these propagators. Similar work has been done in [Rochart and Jussien, 2003] for the STRETCH constraint [Pesant, 2001], as well as in [Rochart et al., 2003] for the ALLDIFF [Régin, 1994] and other constraints. That work, however, only deals with s-nogoods.

We have also independently studied the ALLDIFF constraint in [Katsirelos and Bacchus, 2005] and discuss it further in section 4.3.4.

Note that not all of these propagators enforce GAC on the constraint, as it might be intractable to do so. We will identify for each propagator the level of consistency it enforces.

4.3.1 Less-than

A simple but illustrative example is the less-than constraint between two variables $X$ and $Y$. This constraint simply asserts that $X < Y$. Let $\text{Dom}(X) = \text{Dom}(Y) = \{1, \ldots, d\}$. A propagator for this constraint would prune value $i$ from $\text{Dom}(X)$ when values $i+1, \ldots, d$ have been pruned from $\text{Dom}(Y)$, and it would prune the value $j$ from $\text{Dom}(Y)$ when the values $1, \ldots, j-1$ have been pruned from $\text{Dom}(X)$. There is a clear correspondence between the way the propagator works and the nogoods that should be generated. The nogood labeling the pruned assignment $X \leftarrow i$ would be $\text{Reason}_{<}(X \leftarrow i, X < Y) = \{Y \not\leftarrow i+1, \ldots, Y \not\leftarrow d, X \leftarrow i\}$ while the nogood labeling the pruned assignment $Y \leftarrow j$ would be $\text{Reason}_{<}(Y \leftarrow j, X < Y) = \{X \not\leftarrow 1, \ldots, X \not\leftarrow j-1, Y \leftarrow j\}$. Note that, when we prune, e.g., $X \leftarrow i$, we do not have to worry about pruned values of $X$ and $Y$ other than those contained in this set—these other pruned values (which would also have been included in the nogood produced by GAC-Generic-Nogood) are not relevant. It is also easy to see that these nogoods are minimal with respect to GAC on the constraint $X < Y$. Computing these nogoods does not affect the complexity of the propagator, as they can be precomputed for each value and used to label each pruning in constant time.

We can also compute nogoods that explain why the less-than constraint has become entailed, i.e. why all remaining instantiations have to satisfy this constraint. This would happen if the upper bound of $X$ is less than the lower bound of $Y$, which is written as a
nogood: $\text{Reason}_{\leq}(X < Y) = \{X \not\rightarrow j|j > \text{ub}(X)\} \cup \{Y \not\rightarrow j|j \leq \text{ub}(X)\}$.

We compute $\text{Reason}_{\leq}$ for the $X \leq Y$ constraint in a similar manner.

### 4.3.2 The cardinality constraint on binary variables.

The cardinality constraint over the binary variables $V_1, \ldots, V_n$, requires that at least $d$ of them get the value 1. This constraint is a generalization of propositional clauses, as clauses can be simulated by setting $d = 1$.

**Figure 4.16** Propagator for the cardinality constraint on binary variables. The value of $\text{num-zeroes}$ is 0 at the root of the search tree and restored by the solver on backtracking.

```plaintext
propagate-cardinality-boolean(C, d, ∆)
1. foreach $V \leftarrow x$ in $∆$
2.   if $x \equiv 0$ then continue
3.   $\text{num-zeroes} = \text{num-zeroes} + 1$
4.   if $\text{num-zeroes} \equiv |\text{scope}(C)| - d$
5.     foreach $V \in \text{scope}(C)$ s.t. $V \leftarrow 0$ is not pruned
6.       Prune $V \leftarrow 0$
```

An algorithm that runs in linear time distributed over the length of a branch can be used to enforce GAC on it (see for example [Dixon et al., 2004]). A relatively inefficient implementation of such a propagator is shown in 4.16. The arguments to this algorithm are $C$, the constraint, $d$, the parameter and $\Delta$ the set of values in $\text{scope}(C)$ that were pruned since the last invocation of the algorithm. A counter $\text{num-zeroes}$ is maintained by $\text{propagate-cardinality-boolean}$. The counter is initialized to zero at the root of the search tree. Whenever the value 1 is pruned from the domain of a constrained variable, this counter is incremented. When it reaches $n - d$, the remaining variables are forced to 1, i.e., 0 is pruned. Whenever the solver backtracks to level $l$, it restores the value of the counter to the value it had at that level.

This implementation is still linear distributed over the length of a branch, since the size of $\Delta$ over an entire branch is bounded by $|\text{scope}(C)|$. In practice, such a constraint would be propagated using a generalization of the watch literal techniques used to unit propagate clauses, where we maintain $d + 1$ watches instead of just 2.

To see how we generate labels, we consider the conditions which lead to the pruning of a value. Values are pruned when $n - d$ variables have been forced to the value 0, so the remaining variables will be forced to 1. The nogood la-
bel generated for the pruning of $V_i \leftarrow 0$ must capture the fact that $n - d$ variables have already been forced to 0. This means that $Reason(V_i \leftarrow 0, C) = \{V_i \leftarrow 0\} \cup \{V \neq 1 | V \in scope(C) - \{V_i\} \land V \leftarrow 1$ is pruned}. This is clearly a minimal nogood with respect to GAC on $C$, as pruning cannot happen when fewer than $n - d$ values have been pruned. Having to produce labels, however, is an overhead over propagation, as it takes $O(\text{scope}|C|)$ time to generate a label for each pruning.

4.3.3 The lexicographic ordering constraint.

Given two vectors of $n$ variables $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$, the lexicographic ordering constraint LexLessEq asserts that $X_1 \leq Y_1$ and $\forall k \in [1, n-1]$, $X_1 = Y_1 \land \cdots \land X_k = Y_k \Rightarrow X_{k+1} \leq Y_{k+1}$. A slightly different version is the strict lexicographic constraint LexLess, which asserts that $X_1 \leq Y_1$; $\forall k \in [1, n-2]$, $X_1 = Y_1 \land \cdots \land X_k = Y_k \Rightarrow X_{k+1} < Y_{k+1}$; and $X_1 = Y_1 \land \cdots \land X_{n-1} = Y_{n-1} \Rightarrow X_n < Y_n$. We will deal here with LexLessEq only, as LexLess is so similar.

LexLessEq can be decomposed into $n$ constraints of arity 2, 3, \ldots, 2n that follow exactly the definition, so the $i^{th}$ constraint is $X_1 = Y_1 \land \cdots \land X_i = Y_i \land \cdots \land X_{i-1} = Y_{i-1} \Rightarrow X_{i+1} \leq Y_{i+1}$. However, this decomposition does not maintain GAC for the LexLessEq constraint. A linear time incremental algorithm that maintains GAC on LexLessEq is described in [Frisch et al., 2002].

The algorithm prunes values in two cases. The first case is when there exists $\alpha$ such that $X_i = Y_i$ for all $1 \leq i < \alpha$, in which case it enforces GAC on the constraint $X_\alpha \leq Y_\alpha$. The second case is when there exists an $\alpha$ such that $X_i = Y_i$ for all $1 \leq i < \alpha$ and $X_{\alpha+1} > Y_{\alpha+1}$, in which case it enforces GAC on the constraint $X_\alpha < Y_\alpha$.

In the first case, the nogood label for each value pruned from the domain of $X_\alpha$ or $Y_\alpha$ is the union of the label that GAC produced from the constraint $X_\alpha \leq Y_\alpha$ with the domain prunings that caused GAC to be enforced on this constraint, i.e., the pruned values which forced $X_i = Y_i$ for $i \in [1, \alpha - 1]$. If value $d$ is pruned from $X_\alpha$, then

\[
Reason_{lex}(X_\alpha \leftarrow d, C) = \{X_\alpha \leftarrow d\} \cup Reason(Y_\alpha < d) \cup \bigcup_{1 \leq i < \alpha} Reason(X_i = Y_i)
\]

Note that $X_1 \leq Y_1$ and $X_1 = Y_1 \land \cdots \land X_{i-1} = Y_{i-1} \land \text{LEXLESSEQ}(X, Y) \land X_i \geq Y_i \Rightarrow
$X_i = Y_i$. Therefore, we only need to describe why $X_i \geq Y_i$, which is less restrictive than $X_i = Y_i$. Therefore,

$$\text{Reason}_{lex}(X_\alpha \leftarrow d, C) = \{X_\alpha \leftarrow d\}$$

$$\cup \text{Reason}(Y_\alpha < d)$$

$$\cup \bigcup_{1 \leq i < \alpha} \text{Reason}(X_i \geq Y_i)$$

$$= \{X_\alpha \leftarrow d\}$$

$$\cup \{Y_\alpha \not\leftarrow j | j \geq d\}$$

$$\cup \bigcup_{1 \leq i < \alpha} (\{X_i \not\leftarrow j | j < \max(X_i)\} \cup \{Y_i \not\leftarrow j | j > \min(Y_i)\})$$

(4.1)

Symmetrically, if $d$ is pruned from the domain of $Y_\alpha$, then

$$\text{Reason}_{lex}(Y_\alpha \leftarrow d, C) = \{Y_\alpha \leftarrow d\}$$

$$\cup \text{Reason}(X_\alpha > d)$$

$$\cup \bigcup_{1 \leq i < \alpha} \text{Reason}(X_i = Y_i)$$

subsumed by $\{Y_\alpha \leftarrow d\}$

$$\cup \text{Reason}(X_\alpha > d)$$

$$\cup \bigcup_{1 \leq i < \alpha} \text{Reason}(X_i \geq Y_i)$$

$$= \{Y_\alpha \leftarrow d\}$$

$$\cup \{X_\alpha \not\leftarrow j | j \leq d\}$$

$$\cup \bigcup_{1 \leq i < \alpha} (\{X_i \not\leftarrow j | j < \max(X_i)\} \cup \{Y_i \not\leftarrow j | j > \min(Y_i)\})$$

(4.2)

In the second case, when there exists an $\alpha$ such that $X_i = Y_i$ for all $1 \leq i < \alpha$ and $X_{\alpha+1} > Y_{\alpha+1}$, the nogood label will be the label that GAC produces from the constraint $X_\alpha < Y_\alpha$ unioned with the domain prunings that caused $X_{\alpha+1} > Y_{\alpha+1}$ and $X_i = Y_i$ for $i \in [1, \alpha - 1]$. Specifically,

$$\text{Reason}_{lex}(X_\alpha \leftarrow d, C) = \{X_\alpha \leftarrow d\}$$
Chapter 4. Nogoods in constraint propagation

\[ \begin{align*}
\text{Reason}(Y_\alpha \leq d) \\
\bigcup_{1 \leq i < \alpha} \text{Reason}(X_i = Y_i) \\
\text{Reason}(X_{\alpha+1} > Y_{\alpha+1}) \\
\text{subsumed by} \quad \{X_\alpha \leftarrow d\} \\
\bigcup \text{Reason}(Y_\alpha \leq d) \\
\bigcup_{1 \leq i < \alpha} \text{Reason}(X_i \geq Y_i) \\
\text{Reason}(X_{\alpha+1} > Y_{\alpha+1}) \\
= \{X_\alpha \leftarrow d\} \\
\{Y_\alpha \not\leftarrow j | j > d\} \\
\bigcup_{1 \leq i < \alpha} (\{X_i \not\leftarrow j | j < \max(X_i)\} \cup \{Y_i \not\leftarrow j | j > \min(Y_i)\}) \\
\{X_{\alpha+1} \not\leftarrow j | j \leq \min(Y_{\alpha+1})\} \cup \{Y_{\alpha+1} \not\leftarrow j | j \geq \max(X_{\alpha+1})\}
\end{align*} \tag{4.3} \]

and

\[ \begin{align*}
\text{Reason}_{lex}(Y_\alpha \leftarrow d, C) &= \{Y_\alpha \leftarrow d\} \\
\bigcup \text{Reason}(X_\alpha \geq d) \\
\bigcup_{1 \leq i < \alpha} \text{Reason}(X_i = Y_i) \\
\text{Reason}(X_{\alpha+1} > Y_{\alpha+1}) \\
\text{subsumed by} \quad \{Y_\alpha \leftarrow d\} \\
\bigcup \text{Reason}(X_\alpha \geq d) \\
\bigcup_{1 \leq i < \alpha} \text{Reason}(X_i \geq Y_i) \\
\text{Reason}(X_{\alpha+1} > Y_{\alpha+1}) \\
= \{Y_\alpha \leftarrow d\} \\
\{X_\alpha \not\leftarrow j | j < d\} \\
\bigcup_{1 \leq i < \alpha} (\{X_i \not\leftarrow j | j < \max(X_i)\} \cup \{Y_i \not\leftarrow j | j > \min(Y_i)\}) \\
\{X_{\alpha+1} \not\leftarrow j | j \leq \min(Y_{\alpha+1})\} \cup \{Y_{\alpha+1} \not\leftarrow j | j \geq \max(X_{\alpha+1})\}
\end{align*} \tag{4.4} \]

The nogoods 4.3 and 4.4 are derived from 4.1 and 4.2, respectively, by making two
changes. First, we add \( \text{Reason}(X_{\alpha+1} > Y_{\alpha+1}) \). Second, we substitute \( \text{Reason}(Y_{\alpha} \leq d) \) (resp. \( \text{Reason}(X_{\alpha} \geq d) \)) for \( \text{Reason}(Y_{\alpha} < d) \) (resp. \( \text{Reason}(X_{\alpha} > d) \)).

**Proposition 4.3.1.** The sets 4.1, 4.2, 4.3, 4.4 are minimal nogoods.

**Proof.** All the sets are unions of nogoods generated from simpler constraints and whose scopes are pairwise distinct. We have already shown that the simpler constraints generate minimal nogoods, so 4.1, 4.2, 4.3, 4.4 are also minimal nogoods.

**Complexity.** Generating each of these nogoods takes only as much time as is required to traverse the domains of the variables \( X_1, Y_1, \ldots, X_{\alpha}, Y_{\alpha} \) (or \( X_1, Y_1, \ldots, X_{\alpha+1}, Y_{\alpha+1} \) in the second case). Since \( \alpha < n \), this time is linear in the arity of the constraint. However, the cost of the propagator is spread over the length of a branch. Therefore, nogood learning imposes an overhead over simple propagation.

**Figure 4.17** Comparison of GAC based algorithms on the Social Golfer problem using LEXLESSEQ constraints for symmetry breaking.

<table>
<thead>
<tr>
<th>Problem (w,g,s)</th>
<th>( \text{GAC} )</th>
<th>( \text{GACCBJ+S} )</th>
<th>( \text{GACCBJ+G} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Nodes</td>
<td>Time</td>
</tr>
<tr>
<td>11-6-2</td>
<td>0.23</td>
<td>1776</td>
<td>0.63</td>
</tr>
<tr>
<td>13-7-2</td>
<td>4.82</td>
<td>18730</td>
<td>18.27</td>
</tr>
<tr>
<td>2-7-5</td>
<td>1586.44</td>
<td>35080126</td>
<td>218.04</td>
</tr>
<tr>
<td>2-8-5</td>
<td>-</td>
<td>-</td>
<td>1211.87</td>
</tr>
<tr>
<td>3-6-4</td>
<td>-</td>
<td>-</td>
<td>869.65</td>
</tr>
<tr>
<td>3-7-4</td>
<td>-</td>
<td>-</td>
<td>549.56</td>
</tr>
<tr>
<td>4-5-4</td>
<td>843.4</td>
<td>22525307</td>
<td>91.47</td>
</tr>
<tr>
<td>4-7-3</td>
<td>0.11</td>
<td>1455</td>
<td>0.25</td>
</tr>
<tr>
<td>5-6-3</td>
<td>105.14</td>
<td>1973647</td>
<td>1142.57</td>
</tr>
<tr>
<td>5-8-3</td>
<td>218.86</td>
<td>1869855</td>
<td>277.9</td>
</tr>
<tr>
<td>9-8-4</td>
<td>1.65</td>
<td>2379</td>
<td>2.74</td>
</tr>
</tbody>
</table>

**Empirical evaluation.** We tested a model of the social golfer problem (prob010 in CSPLIB[Gent and Walsh, 1999]) that uses a three dimensional matrix of binary variables and the lexicographic ordering constraint for symmetry breaking [Frisch et al., 2002]. The problem is to create a schedule for \( g \) groups of \( s \) golfers over \( w \) weeks so that no
two players play in the same group more than once. This means that there are \( p = gs \) players overall. The model proposed in [Frisch et al., 2002] has a set of \( wgp = wq^2s \) binary variables \( X_{i,j,k} \). \( X_{i,j,k} \) is true iff player \( i \) plays in week \( j \) in group \( k \). Boolean cardinality constraints are posted to ensure that each player plays in exactly one group each week and that there are exactly \( s \) players in each group. We ensure that each pair of players play in the same group at most once, by introducing auxiliary variables \( X_{i_1,i_2,j,k} = X_{i_1,j,k} \land X_{i_2,j,k} \). \( X_{i_1,i_2,j,k} \) is true if players \( i_1 \) and \( i_2 \) play each other in group \( j \) of week \( k \). We post cardinality constraints on the variables \( X_{i_1,i_2,j,k} \) for each pair of players. Finally, \textsc{LexLessEq} constraints are posted between week planes, player planes and group vectors within each week to break symmetry. A static variable ordering that exploits the structure of the problem is used.

We show results in figure 4.17 for three different algorithms.

- **GAC**: Maintains GAC on all constraints with no learning and no intelligent backtracking.
- **GACCBJ+S**: Maintains GAC on all constraints except \textsc{LexLessEq} and learns \textit{s}-nogoods at conflicts. The GAC algorithm labels pruned values with \textit{s}-nogoods. The decomposition of the \textsc{LexLessEq} constraint is used and propagated by forward checking. The nogoods learned from propagation of this constraint are the ones produced by forward checking, as described in previous chapters.
- **GACCBJ+G**: Maintains GAC on all constraints except \textsc{LexLessEq}, learns \textit{g}-nogoods at conflicts and values by maintaining GAC are labeled with \textit{g}-nogoods. The decomposition of the \textsc{LexLessEq} constraint is used and propagated by forward checking, as in GACCBJ+S. Note that the labels produced by forward checking are \textit{s}-nogoods.

Note that the performance of the algorithm that enforces GAC on \textsc{LexLessEq} when doing nogood learning was worse than all the other algorithms presented here, so we do not present results for it. We explain the reasons for this later in this section.

The table lists the instance parameters for each of the instances we tested and the CPU time and the number of nodes visited by each algorithm to solve the problem. The timeout was set to 2000 seconds. A ‘-’ is shown if the algorithm was unable to solve the problem within the timeout.

We see that except for the easiest instances, GACCBJ+G outperforms GAC by as much as three orders of magnitude and GACCBJ+S by as much as two orders of magnitude (instance
Additionally, in several instances GACBJ+S outperforms GAC significantly, even though the overhead of nogood learning is not always overcome by GACBJ+S (see for example the instance 5-6-3).

As noted, these results highlight another instance of the anomaly pointed out in chapter 3 with regard to comparing algorithms of different power that learn nogoods. Even though we can generate propagator specific g-nogoods from LEXLESS_EQ, these g-nogoods are still too long. By using the decomposition in the nogood learning algorithms, we lose pruning power, but we avoid introducing long nogoods into the nogood database, while still retaining most of the symmetry breaking that LEXLESS_EQ performs. This may also point to a practical disadvantage of nogood learning techniques, namely that they complicate the task of designing a model for a problem.

It should be noted that there exist results that supersede those presented here for the social golfer problem, using a different model and more advanced symmetry breaking techniques [Puget, 2002]. However, these symmetry breaking techniques are orthogonal to nogood learning, so can be applied in conjunction with it, but such a combination will not be studied in this thesis.

### 4.3.4 Alldifferent constraint

The alldifferent constraint (ALLDIFF) is perhaps the most well known global constraint. Such a constraint over the variables $V_1, \ldots, V_n$ asserts that each variable must be assigned a distinct value. Several algorithms have been proposed that efficiently enforce various levels of local consistency on it, such as GAC [Régis, 1994], bounds consistency [Puget, 1998, Lopez-Ortiz et al., 2003] or range consistency [Leconte, 1996] (see [van Hoeve, 2001] for a survey). Most of these are based on Hall’s theorem [Hall, 1935] which states that ALLDIFF has a satisfying assignment if and only if there does not exist a set of variables $K$ such that $|K| > |D|$, where $D = \cup_{V \in K} Dom(V)$. For example, let $C = ALLDIFF(X_1, X_2, X_3, X_4, X_5)$, so that $Dom(X_1) = Dom(X_2) = \{1, 2\}$, $Dom(X_3) = Dom(X_4) = \{2, 3\}$ and $Dom(X_5) = \{1, 2, 3, 4, 5\}$. Then the union of the domains of $\{X_1, X_2, X_3, X_4\}$ is $\{1, 2, 3\}$. Clearly, no assignment can assign different values to all 4 variables, thus the constraint is unsatisfiable.

As a corollary to this, whenever we identify a set of variables $K$, $|K| = |D|$, we can remove the values in $D$ from all variables in $scope(C) - K$. Moreover, this is the only reason that we may remove a value. From here on, we will call Hall sets those sets $K$ for
which $|\mathcal{K}| = |\mathcal{D}|$ holds, in accordance with the Hall intervals defined in [Puget, 1998]. For example, let $C = \text{AllDiff}(X_1, X_2, X_3)$ and $\text{Dom}(X_1) = \text{Dom}(X_2) = \{1, 2\}$, $\text{Dom}(X_3) = \{1, 2, 3\}$. Then $\{X_1, X_2\}$ is a Hall set and so we can prune the values 1 and 2 from the domain of $X_3$.

One possible g-nogood for any assignment that was pruned because of the Hall set $\mathcal{K}$ is easily seen to be the set of pruned values of variables in $\mathcal{K}$ outside of $\mathcal{D}$—it is because these values were pruned that $\mathcal{K}$ must consume all of $\mathcal{D}$. This set of pruned values is easy to compute since it uses information already computed by algorithms that enforce GAC on AllDiff, and it can be a small subset of the total set of pruned values of variables in the scope of the AllDiff (which would have been used by the generic method). It is easy to see that such a nogood is minimal. If any non-assignments are removed from it, then more values will be available for the variables in $\mathcal{K}$, so $\mathcal{K}$ is not a Hall set anymore and cannot cause any pruning.

Generalized Arc Consistency. Let us examine here the algorithm that Régin [Régin, 1994] developed for achieving GAC on an AllDiff constraint that runs in $O(n^{2.5})$ time. The algorithm works by using graph theory to identify Hall sets.

We first introduce some notation. A graph $G = (V, E)$ is bipartite with partitions $A$ and $B$ iff for every edge $(u, v) \in E$ it holds that $(u \in A \land v \in B) \lor (u \in B \land v \in A)$. A matching $M$ is a set of edges such that at most one edge in $M$ is incident on any vertex in $V$. A matching is a perfect matching iff $|M| = \min(|A|, |B|)$.

Formally, the propagator for an AllDiff constraint $C$ of $n$ variables with a domain of possible values of size $d$ works as follows. It constructs an undirected bipartite graph $G(C)$ with partitions $A$ and $B$, where $|A| = n$ and $|B| = d$. Each vertex $V \in A$ corresponds to a variable $V$ and each vertex $x \in B$ corresponds to a value $x \in \{1 \ldots d\}$. Note that we use the same notation for both a variable (value) and its corresponding vertex in the graph. There exists an edge $e = (V, x)$ iff $x \in \text{CurDom}(V)$. For convenience, we use the notation $\text{asgn}(e) = V \leftarrow x$ and $\overline{\text{asgn}}(e) = V \not\leftarrow x$. Conversely, we write $e(V \leftarrow x)$ for the edge that connects $V$ and $x$, when $x \in \text{CurDom}(V)$.

**Proposition 4.3.2** (from [Régin, 1994]). Let $C$ be an AllDiff constraint and $G(C)$ be the corresponding bipartite graph. There exists a perfect matching $M$ for $G(C)$ if and only if $C$ is satisfiable.

We can see why this holds. We can construct a satisfying assignment from
a perfect matching by simply making the assignment \textit{asgn}(e), for each \( e \in M \). From a satisfying assignment we can construct a perfect matching \( M = \{ e(V \leftarrow x) \mid V \leftarrow x \text{ in the satisfying assignment} \} \). In other words, if the satisfying assignment assigns \( x \) to \( V \) then the edge between \( V \) and \( x \) is in the matching. This correspondence between a perfect matching and a satisfying assignment is shown in figure 4.18. In addition, it was shown that a value \( V \leftarrow x \) can be part of satisfying assignment to \( C \) if and only if there exists a perfect matching \( M \) so that its \( e(V \leftarrow x) \) belongs to \( M \).

**Proposition 4.3.3** (from [Région, 1994]). Let \( C \) be an \textsc{AllDiff} constraint and \( G(C) \) be the corresponding bipartite graph. There exists a satisfying assignment to \( C \) that includes \( V \leftarrow x \) if and only if there exists a perfect matching \( M \) for \( G(C) \) such that \( e(V \leftarrow x) \in M \).

**Figure 4.18** Correspondence between a matching in bipartite graph and a satisfying assignment to an \textsc{AllDiff} constraint. The partition \( A \) is on the left and \( B \) is on the right. Vertices of \( A \) are labeled with the variable they correspond to, vertices of \( B \) are labeled with the value they correspond to. Bold edges are part of the matching \( M \).

![Diagram](image)

Satisfying assignment to \( C \): \{\( V_1 \leftarrow 1, V_2 \leftarrow 2, V_3 \leftarrow 4, V_4 \leftarrow 3 \}\)

Assuming that the constraint is satisfiable, the propagator needs to remove values that cannot be a part of any satisfying assignment to \( C \), i.e., generalized arc inconsistent values. This means that it needs to detect which edges cannot be a part of \textit{any} perfect matching. It detects these edges by constructing a directed graph \( G'(C) =< V', E' > \) from \( G(C) \) and a perfect matching \( M \) so that \( V' = V \) and \( (u, v) \in E' \) iff \( (u, v) \in E \land (u, v) \in M \land u \in A \) or \( (u, v) \in E \land (u, v) \notin M \land u \in B \). In other words, \( G'(C) \) is constructed from \( G(C) \) by orienting the edges so that an edge that belongs to the matching is directed from \( A \) to \( B \) and the rest are directed from \( B \) to \( A \). Given \( G'(C) \), Région gave a sufficient condition for an edge to be part of a perfect matching.
Proposition 4.3.4 (from [Région, 1994]). Let $C$ be a satisfiable AllDiff, $G(C)$ be the corresponding bipartite graph, $M$ a perfect matching in $G(C)$, and $G'(C)$ the directed graph constructed from $G(C)$ as described above. An edge $(u, v)$ belongs to a perfect matching if $(u, v) \in M$ or there exists a strongly connected component (SCC) $S$ in $G'(C)$ that contains both $u$ and $v$.

To demonstrate this, we revisit the example of figure 4.18. In figure 4.19, we show the oriented graph $G'(C)$ for the same constraint. There are two SCCs in this graph. One contains the vertices $\{V_1, V_2, 1, 2\}$ and the other the vertices $\{V_3, V_4, 3, 4\}$. The edges $(1, V_3)$ and $(3, V_2)$ connect vertices in different SCCs and thus the values $V_3 \leftarrow 1$ and $V_2 \leftarrow 3$ will be pruned.

Figure 4.19 The directed bipartite graph that is constructed for an AllDiff constraint $C$. The edges that belong to the perfect matching $M$ are bold. The strongly connected components are framed, while the edges that cross SCCs are dashed. The values that correspond to these edges will be pruned.

Generating minimal nogoods for AllDiff. From proposition 4.3.4, we see that a value $V \leftarrow x$ can be pruned if $e(V \leftarrow x)$ is not in $M$ and its incident vertices belong to different SCCs. We already know that the existence of a Hall set is the only possible reason for pruning a value, therefore for an SCC $S$, the set of variables $K$ that correspond to the vertices $S \cap A$ is a Hall set. The set of values $D$ that are consumed by the Hall set are those that correspond to the vertices in $S \cap B$. We can then use this Hall set to label pruned values as described earlier. The algorithm for this is given in figure 4.20. It gets as input the constraint $C$, the pruned value $V \leftarrow x$, the graph $G(C)$, the partitions $A$ and $B$ of $G(C)$ and a function $SCC$ that maps a vertex to the strongly connected component it belongs to. This algorithm simply iterates over all vertices in the same
SCC as \( v(V) \) that correspond to some variable \( V' \) (line 2). For each value \( x \) whose vertex is in a different SCC (line 3), it adds \( V' \not\leftarrow x \) to the nogood (line 4). Note that \( V' \leftarrow x \) is a pruned value, therefore no edge exists between \( V' \) and \( x \) in \( G(C) \).

**Figure 4.20** Generating a minimal nogood label for a pruning caused by the GAC propagator for ALLDIFF

```
Alldiff-Nogood(C, V ← x, G(C), A, B, SCC)
1. Ng = \{V ← x\}
2. for all vertices \( V' ∈ (SCC(V) \cap A) - \{V\} \)
3. for all vertices \( x ∈ B \land SCC(x) ≠ SCC(V) \)
4. Ng = Ng ∪ \{V' \not\leftarrow x\}
5. return Ng
```

**Complexity.** The algorithm that maintains GAC for an ALLDIFF constraint \( C \) runs in time \( O(n^{2.5}) \) for each invocation, where \( n = |\text{scope}(C)| \).

Examine now **Alldiff-Nogood.** Let \( K \) be the Hall set that justifies the pruning of \( V ← x \) and \( k = |K| \). Let \( D \) be the set of values that are consumed by the variables in \( K \). We know that \( |K| = |D| \). Then \( |SCC(v(V))| = 2k \), because it includes one vertex for each element of \( K \) and \( D \). The algorithm iterates over \( k − 1 \) vertices in line 2 and over \( d − k \) vertices in line 4. So the total cost is \( t(k) = (d − k)(k − 1) = dk − k^2 − k \). The maximum value of \( t(k) \) is \( d^2/4 − d = O(d^2) \) when \( k = d/2 \). Since \( d = O(n) \), the complexity of **Alldiff-Nogood** is \( O(n^2) \) for each pruned value. Given an SCC which is a Hall set, the nogood that is generated for all values that are pruned because of it is always the same except for \( V ← x \), the pruned value. Therefore, the common part of the labels can be shared by all values that were pruned because of the same SCC. So the complexity of the labeling algorithm is \( O(n^2 s) \), where \( s \) is the number of SCCs in \( G(C) \). Whenever \( s > \sqrt{n} \) labeling the pruned values imposes an overhead over just pruning.

### 4.3.5 Global Cardinality Constraint

The global cardinality constraint GCC is a generalization of the ALLDIFF constraint. Given a vector of \( n \) variables \( V \) and two vectors of \( d \) values each \( lb \) and \( ub \), the constraint states that value \( i \) will appear at least \( lb[i] \) times and at most \( ub[i] \) times. The ALLDIFF constraint is an instantiation of GCC where \( lb = \{0, \ldots, 0\} \) and \( ub = \{1, \ldots, 1\} \).
Since the two constraints are so similar, the techniques to enforce various levels of
dlocal consistency on them are similar. Régin [Régin, 1996] developed an $O(n^2d)$ GAC
algorithm that is based on identifying a maximum flow in a graph. The graph constructed
is similar to that constructed in the propagator for the AllDiff constraint. It is a
directed graph $G(C)$ with $n + d + 2$ vertices. There exists a vertex $V$ for each variable $V$
and a vertex $x$ for each value $x \in \{1 \ldots d\}$, as well as two vertices $s$ and $t$. The vertices
that correspond to variables are called variable vertices, while those that correspond to
values are called value vertices. There exists an edge $e(V, x)$ iff $x \in CurDom(V)$, in which
case $asgn(e) = V \leftarrow x$. Finally, we have edges from $s$ to each value vertex and from each
variable vertex to $t$ and an edge from $t$ to $s$. Each edge is labeled with a lower bound $l(e)$
and an upper bound (capacity) $c(e)$. For each edge connecting a variable vertex $V$ and
a value vertex $u$, $l(u, V) = 0$ and $c(u, V) = 1$. For each edge from a variable vertex $V$ to
t, $l(V, t) = 1$, $c(V, t) = 1$. For each edge connecting $s$ to a value vertex $x$, $l(s, x) = lb[x]$
and $c(s, x) = ub[x]$. Finally, $l(t, s) = 0$ and $c(t, s) = \infty$.

**Theorem 4.3.5** (From [Régin, 1996]). Let $C$ be a GCC and $G(C)$ the graph associated
with it. $C$ is satisfiable, i.e., there exists a consistent assignment to the variables in
scope$(C)$ if and only if there exists a maximum flow in $G(C)$ from $s$ to $t$ with value
$n = |\text{scope}(C)|$.

We use $G_0(C)$ to represent the graph that corresponds to the state of the constraint
where no values have been removed from any domain.

Given a flow $f$ in $G$, we can construct the residual graph $R(f)$ which has the same
vertices as $G$, but we add edges based on the rule that $\forall(u, v) \in E(G)$:

- If $f(u, v) < c(u, v)$, then $(u, v) \in E(R(f))$
- If $f(u, v) > l(u, v)$, then $(v, u) \in E(R(f))$

**Theorem 4.3.6** (From [Régin, 1996]). Let $G$ be a directed graph in which a lower bound
and a capacity is associated with each edge; $f$ be an arbitrary maximum flow in $G$ from $s$
to $t$; and $(u, v)$ an edge in $G$. For all maximum flows $f'$ in $G$ from $s$ to $t$, $f'(u, v) = f(u, v)$
if and only if $(u, v)$ and $(v, u)$ are not contained in a simple cycle of at least 3 vertices in
$R(f) - \{(s, t)\}$.

An edge is in a cycle of at least 3 vertices if it connects vertices that are in the same
strongly connected component.
Corollary 4.3.7 (From [Régis, 1996]). Given a consistent GCC $C$, and a maximum flow $f$ in $G(C)$, the assignment $V \leftarrow x$ is inconsistent with $C$ if and only if $f(V, x) = 0$ and the vertices $V$ and $x$ belong to different strongly connected components in $R(f) - \{(s, t)\}$.

An example demonstrating how values can be found inconsistent in a GCC is shown in figure 4.21.

**Figure 4.21** (a) The graph $G(C)$. Each edge $e$ is labeled by $l(e)$ and $c(e)$. (b) A maximum flow $f$ on $G(C)$. Each edge is labeled by the flow through it. (c) The graph $R(f) - \{s, t\}$. All SCCs are framed and edges that cross SCCs are dashed. These edges can be removed.

Generating minimal nogoods for GCC. Corollary 4.3.7 gives us the tools to label each pruning that is generated by GCC with a minimal nogood. In the following, we refer to the strongly connected component that a vertex $v$ belongs to in $R(f)$ as $scc(v)$. We define the g-nogood $Reason_{gcc}(V \leftarrow x, C) =$
\( \{V \leftarrow x \} \cup \{ V' \neq y : \text{vcc}(V') = \text{vcc}(V), \text{vcc}(y) \neq \text{vcc}(V) \} \). The procedure that uses \( \text{Reason}_{gcc}(V \leftarrow x, C) \) to label \( V \leftarrow x \) is called \textbf{GCC-Nogood}.

The set of edges represented in \( \text{Reason}_{gcc}(V \leftarrow x, C) \) is a minimal set of edges that are present in \( G_0(C) \) but not in \( G(C) \), whose removal is sufficient to create two SCCs in \( R(f) \), one containing \( V \) and the other containing \( x \). This nogood only explains \( V \) and \( x \) being in different SCCs, but does not explain why \( f(V, x) = 0 \). We show that this nogood is sufficient.

\textbf{Proposition 4.3.8.} Given a consistent GCC \( C \), a maximum flow \( f \) in \( G(C) \), and an inconsistent assignment \( V \leftarrow x \), \( \text{Reason}_{gcc}(V \leftarrow x) \) is a minimal nogood.

\textit{Proof.} Observe that \( \text{Reason}_{gcc}(V \leftarrow x) \) is minimal. If we remove any non-assignments from it, \( \text{vcc}(V) \) and \( \text{vcc}(x) \) are the same. Therefore, we only need to prove that it is a nogood.

Consider the constraint \( C_o \), which is the state of \( C \) where no values have been removed from any of the domains of the constrained variables. \( C_{ng} \) is the reduction of the constraint that results from removing the edges described by \( \text{Reason}_{gcc}[V \leftarrow x] - \{V \leftarrow x\} \). The nogood contains a subset of the values that have been pruned, therefore \( G(C) \) is a subgraph of \( G(C_{ng}) \).

We see that \( f \) is a maximum flow in \( G(C_{ng}) \). By construction, the maximum value of \( f \) is \( d \) in any consistent GCC. The value of \( f \) in \( G(C) \) is \( d \), therefore it is also a maximum flow in \( G(C_{ng}) \).

Since \( f(V, x) = 0 \) and \( V \) and \( x \) belong to different strongly connected components, the conditions for the corollary 4.3.7 are satisfied and GAC on \( C_{ng} \) will remove \( V \leftarrow x \). Therefore \( \text{Reason}_{gcc}(V \leftarrow x) \) is a nogood. \( \square \)

\textbf{Complexity.} The algorithm that maintains GAC on a GCC constraint \( C \) runs in time \( O(n^2d) \) for each invocation. To produce \( \text{Reason}_{gcc}(V \leftarrow x, C) \) we need to iterate exactly once over all domain values for each variable in \( \text{vcc}(V) \). Thus the complexity of \textbf{GCC-Nogood} is \( O(nd) \) for each pruned value. As was the case with \textbf{Alldiff-Nogood} we note that if many values are pruned because of the same SCC, then the labels produced for each of them are identical except for the values themselves. So, we can produce the common part of these labels just once and then in constant time reuse it for all pruned values. Note that at most \( n \) SCCs can be identified at each invocation of the algorithm, as each SCC must contain at least one variable vertex. Thus, at most \( O(n^2d) \) labels
will be generated at each invocation and so the labelling algorithm does not impose an overhead when compared to the propagation algorithm.

### 4.3.6 Specifying counting and occurrence problems as combinations of the RANGE and ROOTS constraints

**Set variables and hybrid consistency.** In [Bessière et al., 2005], the authors propose the use of a combination of simple constraints to specify global constraints. These include constraints over set variables as well as non-set variables. A set variable is one whose domain contains sets rather than single elements. There are several ways to represent set or multiset variables in a CSP [Walsh, 2003]. The most straightforward approach is to represent a set variable $S$ as a variable $V_S$ whose domain contains one element for each set in the domain of $S$. However there are potentially an exponential number of sets in the domain of $S$ so this approach is not feasible in practice.

In the rest of this section, we only study methods that examine the lower and upper bounds of sets, so we give a brief overview of a simple representation that is appropriate for this purpose. Let $S$ be a set variable whose upper bound is $ub(S) = \{x_1, \ldots, x_d\}$, that is $ub(S)$ is the maximal set in the domain of $S$. Let $S_1, \ldots, S_d$ be binary variables, such that $S_i = 1 \iff i \in S$, i.e., the variables $S_1, \ldots, S_d$ represent the characteristic function of $S$.

$$
\begin{align*}
\text{CurDom}(S_i) = \{0\} & \iff i \notin S \\
\text{CurDom}(S_i) = \{1\} & \iff i \in S \\
\text{CurDom}(S_i) = \{0, 1\} & \iff i \notin lb(S), i \in ub(S)
\end{align*}
$$

Since representing the actual domain of a set variable is prohibitively expensive, enforcing GAC is not practical either. Therefore, we define here the notion of *hybrid support* and *hybrid consistency*.

**Definition 4.3.9** (Hybrid support). Let $C$ be a constraint such that $\text{scope}(C) = \{V_1, \ldots, V_n, S_1, \ldots, S_m\}$ so that $V_i$ are non-set variables and $S_i$ are set variables. A *hybrid support* in $C$ is a set of assignments $A$ that assigns to each variable in $V_1, \ldots, V_n$...
a value from its current domain and to each set variable in $S_1, \ldots, S_m$ a set between its lower bound and its upper bound.

In analogy with GAC and bounds consistency, a constraint is hybrid consistent if there exists a hybrid support for each value in the domain of each non-set variable and for the lower and upper bound of each set variable.

The constraint language of the Range and Roots constraints. The basic constraints used are the RANGE and ROOTS constraints. They are defined over a set of variables $X_1, \ldots, X_n$ and two set variables $S$ and $T$. The variables $X_1, \ldots, X_n$ are viewed as a function $\mathcal{X}$ that maps the index of a variable to the value that it gets, while $\mathcal{X}^{-1}$ maps a value to the set of variables that have been assigned that value. So, $\mathcal{X}(i) = a \iff X_i \leftarrow a$ and $\mathcal{X}^{-1}(a) = \{X_j \mid X_j \leftarrow a\}$. The RANGE constraint is defined as follows:

$$\text{RANGE}([X_1, \ldots, X_n], S, T) \iff T = \bigcup_{i \in S} \mathcal{X}(i)$$

The ROOTS constraint is:

$$\text{ROOTS}([X_1, \ldots, X_n], S, T) \iff S = \bigcup_{j \in T} \mathcal{X}^{-1}(j)$$

By definition, in any consistent RANGE and ROOTS constraint, $\text{ub}(S) \subseteq \{1, \ldots, n\}$ and $\text{ub}(T) \subseteq \bigcup_{i \in \{1, \ldots, n\}} \text{Dom}(X_i)$.

A wide array of global constraints can be implemented using a conjunction of these constraints and simple integer constraints (e.g., $X < j$) or set constraints (e.g., $|S| = n$, $S_1 \subseteq S_2$). In many cases, enforcing HC on the conjunction can be equivalent to enforcing GAC on the global constraint. A catalog of such global constraints can be found in [Bessière et al., 2006b].

For example, the AMONG([X_1, \ldots, X_n], [d_1, \ldots, d_m], N) constraint [Beldiceanu and Contejean, 1994], which requires that the values in [d_1, \ldots, d_m] appear $N$ times in the variables [X_1, \ldots, X_n]. This constraint can be encoded using a ROOTS constraint and a cardinality constraint on a set:

$$\text{ROOTS}([X_1, \ldots, X_n], S, \{d_1, \ldots, d_m\}) \land |S| = N$$

That is, we fix $T$ to $\{d_1, \ldots, d_m\}$ so that $S$ contains the indices of all the variables that get assigned one of these values and then constrain the cardinality of $S$ to be equal.
to $N$. It turns out that because $T$ is fixed in this Roots constraint, it is possible to enforce hybrid consistency on it in linear time and enforcing hybrid consistency on this encoding also enforces GAC on the original Among constraint.

The propagation algorithms for Range and Roots are described in [Bessi`ere et al., 2006a].

The Range constraint

Hybrid consistency can be enforced for the Range constraint using an $O(nd+n|lb(T)|^{3/2})$ algorithm. The algorithm is shown in 4.22.

\begin{figure}[h]
\centering
\textbf{Figure 4.22 Enforcing HC on the RANGE constraint}

\textbf{Range-HC($X_1, \ldots, X_n, S, T$)}
1. Introduce the variables $Y = \{Y_i : i \in ub(S)\}$ with $D(Y_i) = D(X_i) \cup \{\text{dummy}\}$
2. Enforce HC on the constraint \textsc{Occurs}($Y, T$)
3. Enforce HC on the constraints $i \in S \iff Y_i \in T$, for all $Y_i \in Y$
4. Enforce GAC on the constraints $(Y_i = \text{dummy}) \lor (Y_i = X_i)$, for all $Y_i \in Y$
\end{figure}

This algorithm introduces a set of auxiliary variables $Y_1, \ldots, Y_n$ in line 1, with domains $Dom(Y_i) = Dom(X_i) \cup \{\text{dummy}\}$. The auxiliary constraint \textsc{Occurs} in line 2 enforces that $T$ is a subset of $\bigcup_{i \in 1..n} Y(i)$, without considering $S$:

\[ \textsc{Occurs}([Y_1, \ldots, Y_n], T) \iff T \subseteq \bigcup_{i \in 1..n} Y(i) \]

The constraints in line 3 ensure that $T = \bigcup_{i \in S} X(i)$. Finally, in line 4, GAC is enforced on the constraints $(Y_i = \text{dummy}) \lor (Y_i = X_i)$ for all $i$: each $Y_i$ either has the value dummy or the same value as $X_i$.

Generating minimal nogoods for the Range constraint. Since the algorithm sequentially enforces HC or GAC on simpler constraints, we can treat these simpler constraints in isolation. Then, when one of these simpler constraints prunes a value, it is sufficient to generate a nogood that justifies why the simpler constraint pruned the value and ignore the fact that it acts as part of the more complex RANGE constraint. Unfortunately, even though we generate minimal nogoods from each simpler constraint, they are not necessarily minimal with respect to the RANGE constraint. The reason is that these constraints have overlapping scopes. Thus, there may exist a set of (non-)assignments
that is not a nogood for any of the simpler constraints, but any possible extension would be a nogood for at least one of them.

We examine first the constraints used in steps 3 and 4 and discuss the Occurs constraint which is used in line 2 in the next paragraph.

**Line 3.** The constraint in line 3 is \( i \in S \iff Y_i \in T \) for a fixed \( i \). When \( i \) is pruned from the upper bound of \( S \), it is because the values of \( Y_i \) are either not in \( T \) or they have been pruned. Hence, the nogood label is 
\[
\text{Reason}(S_i \leftarrow 1) = \{ S_i \leftarrow 1 \} \cup \{ T_j \leftarrow 0 : j \in \text{CurDom}(Y_i) \} \cup \{ Y_i \leftarrow x : x \notin \text{CurDom}(Y_i) \}.
\]
When \( i \) is added to the lower bound of \( S \), it is because all remaining values of \( Y_i \) are in \( T \), so the nogood label is 
\[
\text{Reason}(S_i \leftarrow 0) = \{ S_i \leftarrow 0 \} \cup \{ T_j \leftarrow 1 : j \in \text{CurDom}(Y_i) \} \cup \{ Y_i \leftarrow x : x \notin \text{CurDom}(Y_i) \land x \notin T \}.
\]
The condition \( x \notin T \) means that \( T_x \not\in 1 \) is forced. When an assignment \( Y_i \leftarrow x \) is pruned, it is because \( i \in S \) and \( x \notin T \), so the nogood label is 
\[
\text{Reason}(Y_i \leftarrow x) = \{ Y_i \leftarrow x, S_i \leftarrow 1, T_x \leftarrow 0 \}.
\]
When \( x \) is added to the lower bound of \( T \), it is because \( i \in S \) and \( Y_i \leftarrow x \) is forced, so 
\[
\text{Reason}(T_x \leftarrow 0) = \{ Y_i \leftarrow x, S_i \leftarrow 1, T_x \leftarrow 0 \}.
\]
This is exactly the same nogood as that used to label the pruning of \( Y_i \leftarrow x \). Finally note that we do not have to deal with the case of the upper bound of \( T \) being changed. In [Bessière et al., 2006a] it was pointed out that even though this constraint can potentially prune \( \text{ub}(T) \), this does not happen in \text{Range-HC}.

**Line 4.** The constraint in line 4 is a channeling constraint. When \( X_i \leftarrow x \) is pruned, the nogood label is 
\[
\text{Reason}(X_i \leftarrow x) = \{ X_i \leftarrow x, Y_i \not\leftarrow x, Y_i \not\leftarrow \text{dummy} \}.
\]
When \( Y_i \leftarrow x \) is pruned the nogood label is 
\[
\text{Reason}(Y_i \leftarrow x) = \{ Y_i \leftarrow x, X_i \not\leftarrow x, Y_i \not\leftarrow \text{dummy} \}.
\]
The assignment \( Y_i \leftarrow \text{dummy} \) is never pruned by this constraint.

**The Occurs constraint.** The \text{Occurs}(\{X_1, \ldots, X_n\}, T) constraint, which is used in line 2, is propagated by an \( O(nd + n|\text{lb}(T)|^{3/2}) \) algorithm, which is based on network flows, similar to GCC and \text{ALLDIFF}. To achieve HC on an \text{Occurs} constraint \( C \), we construct the value graph \( G_{\text{Occurs}}(C) \). There exists a vertex \( s \) and a vertex \( t \), which are the source and sink, respectively. There exists a vertex \( Y_i \) for each variable \( Y_i \), a vertex \( d_i \) and a vertex \( T_{d_i} \) for each value \( d_i \) in the \( \cup_j \text{Dom}(Y_j) - \{ \text{dummy} \} \). There exists an edge from \( d_j \) to \( Y_i \) if \( d_j \in \text{CurDom}(Y_i) \) and an edge from \( d_i \) to \( T_{d_i} \) if \( d_i \notin \text{lb}(T) \). Finally, there exists an edge from \( s \) to \( d_i \) for all \( d_i \), an edge from each \( T_{d_i} \) to \( t \) and from each \( Y_i \) to \( t \). All edges have capacity 1. The \text{Occurs} constraint is satisfiable if and only if there
exists a flow $f$ from $s$ to $t$ with value $|\bigcup_j \text{Dom}(Y_j) - \{\text{dummy}\}|$. By theorem 4.3.6, an edge $e$ such that $f(e) = 0$ may have a non-zero flow in a maximum flow if both its edges belong in the same SCC of the graph $R_f - \{s,t\}$. If it is not part of a maximum flow, then it is removed. If an edge $(d_i, Y_j)$ is removed, then the value $Y_j \leftarrow d_i$ is pruned. If an edge $(d_i, T_{d_i})$ is removed, then $d_i$ is added to the lower bound of $T$, which means that $T_{d_i} \leftarrow 0$ is pruned.

**Generating minimal nogoods for the Occurs constraint**  
By proposition 4.3.8, we see that we can produce a minimal set of edges that explains an edge removal by identifying the SCCs of the residual graph $R_f$ for any maximum flow $f$. This theorem is suitable for use in OCCURS, because the graph has the same structure as the one we construct for GCC. Let $e$ be an edge that must be removed and $S(e)$ be the set of edge removals that explains this. We define the function $h(e)$ that maps an edge to a set with one (non-)assignment, so that $h((d_i, Y_j)) = \{Y_j \leftarrow d_i\}$ and $h((d_i, T_{d_i})) = \{T_i \leftarrow 0\}$. For a set $S$, we define $h(S) = \bigcup_{e \in S} h(e)$. The nogood used to label a value pruned by OCCURS is then $\{Y_j \leftarrow d_i\} \cup h(S(d_i, Y_j))$ for an edge $(d_i, Y_j)$ and $\{T_{d_i} \leftarrow 0\} \cup h(S(d_i, T_{d_i}))$ for an edge $(d_i, T_{d_i})$.

Bounds consistency is achieved on the variable $T$ by ensuring that $\text{ub}(T) \subseteq \bigcup_i \text{Dom}(X_i)$. We see that when an element is removed from $\text{ub}(T)$, it is because none of the variables $X$ can get it as a value. Thus the nogood label for that pruning is $\text{Reason}(T_x \leftarrow 1) = \bigcup_i \text{Reason}(X_i \leftarrow x) - \{X_i \leftarrow x\}$.

**The Roots constraint**

It is NP-hard to achieve HC on the ROOTS constraint. However, the authors of [Bessière et al., 2006a] proposed an $O(nd)$ algorithm that enforces a level of consistency that is at least as strong as bounds consistency. This algorithm enforces the following six properties:

1. if $D(X_i)$ does not intersect $\text{ub}(T)$ then $i$ is outside $\text{ub}(S)$

2. if $D(X_i)$ is included in $\text{lb}(T)$ then $i$ is in $\text{lb}(S)$

3. if $i$ is in $\text{lb}(S)$ then $D(X_i)$ is included in $\text{ub}(T)$

4. if $i$ is outside $\text{ub}(S)$ then $D(X_i)$ does not intersect $\text{lb}(T)$
5. if \( v \) is the only possible value for \( X_i \) and \( i \in lb(S) \) then \( v \) is in \( lb(T) \)

6. if \( v \) is the only possible value for \( X_i \) and \( i \notin ub(S) \) then \( v \) is outside \( ub(T) \)

It is straightforward to create an algorithm that simply enforces these properties on each invocation by simply checking each of them in turn. Such an algorithm would be suboptimal, so the algorithm proposed in [Bessière et al., 2006a] enforces the properties incrementally. The details of the algorithm do not concern us here, however.

**Generating minimal nogoods for the Roots constraint.** We encode the properties above as nogoods. Since a value may be pruned only as a result of violating one of these properties, this means that for every pruning, we can identify one of the following nogoods as a valid reason for it. These nogoods are minimal with respect to this propagation algorithm (which does not enforce GAC), in the sense of definition 4.2.1. However, since enforcing GAC on the Roots constraint is \( NP \)-hard, it is \( coNP \)-hard to discover a nogood that is minimal with respect to the constraint, in the sense of definition 4.2.2.

1. \( \{ X_i \not\leftarrow j : j \notin CurDom(X_i) \land j \notin T \} \cup \{ T_j \not\leftarrow \ 1 : j \in CurDom(X_i) \} \cup \{ S_i \leftarrow 1 \} \)

   This encodes the condition that the values of \( X_i \) that have not been pruned are not part of \( T \) and that all elements of \( T \) that are in \( CurDom(X_i) \) have been removed from \( ub(T) \). We take care not to mention values that have been pruned but that are not part of \( T \) either. Note that in this expression \( j \notin T \) means that the assignment \( T_j \leftarrow 1 \) has been pruned.

2. \( \{ X_i \not\leftarrow j : j \notin CurDom(X_i) \land j \notin T \} \cup \{ T_j \not\leftarrow \ 0 : j \in CurDom(X_i) \} \cup \{ S_i \leftarrow 0 \} \)

   This is symmetric to the nogood for condition 1.

3. \( \{ S_i \not\leftarrow 0, T_j \not\leftarrow 1, X_i \leftarrow j \} \forall j \notin T \)

   Unlike the first two properties, each of which causes a single pruning, this property may cause several values of \( X_i \) to be removed at once. Specifically, it prunes values \( j \) of \( X_i \) that are outside \( ub(T) \). Note that \( T \) cannot be modified when we enforce this property.

4. \( \{ S_i \not\leftarrow 1, T_j \not\leftarrow 1, X_i \leftarrow j \} \forall j \in T \)

   This is symmetric to property 3. The expression \( j \in T \) means that the assignment \( T_j \leftarrow 0 \) has been pruned.
5. \( \{X_i \leftarrow v, S_i \neq 0, T_v \leftarrow 0\} \)

6. \( \{X_i \leftarrow v, S_i \neq 1, T_v \leftarrow 1\} \)

Properties 5 and 6 both involve conditions that can be expressed as single (non-)assignments.
Chapter 5

Logical combinations of constraints

It is often the case that CSPs are not expressed by a user directly, but written in a high level modeling language and then converted to a CSP. One of the features common in such languages is the ability to express a problem not only as a simple conjunction of constraints but as a formula that in addition to and uses other logical connectives such as or and not. We discuss here an approach that was proposed recently to take advantage of this structure to achieve better propagation. We build on this work by making it work with the g-nogood learning techniques discussed in previous chapters. We also identify an opportunity to further generalize nogoods and therefore further improve performance.

5.1 The constraint language

The approach we discuss is that of [Bacchus and Walsh, 2005]. In it, the authors suggest a way to specify and propagate constraint expressions using combinations of primitive constraints with the and, or, not logical operators. For example, if $C_1$, $C_2$ and $C_3$ are primitive constraints, we can write an expression such as $C = \text{and}(C_1, \text{or}(C_2, \text{not}(C_3)))$. A constraint expression is itself a constraint and can be used to build further expressions. In this example, the constraint expression $C$ is built up from the primitive constraint $C_1$ and the expression $\text{or}(C_2, \text{not}(C_3))$. A constraint expression is defined over a set of variables, denoted $\text{scope}(C)$ and a set of domains for these variables, $D$. We write $D(X)$ for the domain of variable $X$. A set of assignments $\tau \in D(X_1) \times \ldots \times D(X_n)$ satisfies a constraint expression $C$ if and only if

- $C$ is a primitive constraint $C$ and $\tau \cap \text{scope}(C)$ satisfies $C$. 

118
• \( C = \text{and}(C_1, C_2) \) and \( \tau \) satisfies \( C_1 \) and \( C_2 \).

• \( C = \text{or}(C_1, C_2) \) and \( \tau \) satisfies at least one of \( C_1 \) and \( C_2 \).

• \( C = \text{not}(C') \) and \( \tau \) does not satisfy \( C' \).

An assignment \( X_i \leftarrow a \) is \textbf{inconsistent} with a constraint expression \( C \) in the domain \( D \) if there exists no set of assignments \( \tau \in D(X_1) \times \ldots \times D(X_{i-1}) \times \{X_i \leftarrow a\} \times D(X_{i+1}) \times \ldots \times D(X_n) \) such that \( \tau \) satisfies \( C \). Similarly, an assignment \( X_i \leftarrow a \) is \textbf{valid} for a constraint expression \( C \) in the domain \( D \) if there exists no set of assignments \( \tau \in D(X_1) \times \ldots \times D(X_{i-1}) \times \{X_i \leftarrow a\} \times D(X_{i+1}) \times \ldots \times D(X_n) \) such that \( \tau \) falsifies \( C \). We can see that, by definition, if \( C \) is a primitive constraint \( C \), an assignment is inconsistent with \( C \) in \( D \) if and only if it is GAC inconsistent with \( C \). On the other hand, an assignment is valid for the constraint expression \( C \) if it is GAC inconsistent with the negation of the primitive constraint \( C \), i.e., the primitive constraint that has the same scope as \( C \) and that is falsified (satisfied) by every set of assignments \( \tau \) that satisfies (falsifies) \( C \).

This is easy to see: if no \( \tau \) that includes \( X_i \leftarrow a \) falsifies \( C \), then no \( \tau \) that includes \( X_i \leftarrow a \) satisfies \( \neg C \), therefore the assignment is GAC inconsistent for \( \neg C \). For each assignment \( X_i \leftarrow a \) that is not inconsistent with an expression \( C \), there exists a witness \( \tau \) that includes \( X_i \leftarrow a \) and satisfies \( C \) and for each assignment \( X_i \leftarrow a \) that is not valid for \( C \), there exists a witness \( \tau \) that includes \( X_i \leftarrow a \) and falsifies \( C \). If \( C \) is a primitive constraint \( C \), then the witness for a non-inconsistent assignment \( X_i \leftarrow a \) is a support, while the witness for a non-valid assignment is a support in \( \neg C \).

In order to perform propagation, we need to compute for each constraint expression \( C \) a set of assignments \( \text{Inc}(C, D) \) which contains assignments inconsistent with \( C \) in \( D \) and \( \text{Valid}(C, D) \), which contains assignments valid for \( C \) in \( D \). There are many sets that are legal \( \text{Inc} \) and \( \text{Valid} \) sets, including the empty set. There exist unique maximal sets \( \text{MaxInc}(C, D) \) and \( \text{MaxValid}(C, D) \). The closer the computed sets \( \text{Inc} \) and \( \text{Valid} \) are to \( \text{MaxInc} \) and \( \text{MaxValid} \), the stronger the constraint propagation that is performed. In fact, if \( C \) is a primitive constraint \( C \), \( \text{MaxInc} \) is the set of all assignments that are GAC inconsistent and \( \text{MaxValid} \) is the set of all assignments that are GAC inconsistent for the negation of the constraint.

It is not always practical to compute \( \text{MaxInc} \) and \( \text{MaxValid} \). It is easy to see that it is \textit{coNP}-hard in general to determine whether a specific assignment is in \( \text{MaxInc} \) or \( \text{MaxValid} \). A CSP \( P \) can be seen as a constraint expression that is simply a conjunction
of many primitive constraints. Thus, to determine whether \( \{ X \leftarrow a \} \) is a nogood for \( P \), which is \( \text{coNP}\)-hard (proposition 2.1.3), can be reduced to determining whether \( X \leftarrow a \) is in \( \text{MaxInc} \). Conversely, it is \( \text{NP}\)-hard to determine whether \( X \leftarrow a \) is not in \( \text{MaxInc} \), as we need to find a set of assignments \( \tau \) that satisfies the constraint expression and this is equivalent to determining whether there exists a solution in \( P \) that includes \( X \leftarrow a \). By these two facts, we can tell that to determine whether a given set \( \text{Inc} \) or \( \text{Valid} \) is maximal is complete for the class \( D^P \), which is the intersection of \( \text{NP} \) and \( \text{coNP} \).

We reproduce in figure 5.1 the functions that are proposed in [Bacchus and Walsh, 2005] to compute \( \text{Inc}(C, D) \) and \( \text{Valid}(C, D) \) for a constraint expression.

**Figure 5.1** Computing \( \text{Valid}(C, D) \) and \( \text{Inc}(C, D) \) for constraint expressions

\[
\begin{align*}
\text{Inc}(\text{not}(C_1), D) &= \text{Valid}(C_1, D) \\
\text{Valid}(\text{not}(C_1), D) &= \text{Inc}(C_1, D) \\
\text{Inc}(\text{or}(C_1, \ldots, C_k), D) &= \bigcap_i \text{Inc}(C_i, D) \\
\text{Valid}(\text{and}(C_1, \ldots, C_k), D) &= \bigcap_i \text{Valid}(C_i, D) \\
\text{Inc}(\text{and}(C_1, \ldots, C_k), D) &= \bigcap \text{Inc}(C_i, D) \\
\text{Valid}(\text{or}(C_1, \ldots, C_k), D) &= \bigcap \text{Valid}(C_i, D) \\
\text{ItInc}(\text{and}(C_1, \ldots, C_k), D) &= \text{ItValid}(\text{or}(C_1, \ldots, C_k), D)
\end{align*}
\]

The functions \( \text{Inc}(\text{not}(C_1), D) \) and \( \text{Valid}(\text{not}(C_1), D) \) are derived directly from the definitions of \( \text{Inc} \) and \( \text{Valid} \). Similarly, the functions \( \text{Inc}(\text{or}(C_1, \ldots, C_k), D) \) and \( \text{Valid}(\text{and}(C_1, \ldots, C_k), D) \) are obvious from the definitions: an assignment is inconsistent for a disjunction if it is inconsistent for all disjuncts and it is valid for a conjunction if it is valid for all conjuncts. Finally, \( \text{Inc}(\text{and}(C_1, \ldots, C_k), D) \) and \( \text{Valid}(\text{or}(C_1, \ldots, C_k), D) \) are more complex in that they need to be iteratively computed. Clearly, any assignment that is inconsistent for a conjunct is also inconsistent for the conjunction and any as-
Chapter 5. Logical combinations of constraints

Assignment that is valid for a disjunct is valid for the disjunction. However, an assignment can be inconsistent (valid) for a conjunction (disjunction) even though it is consistent (invalid) for every individual constraint in the conjunction (disjunction). The function above exploits the fact that the constraints can be examined iteratively to discover more inconsistent/valid assignments, although not all such assignments can be detected. For example, consider the expression $C = \text{and}(C_1, C_2)$. Assume that we find that $X \leftarrow a$ is not inconsistent for $C_1$ and $C_2$ in $D$ and the witness for $C_1$ contains $Y \leftarrow b$ and the witness for $C_2$ contains $Z \leftarrow c$. Assume that we now examine $Y \leftarrow b$ in $C_2$ and find that it is inconsistent. Then, $Y \leftarrow b$ is inconsistent for $C$ as well, which means that the witness for $X \leftarrow a$ being true is no longer valid. We now need to reexamine whether $X \leftarrow a$ is consistent for $C_1$ and $C_2$. This is captured by the functions $\text{ItInc}$ and $\text{ItValid}$. Note that if $C = \text{and}(C_1, \ldots, C_n)$, where $C_1, \ldots, C_n$ are primitive constraints, then the computation of $\text{Inc}$ is equivalent to maintaining GAC using $\text{enforce-gac}$ (section 2.1.1, figure 2.3).

5.2 Computing nogoods

The work of [Bacchus and Walsh, 2005] does not address the issue of generating nogood labels for assignments in $\text{Inc}$ and $\text{Valid}$ of a constraint expression. We extend it so that we can compute this information. In order to do this we need to modify the notion of a nogood as presented in previous chapters and introduce goods.\footnote{Contrary to the term nogood, which carries the same meaning in research areas related to constraint satisfaction, the meaning of the term good is not as well established, but other alternatives seem more awkward.} At the intuitive level nogoods are as before, a set of assignments (or non-assignments) that cannot be part of any extended satisfying assignment for the constraint expression. A good, on the other hand, is a set of assignments that cannot be extended to a falsifying assignment. A good can be used to explain why an assignment is valid and to label said assignment, just as a nogood explains why an assignment is inconsistent and is used to label it. To define this more formally, we first note that any set of assignments $\tau$ can be expanded in a way similar to that described by definition 3.1.2, by adding to $\tau$ all non-assignments in the domain $D$ that are not included in $\tau$.

Definition 5.2.1. An expanded set of assignments $\text{expand}(\tau)$ is a set of assignments that includes all the implied non-assignments to variables in $\text{scope}(\tau)$.\footnote{Contrary to the term nogood, which carries the same meaning in research areas related to constraint satisfaction, the meaning of the term good is not as well established, but other alternatives seem more awkward.}
For example, the assignment \( \tau = \{ X \leftarrow a, Y \leftarrow b \} \) with \( D(X) = D(Y) = \{ a, b \} \) is expanded to \( \tau = \{ X \leftarrow a, X \not\leftarrow b, Y \leftarrow b, Y \not\leftarrow a \} \).

A nogood \( n = \text{Nogood}(X \models a, C, D) \) is a set of (non-)assignments \( n \) that includes \( X \models a \) such that \( \forall \tau \in \times_{D \in D} (\text{scope}(C) \subseteq \text{scope}(\tau) \land \text{expand}(\tau) \supseteq n) \rightarrow \neg C(\tau) \). That is, every set of assignments in the cross product of the domains \( D \) that is a superset of \( n \) must falsify \( C \). A good \( g = \text{Good}(X \models a, C, D) \) is a set of (non-)assignments that includes \( X \models a \) such that \( \forall \tau \in \times_{D \in D} (\text{scope}(C) \subseteq \text{scope}(\tau) \land \text{expand}(\tau) \supseteq g) \rightarrow C(\tau) \). This means that every set of assignments in the cross product of the domains \( D \) that is a superset of \( g \) must satisfy \( C \). Note that because valid and inconsistent assignments are defined in terms of the constraint expression and the set of domains examined, we cannot reuse the notation \( \text{Reason}(X \leftarrow a) \) from the previous chapters. Instead, we need to account for the fact that the domains \( D \) may be different from the current domains of the variables and have to use the notation \( \text{Nogood}(A, C, D) \) and \( \text{Good}(A, C, D) \).

In figure 5.2, we present functions that can calculate nogoods and goods for each assignment in \( \text{Inc}(C, D) \) and \( \text{Valid}(C, D) \), respectively. The functions compute the nogood and good for an assignment or non-assignment for an expression \( C \) in domains \( D \). We use \( A \) to denote either an assignment \( X \leftarrow a \) or a non-assignment \( X \not\leftarrow a \). The function \( \text{Nogood}(A, C, D) \) (\( \text{Good}(A, C, D) \)) is used when it has been established from the functions in figure 5.1 that \( A \in \text{Inc}(C, D) \) (\( A \in \text{Valid}(C, D) \)).

**Figure 5.2** Computing \( \text{Nogood}(A, C, D) \) and \( \text{Good}(A, C, D) \) for any assignment or non-assignment \( A \)

<table>
<thead>
<tr>
<th>Function</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Nogood}(A, C, D) )</td>
<td>( \text{Nogood}(A, C, D), C \text{ is primitive } C )</td>
</tr>
<tr>
<td>( \text{Good}(A, C, D) )</td>
<td>( \text{Nogood}(A, \neg C, D), C \text{ is primitive } C )</td>
</tr>
<tr>
<td>( \text{Nogood}(A, \text{not}(C_1), D) )</td>
<td>( \text{Good}(A, C_1, D) )</td>
</tr>
<tr>
<td>( \text{Good}(A, \text{not}(C_1), D) )</td>
<td>( \text{Nogood}(A, C_1, D) )</td>
</tr>
<tr>
<td>( \text{Nogood}(A, \text{or}(C_1, \ldots, C_k), D) )</td>
<td>( \bigcup_i \text{Nogood}(A, C_i, D) )</td>
</tr>
<tr>
<td>( \text{Nogood}(A, \text{and}(C_1, \ldots, C_k), D) )</td>
<td>( \text{Choose } \text{Nogood}(A, C_i, D) )</td>
</tr>
<tr>
<td>( \text{Good}(A, \text{or}(C_1, \ldots, C_k), D) )</td>
<td>( \text{Choose } \text{Good}(A, C_i, D) )</td>
</tr>
<tr>
<td>( \text{Good}(A, \text{and}(C_1, \ldots, C_k), D) )</td>
<td>( \bigcup_i \text{Good}(A, C_i, D) )</td>
</tr>
</tbody>
</table>

There are two distinct cases: the first is that \( C \) is a primitive constraint \( C \), covered in the first two lines of figure 5.2. We can use the techniques in chapter 4 to compute a nogood for \( A \) from the constraint \( C \). In order to compute a good, we treat \( \neg C \) as a primitive
constraint and once again use the techniques described in chapter 4 to compute a nogood for \( A \) from \( \neg C \). This is then a good for \( A \) from \( C \). For example, consider a constraint \( C \) that is represented in extension, as a set of allowed tuples \( T \). \( C \) would be propagated using \text{GAC-Schema}, and nogoods calculated using \text{Nogood-GAC-Schema-Allowed}. The set \text{Valid} would be computed by running \text{GAC-Schema} on the negation of \( C \), which is the constraint \( C' \) that is described in extension by the same set of forbidden tuples \( T \). We can then compute a nogood for \( C' \) using \text{Nogood-GAC-Schema-Conflicts} and use that as a good for the corresponding assignment in \text{Valid}(C,D).

The second case is that \( C \) is not a primitive constraint. We recursively apply the appropriate function from the second part of figure 5.2 to the top level term of the constraint expression. It is not hard to verify that these recursive invocations produce correct goods and nogoods for each type of expression: \text{not}, \text{and}, \text{or}. By definition, it holds that a good for \( C \) is a nogood for \text{not}(C) and vice versa. For any assignment to be inconsistent in a disjunction of constraints, it must be inconsistent for each constraint separately, so the nogood for the disjunction is the union of the nogoods for each constraint. Similarly, an assignment is valid for a conjunction iff it is valid for each constraint, thus the good for the conjunction must be the union of the goods for each constraint. Finally, an assignment is inconsistent for a conjunction if it is inconsistent for any individual constraint, thus we are free to choose any reported nogood from those constraints for which the assignment is inconsistent. The same reasoning applies for the case of a valid assignment for a disjunction of constraints.

5.2.1 Entailment and disentailment

Further simplification of constraint expressions is achieved by observing that if an expression is disentailed in \( D \), i.e., no set of assignments from the domain \( D \) can satisfy this expression, it can be replaced by the constant \text{false}. Similarly, if an expression has become entailed, i.e., no set of assignments from \( D \) may falsify the expression, then it can be replaced by the constraint \text{true}. These conditions can be easily checked. If \text{Inc}(C,D) (resp. \text{Valid}(C,D)) contains all the assignments in \text{scope}(C), then the constraint is disentailed (resp. entailed). This leads to the following augmented rule for the computation of \text{Inc} and \text{Valid}: if the set \text{Inc}(C,D) (resp. \text{Valid}(C,D)) contains all the assignments in the scope of \( C \), then we can replace \text{Inc}(C,D) (resp. \text{Valid}(C,D)) by \( D \) in computations of \text{Inc} (Valid). Recall that \text{scope}(C) \subseteq \text{scope}(D),
therefore $D$ may contain assignments to variables outside the scope of $C$, so this rule may simplify expressions and thus increase propagation. Consider the example from [Bacchus and Walsh, 2005] of the expression $C = or(not(even(X)), odd(Y))$ with the domain $D = \{ \{X \leftarrow 0, X \leftarrow 2\}, \{Y \leftarrow 1, Y \leftarrow 2\} \}$. The functions in figure 5.1 calculate that $Inc(C, D) = \emptyset$. Note however that $even(X)$ is entailed, thus $not(even(X))$ is disentailed. By setting $Inc(not(even(X)), D) = D$, we are able to compute that $Inc(C, D) = \{Y \leftarrow 2\}$.

We need to adjust the computation of nogoods to account for the entailment and disentailment rules. Let us first examine how disentailment affects computation of nogoods. We will then apply the same reasoning to see how entailment affects good computation.

Consider the expression $C = or(C_1, \ldots, C_j, \ldots, C_k)$ and an assignment $A \in Inc(C, D)$. Assume that $C_j$ is disentailed under $D$. There are two possibilities: either $\text{var}(A) \in \text{scope}(C_j)$ or not. In the former case, we can just use $\text{Nogood}(A, C_j, D)$ in our computation, as shown in figure 5.2. In the latter case however, $C_j$ is unable to provide a nogood for $A$. We need instead to compute a nogood for why $C_j$ is disentailed and use that instead of $\text{Nogood}(A, C_j, D)$. One possible choice is to pick a variable $V \in \text{scope}(C_j)$ and use the nogood $\cup_{x \in \text{Dom}(V)}(\text{Nogood}(V \leftarrow x, C_j, D) - \{V \leftarrow x\})$. This is valid, because wiping out the domain of any variable in the scope of $C_j$ is enough to make it disentailed. However, this presents a problem in that our choice of which variable to use to create the nogood may affect its minimality. It is also conceivable that no choice would yield a minimal nogood, even though each individual nogood for the assignments of $V$ is minimal. It would be better if a constraint could report a minimal reason for why it has become disentailed. Therefore, when a constraint is disentailed, the expression $\text{Nogood}(C, D)$ is a reason why it has become so. Even if a primitive constraint is not able to provide $\text{Nogood}(C, D)$, the computation can fall back to the procedure described above. This suggests that one could use this potentially improved nogood even in the case when $\text{var}(A) \in \text{scope}(C_j)$.

### 5.2.2 Minimality

The functions to compute nogoods from figure 5.2 imply at several points that we have to make a choice between two or more possible nogoods, but no indication is given on how to choose. A simple heuristic is to choose the shortest nogood or one that involves the smallest number of variables. But that is not necessarily the best choice.
Even when no choice is involved, it is not easy to produce a minimal nogood. For example, consider a conjunction of primitive constraints. Each constraint generates a minimal nogood. SAT can be reduced to this problem, so that each constraint in the expression is exactly a clause. Therefore, determining a minimal nogood for even a simple expression is NP-hard.

5.3 GAC-Schema as a constraint expression

We examine now the behaviour of this language in terms of both propagation and the nogoods generated compared to the GAC-Schema constraint presented in the previous chapter, as the extentional versions of GAC-Schema can be expressed as constraint expressions. This is a meaningful exercise as it shows the limits of the method described in section 5.2.

Observe that GAC-Schema-Allowed is a disjunction of conjunctions of primitive constraints \( V_i = j \). For example the constraint over \( V_1, V_2, V_3 \) with allowed tuples \( \{\{1,2,3\},\{2,3,4\}\} \) is equivalent to the expression \( C = or(and(V_1 = 1, V_2 = 2, V_3 = 3), and(V_1 = 2, V_2 = 3, V_3 = 4)) \). First, we observe that \( Inc(C, D) = MaxInc(C, D) \) because, according to [Bacchus and Walsh, 2005], it satisfies the following necessary conditions:

- We can compute \( MaxInc(V_i = j, D) \).
- The equality constraints in the conjunction \( and(V_1 = i_1, \ldots, V_k = i_k) \) have no variables in common, therefore we can compute \( MaxInc \) for these subexpressions as well.
- The top level is a disjunction.

The nogood generated for this expression is equivalent to the one generated by Nogood-GAC-Schema-Allowed-naive, shown in figure 4.11. Specifically, exactly one assignment is chosen from each removed tuple. This is potentially bigger than the nogood generated by more specialized functions such as Nogood-GAC-Schema-Allowed-logapprox. This suggests that the nogood computing function shown in figure 5.2 can be improved using the same methods. However, the technique only applies to disjunctions of conjunctions, so has limited applicability.
Chapter 5. Logical combinations of constraints

The same results do not hold for **GAC-Schema-Conflicts**, which is a conjunction of disjunctions. The constraint over variables \(V_1, V_2, V_3\) with disallowed tuples \(\{1, 2, 3\}, \{2, 3, 4\}\) is equivalent to the expression \(C = \text{and}(\text{or}(V_1 \neq 1, V_2 \neq 2, V_3 \neq 3), \text{or}(V_1 \neq 2, V_2 \neq 3, V_3 \neq 4))\). Since the scope of each of the disjunctions is the same, we cannot compute \(\text{MaxInc}\) for the conjunction. In fact, propagation on the expression is only as powerful as applying forward checking to the constraint. The nogood computed in that case is exactly the disallowed tuple that would cause the pruning.

### 5.4 Further generalization of nogoods

Consider the expression \(\text{or}(C_1, C_2)\), where \(C_1\) is disentailed and \(A \in \text{Inc}(C_2)\). Recall that \(C_1\) being disentailed means that there exists no assignment that can satisfy the constraint, therefore the constraint is equivalent to the constant \(\text{false}\). We compute a nogood for \(A\), \(\text{Nogood}(A, \text{or}(C_1, C_2), D) = \text{Nogood}(C_1, D) \cup \text{Nogood}(A, C_2, D)\). Notice that \(C_1\) reports a specific reason for why it is disentailed and we use that in the computation of \(\text{Nogood}(A, \text{or}(C_1, C_2), D)\). However, any time that \(C_1\) is disentailed, and \(\text{Nogood}(A, C_2, D)\) is true, we can prune \(A\). There may exist an exponential in the arity of \(C_1\) number of nogoods that explain the fact that \(C_1\) is disentailed, but each computed nogood can only use one of these.

We can then further generalize nogoods to capture the fact that a constraint is entailed or disentailed by reifying constraint expressions. In the example above we can compute the nogood \(\{\neg C_1\} \cup \text{Nogood}(A, C_2, D)\). The intuitive meaning of this is that whenever \(C_1\) is disentailed and the assignments in \(\text{Nogood}(A, C_2, D)\) are true, there is no solution. We call nogoods of this form **c-nogoods**. This generalization corresponds to implicitly adding the boolean variables \(V_{C_1} \equiv C_1\) for each constraint, which we call a **c-variable**.

Further, we can rewrite the nogood computed above as \(\{\neg C_1, \neg C_2 | A\}\). We write \(C_2 | A\) to denote the reduction of the constraint \(C_2\) to \(A\). Intuitively, this means that when \(C_1\) is disentailed and \(A \in \text{Inc}(C_2)\), there is no solution. This corresponds to implicitly adding the boolean c-variables \(V_{C_1 | A} \equiv (C_1 \land A)\) for all constraints and assignments.

In figure 5.3 we show the functions that compute c-nogoods and c-goods.

As is the case when comparing g-nogoods to s-nogoods, we see that c-nogoods are more expressive than g-nogoods. The reason is fundamentally the same: when we choose to resolve \(\text{Reason}(V \leftarrow a)\) with \(A \cup \{V \neq a\}\) to get \(A \cup \text{Reason}(V \leftarrow a) - \{V \neq a\}\), we represent only a single possible reason for the pruning of \(V \leftarrow a\), among a potentially
Figure 5.3 Computing Nogood($A, C, D$) and Good($A, C, D$), using c-nogoods.

<table>
<thead>
<tr>
<th>Nogood($A, \neg(C_1), D$)</th>
<th>=  Good($A, C_1, D$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nogood($A, C_1, D$)</td>
<td>=  Nogood($A, C_1, D$)</td>
</tr>
<tr>
<td>Nogood($A, \lor(C_1, \ldots, C_k), D$)</td>
<td>=  $\cup_i$(Choose ${\neg C_i \mid A}, {\neg C_i}$)</td>
</tr>
<tr>
<td>Nogood($A, \land(C_1, \ldots, C_k), D$)</td>
<td>=  Choose $\neg C_i \mid A, C_i$</td>
</tr>
<tr>
<td>Good($A, \lor(C_1, \ldots, C_k), D$)</td>
<td>=  Choose $C_i \mid A, C_i$</td>
</tr>
<tr>
<td>Good($A, \land(C_1, \ldots, C_k), D$)</td>
<td>=  $\cup_i$(Choose ${C_i \mid A, C_i}$)</td>
</tr>
</tbody>
</table>

exponential number of them, most of which we have not yet discovered. Similarly, with c-nogoods, we can use the following implied nogoods to resolve away the extra variables.

\[ \{ \text{Nogood}(C_i, D), V_{C_i} \} \]
\[ \{ \text{Good}(C_i, D), \neg V_{C_i} \} \]
\[ \{ \text{Nogood}(A, C_i, D), V_{C_i \mid A} \} \]
\[ \{ \text{Good}(A, C_i, D), \neg V_{C_i \mid A} \} \]  

(5.1)

If we choose to use Nogood($A, C_1, D$) instead of $\neg C_1 \mid A$, we represent a specific reason for the pruning of $A$ by $C_1$ among a potentially exponential number of them. In fact we can state the following, similarly to proposition 3.2.2.

**Proposition 5.4.1.** Let $P$ be a CSP $P = \langle V, D, C \rangle$, $P_c$ be $P$ augmented with the boolean variables $V_{C_i}$ and $V_{C_i \mid A}$ for all $i$, and a partial assignment $A$ that causes a conflict after constraint propagation and unit propagation, while no subset of $A$ satisfies this condition. Let constraint propagation label prunings with c-nogoods in $P_c$. Consider now the two nogoods $NG$ and $NG_c$ such that $NG_c$ can be derived from the conflict by trivial resolution (defined in section 2.2) in $P_c$ and $NG$ can be derived from $NG_c$ by resolving away all the Boolean variables $V_{C_i}$ and $V_{C_i \mid A}$ from $NG_c$. Then,

1. $NG$ is a nogood for $P$

Further, let $P' = P \cup \{NG\}$ and $P'_c = P_c \cup \{NG_c\}$. Then

2. Every branch that is a conflict for $P'$, is also a conflict for $P'_c$ after constraint and unit propagation, but there exist branches where the converse is not true.

3. $P'$ and $P'_c$ are incomparable with respect to propagation. There exist branches where some values are pruned in $P'$ but not in $P'_c$ and branches where some values are pruned in $P'_c$ but not in $P$. 
Chapter 5. Logical combinations of constraints

Proof. 1. Note that $\mathcal{P}_e$ is not any more constrained than $\mathcal{P}$. Specifically, every solution of $\mathcal{P}$ can be mapped to exactly one solution in $\mathcal{P}_e$ and every non-solution of $\mathcal{P}$ can be mapped to exactly one non-solution of $\mathcal{P}_e$. This is because the variables $V_{C_i}$ and $V_{C_i|A}$ are uniquely determined by a complete assignment to the variables of $\mathcal{P}_e$ that are shared with $\mathcal{P}$. This also means that we can map every solution and non-solution of $\mathcal{P}_e$ back to $\mathcal{P}$.

We now see that $NG$ is a nogood of $\mathcal{P}_e$, because it is derived by resolution. Since it contains none of the additional Boolean variables, it is also a nogood of $\mathcal{P}$.

Items 2. and 3. are a corollary of proposition 3.2.2, since $NG$ is derived by trivial resolution from $NG_e$.

5.4.1 Branching

A potentially significant improvement in performance can result by allowing the solver to branch on constraints [Järvisalo et al., 2005]. This means that the branching heuristic chooses a constraint $C$ and creates two branches. In one branch $C$ holds and in the other $\neg C$ holds. In general, this is possible in any solver. However, finding an effective heuristic that does this is hard, because the number of choices is very large. On the other hand, when a CSP is expressed in this constraint algebra, the set of subexpressions that appear in the problem is a set of candidates that are likely to be suitable for branching. Even though the set of choices has not shrunk at all, we now have a reasonable way to restrict the choices that a branching heuristic may need to examine. Still, other than the structure of the constraint expression, there is no differentiating factor in order to make a choice. Thus, it is still hard to define an effective heuristic that determines not only which constraint to branch on, but also whether to branch on a constraint or a variable.

Learning c-nogoods provides a possible solution to this problem. The implicit variables that are added to the problem allow us to treat the constraints that may be candidates for branching as variables. Thus, any metrics that are valid for other variables are valid for the implicit ones as well.
Chapter 6

Conclusions

In this thesis, we have examined how the technique of clause learning from SAT translates to CSPs. The impressive results seen in SAT using clause learning have not so far been seen in CSPs with nogoods, which is the corresponding notion to clauses. One of the reasons for this is that the notion of nogoods, as defined in the CSP literature, is restricted in a way that limits the gains CSP solvers can achieve from nogood learning. Specifically, a CSP algorithm that performs standard nogood learning is bounded by negative resolution, a refinement of resolution that is superpolynomially slower than unrestricted resolution in some instances. This partially explains the lackluster performance gains of adding nogood learning to a CSP solver, compared to using the equivalent technique of clause learning in a SAT solver.

We introduced the notion of generalized nogoods and showed that CSP algorithms that perform generalized nogood learning are not bounded by negative resolution. We also explored the relationship of nogood learning and global constraints. We explained why standard nogood learning degrades when combined with global constraints and showed that generalized nogood learning allows us to alleviate this problem. To achieve even better results, it is desirable to discover minimal nogoods whenever possible. To this end, we augmented several common constraint propagators to label prunings using minimal generalized nogoods.

Finally, we showed that nogoods can be generalized further when the input problem is expressed in a constraint algebra that allows the user to specify negations, disjunctions and conjunctions of constraints. We can include the satisfaction of constraints in nogoods, so that for a nogood to be true, the constraints it contains must be satisfied. We showed that this generalization also makes the nogoods learned more powerful.
Future work. The work in this thesis presents several possible avenues of future research. In order for generalized nogood learning to become a common technique among CSP solvers, we must have available a wide range of predefined constraints each with propagators that are able to label pruned values with effective nogoods. Besides the benefit of increasing the general usefulness of a g-nogood learning solver, this work may also provide further algorithmic insights.

Going further, one area not addressed here is that of branching heuristics that take into account the nogoods that have been learned. Branching heuristics based on VSIDS have been very successful in SAT solvers, but experiments that we performed with it suggest that a direct application of that heuristic in CSPs does not have the same results. One possible explanation is that this heuristic does not consider other information that is available to a CSP solver, such as the existence of constraints more complex than clauses and the fact that literals are grouped together into variables. Therefore, one possible future avenue of research is to investigate the question of whether or not VSIDS-like heuristics achieve the same purpose in CSPs as they do in SAT, which is to generate clusters of resolvable clauses. Similarly, we can investigate what kind of nogood learning behavior is encouraged by CSP heuristics. This line of research is promising, as similar approaches have yielded encouraging results. Specifically, the successful weighted degree heuristic collects information similar to the VSIDS heuristic. On the other hand, providing domain size information to a SAT solver has also been shown to improve performance. A related question, both in theory and in practice, is whether 2-way branching is more powerful in the presence of generalized nogood learning and whether we can invent 2-way branching heuristics that exhibit better behavior in practice than the commonly used d-way branching heuristics.

One of the trends in CSP research recently has been exploitation of the symmetry present in some instances by avoiding branches of the search tree that are subsumed under symmetry by branches we have already visited. One of the basic algorithms for this, SBDD, is based on nogood learning. The nogoods learned, however, are always maximal. We can use intelligent backtracking techniques to discover smaller nogoods and thus prune more of the search tree. But standard intelligent backtracking techniques often end up discovering very large nogoods, so the use of generalized nogoods in SBDD may provide an immediate improvement. Moreover, in SBDD even nogoods that are maximal with respect to the branch they were discovered in, can prune an exponential number of paths because they represent all other nogoods that are symmetric to the
stored one. Since a g-nogood may itself expand to an exponential number of s-nogoods, the effect may be even more pronounced.

Regarding the use of a constraint algebra to express CSPs, more questions still need to be answered. We saw that reifying subexpressions as boolean variables and including these variable in nogoods can increase the power of the nogoods. One possible question is whether there exist families of problems where this power results in a superpolynomial separation. Furthermore, including the satisfaction of a constraint in a nogood can be used by global constraints as a method of generating more succinct explanations for the prunings that they generate. This can in turn can make nogoods that are discovered later that much more powerful. There exists evidence that being able to branch on the satisfaction of a constraint (for example, explore one branch where the constraint $X < Y$ is posted and a branch where $X \geq Y$ is posted) can produce significantly shorter refutations [Järvisalo et al., 2005]. However, in the absence of any other information, the space of choices is too large to construct an effective branching heuristic that branches on the satisfaction of constraints. By including the satisfaction of constraints in nogoods, we find that measures such as VSIDS now apply to constraints as if they were variables, so there exists a much richer set of metrics to help us design branching heuristics.
Bibliography


[Bessi`ere et al., 2006a] Christian Bessi`ere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. The Range and Roots constraints: Algorithms and im-


