Improving the efficiency of learning CSP solvers

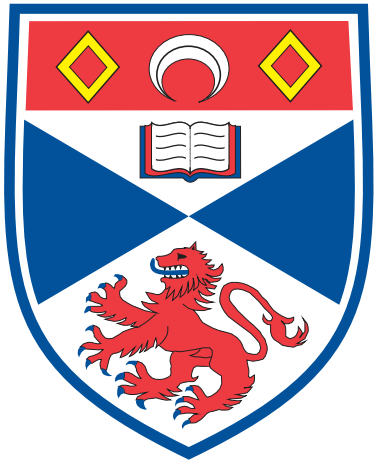
Thesis by

Neil C.A. Moore

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



University of St Andrews

School of Computer Science

2011

Abstract

Backtracking CSP solvers provide a powerful framework for search and reasoning. The aim of constraint learning is increase global reasoning power by learning new constraints to boost reasoning and hopefully reduce search effort. In this thesis constraint learning is developed in several ways to make it faster and more powerful.

First, lazy explanation generation is introduced, where explanations are generated as needed rather than continuously during propagation. This technique is shown to be effective is reducing the number of explanations generated substantially and concequently reducing the amount of time taken to complete a search, over a wide selection of benchmarks.

Second, a series of experiments are undertaken investigating constraint forgetting, where constraints are discarded to avoid time and space costs associated with learning new constraints becoming too large. A major empirical investigation into the overheads introduced by unbounded constraint learning in CSP is conducted. This is the first such study in either CSP or SAT. Two significant results are obtained. The first is that typically a small percentage of learnt constraints do most propagation. While this is conventional wisdom, it has not previously been the subject of empirical study. The second is that even constraints that do no effective propagation can incur significant time overheads. Finally, the use of forgetting techniques from the literature is shown to significantly improve the performance of modern learning CSP solvers, contradicting some previous research.

Finally, learning is generalised to use disjunctions of arbitrary constraints, where before only disjunctions of assignments and disassignments have been used in practice (g-nogood learning). The details of the implementation undertaken show that major gains in expressivity are available, and this is confirmed by a proof that it can

save an exponential amount of search in practice compared with g-nogood learning. Experiments demonstrate the promise of the technique.

Declaration

Only kings, presidents, editors, and people with tapeworms have the right to use the editorial 'we.'

Mark Twain

I, Neil C.A. Moore, hereby certify that this thesis, which is approximately 48381 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

Date ..... Signature of candidate .....

I was admitted as a research student in September 2007 and as a candidate for the degree of Doctor of Philosophy in September 2008; the higher study for which this is a record was carried out in the University of St Andrews between 2007 and 2010.

Date ..... Signature of candidate .....

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date ..... Signature of supervisor .....

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the electronic publication of this thesis: Access to printed copy and electronic publication of thesis through the University of St Andrews.

Date ..... Signature of candidate .....

Date ..... Signature of supervisor .....

Copyright

This is dedicated to my family and friends, but especially to those who never got a chance to see its completion.

Acknowledgements

I'd like to thank my supervisors, Ian Gent and Ian Miguel, for helping me to make the transition from being told what to do to being an independent researcher. Thanks to George Katsirelos for taking the time to give me detailed expert assistance at key moments. Thanks to Chris Jefferson and Pete Nightingale for their patient help on programming minion and understanding CP literature. Thanks to Patrick Prosser for giving me an interest in constraints that has persisted ever since we met. Thanks to Özgür Akgün, Andy Grayland, Lars Kotthoff, Yohei Negi and Andrea Rendl for their help and companionship.

This research was funded by UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/E030394/1 named "Watched Literals and Learning for Constraint Programming".

Publications

I have had the following peer-reviewed articles in journals, international conferences and workshops published during the course of this PhD. The first four entries are based on work contained in this thesis and are all based upon my own work, the other authors providing supervision and assistance with writing. The remaining entries are work peripheral to the PhD in decreasing order of my contribution.

- I.P. Gent, I. Miguel, and N.C.A. Moore. Lazy explanations for constraint propagators. In *PADL 2010*, number 5937 in LNCS, January 2010
- Ian P. Gent, Ian Miguel, and Neil C.A. Moore. An empirical study of learning and forgetting constraints. In *Proceedings of 18th RCRA International Workshop on "Experimental Evaluation of Algorithms for solving problems with combinatorial explosion"*, 2011
- Neil C.A. Moore. C-learning: Further generalised g-nogood learning. In Alan Frisch and Barry O'Sullivan, editors, *Proceedings of the ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, pages 103–119, 2011
- Neil C.A. Moore. Learning arbitrary constraints at conflicts. In *Proceedings of the CP Doctoral Programme*, September 2008
- Neil C.A. Moore. Propagating equalities and disequalities. In *Proceedings of the CP Doctoral Programme*, September 2009
- Neil Moore and Patrick Prosser. Species trees and the ultrametric constraint. *Journal of Artificial Intelligence Research*, 32:901–938, 2008
- Christopher Jefferson, Neil C.A. Moore, Peter Nightingale, and Karen E. Petrie. Implementing logical connectives in constraint programming. *Artificial Intelligence Journal (AIJ)*, 174:1407–1420, November 2010

- Ian P. Gent, Lars Kotthoff, Ian Miguel, Neil C.A. Moore, Peter Nightingale, and Karen Petrie. Learning when to use lazy learning in constraint solving. In Michael Wooldridge, editor, *European Conference on Artificial Intelligence (ECAI)*, 2010
- Lars Kotthoff and Neil C.A. Moore. Distributed solving through model splitting. In *3rd Workshop on Techniques for implementing Constraint Programming Systems (TRICS)*, pages 26–34, 2010

Contents

Chapter 1

Introduction

This thesis is about the *constraint satisfaction problem* (CSP). The CSP has enduring practical appeal because it is a natural way of encoding problems that occur spontaneously in practice. This is because the world is full of constraints, like credit limits on bank accounts, the amount of shopping a person can carry and the force of gravity. CSPs are useful because people may wish to know whether they can achieve their aims subject to the constraints imposed on them, like whether there is a travel itinerary visiting New York, London and Paris for under £1000.

So far, all I have done is to write down common sense since everyone copes with constrainedness every day. However some problems of this type are extremely difficult in practice, e.g. often finding a feasible school timetable is a difficult undertaking due to the difficulty of reconciling shortage of space, time and teachers. The idea of using computers to find solutions to such problems is very attractive, imagining that they can try out thousands of combinations per second. However even relatively simple looking practical problems can have huge numbers of possible guesses and very few solutions that satisfy the constraints. Sudoku is a CSP that many people are familiar with. The object of the game is to fill in a grid (e.g. Figure 1.1) with numbers, such that each row, column and box contains each number between 1 and 9. A sudoku has a single solution, but at most 11,790,184,577,738,583,171,520,872,861,412,518,665,678,211,592,275,841,109,096,961* complete assignments that are wrong! More formally, CSP is an NP-complete problem,

*this is based on a sudoku with 17 clues: there are 64 remaining unknowns with 9 choices for each, hence 9^{64} possible solutions

so it is strongly suspected that no algorithm exists that can solve any CSP quickly (in polynomial time).

In spite of the likely impossibility of finding a fast algorithm for CSP that works in all cases, general purpose CSP solvers exist and are used to solve many problems in practice. But why use a CSP solver, when you can write a solver just for timetabling, or just for planning road trips? The reason is that general purpose solvers:

- incorporate the distilled wisdom of experts, e.g. efficient implementation and effective general-purpose heuristics to help find a solution;
- adapt to solve variants of the original problem, something that a solver tailored to one problem cannot easily do; and
- save programmer's time because they can be taken off the shelf.

However, it is the burden of writers of CSP solvers to be constantly striving to solve a larger range of instances, and faster, by incorporating clever new tricks into their solver. In this thesis I attempt to do this for a certain type of CSP solver, as I will now explain.

1.1. CSP solvers in practice

In this thesis, I will consider cases where the unknowns are integers and the constraints may take any form, provided they are easy to check. For example, it is easy to verify a solution to equation $x^2 - 2 = y^m$ such that x, y are integers and $m \geq 3$ by arithmetic, but it is an unsolved problem to find one (according to [**Coh07**]). Many commonly used CSP solvers use a variation on *backtracking search* to find a solution. Typically they make a sequence of guesses to fill in unknowns, retracting decisions that don't lead to a solution and finishing when a solution is found or no more guesses are possible. In this way every possible solution is eventually tried.

In its basic form this naïve strategy can, for example, hit the worst case for sudoku where 1.2×10^{61}[†] different assignments are tried. One of the primary reasons why CSP solvers are able to be efficient in practice is that they integrate reasoning algorithms for each constraint individually into backtracking search. For example, consider the sudoku shown in Figure 1.1. The coloured numbers can be inferred to be

[†]explained in footnote on previous page

2	5	9	9	3	9	9	9	1
	1				4	9	9	9
4		7				2	9	8
		5	2			9		
				9	8	1		
	4				3	9		
			3	6		9	7	2
	7					9		3
9		3				6		4

FIGURE 1.1. Example sudoku grid, clues are printed in black, impossible entries in colour

impossible because of the (black) 9 already assigned. The red values are impossible by reasoning with the constraint over the first row: no duplicate 9s are allowed. The blue values are impossible by the constraint over the top-right box. Finally the green values are ruled out by the constraint over the 7th column. This type of simple reasoning has drastically cut the number of possible assignments, removing 284,294,103,884,805,425,687,795,982,353,378,114,796,118,426,545,705[1] or 75% of the possible assignments straight away! In general-purpose CSP solvers with this type of "all different" reasoning can solve almost all sudokus very easily, however creating hard sudoku puzzles by computer is still a challenging problem. The process of removing values from consideration by inference is known as *propagation*.

Propagation is important to CSP solvers because it allows separation of concerns: specialised inference algorithms are added in a controlled way, because there is one for each constraint. There is no need to consider, for example, what to do with combinations, such as $x + y = z$ s.t. $x \neq y$. The solver simply has one algorithm

[1]original search space: 9^{53}, new search space: $8^{12} \times 9^{41}$, difference: $9^{53} - 8^{12} \times 9^{41}$

for $x + y = z$ and another one for $x \neq y$. There is, however, a limit to how far this *local* reasoning can take you. Sometimes there is a need for *global* reasoning, where knowledge about groups of constraints is combined to boost the solver's performance. This is what *constraint learning* does.

Learning is when the solver is allowed to proceed until it makes a mistake from which it cannot recover. Then, the solver identifies a set of the assignments that it made, that are incompatible with each other—these are called *nogoods*.

Learning works by waiting for a failure to occur, such as all the possible values for a choice being ruled out, meaning progress is now impossible. All the while, the CSP solver is introspecting its own inferences, producing and storing what are called *explanations* for the inferences. When the failure occurs, the root causes are traced by inspecting the explanations and this set of root clauses is *learned* and avoided in future.

My aim in this thesis is to improve the practical performance of CSP learning solvers in 3 ways: by producing explanations lazily, by forgetting constraints and by generalising the constraints learned.

1.2. Hypotheses on learning in CSP search

I will now briefly describe each idea and state the hypotheses which I will be following up throughout the thesis. This section will be a little more technical than the last, and here I will give an overview of the main hypotheses of this thesis, and why I believe that they are worthwhile questions. However I have left references out of this section and they are instead provided in subsequent chapters.

The object of stating my hypotheses is to clearly differentiate the overall aims of my research from the means used to achieve them. Since the bulk of the thesis describes detailed work used to achieve these aims, there is a danger they will become obscured. The hypotheses are all written in such a way that their validation is associated with a speedup in the solver, that is, I investigated these questions because I hoped they would be true. During the research, I investigated other questions that turned out to be inconclusive or uninteresting and I have left some of these out of the thesis.

1.2.1. Lazy learning. Producing explanations lazily basically means that instead of generating and storing explanations for inference on the fly (eagerly), as has been done in the past in general purpose CSP solvers (see §3.3 for a complete bibliography), a minimum of required information is stored up front and the explanation is only fully computed *when needed*.

There is an analogy here with police detective work. The police do not attempt to, and cannot, collect all information as they go along that may be relevant to their investigation. For instance, detectives don't exhaustively question every connected person immediately. Instead they take names and contact details, and when appropriate they can revisit a witness for further questioning in order to open up new lines of enquiry. In this way, they can work backwards from the scene of the crime, inferring facts about the case from what they have discovered, until they find the perpetrator. At least, that's the way it works in Sherlock Holmes stories; the real world is more complex.

Hence the broad idea of learning lazily is sensible. Implementing it is complicated but possible, as I will show. But whether it is useful, as in the case of police work, or completely useless, depends on testable hypotheses:

Hypothesis 1. In a constraint learning CSP solver solving practical CSPs, most of the explanations stored are never used to build constraints during learning.

When I say "practical CSPs", I mean CSPs that people are interested in solving, e.g. those from solver competitions and/or those associated with problems of practical interest. If this is the case, there is a chance for lazy learning to be fast, because time can be saved by avoiding computation. This relies on lazy and eager explanations taking about the same time to compute, because if lazy uses fewer longer computations it is not necessarily faster overall:

Hypothesis 2. The asymptotic time complexity of computing each explanation lazily is no worse than eager computation, or the practical CPU time to compute each lazy explanation for practical CSPs is no worse.

If these hypotheses are valid, then lazy learning will be successful in speeding up the learning solver.

1.2.2. Forgetting. Learning nogoods takes up memory and requires CPU time to check if the current assignment is ruled out. Since CSP solvers often search many thousands or millions of nodes, it is necessary to remove constraints during search to avoid running out of memory or spending too much time checking nogoods that are not currently relevant. The removal of learned constraints is called *forgetting* in this thesis. Forgetting has been tried before in learning for CSP and SAT, but I will build upon the previous research in this area.

Firstly I will examine several questions that motivate the use of learning and explain why it works well. The first hypothesis is as follows:

Hypothesis 3. Nogoods vary significantly in the amount of inference they do.

If this is the case then removing the least propagating constraints should remove overheads but cause less than a proportionate increase in search size. This is because if all the constraints were similar, removing $k\%$ would reduce inference by $k\%$. However if they are different, removing $k\%$ carefully would reduce inference by $< k\%$. In order to achieve an improvement in speed, one would also hope that these less effective constraints take a lot of time to process.

Hypothesis 4. Weakly propagating nogoods occupy a disproportionate amount of CPU time, relative to their level of propagation.

If this is the case then removing some of the worst performing nogoods should disproportionately improve the performance of the solver. Strategies for doing this in practice have been tried before, but in a slightly different setting using relatively inefficient propagators for learned constraints, different strategies for learning and/or a smaller set of benchmarks. Hence I will reevaluate these strategies from the literature in a new setting to determine their usefulness:

Hypothesis 5. There are forgetting strategies that are successful in reducing the time spent solving CSPs of practical interest.

Assuming this hypothesis is correct, there exist strategies in practice to achieve the hypothetical saving in time associated with the previous two hypotheses.

1.2.3. Generalising learning. Generalisation is a powerful theme in learning. Nogoods can be generalised by removing irrelevant assignments, so that they rule out more branches of search. They can also be generalised by changing the type of information of which nogoods are composed. For example, rather than storing only assignments they can be generalised to both assignments ($x = a$) and disassignments ($x \neq a$). In this thesis I will develop the idea of further generalising nogoods to be composed of arbitrary constraints. This allows nogoods to be more expressive, so that they can rule out more paths leading to failure and do so just as compactly.

Hypothesis 6. Using nogoods composed of arbitrary constraints, as opposed to assignments and disassignments, can significantly reduce the amount of search required to solve some CSP instances.

To validate this hypothesis I had to develop a further generalised learning framework, and experiment on CSPs to test its effectiveness.

I will return to these hypotheses at the end of each chapter and at the end of the thesis, to discover if they have been verified.

1.3. Contributions

I will now summarise the contributions of this thesis, in order of increasing chapter breaking ties by decreasing importance:

- Introduction of lazy explanations for CSP solvers (Chapter 3)
  - I prove empirically that, in the context of g-nogood learning, lazy explanations result in significantly less work and save time over a wide range of benchmarks.
  - I describe how to compute explanations lazily for a range of commonly used constraints including lexicographical ordering, table constraint and alldifferent.
  - I describe how to implement efficient lazy explanations in a solver in practice.
- Experiments on forgetting constraints (Chapter 4)

  - I carry out experiments showing why forgetting is likely to be successful, ruling out other possible explanations:
    * to prove empirically, for the first time in either CSP or SAT, that a small number of learned constraints are generally reponsible for much of the search space reduction associated with learning; and
    * to prove that, using watched literal propagation, the most weakly propagating constraints overall account for the majority of the propagation time.
  - I show that simple strategies like size- and relevance-bounded learning from the literature are successful in speeding search up overall, with a modest increase in search space.
- Advance understanding of c-nogood learning, where learned constraints can contain arbitrary constraints (Chapter 5)
  - I prove that there is an exponential separation between c- and g-nogood learning; and also that empirically that this can translate to a massive time saving.
  - I describe how to implement c-nogood learning for the first time.
  - I describe c-explainers for all different and occurrence constraints and compare their expressivity with the best g-explainers.
  - I experimented with c-nogood learning on practical problems.

1.4. Thesis structure

The structure is as follows:

Chapter 2: I undertake a literature review including the basic concepts of CSP and fundamental algorithms for backtracking search. Following this is a comprehensive survey of the literature on learning in CSP, SAT, SMT and related fields. In this review I emphasise the underlying similarity of many learning and backjumping algorithms, unified by the concept of explanations.

Chapter 3: In this chapter I describe the first fundamental contribution of this thesis. Here I introduce the idea of lazy explanations for general purpose CSP

solvers, and develop the technique. This involves demonstrating the feasibility of lazy learning by describing a framework and developing algorithms for explaining global constraint propagators lazily. These lazy explanation algorithms implemented within the described framework are shown to be effective in reducing the amount of time spent calculating explanations in practical usage. This is done using a comprehensive experiment testing their effectiveness over many constraints and problem types. Lazy explanation algorithms also find application in SAT modulo theories (SMT) and the work in this thesis is the first published progress in this area.

Chapter 4: Several practical contributions to knowledge in the area of clause forgetting are developed. First I provide evidence to suggest that constraint forgetting will be effective on CSP by showing that, in practice, a small number of the highest propagating constraints are responsible for much of the search space saving associated with learning. Next, I show that the constraints that are doing little propagation are nevertheless occupying much of the solver's time. This is a surprising result which motivates using forgetting to avoid wasting this time. Finally I experiment on size- and relevant-bounded learning and the minisat conflict driven strategy, which are all strategies for forgetting from the literature. These strategies are thoroughly evaluated in a modern learning CSP solver (using g-nogoods) for the first time and, contrary to previous evidence, are shown to be extremely effective in speeding up search.

Chapter 5: The idea of c-nogood learning (c-learning)[2] is developed. In this chapter I prove the theoretical promise of c-learning by proving that there exist problems that c-learning can solve in polynomial time, but that g-learning solvers cannot solve in less than exponential time. Next, I develop the first practical framework for implementing c-learning, and show how to produce c-explanations for several global constraints, analysing the improvement in expressivity compared with g-explanations for the same constraints. Finally I present evidence regarding c-learning's practical usefulness.

[2]the "c" in c-learning stands for "constraint"

Chapter 6: Conclude the thesis with a critical summary of its contributions and suggestions for future work.

Chapter 2

Background and related work

I have attempted to make this thesis self-contained so that a reader familiar with general computer science but not constraints or AI can read it. The following background chapter is intended to be a fairly gradual introduction to constraints and to learning in constraints. However as a more comprehensive reference I can recommend the following titles [**Lec09, RvBW06**].

2.1. Basic definitions

The constraint satisfaction problem (CSP) is an NP-complete problem. Practically, it is used as a universal and practical means of encoding problems from the class NP, i.e. the set of problems whose solutions can be verified in polynomial time, but which may take exponential time to find.

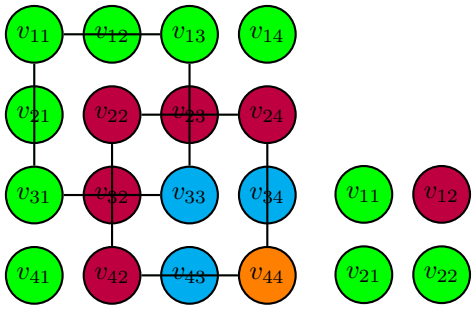
Definitions 2.1 (Basic CSP definitions)**.** A finite integer CSP is a triple (V, D, C) where:

- V is the finite set of *variables*, and
- the *domain* $D : V \to 2^{\mathbb{Z}} \setminus \{\emptyset\}$ is function from each variable to a finite set of integer values[1],
- C is the finite set of *constraints*.

Each constraint $c \in C$ is defined by

- its *scope* $\text{scope}(c) = (v_{c_1}, \ldots, v_{c_k})$ s.t. $\forall i \in [1..k]$, $v_{c_i} \in V$ and

[1] $2^{\mathbb{Z}}$ represents the powerset of the set $\mathbb{Z}$ of integers; hence the codomain is the set of non-empty sets of integers



(a) NMR-4-4-4 (b) NMR-2-2-2

FIGURE 2.1. Solutions for various NMR problems

- its *relation* $\text{rel}(c) \subseteq D^k = \underbrace{D \times \ldots \times D}_{k \text{ times}}$ where $k = |\text{scope}(c)|$, i.e. a subset of possible k-tuples containing values from D.

A (partial) *assignment* A is a (partial) function $A : V \to D$.

A *solution* is an assignment S, such that

$$\forall c \in C, (S(v_{c_1}), \ldots, S(v_{c_k})) \in \text{rel}(c)$$

i.e. the values assigned to variables in $\text{scope}(c)$ form a tuple in the constraint's relation. In this case, I will say that constraint c has been *satisfied*[2].

A *binary CSP* is a CSP consisting only of constraints with a scope of at most two variables.

I now introduce a running example to illustrate each of these concepts.

Example 2.1 (Non-monochromatic rectangle [**Gas09**])**.** *The problem is to colour the cells of an m by n grid, using k colours and such that the cells at the corners of any rectangle may not be* monochromatic*, i.e. all take the same colours. I will call the instance with a m by n grid using k colours NMR-m-n-k. An example solution to NMR-4-4-4 is shown in Figure 2.1(a)*[3]*. Notice that a couple of rectangles are highlighted using lines on each side and are indeed non-monochromatic.*

NMR-m-n-k can be encoded as a CSP (V, D, C) as follows:

[2]hence constraint *satisfaction* problem

[3]note that NMR-4-4-2 is also solvable, so it can be done with 2 fewer colours

- *A variable* $v_{ij} \in V$ *for each cell* (i, j) *in the grid. Here the row* i *is indexed first.*
- *A value* $val_c \in D$ *for each colour* c*. For example,* 1 *for blue and* 2 *for red.*
- *For all choices of row indices* i *and* j*, and column indices* k *and* l *such that* $i < j$ *and* $k < l$*, a constraint* $c_{ijkl} \in C$ *to ensure that the variables* v_{ik}*,* v_{il}*,* v_{jk} *and* v_{jl} *are not all the same*[4]*.*

Let $c_{ijkl} = v_{ik} \neq v_{il} \vee v_{ik} \neq v_{jk} \vee v_{ik} \neq v_{jl}$ *then* $scope(c_{ijkl}) = (v_{ik}, v_{il}, v_{jk}, v_{jl})$ *and* $rel(c) = D^4 \setminus \{(1,1,1,1), (2,2,2,2), \ldots, (k,k,k,k)\}$[5]*.*

$A = (v_{11}, 1), (v_{12}, 2), (v_{21}, 1), (v_{22}, 1)$ *is a solution to NMR-2-2-2, shown in Figure 2.1(b).*

2.2. Fundamental CSP algorithms

Constraint programming is the use of the CSP[6] for solving practical problems. Typically this involves modelling a problem as a CSP, as above for NMR, and then using a *solver* to find one or more solutions. However sometimes instead what is required is to satisfy the maximum number of constraints at once, optimise the solution according to an optimisation function, etc. Solvers can be *complete* or *incomplete*. Given enough time and memory, complete solvers guarantee to find a solution, if one exists, or to report conclusively that none exists; incomplete solvers make no such guarantees. In this thesis I will exclusively be concerned with complete solvers.

Complete algorithms for the CSP can be broadly categorised as either *backtracking search* or *dynamic programming* solutions. Backtracking search algorithms predominate in practice because they have been found to be more memory efficient; more flexible when it comes to solving related problems such as to find only the first or optimal solution; and more time efficient in general [**vB06**].

Before introducing the algorithms I need to give some notations used throughout:

---

[4] called "not all equal" constraint in [**BCR10**]

[5] c_{ijkl} correctly models "not all same" because it's the negation of $(v_{ik} = v_{il} \wedge v_{ik} = v_{jk} \wedge v_{ik} = v_{jl})$ which is only true when the variables are all equal

[6] and its extensions such as MAX CSP, weighted CSP, etc.

Definitions 2.2. In search algorithms I will denote the domain of a variable v by initdom(v). Simply, $D(v) = \text{initdom}(v)$. In many algorithms there exists a concept of the domain being narrowed, as values are ruled out, hence each variable v has its own domain dom(v), which is the subset of initdom(v) that has not already been ruled out at the current point in search.

If a constraint c is satisfied by all possible assignments to variables in scope(c) from their respective domains, c is said to be *entailed*.

I will first describe the classical backtracking (BT) search algorithm, before proceeding to describe the many available enhancements.

2.2.1. Chronological backtracking. *Chronological backtracking* (BT) is a simple and effective base algorithm for solving CSPs. A BT solver for CSP will repeatedly pick a variable by some means and then assign the variable one of its available values. There are now 3 possible states:

(1) A solution has been found, in which case it is reported to the user; the solver terminates[7].
(2) One or more constraints are fully assigned but unsatisfied, the solver must backtrack and try again.
(3) Neither of the above apply, the solver makes its next assignment.

Practically, the BT algorithm is implemented by pushing and popping the current state of the variables. So when v is set to to value a, dom(v) = $\{a\}$, but after the solver backtracks beyond the point when this occurred, dom(v) will be restored to *initdom*(v). Pseudocode is given as Algorithm 1[8].

I will now present an example of the progress of BT search on NMR-2-2-2.

Example 2.2. *The entirety of a search process can be depicted as a* search tree. *In a search tree the nodes are decision points, and the child subtrees represent the search in a recursive call. The search tree for NMR-2-2-2 with BT search is shown in Figure 2.2. There are few enough variables and values in NMR-2-2-2 that the state of the*

[7]This is not essential, a solver could also proceed to look for more solutions.

[8]initially domains are assumed to be non-empty and not an inconsistent (complete) assignment

Algorithm BT-SEARCH

Search()
A1 **if** $\forall v \in V, |\text{dom}(v)| = 1$
A1.1 output solution
A1.2 **exit**
A2 choose a variable v to branch on s.t. $|\text{dom}(v)| > 1$
A3 **for** $val \in \text{dom}(v)$
A3.1 push solver state
A3.2 set $\text{dom}(v) = \{val\}$
A3.3 **if** $\neg\exists c \in C$ s.t. c is fully assigned and unsatisfied
A3.3.1 **Search()**
A3.4 pop solver state
A3.5 **return**

Algorithm 1: Backtracking search

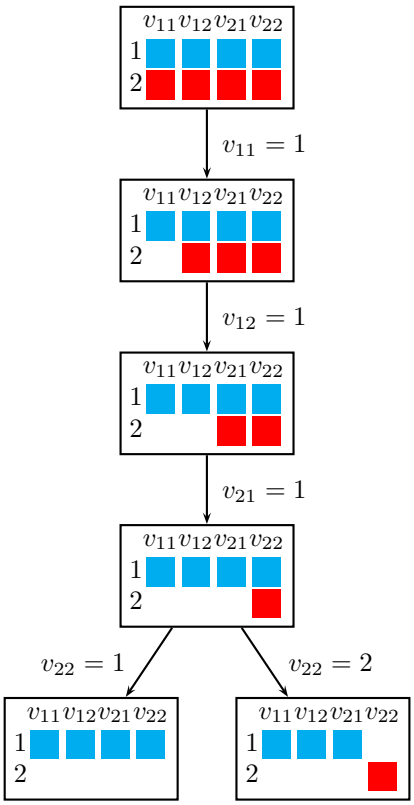


FIGURE 2.2. Search tree for BT search on NMR-2-2-2

domain at each node can be depicted: in Figure 2.2 the variable name is shown above the colours represented by the values still remaining in its domain.

The first decision is to assign v_{11} *to* 1*, i.e. blue, at A3.2 in Algorithm 1. The sole constraint is not yet fully assigned so a recursive call is made (line A3.3.1). The second decision is to assign* v_{12} *to* 1*. Again, no constraint is unsatisfied. The*

third decision is to assign v_{21} to 1 and again no constraint is unsatified. The fourth decision is the same: assign v_{22} to 1, but this time the constraint is fully assigned and unsatisfied. Hence the state is restored, and v_{22} is assigned to 2, i.e. red. Now the recursive call happens and the solution is output at A1.1 and the solver can terminate.

Given enough time and space, BT search can solve any CSP. However several improvements are available, and I will survey them briefly in the following sections.

2.2.2. Propagation. *Propagation* is when a constraint solver infers that one or more values in the domain of a variable v cannot possibly be part of a solution to the CSP. Those values are then removed from $\mathrm{dom}(v)$. In this way

- fewer incorrect assignments are available resulting in less fruitless search, and
- domain might be emptied allowing for immediate backtrack.

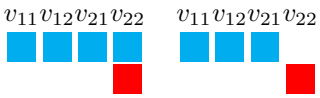
Various different levels of propagation are available (see [**Bes06**] for a general survey), I will now describe several levels of consistency that will be relevant for this thesis, with examples of the effect of each.

2.2.2.1. *Generalised arc consistency.* In short, *generalised arc*[9] *consistency* (GAC) on a constraint c ensures that given the current domains of the variables, no value remains that cannot be part of a complete assignment to all the variables in the scope of the constraint.

Definition 2.1 (Generalised arc-consistency, adapted from [**Bes06**])**.** Given a CSP (V, D, C), a constraint $c \in C$ and a variable $v \in \mathrm{scope}(c)$,

- A *valid tuple* is a tuple $\tau \in \mathrm{rel}(c)$ s.t. $\forall i$, $\tau[i] \in \mathrm{dom}(\mathit{scope}(c)[i])$.
- A valid tuple τ is a *support for value* $\mathit{val} \in \mathit{dom}(x)$ iff $\exists i$ s.t. $x = \mathit{scope}(c)[i]$ and $\mathit{val} = \tau[i]$.
- A value $\mathit{val} \in \mathrm{dom}(v)$ is GAC with c iff there exists a valid tuple $\tau \in \mathrm{rel}(c)$ s.t. $\tau[i] = \mathit{val}$ where i is the index of v in $\mathrm{scope}(c)$.
- A variable v is GAC with c iff $\forall \mathit{val} \in \mathrm{dom}(v)$, val is GAC with c.
- The CSP (V, D, C) is GAC iff $\forall c \in C$, $\forall v \in \mathrm{scope}(c)$, v is GAC with c.

[9] "arc" refers to the constraint graph: consisting of a vertex for each variable and a hyper arc/edge for each constraint



(a) Before (b) After

FIGURE 2.3. Before and after running GAC on the NMR-2-2-2 problem

- When the CSP (V, D, C) is GAC but $\exists v \in V$ s.t. $\text{dom}(v) = \emptyset$, (V, D, C) is said to be *arc inconsistent*, or just *inconsistent*.
- On a binary CSP, GAC is called arc consistency (AC) but is otherwise the same.

Various algorithms are available to *enforce* GAC on a CSP [**Bes06**] including AC3 [**Mac77**] and AC5 [**HDT92**]. Enforcing GAC means to take a set of domains (one for each variable) and to output a set of domains with all GAC inconsistent values removed, but no consistent values removed. Hence all the solutions of the input domains are still present, but values may have been removed. AC3 and AC5 are variations on a simple theme: GAC can be enforced on the CSP by enforcing it on constraints, variables and values until a fixed point is reached [**Bes06**].

I now give an illustration of these definitions on NMR-2-2-2 running example.

Example 2.3. *Recall that NMR-2-2-2 has just one constraint that variables* v_{11},v_{12}, v_{21} *and* v_{22} *must not all be the same. In Example 2.1 I said that this constraint* c *has* $scope(c) = (v_{11}, v_{12}, v_{21}, v_{22})$ *and* $rel(c) = \{1, 2\}^4 \setminus \{(1, 1, 1, 1), (2, 2, 2, 2)\}$.

Suppose that GAC is enforced on NMR-2-2-2 when the domains are as shown in Figure 2.3(a). $(1, 1, 1, 2)$ *is a valid tuple in* $rel(c)$ *and hence* $1 \in dom(v_{11})$, $1 \in dom(v_{12})$, $1 \in dom(v_{21})$ *and* $2 \in dom(v_{22})$ *have been shown to be GAC for* c. *Hence variables* v_{11}, v_{12} *and* v_{21} *are GAC for* c *because all their values are GAC.*

It now remains to resolve whether $1 \in dom(v_{22})$ *is GAC or not. Clearly no valid tuple includes this value because the remaining 3 variables are already assigned to blue then to set* v_{22} *to blue would mean all 4 were the same. Hence* $1 \in dom(v_{22})$ *is not GAC.*

After GAC is enforced the domains are as shown in Figure 2.3(b).

In many constraint solvers GAC is maintained throughout search: it is enforced before every decision is made. GAC has been shown to be a practically useful level of consistency to enforce during BT search because:

- It is cheap to enforce. The worst case time complexity of enforcing GAC on a binary CSP is $O(ed^2)$, where $e = |C|$ and $d = |D|$ (using AC4 [**MH86**] or AC2001 [**BR01**]).
- On many CSPs of practical interest enough values are removed by AC to result in faster search overall [**SF94**].

Algorithm MAC-SEARCH

```
Search()
A1    if ∀v ∈ V, |dom(v)| = 1
A1.1      output solution
A1.2      exit
A2    enforce GAC on all c ∈ C
A3    if ∃v ∈ V s.t. dom(v) = ∅
A3.1      return
A4    choose a variable v to branch on s.t. |dom(v)| > 1
A5    for val ∈ dom(v)
A5.1      push solver state
A5.2      set dom(v) = {val}
A5.3      Search()
A5.4      pop solver state
A6    return
```

Algorithm 2: Backtracking search maintaining arc-consistency

With GAC incorporated in the search process the search algorithm is as shown in Algorithm 2. It is known as MAC, for *maintaining arc consistency*. The differences compared to Algorithm 1 (BT search) are as follows:

- GAC is enforced at line A2.
- If the CSP is GAC inconsistent, the solver backtracks (line A3.1).
- The recursive call at A5.3 is no longer conditional: the most recent assignment cannot complete an incorrect assignment, otherwise GAC would have removed the offending value[10].

[10]Note, however, the entire CSP may now be GAC inconsistent and this is discovered in the recursive call.

GAC is the strongest possible level of domain consistency that can be enforced by analysing constraints individually [**Bes06**].

2.2.2.2. *Constraint propagators.* In practice, consistency on a constraint c is usually enforced by a constraint propagator for c. A constraint propagator takes as input a set of domains and returns domains where zero or more inconsistent values are removed. Constraint propagators in an AC3 framework [**Mac77**] are told nothing about how the domains have changed since they were last invoked. In more modern solvers, propagators are told what has changed, hence they must maintain state if they want to take full advantage. In Example 2.3 I gave an example of GAC propagation of the "not all same" constraint which ensures its variables are not all equal, I now present a GAC propagator for it.

Algorithm NOT-ALL-SAME-PROPAGATOR

```
BT<bool> allAssgsSame = true;
int lastAssg;
BT<int> howManyAssgsSame = 0;
when v_i ← a:
A1    if(!allAssgsSame)
A1.1      return
A2    if(howManyAssgsSame == 0)
A2.1      lastAssg = a;
A2.2      howManyAssgsSame = 1;
A3    else if(lastAssg == a)
A3.1      howManyAssgsSame++;
A4    else
A4.1      allAssgsSame = false;
A4.2      return;
A5    if(howManyAssgsSame == arity - 1)
A5.1      prune a from variable v_j that is not already set to a;
A5.2      allAssgsSame = false;
```

Algorithm 3: Propagator algorithm for "not all same" constraint

Example 2.4. *A propagator for "not all same" is given as Algorithm 3. First I describe the declarations. allAssgsSame is a backtracking boolean that is true iff all variables in the scope of the constraints so far have been assigned the same value. lastAssg is the last value assigned, unless allAssgsSame is false. howManyAssgsSame is a backtracking integer that is the number of assignments known to be the same, unless allAssgsSame is false.*

Now I describe the algorithm. At line A1, the algorithm stops immediately if the constraint is already satisfied, i.e. two assignments are known to be different. Between lines A2 and A4.1 the variables are updated to take into account the new assignment $v_i \leftarrow a$. *If it's the first assignment, they are initialised at A2. Otherwise at A3 when the assignment is to the same value as before howManyAssgsSame is advanced, or at A4 when it is different the constraint is now satisfied so allAssgsSame is set accordingly and the propagator stops (A4.2). Finally, the propagator checks if any propagation is necessary, which is the case when all but one variable is assigned the same.*

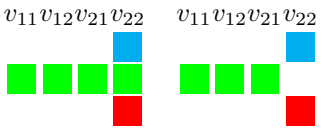
This algorithm is amortized linear time down a branch of the search tree. Clearly everything is constant time except for line A5.1. A5.1 takes time linear in the number of variables, but the propagator must have run a linear number of times down the current branch of the search tree in order to reach A5.1. Hence the time to check each variable can be amortized against the earlier propagator invokations, to make each invokation of the propagator amortized $O(1)$ time and overall $O(n)$ down a complete branch.

I will now complete the description of propagation by mentioning other, weaker consistencies.

2.2.2.3. *Consistencies weaker than GAC.* Solvers are at liberty to enforce any level of consistency that works well in practice, provided only that an incorrect complete assignment will lead to an immediate backtrack [**SC06**]. Consistency may be enforced selectively on values based on their relative order; at other times an ad-hoc level of consistency is chosen to perform a subset of available prunings *easily*.

I will call an algorithm that enforces consistency on a constraint c a *propagation algorithm* for c. As described above a propagation algorithm takes as input the variable domains, and outputs reduced domains.

For example, there is a consistency level called *bound(D) consistency* (abbreviated to BC(D)) that guarantees only that the smallest and largest values belong to a support, but not necessarily the ones in between. This is exploiting the numerical *order* of values.



(a) BC(Z) (b) GAC

FIGURE 2.4. The NMR-2-2-3 problem in BC(D) and GAC consistencies

Definition 2.2 (Bound(D) consistency (BC(D)), adapted from [**Bes06**])**.** For a CSP (V, D, C),

- A variable v is BC(D) with c iff $\min(\text{dom}(v))$ is GAC with c and $\max(\text{dom}(v))$ is GAC with c.
- The CSP (V, D, C) is BC(D) iff $\forall c \in C$, $\forall v \in \text{scope}(c)$, v is BC(D) with c.
- When (V, D, C) is BC(D) but $\exists v \in V$ s.t. $\text{dom}(v) = \emptyset$, (V, D, C) is said to be *bounds inconsistent*.

BC(D) and related consistencies like BC(Z) and BC(R) are designed to find "low hanging fruit", that is inconsistent values that are easy to find, and are not intended to be complete except coincidentally. They are very important in practical constraint solvers because it is NP-hard to enforce GAC on some constraints [**BHHW04**]. Even when not NP-hard, the cheapest GAC propagator may be a slow polynomial time algorithm.

I will now illustrate the definition with the NMR running example.

Example 2.5. *Recall that NMR-2-2-3 is the same as the NMR-2-2-2 example discussed earlier, except it has 3 values (i.e. colours) for each variable (cell).*

In Figure 2.4(a) the domains shown are BC(D) consistent. However they are not GAC, because the middle (green) value in dom(v_{22}) is not part of a valid tuple. The domains shown in Figure 2.4(b) are GAC.

2.2.2.4. *Consistencies stronger than GAC.* There are many other consistencies available for CSP that enforce a higher level of consistency than GAC. Since GAC is the highest possible level of consistency that can be enforced on a single constraint, these higher forms of consistency work by processing groups of constraints to find disallowed assignments. I will briefly summarise a few of the most widely used consistencies stronger than GAC.

Singleton arc consistency (SAC). Singleton arc consistency [**DB97**] is able to identify values that cannot appear in any solution.

Definition 2.3 (Singleton arc consistency)**.** Given a CSP (V, D, C), a variable $v \in V$ and a value $a \in \text{dom}(v)$: An assignment $v \leftarrow a$ is singleton arc consistent if and only if (V, D, C) is GAC when $v \leftarrow a$, i.e. $\text{dom}(v) = \{a\}$; otherwise $v \leftarrow a$ is singleton arc inconsistent.

Any singleton arc inconsistent (SAI) assignment can be ruled out of future search, i.e. if $v \leftarrow a$ is SAI then remove a from $\text{dom}(v)$. SAC is time consuming to enforce, requiring AC to be enforced for each variable and value pair. For this reason it is typically enforced only at the root node but successful results have been shown for both root node and maintaining SAC during search [**LP96**].

2.2.2.5. *Other consistencies.* Consistency and propagation is a central topic of constraint satisfaction. Hence I have given only a brief overview of consistencies, especially where they are relevant to this thesis. More information can be found in [**Bes06**].

2.3. Learning CSP algorithms

Having completed a brief survey of the components of a complete CSP solver, I will now focus on the topic of this thesis: solvers that learn from experience. Previous applications of learning in CSP search can be broadly classified as being either:

Backjumping: A normal solver will step back once after inconsistency is discovered. *Backjumping* algorithms are sometimes able to make multiple steps after an inconsistency.

Learning new constraints: A normal solver retains the same set of constraints throughout search: those of the original problem. However it is possible to achieve increased inference by augmenting the set.

Heuristics: The order in which variables and values are picked for assignment can make a big difference: with an oracle a solution can always be found in polynomial time just by assigning the variables correctly first time!

Some heuristics learn from experience to attempt to reduce search size, e.g., [**BHLS04**].

Historically, backjumping and learning new constraints (which I will call simply *learning* from now on) have been closely related, because both rely on an analysis of the decisions and propagation that led to inconsistency, hence having done the analysis for, say, constraint learning, it can make sense to also do backjumping. So-called *explanations*, in some form, are a unifying concept in both constraint learning and backjumping and for this reason I will review explanations first. Techniques for explanations are spread and often hidden throughout the history of learning.

For this reason I take a unconventional approach to reviewing this field. Often two learning or backjumping algorithms can be viewed as identical or similar, except that the explanation mechanism is varied. For this reason I first extract the explanation algorithms from the mess of learning and backjumping algorithms. After that I describe the essential techniques of backjumping and learning new constraints, based on the understanding that often any explanation technique or a mixture of them can be used.

2.4. Explanations

Explanations are a means for discovering *why* a constraint solver makes an inference, why it cannot find a solution, etc. Such techniques have been used to allow a solver to introspect and learn, but also for user feedback to assist with modelling and debugging (e.g. [**Jun01**]). Explanations are dual to the common CSP concept of *nogood*, as I will describe shortly.

First I define the possible aspects of a solver's current state that an explanation pertains to:

Definition 2.4. If $\mathrm{dom}(v) = \{a\}$ for some variable $v \in V$ then a is said to be *assigned* to v; this is called an *assignment* and written $v \leftarrow a$. Similarly, if $a \notin \mathrm{dom}(v)$ then a is said to be *disassigned* to v; this is called a *disassignment* and is written $v \nleftarrow a$. Collectively I will call them (dis-)assignments.

When $\exists v \in V$ s.t. $\mathrm{dom}(v) = \emptyset$ for any valid reason (e.g. CSP is GAC inconsistent), I will say the CSP has *failed*.

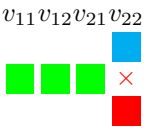


FIGURE 2.5. Illustration of disassignment for NMR-2-2-3

Failures, assignments and disassignments are all *solver events*.

Explanations are intended to connect (dis-)assignments and failures with their causes, which for present purposes will be other (dis-)assignments.

Definition 2.5. An *s-explanation*[11] for a solver event e is a set of assignments that are sufficient for the solver to infer e by some unspecified method. An explanation for e is *minimal for propagator* P if no event can be removed from it while still allowing a specific propagator P to infer e. An explanation is simply *minimal* if it is minimal for a GAC propagator. An explanation for event e is said to "label" the event.

A solver will usually make such an inference as a result of propagation, but the definition leaves open any form of inference.

Example 2.6. *Figure 2.5 shows an example of GAC propagation on NMR-2-2-3 from earlier Example 2.5. GAC has inferred that because* v_{11}, v_{12} *and* v_{21} *are assigned to 2=green,* v_{22} *must be assigned to anything but green, in order to satisfy the constraint. Hence the explanation for* $v_{22} \nleftarrow 2$ *is* $\{v_{11} \leftarrow 2, v_{12} \leftarrow 2, v_{21} \leftarrow 2\}$.

This type of explanation was used for most of the history of learning in CSP, until g-learning[12] was introduced by Katsirelos and Bacchus [**Kat09**] in a major breakthrough that I will describe later. g-learning uses explanations that can be composed of both assignments and disassignments:

Definition 2.6. A *g-explanation* for a solver event e is a set of (dis-)assignments that are sufficient for the solver to infer e.

Clearly an s-explanation is a type of g-explanation. Inference is often based on disassignments, not only assignments. For this reason g-explanations provide a closer match for certain consistencies, e.g. GAC.

[11] the s stands for *standard* and the name is due to Katsirelos [**Kat09**]

[12] *generalised* learning

Example 2.7. *Suppose that a CSP contains the constraint $x = y$. Suppose further that $x \nleftarrow 1$. By GAC the solver can then infer that $y \nleftarrow 1$. The explanation for $y \nleftarrow 1$ is $\{x \nleftarrow 1\}$. This cannot be expressed by an s-explanation involving only variables x and y, because there has been no suitable assignment to x (indeed there has been no assignment to x at all).*

Explanations are not necessarily unique, there may be several minimal explanations for an event. This fact is exploited by at least one learning algorithm [**SV94**].

It is important to emphasise an essential property of explanations when they are used in practice (see [**NOT06**] for an equivalent property used in SMT solvers). Suppose explanation $\{d_1, \ldots, d_k\}$ labels pruning $v \nleftarrow a$:

Property 2.1. All of $d_1, \ldots, d_k$ must first occur before $v \nleftarrow a$.

Remark 2.1. Ensures that causes must precede effects[13]. This simply takes into account that there may exist multiple possible explanations, but I am presently interested in the unique explanation that was actually used to do an inference.

I will now proceed to describe schemes for generating explanations and their characteristics and merits.

2.4.1. Generic techniques. Suppose that all decisions made by the solver are assignments, then $\{v \leftarrow a : v \leftarrow a \text{ is a decision}, v \leftarrow a \neq e\}$ is an s-explanation for any event e, i.e. the explanation for e is all the decision assignments excluding the event itself (if necessary). This is a rather pointless explanation which usually says little about the intuitive reason for e. It works because if all the decision assignments were repeated, assuming the same level of propagation was enforced, the same event must eventually be obtained.

As shown in [**Kat09**], such an explanation can be improved using a generic minimisation technique called quickXplain [**Jun01**]. quickXplain may need to enforce consistency on a set of propagators $O(k^2)$ times where k is the size of the explanation to begin with. Katsirelos claims that this overhead is too large for minimisation to

[13] avoiding cycles in the g-learning implication graph [**MSS96, Kat09**] to be defined later in §2.6.1

be practically worthwhile [**Kat09**]. Also, as I describe in the following sections, often a minimal explanation can be computed directly and efficiently.

Another generic technique is used in Dechter's graph-based backjumping (GBJ) (Section 2.5.4) and *graph-based shallow learning* [**Dec90**]. It exploits the observation that an explanation for a (dis-)assignment to variable v can only include assignments to variables connected to v through the scope of one or more constraints. All such assignments are included in the explanation, hence the result is a subset of that obtained by the above generic scheme for s-explanations. The explanations are not necessarily minimal, but are easy to compute. In [**Dec90**] such an explanation for an assignment is called its *graph-based conf-set*.

Dechter [**Dec90**] also suggests how to explain a failure due to variable f[14] having no remaining consistent values: Start with the partial assignment at the point in search where the failure occurs. Remove any assignment from A that is consistent with all values in the failed var f; or, alternatively, ensure that A contains only assignments that are in conflict with at least one value in f. Formally, $v \leftarrow a$ can be removed from A if constraint c with $\text{scope}(c) = (v, f)$[15] is such that $\forall val \in \text{dom}(f)$, $(a, val) \in \text{rel}(c)$, i.e. $v \leftarrow a$ does not conflict with any value of f. This technique is used in *full shallow learning* in conjunction with forward checking [**HE79**].

2.4.2. Generic techniques based on propagation. Since many solver events are derived by propagators, explanations for propagation should be available. As shown in Example 2.7 the reasoning behind propagation routinely cannot be directly expressed as an s-explanation and it was not until Katsirelos and Bacchus' work on g-learning [**KB03**] that more expressive g-explanations were used for this purpose. Katsirelos gave a generic scheme for explaining propagators in his thesis [**Kat09**]: Suppose that event $v_k \leftarrow a$ (or $v_k \nleftarrow a$) is forced by the propagator for constraint c which has $\text{scope}(c) = (v_1, \ldots, v_k)$. The propagator can only have used (dis-)assignments to variables in the set $\{v_1, \ldots, v_{k-1}\}$ in its reasoning. Hence the explanation is the set of all assignments (if possible) and disassignments to those variables. It should be clear that some of these (dis-)assignments had no effect on the

[14] called f because it caused the failure

[15] CSPs are assumed to be binary

propagation in question and so the explanation is imprecise. I will illustrate with an example:

Example 2.8. *Suppose that a CSP contains the constraint $x = y$. Suppose further that $x \not\leftarrow 1$ and $x \not\leftarrow 2$. The propagator for $x = y$ is able to infer that $y \not\leftarrow 1$ and an explanation of $\{x \not\leftarrow 1, x \not\leftarrow 2\}$ would be produced by the above generic scheme. However, as shown in Example 2.7, just $\{x \not\leftarrow 1\}$ is a valid and shorter explanation.*

These explanations can again be improved by a generic minimisation routine like quickXplain.

Prosser's CBJ in its various forms (CBJ, FC-CBJ, MAC-CBJ) can also be viewed as explanation-producing algorithms [**Pro93b, Pro95**]. Indeed, they were used as such in [**FD94**]. I will revisit these algorithms later when I review backjumping (Section 2.5) but for now describe how MAC-CBJ produces explanations for assignments and failures[16].

Each variable v has a *conflict set* $CS(v)$, which is the set of variables whose assignments have directly or indirectly caused the removal of a member of $\mathrm{dom}(v)$:

- When a variable v is assigned in a decision $v \leftarrow a$, $CS(v)$ is set to $\{v\}$. This reflects that the assignment itself caused the removal of several values.
- When a propagator for constraint c with $\mathrm{scope}(c) = (v_1, \ldots, v_k)$ causes an (dis-)assignment to variable v_j, $CS(v_j)$ is assigned to $\bigcup_{i=1}^{k, i \neq j} CS(v_i)$. The new conflict set incorporates the reasons why all the variables in $\mathrm{scope}(c)$ have the values they do, and hence why the propagation happened.

Now the explanation for a failure in variable v (if v has failed) or an assignment to v (if it is assigned) is just the set of assignments variables in $CS(v)$. Such explanations may not be minimal, but are more precise than GBJ or the other generic schemes so far described.

I now proceed to describe explanation techniques that are even more precise, because they take into account exactly why propagators behave as they do.

[16] it is quite easy to extend this to explanations for disassignments as well

	[**OSC07**]	[**Kat09**]	[**Sub08**]	[**Vil05**]	[**RJL03**]	[**SFSW09**]	[**GJR04**]
Product	Y(es)						
Inequality	Y	Y					
Lex$\leq$		Y					
Alldiff		Y			Y		
GCC		Y					
Roots+range		Y					
Table		Y					
BDD			Y				
Scheduling				Y		Y	
Stretch					Y		
Flow							Y

TABLE 2.1. Global constraints and the papers in which they have been given invasive explanations

2.4.3. Specialised techniques based on propagation. The next level of explanation technology is to generate explanations for (dis-)assignments with full knowledge of the propagation that occurred, in other words, inside the propagator. This affords the opportunity to produce explanations with no superfluous (dis-)assignments and to directly encapsulate propagation reasoning.

This general approach has been used before with:

- Propagators enforcing GAC [**Kat09, OSC07**], bound consistencies [**OSC07, Kat09**] and specialised consistencies [**Vil05**].
- Many different constraints, see Table 2.1.
- Different types of explanation including g- and s-explanations.

The essence of the approach is that the propagation algorithm is adapted to not only do disassignments, but also to store an explanation for each disassignment [**JDB00a**]. I will call these *invasive explanations*. Invasive explanations:

- need not be minimal (e.g. Nogood-GAC-Schema-Allowed-log-approx from [**Kat09**]);
- may be computed at the time of propagation (usually) or retrospectively (e.g. [**Vil05, NO05**]); and
- may be either g-explanations [**Kat09**], s-explanations [**Sub08, RJL03**] or something domain-specific [**Vil05**].

Advances in invasive explanations will form a major part of this thesis, however since many of the algorithms of Table 2.1 will be revisited in Section 3.5 in detail I will not describe them here. However to provide an introduction I will now illustrate the technique by continuing the NMR running example.

Algorithm NOT-ALL-SAME-EXPLAINING-PROPAGATOR

```
...    ...
A5     if(howManyAssgsSame == arity - 1)
A5.1       remove a from dom(v_j) where remaining variable is v_j;
A5.2       store explanation {v_i ← a : v_i ∈ V, i ≠ j} for v_j ↚ a
A5.3       allAssgsSame = false;
```

Algorithm 4: Explaining propagator algorithm for "not all same" constraint

Example 2.9. *Example 2.4 contains a propagation algorithm for the "not all same" constraint. An excerpt is reproduced as Algorithm 4. At line A5.2 the explanation is stored.*

This intrusive explanation code is typical: when the propagator removes a value it must also store an explanation. In this case, the amortized complexity of the propagator is unchanged. The explanation is minimal.

2.4.4. Nogoods. The concept of a *nogood* has been common in CSP learning literature (e.g. [**Kat09, Dec03, Dec90**] and others). Nogoods are closely related to explanations:

Definitions 2.3 (Nogood)**.** A *g-nogood* for (V, D, C) is a set of (dis-)assignments that cannot all be true in any solution.

An *s-nogood* is a g-nogood containing only disassignments.

I will now make precise the relationship between explanations and nogoods.

Lemma 2.1. *E is an g-explanation for $v \leftarrow a$ iff $E \cup \{v \nleftarrow a\}$ is a g-nogood.*

Proof. ($\Rightarrow$) If everything in E is true, then the propagation logically determines that $v \leftarrow a$ must hold. Hence $v \nleftarrow a$ must not hold along with E and $E \cup \{v \nleftarrow a\}$ is a g-nogood.
($\Leftarrow$) If $E \cup \{v \nleftarrow a\}$ is a g-nogood, then supposing everything in E were true there

exists a level of inference, for example unit propagation, which would infer $\{v \leftarrow a\}$ using only E, hence it is an explanation for $v \leftarrow a$. □

I now continue to a review of backjumping, which is conceptually easier than constraint learning, and arguably played a bigger part in the early history of CSP solvers.

2.5. Backjumping

Before describing several concrete backjumping algorithms I will illustrate the technique by an example.

Example 2.10. *Consider the problem NMR-2-3-2, but with the additional constraint $v_{13} = v_{23}$. Suppose chronological backtracking (BT) plus an unspecified type of backjumping is used for search. A whole search tree is shown in Figure 2.6. The sequence of decisions and inferences is as follows:*

(1) *Set $v_{23} = 2$.*
(2) *Set $v_{11} = 2$.*
(3) *Set $v_{21} = 2$.*
(4) *Set $v_{12} = 1$.*
(5) *Set $v_{13} = 1$. The constraint $v_{13} = v_{23}$ is failed. The solver backtracks.*
(6) *Set $v_{13} = 2$. The "not all equal" constraint with scope $(v_{11}, v_{13}, v_{21}, v_{23})$ is failed, because all cells are red. The most recent choice point with options still remaining is for variable v_{12}, but changing this option does not help at all: it had no influence on the failure and the solver still cannot give a consistent assignment to v_{13}. Hence it is preferable to* backjump *to reassign v_{21} instead.*
(7) *Eventually a solution will be found after the assignment $v_{21} = 1$.*

At step 6, if the solver were to instead try a different choice for v_{12} it would proceed to fail for the same reason. In BT search this is called *thrashing*. Generally, search algorithms like this which skip parts of the search tree by making multiple steps are called *backjumping* algorithms. Such algorithms avoid thrashing, because they avoid some making some assignments which are certain to fail (however thrashing is still possible even when a form of backjumping is used).

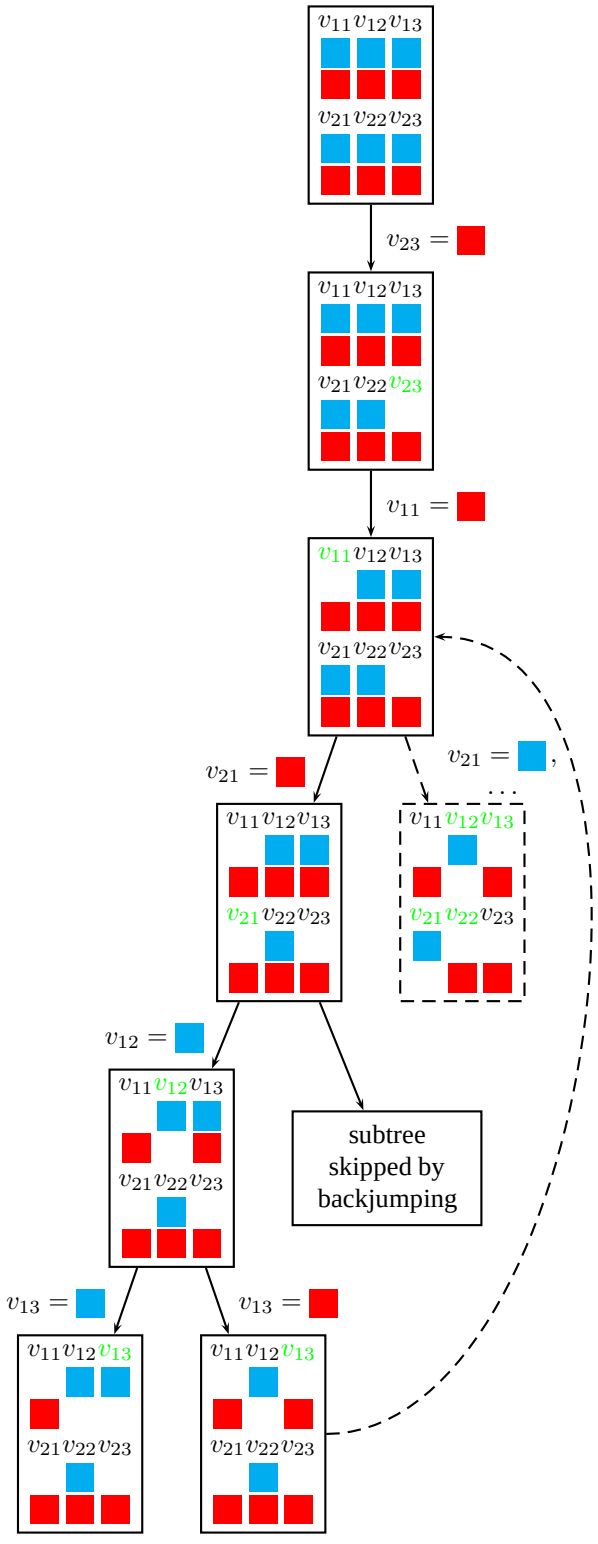


FIGURE 2.6. Partial search tree for BT search with backjumping on NMR-2-3-2

I will now describe 4 standard backjumping algorithms from the literature.

2.5.1. Gaschnig's backjumping. Gaschnig's algorithm (see [**Dec03**], abbreviated to BJ henceforth) is a backjumping algorithm designed for plain BT search, i.e. no propagation. BJ is not precise enough to perform the level of backjumping described in Example 2.10 and furthermore it is only able to backjump from inconsistent leaf nodes, not from internal nodes that have run out of options. However it is space and time efficient, requiring only $O(|V|)$ space and little or no additional overhead over existing consistency checking.

The algorithm works as follows[17]. A leaf decision $v = a$ at depth j is found to be in conflict with a set of earlier decisions, in the sense that the decisions form an s-explanation for $v \nleftarrow a$.

At the conflicting leaf node $v = a$ at depth j, BJ obtains the *most recent* conflicting decision. It does so by maintaining for each variable v a pointer $latest(v)$ which is

- the most recent variable checked against v (when v has not yet been assigned consistently); and
- v itself (once it has been assigned consistently).

In search when a variable v cannot be assigned, the solver jumps back to reassign $latest(v)$, rather than the previous variable as in standard BT.

As I said before the weakness here is that once a variable is assigned consistently once, the solver can only step back from it henceforth. This is a weakness because it could be imagined that after backjumping once, the target variable could not be assigned consistently, resulting in another backjump.

2.5.2. Conflict directed backjumping. CBJ [**Pro93b**] is an advance on BJ, that allows jumps at internal nodes as well as at leaf dead-ends. I will describe the variant that is compatible with MAC search (Algorithm 2), called MAC-CBJ [**Pro95**].

I introduced the explanation subsystem of MAC-CBJ in Section 2.4.2, where I described how to update conflict sets to correctly store s-explanations for GAC propagation. Whenever $\exists u \in V$ s.t. $\mathrm{dom}(u) = \emptyset$, the solver will backjump and reassign

[17] I have adapted it slightly to work with arbitrary variable assignment ordering

the most recently assigned variable v in $CS(u)$ that has other values available. Suppose this assignment is $v \leftarrow a$. The invariant that $CS(v)$ is a valid conflict set has to be maintained, hence it must be updated to explain why the decision was ruled out. To extend this to backjumping, $CS(v)$ must be revised to include variables whose assignments justify $v \not\leftarrow a$, as follows.

Notice that the assignments to variables in $CS(u)$ are not allowed together in any solution, since propagation inferred a failure from them, hence they are a nogood. The nogood can be transformed by logical laws as described in Section 2.4.4 to obtain the explanation $CS(u) \setminus \{v\}$ for $v \not\leftarrow a$. When the backjump occurs, $a \in \mathrm{dom}(v)$ is ruled out, so $CS(v)$ must be updated by setting $CS(v)$ to $CS(v) \cup CS(u) \setminus \{v\}$ after backjump.

MAC-CBJ performs the largest possible backjump based on the evidence in the conflict sets [**Dec03**]. MAC-CBJ has been found to be more advantageous when a low level of consistency is enforced: this is because the benefits of backjumping and consistency are not orthogonal as both have the effect of avoiding failing assignments (see [**BR96**]). MAC-CBJ has been shown to work on some problems, and it is a feature of most learning solvers because once the conflict sets are already available the backjump is easy to compute.

2.5.3. Dynamic backtracking. Dynamic backtracking [**Gin93**] is a backjumping technique that avoids erasing assignments between the current assignment and the most recent variable identified as being part of the reason for a conflict, unlike CBJ. I will not describe it in detail. It has also been adapted to work in combination with arc consistency [**JDB00b**].

2.5.4. Graph-based backjumping. GBJ [**Dec90**] works along similar lines to CBJ, but the requirement to calculate specific explanations for propagation is removed, as the explanation scheme used is the GBJ scheme described in Section 2.4.1. Hence GBJ jumps more conservatively than CBJ, but is theoretically useful as a tool for calculating worst case bounds on the size of a search tree [**Dec03**]. Dechter [**Dec90**] summarises GBJ beautifully using one sentence:

> In short, [GBJ] backs up to the most recent variable among those that are both connected to it by a path of preceding variables, and from which it can continue forward.

2.5.5. Restart. The *restart* backjumping algorithm just backtracks past the root node, to revoke all current branching decisions and start again! Hence it is the least informed scheme possible. However it is useful in practice for various reasons:

- The solver is given a chance to start again, perhaps making better early decisions that will lead it to a solution more directly. This can be useful in the presence of other learning algorithms like heuristics and constraint learning (see start of Section 2.3), where the solver may "understand" the problem better than before.
- When randomised behaviour is present (e.g. a randomised branching heuristic) the amount of time remaining to get to a solution can vary widely, and a restart affords an opportunity to "get lucky" by choosing a different sequence of decisions that lead quickly to a solution.

Indeed these benefits may be afforded by even partially restarting, i.e. jump back to any point between the current decision and the first. A disadvantage is that restarts make search incomplete because the same space may be repeated, however constraint learning can prevent repetition [**LSTV07a, KB03**].

The reasons for the success of restarts are complex and outwith the scope of this thesis, but restarts are a technique that are commonly used in learning solvers. See [**vB06**] for a survey.

2.6. Learning

The type of learning that SAT solvers currently do played a bigger part in the development of current CSP learning solvers (including new developments in this thesis), than learning technology specifically developed for CSP. This is because modern CSP learning solver use algorithms adapted from SAT solvers rather than adapted from algorithms specifically for CSP. For this reason I begin with a discussion of learning in the SAT problem. I will return to CSP-specific learning later in this section.

2.6.1. Learning in satisfiability. My review of learning in CSP begins with learning for the *satisfiability problem* (SAT), which is CSP restricted to Boolean variables and constraints of a specific form:

Definitions 2.4 (Basic SAT definitions)**.** A SAT problem is a CSP where $D = \{0, 1\}$ and $\forall c \in C$, c is a *clause*. A clause is a constraint of the form $(v_{c_1} \vee \ldots \vee v_{c_p} \vee \neg v_{c_{p+1}} \vee \ldots \vee \neg v_{c_{p+n}})$[18]. Such a clause is satisfied when $\exists i \in [1, \ldots, p]$ s.t. $v_i = 1$ or $\exists i \in [1, \ldots, n]$ s.t. $v_{p+i} = 0$. Each v_i and $\neg v_j$ is called a *literal*; also a *positive* and *negative* literal respectively. In effect, the value 0 means False and 1 means True, and a clause is satisfied when any of its constituent literals is True.

I will now introduce a running example of a SAT problem:

Example 2.11. *Suppose you are a civil servant who has been asked to produce invitations for an embassy ball. You decide to encode the problem as a SAT, letting variable X mean the ambassador whose country begins with X, where the countries are Belgium, France, Germany and the Netherlands. For example, $B = 1$ in a solution means the Belgian ambassador attends.*

You are told:

- *You must invite* somebody*: $B \vee N \vee F \vee G$.*
- *The ambassador asks you to invite a Francophone ambassador so his daughter can practice her French: $B \vee F$.*
- *The Belgian, German and Dutch ambassadors are badly behaved when they get together, so they mustn't all be invited: $\neg B \vee \neg G \vee \neg N$.*
- *If you invite the Dutch ambassador, you must also invite the Belgian ambassador: $N \Rightarrow B \equiv \neg N \vee B$*

Given these constraints, one of many possible solutions is that the French and German ambassadors attend, i.e. $B = 0$, $F = 1$, $G = 1$ and $N = 0$.

As with CSP, SAT problems can be solved by complete and incomplete methods, and the foundational algorithm for most complete methods is BT search. Obviously,

[18] strictly speaking, the constraints in a SAT problem can be any Boolean expression involving $\wedge$, $\vee$, $\neg$, variables and parenthesis, but clausal (CNF) format is standard

since every SAT is a CSP, Algorithms 1 and 2 are both complete SAT algorithms. Algorithm 2 (MAC) is a realistic starting point for discussion of a modern SAT solver.

MAC involves enforcing GAC on the set of clauses. When the constraints are restricted to be clauses, as in the SAT problem, only one type of propagator is required. Propagators for clauses use *unit propagation* (UP). Unit propagation is based on the observation that there is only one disallowed assignment for a clause: when every literal is falsified. Consistency is enforced by doing nothing until all but one literal is falsified, and then forcing the remaining literal to be true, so that the whole clause is satisfied. When a clause is able to unit propagate, it is said to be *unit*. Unit propagation is the highest possible level of inference that can be enforced on a single clause.

Note that this propagation algorithm is quite reminiscent of Algorithm 4 which propagates the "not all same" constraint, since both clause and "not all same" are types of disjunction. Generalised schemes for propagating disjunction constraints have been described in [**JMNP10**].

The first major departure from MAC is that learning is done after each conflict, a technique first introduced in the GRASP solver [**MSS96**]. It is assumed that any propagator will ensure that an appropriate explanation is stored when it makes an inference. In the case of UP, when a clause $(l_1 \vee \ldots \vee l_k)$ is used to make literal l_i true, the explanation is the negative of the remaining literals, i.e. $\{\neg l_1, \ldots, \neg l_{i-1}, \neg l_{i+1}, \ldots, \neg l_k\}$.

Example 2.12. *Suppose that the clause $B \vee F$ from Example 2.11 unit propagates to set B true. The explanation is $\{\neg F\}$.*

When $B \vee N \vee F \vee G$ sets B true, the explanation is $\{\neg N, \neg F, \neg G\}$.

Notice that only one explanation is required per variable v. This is because when a Boolean is assigned to b, $1 - b$ is ruled out immediately. That is, $v \leftarrow 0$ is *the same* event as $v \nleftarrow 1$. For example, the explanation for $v \leftarrow 0$ is also valid for $v \nleftarrow 1$.

At any point in search, such explanations can be combined into an *implication graph* (IG), which is a central idea in contemporary constraint learning:

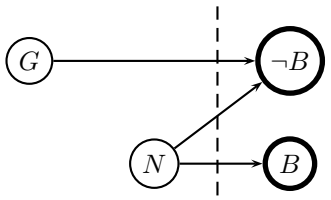


FIGURE 2.7. Implication graph for Example 2.13: IG cut shown by dashed line

Definition 2.7 (SAT implication graph)**.** An *implication graph* for the current state of variables is a directed acyclic graph where

- each node is a currently true literal, e.g. v when variable $v \leftarrow 1$, and
- there is an edge from u to v iff u appears in the explanation for v.

Implication graphs are usually drawn with edges going from left to right, for reasons that will become clear shortly.

Example 2.13. *In the SAT of Example 2.11, suppose the search process has set G, i.e. invite the German ambassador. No unit propagation results. Suppose next decision N, i.e. invite the Dutch ambassador. $\neg N \vee B$ can propagate, to assert B, i.e. invite the Belgian ambassador. However now the clause $\neg B \vee \neg G \vee \neg N$ can unit propagate asserting $\neg B$ (or $\neg N$ or $\neg G$) to prevent the terrible trio of ambassadors being reunited*[19]*. The search process will stop and backtrack at this stage.*

The IG is shown in Figure 2.7.

When a conflict occurs, the IG shows *why* the conflict happened: tracing back from mutually inconsistent nodes (e.g. B and $\neg B$ in Figure 2.7) to decision assignments via inferences. I now define IG *cut*:

Definition 2.8 (Implication graph cut)**.** A *cut* of an IG (V, E) containing mutually inconsistent nodes is a partition (S, T) of V such that

- all nodes corresponding to decision assignments belong to S,
- the mutually inconsistent nodes belong to T, and
- if a node $x \in T$, either all its direct predecessors are in T or all its direct predecessors are in S.

[19] it is equally valid here to stop the search process because that clause is definitely unsatisfied, rather than propagating it to cause a domain wipeout

A cut can be drawn on a graph as a line through the edges in the cut-set, i.e. edges $(u, v) \in E : u \in S, v \in T$. Since an IG is a directed acyclic graph, the cut can equally well be characterised by the vertices in S that are incident to edges in the cut-set.

Example 2.14. *The dashed line in Figure 2.7 is the cut* $(\{G, N\}, \{\neg B, B\})$ *or alternatively just* $\{G, N\}$ *to state the vertices in* S *incident to the cut-set.*

In this thesis, when I say "cut" from now on I am referring to the vertices incident to edges in the cut-set.

Recall from Definition 2.6 that repeating the events of an explanation will result in the same propagation happening again. Hence, repeating the events of any *cut* of the IG will cause identical propagation and failure.

Lemma 2.2. *Asserting all the events of a cut of a failed IG and enforcing the same consistency level as the explanations were built with will lead to the same failure.*

Proof. *By induction* Let $P(k)$ be the statement that if the shortest path from a node n to the furthest node in a cut c is at most k edges, then n can be derived from c by propagation.

Basis step $P(1)$. Immediate from the properties of explanations in Definition 2.6: the events of the explanation are in the cut and are all true, so the event n can be re-derived.

Inductive step Assume $P(k-1)$ is true. The inductive hypothesis shows that all n's direct predecessors can be proved from the cut. Hence n can be proved using them, similarly to the basis step. Hence $P(k-1) \Rightarrow P(k)$.

In particular the inconsistent nodes can be derived again. Hence the same failure will result. □

The previous lemma is a standard result, although the proof is my own.

To avoid failures the solver rules out the conjunction of events in certain cuts, i.e. for cut $\{e_1, \ldots, e_k\}$ clause $\neg e_1 \vee \ldots \vee \neg e_k$ is posted, ensuring that they cannot all become true at once.

The cuts that are typically used in SAT solvers are built in a special way, and have various special properties. Before I reproduce the algorithm, several definitions are required:

Definition 2.9. The *decision depth* of a (dis-)assignment d is how many decision assignments have been made when d occurred (if d is itself a decision assignment it is included in the count). The *sequence number* of d is the number of (dis-)assignments that have happened since the last decision assignment (or 0 for the decision itself).

For any (dis-)assignment d, $depth(d)$ is a pair written $dd.sn$ where dd is the decision depth of d and sn is the sequence number. For example the first decision assignment is 1.0 and the third event after the third decision is 3.3.

Let $expl(d)$ be the explanation recorded for event d.

The type of cut used in SAT solvers is known as a *firstUIP* cut, since it contains the first *unique implication point* (UIP) encountered in a traversal starting with the conflict. A UIP is a node which is on every path from the decision at the current depth to both confict nodes. The firstUIP algorithm to derive such a cut is shown in Algorithm 5. It finds a cut to learn where the initial cut c consists of the negative of all the literals in a failed clause. For example if clause $x \vee y \vee z$ is failed then $c = \{\neg x, \neg y, \neg z\}$.

Algorithm FIRST-UIP-CUT	
A1	**let** c be the set of events directly causing the initial failure
A2	**let** cd be the depth of the most recent decision assignment
A3	**while** $\exists d \in c$, $\exists e \in c$ s.t. $d \neq e$, $depth(d) \geq cd$ and $depth(e) \geq cd$
A3.1	**let** $deepest = f \in c$ with maximum depth
A3.2	$c \leftarrow c \setminus \{deepest\} \cup expl(deepest)$
A4	**return** c

Algorithm 5: Find firstUIP cut

Lemma 2.3. *Algorithm 5 terminates and finds a valid cut.*

Proof. The initial value of c set at line A1 is a cut, by definition. The loop maintains this invariant, because whenever a node is notionally put into set T of the cut, all of its direct predecessors are put into set S. The algorithm must terminate since the loop condition is true at line A1 and must eventually become false, since

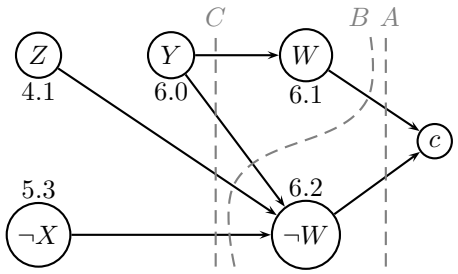


FIGURE 2.8. Implication graph for Example 2.15

there are a finite number of nodes from the current decision depth and the depth of the deepest node is decreasing by Property 2.1. Hence eventually the loop must exit and the algorithm terminates. □

Example 2.15. *The IG in Figure 2.7 is not large enough to provide an interesting input for Algorithm 5. For this reason in Figure 2.8 I introduce a new IG, but do not describe in detail how it could happen during search. Each event d is labelled by depth(d). In addition, this figure shows the cuts Algorithm 5 would have at the start of each repetition of the loop, when it runs on the IG. Cut A consists of the events causing the initial failure. Cut B results from replacing the most recent event ¬W by its explanation {Y}. However this cut has two events from depth 6. Another cut C is produced by replacing W by its explanation {¬X, Y, Z}. Now there is only one event from depth 6, so the final cut has been reached.*

Whenever a failure occurs, a firstUIP cut is built in this way, and a new constraint is added. This firstUIP constraint has the property that when the solver backtracks, the constraint will unit propagate immediately. This is because there is a single disjunct d that became satisfied at the current decision depth and, after backtrack, all but d are false and constraint can unit propagate d. This has the effect of forcing the solver to reverse an earlier decision assignment, hence it has effectively done a right branch.

Example 2.16. *Cut C from Example 2.15 is used to create a new constraint* $X \vee \neg Y \vee \neg Z$. *After a single backtrack: X is still false; ¬Z is still false; but Y is neither false nor true. Hence the constraint is unit, and it will propagate to force* $\neg Y$.

SAT solvers that use conflict driven clause learning now predominate in competitions to evaluate complete SAT solvers, especially on industrial problems [**BRS10**]. This domination owes much to various enhancements that I will describe in Section 2.6.4, but most successful learning SAT solvers that I know of are based on the algorithms in this section. The idea of implication graphs and discovering cuts to disallow were introduced by the GRASP solver [**MSS96**]. The idea of using only the firstUIP cut to learn originates from the Chaff solver [**ZMMM01**]; GRASP used a different cut consisting of all the decision assignments linked to the failure, and other schemes are possible.

The study of *proof complexity* provides a possible explanation for the success of modern SAT solvers. BT search can be viewed as a proof process, so that a complete failed search tree is equivalent to a proof of the empty clause (). This is made precise in [**BKS03**]. Proof complexity is a measure of the *potential* of a proof technique: when the smallest proof*the size of a proof is the number of clauses which are resolved together in order to prove* () for system A is always smaller than the smallest proof for system B then A has a superior proof complexity. The practical importance is that although it may be hard to find a small proof in practice, if no small proof exists it can't be found!

Several proof complexity results relate to SAT solvers. BT solvers produce tree-like resolution proofs [**BKS03, PD09**]. However learning solvers, with restarts (Section 2.5.5), are capable of producing a proof no more than a polynomial factor larger than the smallest general resolution proof. General resolution proofs can be exponentially smaller than tree-like proofs of the same theorem [**BOP03**]. Hence constraint learning solvers are theoretically capable of producing much smaller proofs, and do so in practice.

2.6.2. Various learning schemes for CSP invented by Dechter et al. IGs, as described in the previous section, have been explicitly used in some recent CSP learning algorithms. However in this section I review some algorithms that take a different approach.

Dechter and Frost [**Dec90, FD94**] have created several learning algorithms that can be added to BT (Algorithm 1) to allow one or more nogoods to be learned at each

dead-end. As described in Section 2.4.1, the partial assignment A is an explanation for failure. However it is not advantageous to learn this as a nogood, because BT will not explore this path of search a second time. Dechter and Frost give various schemes for minimising A so that it might reduce search in future, if learned as a nogood:

Graph-based shallow learning [**Dec90**]**:** Remove any assignment from A that does not share a constraint with the failed variable, because it must be consistent with every value in the failed variable. Time complexity: linear in size of assignment.

Full shallow learning [**Dec90, FD94**]**:** Use the *full shallow learning* explanation scheme from Section 2.4.1 to find a set of assignments that cause the failure. Time complexity: quadratic in size of conflict set (defined page 27) and number of values.

Jump-back learning [**FD94**]**:** Only include assignments to variables in the CBJ conflict set $CS(f)$ (see Section 2.5.2 for definition). Time complexity: no additional complexity if CBJ is already being done.

Full deep learning [**Dec90, FD94**]**:** Learn all minimal conflict sets. No implementation is suggested in either [**Dec90**] or [**FD94**], however Dechter suggests doing it by enumeration in [**Dec03**]. Time complexity: potentially exponential *at each dead-end*.

These are given in rough order by how much inference is added by that form of learning. Note that the amount of space required grows with the number of dead-ends so far in search, and additional consistency checking will be required for each additional nogood. Hence speedups are reliant on reducing the number of nodes explored sufficiently to compensate for the overhead of learning. In [**FD94**] experiments show that all the above learning techniques reduce search. However only for jump-back learning is search time reduced and the reduction in search time is not large on the benchmarks they tried (less than an order of magnitude). Nevertheless, this is an advance on the experiments in [**Dec90**] which only showed a reduction in search, but do not include any data on search time.

Dechter and Frost's learning is theoretically interesting because an analysis of worst case search time, search space, reduction in search space size, etc. are included

in the analysis. However Dechter and Frost's experiments show that time savings are modest, in contrast to the practically important speedups obtained by adding learning to SAT solvers. This could be because of the benchmarks used, which are few and academic in nature.

CBJ and jump-back learning has also been incorporated into SAT solvers [**BS97**]. Practical results on industrial problems are good, and this success inspired the type of learning SAT solvers described in §2.6.1.

In the following section I describe newer techniques that unify learning in SAT and CSP, with more promising practical results.

2.6.3. Generalized nogood learning. Katsirelos and Bacchus' work on learning *generalized nogoods* (g-nogood) unifies SAT and CSP learning [**Kat09**]. The fundamental contribution of this work is to introduce g-explanations (Section 2.4); to observe that s-explanations are fundamentally limited in comparison, in several important ways; and to invent a learning scheme for CSP that is significantly superior to those of Section 2.6.2. The superiority of g-explanations over s-explanations is of interest because in the best case Dechter and Frost's learning scheme of the previous section is restricted to using the latter.

First, Example 2.8 showed that the representational power of s-explanations is less than g-explanations. At best, a g-explanation can generalise an exponential number of s-explanations, as shown in the next example.

Example 2.17 (Adapted from [**Kat09**])**.** *Suppose that* $\forall i \in [1, \ldots, m], dom(v_i) = \{0, \ldots, n\}$*. Suppose that each assignment of variables* $v_1, \ldots, v_m$ *to values in the range* $1, \ldots, n$ *is disallowed, i.e.* $\{v_1 \leftarrow 1, \ldots, v_m \leftarrow 1\}$ *through* $\{v_1 \leftarrow n, \ldots, v_m \leftarrow n\}$ *are all s-nogoods. This is a total of* m^n *s-nogoods.*

The single g-nogood $\{v_1 \nleftarrow 0, v_2 \nleftarrow 0, \ldots, v_m \nleftarrow 0\}$ *is failed if and only if any of the above s-nogoods is failed.*

Thus fewer g-nogoods are required to prevent a certain cause of failure, and what g-nogoods are needed are no larger, since an s-nogood is a type of g-nogood.

A second advantage of g-nogoods is that they can propagate better. This follows from Example 2.17, because the g-nogood in that example will become unit before any

of the s-nogoods do, and hence will propagate sooner. This can be proved at follows: for any s-nogood to be unit, all but one variable must be assigned to something other than 0, hence the g-nogood will already be unit by definition; conversely, if only 0 is pruned for all but one variable then the g-nogood will be unit but no s-nogood will be unit because no assignment has yet been made.

Thirdly, g-nogoods corresponding to g-explanations interact more readily during propagation to cause other g-nogoods to propagate. s-nogoods unit propagate as a result of assignments only. When s-nogoods unit propagate they cause disassignments. Hence s-nogoods cannot directly cause others to propagate. They can do so indirectly through other constraints that might infer assignments from disassignments, e.g. the constraint that infers an assignment from the elimination of all but one value. This is not a limitation of g-nogoods, since they can propagate as a result of both assignments and disassignments, and also unit propagate to infer both.

To obtain some of the other advantages of g-explanations it is necessary to put them in the context of a specific learning scheme. Hence I will now describe how Katsirelos and Bacchus incorporated them into their g-nogood learning (g-learning) scheme.

The g-learning scheme [**Kat09, KB03, KB05**] can be characterised as CSP generalisation of GRASP's conflict driven clause learning (described page 36). The major differences are as follows:

Values The values are no longer restricted to 0 and 1. g-explanations must be stored for all events.

Events In SAT, the connection between assignments and disassignments is that an assignment to i is the same event as a disassignment to $1 - i$. In CSP, the connection between assignments and disassignments is more subtle. An assignment $v \leftarrow a$ leads directly to the removal of every value $b \in \mathrm{dom}(v)$ s.t. $b \neq a$. These disassignments must be given the explanation $\{v \leftarrow a\}$. Next, when just one value c remains in $\mathrm{dom}(v)$, assignment $v \leftarrow c$ is implicit, and must be labelled by explanation $\{v \nleftarrow i : i \in \mathrm{dom}(v), i \neq c\}$.

Consistency The constraints might use any level of consistency, not just unit propagation/forward checking. g-explanations must be stored for all inferred events.

In other respects the algorithms are the same: both use implication graphs, the same algorithms for obtaining a cut and create a new constraint after each failure.

The following example describes a complete search using g-learning.

Example 2.18. *The complete search tree is shown in Figure 2.9. The first three decisions have been combined into one step in the tree. They result in setting the first row uniformly to 1 (red). No additional consistency can be enforced. Next* $v_{21} \leftarrow 1$*, resulting in* v_{22} *and* v_{23} *being set to 2 (blue), to avoid the 4 corners of two different rectangles being uniformly red. Finally in this branch of search,* $v_{31} \leftarrow 1$*. Now looking at rectangle* $v_{11}, v_{12}, v_{31}, v_{32}$ *the solver must set* v_{32} *to 2 (blue) to avoid all 4 corners being red. However variable* v_{33} *is involved in two rectangles* $v_{11}, v_{13}, v_{31}, v_{33}$ *and* $v_{22}, v_{23}, v_{32}, v_{33}$*, the former has 3 corners red and the latter has 3 corners blue. Hence propagation will remove values* 1 *and* 2 *from* $dom(v_{33})$ *leaving it empty. This failure is depicted as an implication graph in Figure 2.10. The edges collectively forming one explanation are coloured the same.*

Initial cut A has as little as possible on the conflict side, being immediately to the left of the conflicting nodes $v_{33} = 2$ *and* $v_{33} = 1$*. This cut has two nodes at the current decision depth* 5*,* $v_{32} = 2$ *at 5.1 and* $v_{31} = 1$ *at 5.0. Hence posting this cut as a g-nogood would not be correct, because it would not be unit after backtrack. To address this, the deepest node* $v_{32} = 2$ *is replaced by the nodes in its explanation. This is depicted as cut* B*. This cut has a unique node at the current depth* 5*.*

The cut is converted to nogood

$$n = \{v_{11} \leftarrow 1, v_{12} \leftarrow 1, v_{13} \leftarrow 1, v_{22} \leftarrow 2, v_{23} \leftarrow 2, v_{31} \leftarrow 1\}.$$

I have coloured these nodes pink to make them easy to find at a glance.

The solver now backtracks once to the 3rd node from the top, and posts the new nogood. The nogood is unit, as all but the final literal remain true, and so the g-nogood unit propagates, to force $\neg(v_{31} \leftarrow 1) = v_{31} \not\leftarrow 1 = v_{31} \leftarrow 2$*. This propagation is shown on the right edge as pseudo-decision assignment* $v_{31} = 2$*.*

Finally, another decision is made, the solver finds a solution and stops.

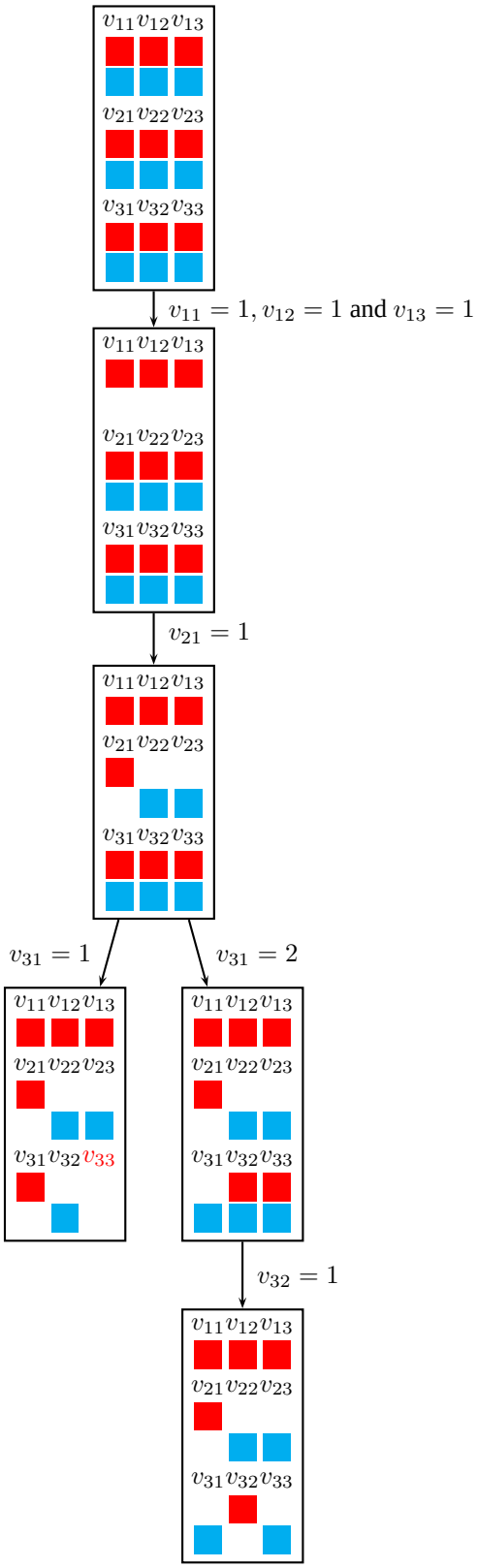


FIGURE 2.9. Search tree of NMR-332 for Example 2.18

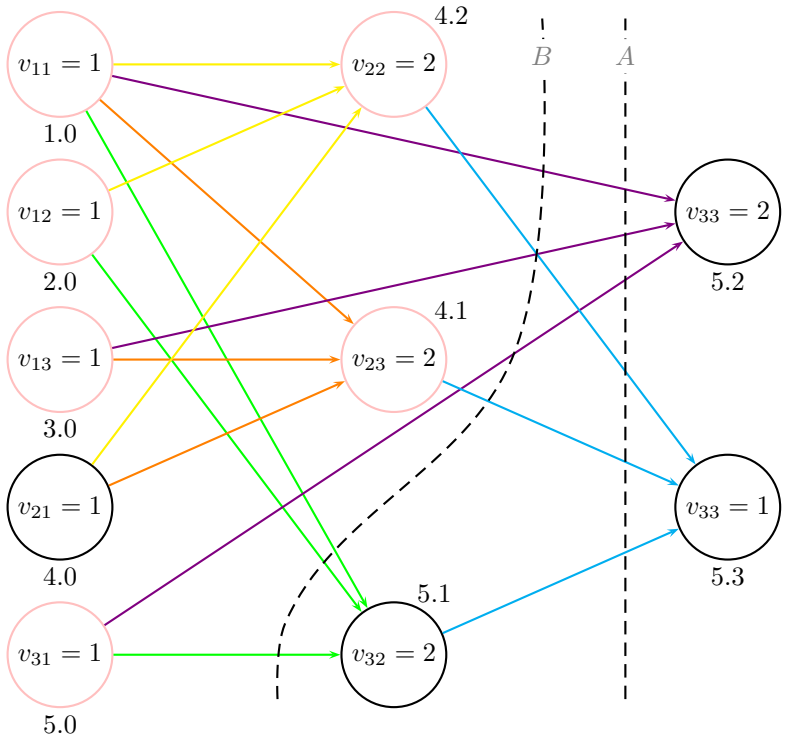


FIGURE 2.10. Implication graph for Example 2.18

Remark 2.2. In the previous example, just one nogood is learned and it is used just once. This is far from ideal, because the advantage of learning comes from nogoods propagating repeatedly and avoiding search in future, and also being themselves incorporated into the implication graph and being generalised further into new nogoods. This is very hard to show in a short example, but is crucial to learning's effectiveness in practice. If learned constraints only ever propagate once, they might as well not be learned, as they are just providing a shortcut to propagation which is already polynomial time.

Katsirelos [**Kat09**] gives several interesting theorems comparing the merits of learning using g-explanations and learning using s-explanations.

The first he calls *local power* which compares the potential for a single learned constraint to cut off branches of search and to propagate. It compares the effect of

- a single jump-back nogood JB (see §2.6.2), which is composed only of assignments; and

- a single nogood FD created using the *first-decision scheme*, which is composed of (dis-)assignments and is therefore a g-nogood.

The theorem shows that anywhere that JB is true, FD is also true. Hence if JB causes a failure, so too will FD. However JB and FD are incomparable in terms of unit propagation, in the sense that either one can propagate when the other does not. The practical repercussions of this are unclear. Since most learning solvers will use unit propagation, FD appears to be no better than JB. I will shortly describe Katsirelos' experiments which suggest FD really is better.

The conclusion from his next theorem is much clearer, it concerns the *global power* of g-learning versus s-nogood learning (s-learning). Global power compares the best and worst case search time with arbitrary variable and value orderings. It is basically a measure of the *proof complexity* of these search methods, defined at the end of §2.6.1. The theorem is that there exists a family of CSPs such that s-learning takes $\Omega(n^{\log n})$ time in the best case whereas g-learning can prove unsatisfiability in $O(n^3)$ time. Conversely, s-learning is a special case of g-learning, so if s-learning can solve a problem quickly then so can g-learning.

As Katsirelos also points out, the weakness of these analyses is that the right is reserved to consider the best possible behaviour of g-learning. Hence in practice, in the presence of different branching choices and different schemes for deciding which cut of the IG to learn, either can still win. However it does show that g-nogood learning has significantly more *potential* than s-learning.

Katsirelos presents various experiments confirming the practical interest of the theoretical results: for many problem classes drawn from a CSP solver competition g-learning usually explores a smaller search tree as expected from the global power theorem. g-nogoods tend to propagate more often than s-nogoods, as expected from the local power theorem.

2.6.4. Enhancements to learning solvers. I have now described the core algorithms used in learning SAT and CSP solvers, however a panoply of enhancements to these basic algorithms contribute a great deal to their efficiency in practice. I will describe the most important of these in this section.

Watched literal unit propagation All the solvers in §2.6 post disjunctions of literals, usually many thousands or millions of them. Furthermore, as observed in [**MMZ+01**] almost all of a solver's time is spent on propagation[20]. To address this bottleneck, a highly efficient means of propagating disjunctions of literals was introduced in the Chaff solver [**MMZ+01**], using watched literals. I will delay describing how it works until §4.3.3 (page 117) when it is in context. It is known to be far superior to alternative techniques. The efficiency benefits are believed to be larger for longer clauses, which are common inside learning solvers (see, for example, [**Kat09**], Figure 3.16), and also for clauses that are currently inactive[21].

Restarts Certain restart strategies are known to improve modern learning SAT and CSP solvers [**Kat09, PD09, Hua07**]. The advantages of restarts were described in §2.5.5.

Heuristics The order in which variables and values are picked for branching decisions can make an enormous difference to the size of the search tree. For example a perfect ordering might obtain a solution without backtracking if one exists, whereas a bad ordering might cause exponential search before a solution is found. Good orderings also help in branches where no solutions exist, because they can induce a failure as soon as possible to avoid unnecessary branching.

Approaches to heuristics vary widely by problem and solver type, but the consensus in the SAT and CSP communities is that constraint learning and heuristics that gain information from recent past failures are complementary. For example, many SAT solvers use a variation on VSIDS [**MMZ+01**] or Berkmin [**GN07**] which both favour assignments that satisfy recently added constraints, in fact Berkmin aims to satisfy *all* learned constraints before trying to satisfying the non-learned ones. In CSP solvers, the domain over weighted degree heuristic [**BHLS04**] has also proved successful.

Unfortunately I cannot cite any evidence for the complementarity of learning heuristics and constraints together, however the SAT solver competition is dominated by such solvers [**BRS10**].

---

[20] this is widely known by people who write solvers, based on informal experiments

[21] I cannot give a reference because this is part of the folklore!

Bounding learning If a constraint is learned at every conflict, the number of constraints in total can grow very quickly and be exponential in the worst case over a complete search tree. Hence various techniques have been implemented for bounding learning, either when constraints are first discovered (e.g. only short constraints are allowed [**Dec90**]) or later on (e.g. only constraints that are involved in many failures are kept [**ES03**]). This decision has been shown to make a big difference to the efficiency of SAT solvers [**GN07**] and a g-learning solver without bounding of some sort runs out of memory fairly quickly. I defer further discussion of this issue until Chapter 4 which is entirely about this subject.

Completeness A solver which both restarts and bounds learning is in danger of being incomplete. After a restart it is possible for the solver to repeat past search forever. Bounded learning allows this problem, because if the learned constraint that rules out a particular failed branch is removed then there is nothing to prevent the branch being repeated.

A correct solution is to always increase the interval between restarts or the number of constraints allowed, so that eventually the entire search space can be traversed before an incompleteness is introduced (assuming there is enough memory available). So that a free choice of both strategies is available a technique has been devised [**LSTV07a**] to learn some constraints immediately before a restart, to prevent the space explored since the last one ever being explored again, even partially. The advantage of the approach is that an exponential amount of search can be ruled out by constraints that occupy cubic space (specifically $O(n^2d)$ space where $n = |V|$ and $d = |D|$).

The technique works as follows. At the point when a restart is taken, the solver has a current sequence of decisions $d_1, \ldots, d_n$. Some of these will be left branches (assignments) and some right branches (disassignments), where the solver has already tried a left branch and been forced to try the opposite in order to succeed. The right branches correspond to left branches that have already been fully explored and a nogood is added for each one to say that no solution exists for it. It is best illustrated by an example, for the technique is simple.

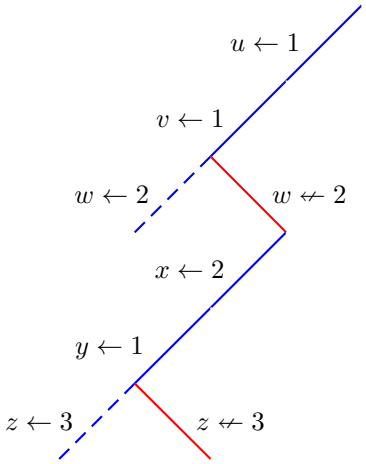


FIGURE 2.11. Search tree for Example 2.19

Example 2.19. *Suppose the current sequence of decisions is* $u \leftarrow 1, v \leftarrow 1, w \nleftarrow 2, x \leftarrow 2, y \leftarrow 1, z \nleftarrow 3$ *where the 3rd and 6th decisions are right branches. This tree is depicted in Figure 2.11. The subsequence* $u \leftarrow 1, v \leftarrow 1, w \nleftarrow 2$ *ends with a right branch. From this it can be inferred that no solution exists extending* $u \leftarrow 1, v \leftarrow 1, w \leftarrow 2$*, else the solver would have found it and terminated during the corresponding left branch. Hence* $\{u \leftarrow 1, v \leftarrow 1, w \leftarrow 2\}$ *is a nogood and can be posted to prevent the left branch being explored again.*

The same applies to the complete sequence $u \leftarrow 1, \ldots, z \nleftarrow 3$*: if any solution existed including* $u \leftarrow 1, \ldots, y \leftarrow 1$ *plus the already explored left branch* $z \leftarrow 3$*, it would have been found already. Hence* $\{u \leftarrow 1, v \leftarrow 1, w \nleftarrow 2, x \leftarrow 2, y \leftarrow 1, z \leftarrow 3\}$ *is a nogood.*

I haven't described how such nogoods can also be *reduced* to make them shorter but equally effective. Details in [**LSTV07a**].

Finally, some solvers ignore this issue and allow an incompleteness, relying instead on luck and good heuristics to find a solution [**BB08**].

Forms of caching Clause learning can be classed as a form of caching, where earlier failures are remembered, but what about earlier successes?

#SAT is the problem of counting the number of solutions to a SAT, without necessarily finding them. A solver has been created [**SBB$^+$04**] that learns by remembering problem components that it has already seen and using the counts again.

It is common in SAT solvers to recall which value of a variable was last successfully assigned, and to do the same next time [**PD07**]. This is useful because considerable search may be needed to assign a subset of variables consistently, and it is better not to repeat this work after a restart. This technique has been shown to speed up search. *Dynamic backtracking* is a related technique where assignments can be maintained during backtracking, meaning that fewer consistent assignments are lost during backtracking.

Multiple constraints per conflict Most of the learning SAT and CSP solvers descibed above learn only one constraint per conflict. Dechter's full deep learning of §2.6.2 is the exception which adds a constraint corresponding to every minimal conflict set. In addition, [**ZMMM01**] surveys two schemes to learn multiple constraints per conflict. The first is called allUIP and involves adding not only the firstUIP (see §2.6.1) but also cuts for every other UIP at the current decision level. The second is "GRASP" learning which originates from the GRASP solver [**MSS96**]. I will not go into fine details as I do not believe they are instructive; both these techniques were found to be inferior to firstUIP alone [**ZMMM01**]. In [**FD94**] full deep learning was inferior to the schemes that learned just one constraint. To my knowledge this technique has not been tried in learning CSP solvers.

Far backjumping MAC-CBJ is described in §2.5.2. Some practical learning solvers (e.g. [**ES03**]) perform *far backjumping*[22] meaning that they jump a little bit further than CBJ. Suppose that $CS(v_{11}) = \{v_1, \neg v_3, v_5\}$ and for the sake of argument these assignments were made at depths 1, 3 and 5 respectively. In this situation CBJ would jump back to depth 5 and set $\neg v_5$. However modern solvers would also revoke the decision at depth 4 before setting $\neg v_5$. Abstractly, the solver backjumps as far as possible while ensuring that the new constraint unit propagates, whereas CBJ backjumps as little as possible ensuring that the new constraint unit propagates. However some SAT solvers do exactly the same level of backjump as CBJ (e.g, [**MSS96**]).

Both variants are complete. The authors of [**ES03**] assert that far backjumping is more effective in practice in conventional SAT solvers, but the authors of [**SBK05**]

[22] name originates from [**SBK05**]

assert that in a #SAT solver far backjumping is inferior. I have not seen experiments empirically evaluating SAT solvers on the basis of backjump depth.

2.6.5. State based reasoning. There exists a strand of research in CSP which is complimentary to g-nogood learning. It is characterised by representing conditions for inconsistency using *partial states*:

Definitions 2.5 (Partial state, dominance and inconsistent partial state)**.** A *partial state* (PS) Δ is a set of variable and subdomain pairs (where each variable appears at most once) denoted $\{v \in \{a_1, \ldots, a_k\} : v \in V, \{a_1, \ldots, a_k\} \subseteq \text{initdom}(v)\}$. A domain is dominated by a partial state Δ when $\forall v \in V$, if $v \in \{a_1, \ldots, a_k\}$ is found in Δ then $\text{dom}(v) \subseteq \{a_1, \ldots, a_k\}$. An inconsistent partial state (IPS) Δ is a PS such that every domain dominated by Δ has no solutions.

Example 2.20. *Consider a CSP with* $V = \{x, y\}$ *and* $initdom(x) = initdom(y) = \{1, 2, 3, 4\}$. *The following is a PS:* $\{x \in \{3, 4\}, y \in \{1\}\}$.

When the current domain is dominated by an IPS then the CSP has no solutions based on that domain, and hence search can backtrack without considering that branch any further.

Example 2.21. *This example uses the same variables and domains as the previous one. Suppose that* $\{x \nleftarrow 1, x \nleftarrow 2, y \leftarrow 1\}$ *is a g-nogood. An the equivalent IPS is given by* $\{x \in \{3, 4\}, y \in \{1\}\}$. *Intuitively, this means domains where* x *is either 3 or 4 and* y *is 1 cannot lead to a solution.*

Hence IPS is a dual concept to nogoods, because both describe a condition under which no solutions are possible. In fact, g-nogoods and IPSs each can be converted straightforwardly into the other [**Lec09**] (page 470).

IPSs can be used in two ways during search: to cut off branches which are dominated and to prune future failing paths by enforcing a consistency on them. Conversely g-nogoods discovered using an implication graph have to my knowledge only ever been used for the latter purpose.

To prune failing paths (not enforcing consistency), IPSs can be stored in a table: all known IPSs are stored in a hash table and at each node of search if the PS for the

current node is found in the hash table then search can backtrack immediately. This representation is called *transposition tables* [**LSTV07b**].

Several GAC propagators been proposed to propagate IPSs as normal constraints: one using watched literals ([**Lec09**], p466) and one not using WLs [**RM03**]. Interestingly when g-nogoods are propagated using a standard clausal propagator, GAC may not be obtained [**Lec09**] (page 469–470).

Now I have described what IPSs are and how they are propagated. It now remains to give a brief overview of how they are discovered. Published techniques begin with an IPS corresponding to the current domains at the time of a dead end. Parts of the IPS are dropped using these algorithms, in an attempt to make it generalise the node at which it is found, so that it will apply later to other similar branches. I will not describe these techniques but a detailed description can be found in [**Lec09**] (Chapter 11, Section 2).

2.6.6. Lazy clause generation. Recently, a series of solvers primarily identified as doing *lazy clause generation* has emerged [**OSC07, OS08, OSC09, FS09**]. These could broadly be described as an attempt to make g-learning robust and well engineered. I will describe below various novel implementation techniques as well as new techniques in learning. This work gives further evidence that nogood learning is a valuable technique for CSP solvers and that use of CSP rather than SAT need not result in slower search. The published work, however, suffers from a lack of justification for many of the techniques introduced, so that it is in some cases unclear why a technique is being used. Below, when I describe the differences compared to g-learning, I will say if an empirical justification is available.

Lazy clause generation (LCG) The principal selling point of the work is a new way of propagating constraints combined with explanation mechanism. Examples 2.3 and 2.4 show how conventional propagators work in CSP solvers: a piece of code runs, and any inconsistent values are removed from the domains. When explanations are required (as shown in Example 2.6) an additional algorithm is used to obtain an explanation. LCG takes a completely different approach. Instead of doing a pruning directly, a propagator just returns explanations for the pruning it wishes to make.

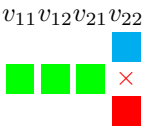


FIGURE 2.12. Illustration of disassignment for NMR-2-2-3

The explanation is then converted to a nogood (§2.4.4) and posted *into a SAT solver* which does all the propagation.

Some constraints may be converted to SAT entirely before search begin, rather than being translated lazily. In [**OS08**] the authors say that this works well when constraints have a small extension, but this is not empirically justified.

The overwhelming advantage of LCG technique is that the CSP solver inherits the learning, heuristics and propagation efficiency of the SAT solver, but with the addition of advanced constraints propagators from the CSP side. There is nothing in a SAT solver that couldn't be done in a CSP solver *in principle*, but SAT solvers have already benefitted from years of work and there is no point in doing it again.

Variable representation Before providing an example of LCG, I must describe how CSP variable states are represented in SAT. Two types of Boolean variables are available:

equality: $\forall v \in V$, $\forall a \in \mathrm{dom}(v)$ $[\![v = a]\!]$ is true iff $v \leftarrow a$

inequality: $\forall v \in V$, $\forall a \in \mathrm{dom}(v)$ $[\![v \leq a]\!]$ is true iff v takes a value less than a

Hence assignment, disassignment and bound events can all be represented by literals and negated literals using these variables. In early versions of LCG [**OSC07**] all of these variables were created before the start of search, however the space requirements are high for large domains, so in an implementation [**FS09**] they are created only when needed during search. The addition of single literals for inequalities allow LCG explanations to be more concise than in g-learning.

Example 2.22. *The LCG expression* $[\![x \leq 500]\!]$ *where* $dom(x) = \{1, \ldots, 1000\}$ *expands to* $x \nleftarrow 501 \wedge x \nleftarrow 502 \ldots x \nleftarrow 1000$ *in a g-explanation.*

Propagation I can now give an example of how propagation works in LCG:

Example 2.23. *Recall Example 2.6 using NMR-2-2-3 where the domains before propagation are as shown in Figure 2.12. A LCG solver invokes the "not all same" propagator, which instead of pruning* $2 \in dom(v_{22})$ *will instead return the clause* $[\![v_{11} = 2]\!] \wedge [\![v_{12} = 2]\!] \wedge [\![v_{21} = 2]\!] \rightarrow \neg[\![v_{22} = 2]\!]$. *The clause is then posted into the SAT solver component which unit propagates immediately to force* $v_{22} \nleftarrow 2$.

Search control In the early versions of LCG, search was controlled by the SAT solver component, with CSP propagators being invoked only when the SAT solver reached a unit propagation fixpoint but not a failure. In the latest version described in [**FS09**] a CSP solver controls search and the SAT solver is used mainly to unit propagate clauses, produce conflict clauses and for heuristic guidance by VSIDS, depending on the heuristic enabled. The latter variant was shown to be faster in [**FS09**].

Empirical evaluation LCG solvers have been shown to be faster than the alternative of solving the static SAT conversion of a CSP [**OSC07**]. Practical results vary in comparison to conventional non-learning CSP solvers. In the authors' experiments, LCG usually beats the conventional solver. However in the authors' own competition [**SBF10**] a state-of-the-art solver with no learning wins overall. It is impossible to tell how a conventional CSP solver with learning compares with LCG, for they have not been directly compared in any publication.

Results in [**FS09**] suggest that search heuristics and learning are reponsible for the time saving, rather than the new propagation technique itself. This is because on any instance where the conventional search tree is no more than 10 times larger than the LGC tree, the conventional solver takes less time, suggesting that is is faster at doing the combination of search, propagation and backtracking. However on many instances the LCG search tree is much smaller, which can be attributed to learning.

Symmetry Symmetry breaking techniques avoid searching parts of the search tree that are symmetric to those already explored, for example, if variables x and y are involved in the exact same constraints and hence whenever assignment $x \leftarrow a$ is inconsistent, so is assignment $y \leftarrow a$. Dynamic symmetry breaking algorithm SBDS [**GS00**] posts s-nogoods after each failure, ruling out each symmetric equivalent of the set of all assignments to the failure. In [**SG10**], this algorithm is applied to firstUIP

g-nogoods rather than the "all decisions" s-nogood. In practice this results in shorter nogoods and more propagation. The associated implementation in an LCG solver is shown to solve several benchmarks faster than just SBDS or LCG alone.

2.6.7. Satisfiabiity modulo theories. Satisfiability modulo theories (SMT) is a technique that is similar to CP in many ways, including that both

- are constraint problems richer than SAT; and
- commonly use backtracking search, propagation, backjumping and learning [**NOT06**].

However the practical usage and fine details of their respective solvers differ greatly. In SMT the emphasis is on extending SAT with a selection of *theories* to allow selected problems to be modelled more directly and solved more efficiently. In CSP, the emphasis is on providing a rich and general set of constraints for modelling any appropriate problem, and that the solvers should expose many options and strategies to the user. I will first give an example of a typical SMT model, and then describe the highlights of the solvers.

Example 2.24 (Based on [**Gor09**])**.** *SMT commonly finds application in hardware and software verification problems. In programming language type systems, variables may be given a complex* refinement type *such as* $x : x \geq 0$ *(read "*x *such that* $x \geq 0$*"). The requirement on the type system is to prove that given the input types and operations carried out, the output can never differ from its type. Consider the following code and suppose* $+ve$ *is the positive integer type:*

$$\textbf{\textit{function}}\ foo(x : +ve, y : +ve) : +ve = \textit{if}\ x > y\ \textit{then}\ y - x\ \textit{else}\ 42$$

Now the aim is to automatically prove that the result must be +ve. This is done by trying to find an assignment to x *and* y *such that the result is -ve, which can be*

modelled as:

$$\begin{aligned} & x > 0 \\ \wedge\ & y > 0 \\ \wedge\ & (x > y \rightarrow y - x \leq 0) \\ \wedge\ & (x \leq y \rightarrow 42 \leq 0) \end{aligned}$$

The first and second terms type x and y. The third term checks the result if the condition is true. The fourth term checks the type if the condition is false. If all 4 can be satisfied at once it denotes a type error. Hence, if a solution is found, a type error exists; if the solver finds no solution, the program is well-typed. In this case a solution exists when $x = 10$, $y = 9$ so the program is wrongly typed.

This SMT example is basically a SAT, except literals have been replaced by *theory terms* like $x > y$. In this case the theory is *linear arithmetic* [**KS08**] since only numbers and operators like $+$ and $<$ are used. It is also a type of CSP, where domains are infinite. Many other SMT theories exist [**KS08**], and there is an ongoing challenge to create a theory of CSP constraints, so that SMT will fully encompass CSP [**NORCR07**].

I will now describe in outline how contemporary SMT solvers work [**NOT06**].

Search A standard SAT solver drives the search, by ignoring the meaning of the theory atoms, and just trying to find a consistent assignment to the disjuncts. In Example 2.24, in order to satisfy the underlying SAT, atom $x > 0$ *must* be true. Once the SAT solver finds a consistent assignment to the SAT part, it invokes a decision procedure on the theory to see if it's a solution to the whole problem.

Decision on theory If the theory part is consistent then the theory solver would return true. However if two conflicting theory atoms, e.g. $x > 0$ and $x + 1 < 2$, were both set true by the SAT solver the theory solver would return a nogood which would be added to the SAT in order to avoid that mistake in future.

Theory propagation The theory solver may be used in an *on-line* mode whereby every SAT decision is communicated to it. If it finds that one decision forces another on the theory side, e.g. $x > y$ forces $y - x \leq 0$ to false, it would notify the SAT solver

of the new assignment. This is an effective technique that ensures wrong decisions are discovered quickly.

Learning and theory explanations The SAT solver does learning as described in §2.6.1. Explanations must always be available for theory propagations, so that conflict analysis can find the cause of the failure. These explanations are produced by the theory solver on demand. I will discuss theory propagation and explanations in more detail in Chapter 3.

2.7. Conclusion

This concludes my review of the relevant background literature in CSP and learning. However in each of the following 3 research chapters I will include a section called "Context" with specialised literature specific to that chapter.

This chapter has shown that learning in CSP has a long history and that the literature is well developed. Explanations are a fundamental concept that appear in disparate areas including learning, backjumping, dynamic CSP and user interaction. Bounding the number of constraints that are learned has also been an enduring subject for research in CSP and SAT. However I have identified some new ideas that I will develop in this thesis.

First, explanations are widely used but have never been computed lazily in a CSP solver, i.e. only when needed. Chapter 3 shows that putting this insight into practice yields significant improvements in CSP technology.

Second, there is a need for consolidation in the area of forgetting constraints. There has been little research focussed on throwing away learned constraints in combination with g-nogood learning. Furthermore forgetting techniques, although effective, are relatively poorly understood. In Chapter 4, I will address these problems.

Finally, it is curious that in all the constraint learning algorithms surveyed, the set of learned constraints is essentially a SAT representation of the conflicts of the CSP. Chapter 5 develops the idea of making this representation more general by allowing arbitrary constraints to be learned.

Chapter 3

Lazy learning

Having gathered these facts, Watson, I smoked several pipes over them, trying to separate those which were crucial from others which were merely incidental. There could be no question that the most distinctive and suggestive point in the case was the singular disappearance of the door-key. A most careful search had failed to discover it in the room. Therefore it must have been taken from it. But neither the Colonel nor the Colonel's wife could have taken it. That was perfectly clear. Therefore a third person must have entered the room. And that third person could only have come in through the window.

Sherlock Holmes

The Crooked Man

by Arthur Conan-Doyle

In solving a problem of this sort, the grand thing is to be able to reason backwards. That is a very useful accomplishment, and a very easy one, but people do not practice it much.

Sherlock Holmes

A Study in Scarlet

by Arthur Conan-Doyle

In the preceding literature review on backjumping (§2.5) and learning (§2.6) I showed that explanations are a useful concept in learning constraint solvers: they provide a way for the solver to introspect into *why* it made the decisions it did, and, when used to explain the reasons for a conflict, can be built directly into learned constraints or used to backjump. In this chapter I introduce a new technique for calculating explanations called *lazy learning*, which can dramatically reduce the time and space overhead of using explanations. To prove that the technique is useful in practice I will describe how laziness can be applied to various global constraint propagators, and say for each one when it can be expected to perform better than the alternatives. Finally, I will show experimentally that it is successful in speeding up a g-learning constraint solver in the average case.

3.1. Motivation

Recall that propagators remove values and cause assignments and, in a learning solver, an explanation must be available for each (dis-)assignment. To date, all constraint solvers and most other solvers have always stored these explanations at the same time as the propagation occurs. However, there is no guarantee that every explanation will be used, and hence if the effort of producing explanations can be delayed in a work-efficient way then time could be saved. The technique that is the subject of this chapter, *lazy learning*, seeks to achieve this by storing the minimum of data at the time of propagation, so that the explanation can later be reconstructed. Before describing how it works, I will briefly justify the potential gains available.

Figure 3.1 depicts, for a large set of instances to be formally introduced in §3.6.2, the number of explanations computed eagerly (at the time of propagation) divided by the number computed lazily (only when needed) when solving instances using g-nogood learning (described in §2.6.3). This shows that usually fewer than half the total number are ever needed, and sometimes fewer than 1%. This roughly corresponds to a target speedup of between 2 and 200 times in the explanation subsystem, depending on the instance, being optimistic and assuming that work can be done later in a comparable amount of time. But explanations are only one part of the constraint system, so is a speedup even worthwhile? In fact, producing explanations is likely to

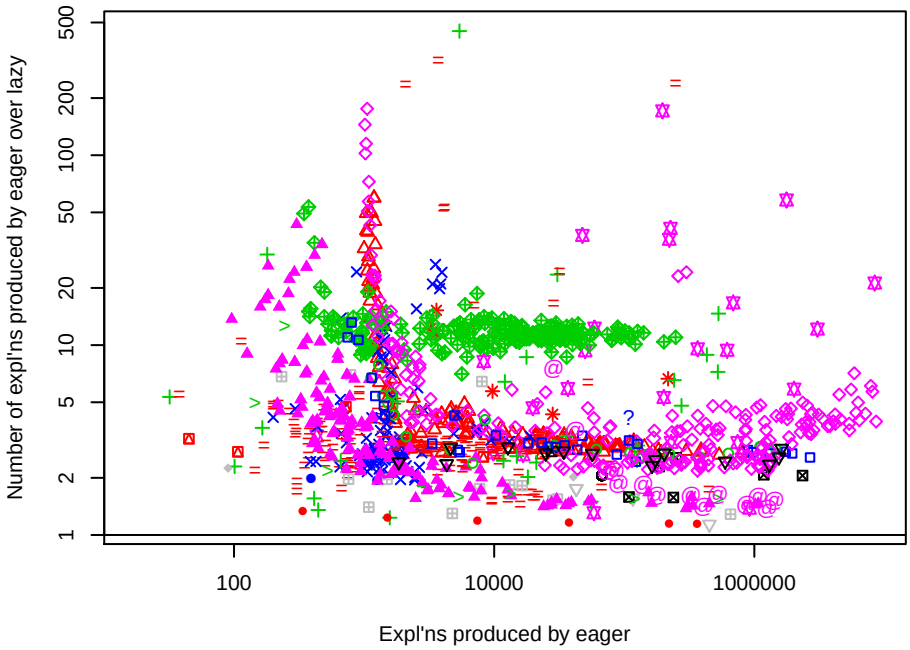


FIGURE 3.1. Scatterplot of various instances. x dimension is time to solve for eager solver. y dimension is ratio of explanations produced by eager solver over explanations produced by lazy solver. Legend is omitted until instances are introduced: each colour and symbol pair denotes an instance family.

be the second biggest task a learning solver must perform, after propagating learned clauses. Suppose that $p\%$ of the solver's time is spent generating explanations, then the expression $100/(100-p)$ is the target speedup for the whole solver available from a zero-cost implementation of explanations, and this is the target of lazy explanations.

In §3.5 I will analyse exactly how efficient lazy learning is compared to eager learning, for each individual global constraint propagator. However it is also interesting to speculate when lazy learning might be particularly good and bad, and why:

Good:

when explanations are most expensive to produce: time wasted calculating them is greatest

when explanations are least expensive to produce: it might be more expensive to store the data than to recompute

when particular variables and values are rarely involved in conflicts: their stored explanations are rarely needed to explain conflicts

when restarts are used: occasionally all explanations are thrown away

less backtracking: the more assignments that are solutions, the less conflict analysis is needed

scarce memory: if memory is very scarce, better not to store speculatively

decoupling: it is possible to completely decouple the implementations of propagators and explanations

Bad:

easy to store: perhaps after doing the propagation it is very easy to do the extra step of storing the explanation, but with laziness the effort will have to be repeated

3.2. Design

I will now describe the basic idea of lazy learning. The essence of lazy learning is that when a propagator does a (dis-)assignment, it must provide a data record and function to the solver runtime system, that can be used to compute the explanation later[1]. Subsequently in the same branch, i.e. when *additional* assignments and prunings have been done, if an explanation is requested the solver will execute the function using the data as a parameter[2]. This function is expected to return a valid explanation, e.g. a g-explanation. The reason for having Property 2.1 (page 25) is now clearer: lazy explainers could potentially pick (dis-)assignments that happened after the event it is trying to explain, however part of the proof that learning is complete (Lemma 2.3 (on page 39) relies on the fact that they don't. I will assume from now on that the data always includes the event to be explained and the propagator that caused the event. When the record contains only these I will call it a *minimal record*. The function may access propagator state, domain state and its parameter in order to compute the explanation.

Contrast this with an explanation stored eagerly: at propagation time, a complete explanation is stored. When it is requested later on it is returned from storage. Hence eager explanations are a special case of lazy explanations, where the record consists

[1] this is very similar to a common implementation of a *thunk*, or postponed computation, in lazy functional language implementation

[2] for example, in my implementation the data is an object and the function is a polymorphic function of the object

of the entire explanation, and the function just returns it. This is practically useful because if individual constraint types don't benefit experimentally on the instances that the solver is designed for, eager explanation can be used for those constraint types alongside lazy explanation for the remainder. That is, for each propagator, the implementer chooses how lazy to be to maximise efficiency. Hence there is no risk in building the generality of laziness into the solver, with a few exceptions I will describe in §3.3.

In order to show that lazy explanations are correct, it must be shown that every explanation, however it is generated, conforms to the appropriate definition of an explanation, e.g. Definition 2.6 and Property 2.1 for g-learning. This I will do for individual propagators in §3.5.

I will now give an outline of the execution of a lazy learning solver, based on Example 2.18 on page 45. The idea of the example is to show that the implication graph is built up gradually as it becomes relevant.

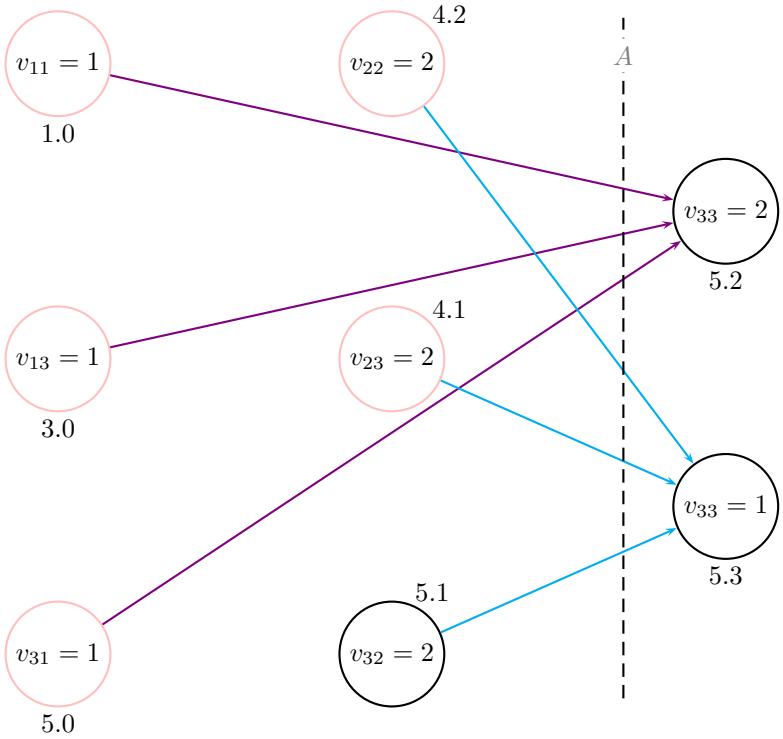
Example 3.1. *This example is on instance NMR-3-3-2. The search tree explored can be seen in Figure 2.9 and a complete IG for the first failure is in Figure 2.10.*

Figure 3.2(a) depicts the known explanations after the first cut has been computed. All the other explanations are simply unknown. The deepest node in the cut is $v_{32} = 2$*. Hence its explanation is generated by calling the lazy explanation function associated with the "not all same" constraint. The unique valid explanation is then built retrospectively: variables* v_{11}, v_{12} *and* v_{31} *are all assigned to the same value so* v_{32} *must be different, and the explanation is just the set* $\{v_{11} \leftarrow 1, v_{12} \leftarrow 1, v_{31} \leftarrow 1\}$ *of assignments. This explanation is returned and incorporated into the implication graph. Now the known implication graph is as shown in Figure 3.2(b).*

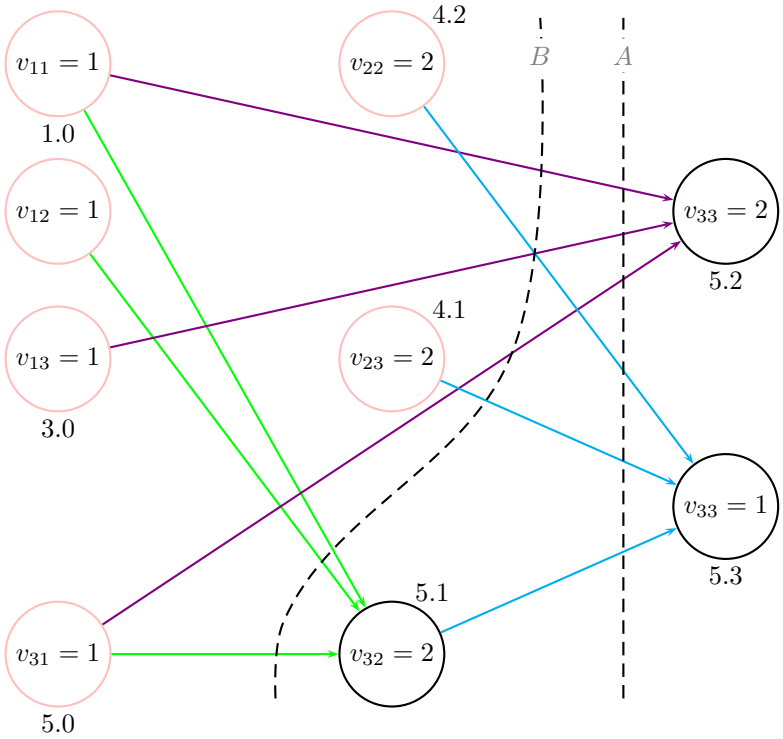
Hence the implication graph is built as it is needed, in this case until a firstUIP cut is obtained.

3.3. Context

§3.2 describes the basic ideas of lazy explanations. The idea is simple, powerful and has never before been tried in a constraint solver. The purpose of this section is to review similar ideas which have appeared in the past.



(a) Explanations needed to compute first cut



(b) Explanations needed to compute second cut

FIGURE 3.2. Implication graph being lazily computed

3.3.1. Jussien's suggestion. The broad idea of using lazy explanations in a constraint solver was suggested but never pursued by Jussien in the "Future work" section of [**Jus03**]:

> regarding explanation computation, the following topics spring to mind [...] computing explanations in a lazy way (keeping less information).

3.3.2. Explaining theory propagation in SMT. I gave an overview of SMT in §2.6.7. In [**NOT06**] two strategies for generating explanations are contrasted:

(1) When a theory propagation is due, add and propagate the corresponding "theory lemma" to the set of clauses rather than just setting the appropriate literal. The theory lemma is the clausal equivalent of the explanation for the propagation, that is, the clause $\neg l_1 \vee \ldots \vee \neg l_n \vee l$ where $l_1 \wedge \ldots \wedge l_n$ is the explanation for l (see §2.4.4). This lemma participates in conflict analysis like a normal clause. This is analogous to an eager explanation. Note that lazy clause generation constraint solvers work similarly [**OSC09**], replacing "theory" by "constraint".

(2) Do not generate an explanation immediately, but instead wait until the explanation is needed in the course of conflict analysis. This is based on the observation that theory propagations are "around 250 times more frequent than resolution steps with explanations"[**NOT06**].

It cannot be disputed that this is exactly the same idea as lazy explanations in spirit. However g-learning constraint solvers differ from SMT solvers because the set of constraints are different and most of the difficulty in using and evaluating lazy explanations is in choosing the correct way of explaining individual constraints. Indeed, there is considerable interest in using CSP propagation algorithms in SMT solvers [**NORCR07, BM10**] and this chapter and associated published papers describe some of the foundational algorithms needed to achieve this.

There are a couple of other occasions that I know of where constraints have been explained lazily in the past and it is the subject of the next two sections.

3.3.3. Lazy explanations for the unary resource constraint. In [**Vil05**] algorithms are given for computing explanations for the unary resource constraint (which I will not describe), used in scheduling, for the purpose of backjumping. The explanations are reconstructed when needed from simpler data stored eagerly during search. In associated experiments between 25 and 80% of explanations are eventually needed, however eager explanations are not tried, so it's impossible to say if lazy explanations were worthwhile. The solver in question was a specialised constraint-based scheduler.

3.3.4. The patent of Geller and Morad. Geller and Morad (IBM) have been granted a patent [**GM09**] for a technique for deriving an explanation for an arbitrary propagator non-invasively and only when required (i.e. lazily). The idea works as follows, suppose that a trail is maintained[3], allowing the domain state of all variables to be reconstructed later. If an explanation is required for an arbitrary propagation $v \nleftarrow a$, it can be obtained as follows:

- $E \leftarrow \emptyset$
- for each variable $w \neq v$ in the constraint
  - set the state of every variable in the constraint to what it was *before* the propagation happened
  - replace the domain of w with its initial domain, i.e. no values removed
  - run the propagator on these domains
  - if pruning $v \nleftarrow a$ does not occur, add all prunings from w to E, i.e. E becomes set to $E \cup \{w \nleftarrow val : val \in initdom(v) \setminus dom(v)\}$.
- return E

This algorithm works because if the pruning is not repeated by the propagator if the domain of w is untouched this proves that at least one pruning from dom(w) had something to do with $v \nleftarrow a$, and they are conseqently *all* added to E.

This explanation algorithm requires the entire propagation algorithm to run v times, which is a major overhead. It can produce explanations that are very bad: consider the case where each domain contains just one pruning that is responsible for

[3]called a log in the patent documentation

$v \nleftarrow a$, the algorithm will add all prunings to all domains when only 1 from each is necessary. Hence it is an impractical algorithm. I know of no experimental evaluation. My solver does not implement this patented algorithm.

3.3.5. Lazy generation for BDD propagation. [**GSL10**] described a propagator for BDDs that evaluates explanations lazily, in a technique they call *lazy generation*. This work was published after my own paper on lazy explanations [**GMM10**] and does not contribute any new technique: the existing explanation algorithm of [**Sub08**] is amended in a similar way to how I amend, for example, the GAC schema propagator in §3.5.3.

3.3.6. Lazily calculating effects of constraint retraction. [**NB94**] contains a technique for calculating the effect of revoking a constraint on a set of domains made arc consistent, i.e. restoring all the values that are no longer arc inconsistent once the constraint is removed. This paper considers how to solve this problem for binary constraints specified by extension. According to [**NB94**], prior to this paper, "reason maintenance systems" were used to store explanations for value removals. However the algorithm in this paper does not store information ahead of time. Instead, when a constraint is retracted candidate values for restoration are found, picked because the removed constraint no longer rules them out. This process is repeated across the constraint network and once all candidates are found a standard AC algorithm is used to remove any candidates identified incorrectly, i.e. a candidate that is inconsistent by two constraints might still be inconsistent when one of the constraints is revoked. This technique is quite similar in spirit to lazy explanations, but instead of producing explanations per-se, it is really finding candidate values that are *possibly* explained by other candidates.

I will now describe some fundamental issues in integrating lazy explanations into constraint solvers.

3.3.7. Integrating lazy explanations into constraint solvers. §2.5 and §2.6 describe two applications of explanations to constraint programming: backjumping and learning. I will now describe difficulties with its integration into various learning algorithms other than g-learning.

3.3.7.1. *Various learning schemes for CSP invented by Dechter et al.* There are a few issues with implementing s-nogood learning (§2.6.2) with lazy explanations. The first problem is that, as shown in Example 2.7, s-explanations for prunings cannot necessarily be expressed using only the variables in the scope of the constraint. Hence it is impossible to be sure that an explanation can be produced using a minimal record. Rather the state of all the variables and all the constraints may need to be taken into account. However s-nogood learning is obsolete (see justification in §2.6.3).

3.3.7.2. *Conflict directed backjumping.* CBJ has a similar drawback to Dechter et al.'s learning, because it uses s-explanations. However once lazy s-explanations are available, it can easily be implemented.

In g-learning, the backjump target is found by analysing the lazily-built implication graph.

3.3.7.3. *Lazy clause generation.* Lazy explanations are a bad fit for lazy clause generation (LCG) (§2.6.6). This is because in LCG, clauses corresponding to the propagation are posted into the solver and these also act as explanations. Since they are expected to propagate immediately and repeatedly the effort cannot be delayed. However in [**FS09**], one of the solver options is to delete clauses after backtracking and for this lazy explanations would be ideal, since lazily posted clauses can be replaced with lazy explanations.

Hence apart from when clauses are deleted after backtracking, the LCG approach to propagation is fundamentally to normal CSP solvers and obviates lazy explanations.

I think it is worth noting that whilst lazy clause generation and SMT are very similar technologies, a wedge can be driven between them mainly in their use of explanations. SMT solvers very sensibly use lazy explanation, whereas for lazy clause generation lazy explanation is inappropriate.

3.4. Implementation of lazy learning

I will now describe how I implement the lazy learning framework in minion, since there are various interesting design choices to make, as well as other important choices like which variable ordering heuristic to use.

3.4.1. Framework. The g-learning solver used is based on release 0.7 of the minion solver, a highly optimised solver that didn't originally contain any learning or explanation mechanisms [**GJM06**]. By convention, I will call the eager learning variant "minion-eager" and the lazy variant "minion-lazy". Implementation decisions are made so that compared to the experiments in [**Kat09**], which use eager explanations, only the method used to produce explanations is varied. Hence dom/wdeg variable ordering [**BHLS04**] and far backtracking as described in [**Kat09**] are used. The solver learns the firstUIP cut. From personal correspondence I know that Katsirelos' solver also uses firstDecision cuts if a loop is detected but the details are unpublished [**Kat08**]. Finally node counts are not directly comparable because I do not know how they were calculated.

Learned clauses are propagated by the 2-watch literal scheme [**MMZ$^+$01**].

3.4.2. Storage of depths and explanations. Recall that an explanation and a depth must be available for each and every (dis-)assignment that occurs. It is a very common operation to request this information, the depth being requested once for each (dis-)assignment involved in the derivation of a new g-nogood and the explanation being requested for a subset of these.

How to implement depth and explanation storage depends on whether the constraint solver uses *copying* or *trailing* to maintain backtrackable state (see [**RSST09**] for a detailed discussion of the choice). Copying means that the entire backtrackable state is copied before a new decision is made, and when search backtracks to that point again it can be copied into place to undo any changes that occurred. In trailing, whenever the state is changed, a record is pushed onto a stack, consisting of the address changed plus the old value. When a new decision is made a NULL record will be pushed. Now when search backtracks the solver will pop records off the stack until the NULL record is reached, restoring the value to the appropriate address each time. In this way the state is restored to its original value.

With trailing, each (dis-)assignment is written onto the stack and is thus implicitly labelled with a depth, since the order on the stack mirrors the order of events. Explanation records can be pushed onto the stack for each (dis-)assignment. It may

seem that the complexity of obtaining the depths and explanations will be unreasonably high: linear in the size of the trail in the worst case, which can contain $|V|.|D|$ records in the worst case, one for each possible (dis-)assignment. It could be even worse if propagators and other solver code have backtrackable state, e.g. the α pointer for the GAC propagator for lexicographic ordering [**FHK$^+$06**]. However, recall from Algorithm 5 that during the firstUIP algorithm, the (dis-)assignments are processed deepest first. Since the explanations and depths are only needed by the firstUIP algorithm, this algorithm can be combined with restoring the trail: the trail can be unstacked until a (dis-)assignment in the current cut in found, at which time the depth and explanations are found. Hence no additional cost is incurred obtaining them, since the trail must be unstacked anyway. Katsirelos [**Kat09**] implements depth and explanation storage using the stack in this way.

Minion uses copying [**GJM06**] and so the stack implementation is inefficient: time spent searching the trail *cannot* be amortized against unstacking it. Hence in my implementation there are two variables for each assignment and each disassignment, one for the depth and one for the explanation record, organised as arrays which are *not* backtracked. Explanations and depths can be obtained in $O(1)$ time. In the trail implementation *validity* is not a problem: if an explanation is on the stack it is for a current pruning. However in minion's implementation of copying invalid explanations and depths are left in the array, and not deleted once they become invalid. This is because copying treats the whole domain state as a piece of uninterpreted memory which is copied into place verbatim, hence the backtracking memory system does not know which values are being restored to the domains. Fortuitously, finding out if an explanation or depth is valid is simple: an explanation for assignment $x \leftarrow a$ (disassignment $x \nleftarrow a$) is valid if and only if x is currently assigned to a (a is not in $\mathrm{dom}(x)$). Hence, invalid depths and explanations can efficiently be left in the arrays but ignored.

3.4.3. Explanations for internal solver events. In minion, variable types implement a couple of rules internally:

- if $x \leftarrow a$, must force $x \nleftarrow b$, $\forall b \in \mathrm{dom}(x) \setminus \{a\}$, and
- if $x \nleftarrow b$, $\forall b \in \mathrm{initdom}(x) \setminus \{a\}$, must force $x \leftarrow a$.

These can be summarised as, respectively, x can take *at most one value* (AMOV) and x *must have a value* (MHAV). They can be treated like clausal constraints, for each variable x the solver implements

- the AMOV constraints: $\forall a, b \in \text{initdom}(x)$, $x \nleftarrow a \vee x \nleftarrow b$, and
- the MHAV constraint: $\bigvee_{val \in \text{initdom}(x)} x \leftarrow val$.

Explanations are then normal explanations for a clause as described in Example 2.12.

3.4.4. Eager and lazy explanations. In my solver, each explanation record is an object representing a (dis-)assignment DA, equipped with methods to return the explanation and depth of DA on demand.

As a final point, [**NOT06**] says that in SMT with lazy explanation "each theory propagated literal may occur in more than one conflict". This suggests that there may be a benefit to keeping an explanation that has been computed lazily, keep it in case it is needed again, i.e. the computational technique of *memoization*. However the authors of this paper appear to be mistaken, as they use firstUIP and the following theorem shows that each explanation can be used at most once.

Theorem 3.1. *Using firstUIP learning, each explanation for a specific solver event is needed at most once.*

Proof. Clearly, in Algorithm 5 any (dis-)assignment for which the explanation is requested is at the current depth (in the algorithm, $depth(e) \geq cd$). These explanations are requested exactly once. However after the constraint is built, the solver backtracks at least one level, and hence the explanations at current depth all become invalid and will never be requested again. □

To be clear, I do not dispute that the same literal may be inferred multiple times using the exact same explanation in different branches of search; only that exactly the same (dis-)assignment can appear twice in different conflict analyses.

3.4.5. Failure. Conflict analysis (Algorithm 5) begins with a set of events that directly caused the initial failure. I will now describe issues with obtaining such a set, also touching on consistency of state at failures. There are two types of failure:

(1) failure detected by constraint, where a constraint detects that it has no remaining consistent assignments and stops immediately rather than removing values; and
(2) inconsistent state, i.e. a variable has no values left in its domain, I will call this a *domain wipeout* in variable x.

3.4.5.1. *Constraint detects.* The first type of failure is dealt with by ensuring that the constraint must return a set of events that are inconsistent. The lexicographical ordering constraint works in this way and I will defer discussion until §3.5.2.3.

3.4.5.2. *Inconsistent state.* The second type of failure is similar because if variable x wipes out then the built-in MHAV constraint is failed. The set of events in the failure is the set of all disassignments to x. However there are some subtleties in the implementation.

Minion deals with 3 types of failure due to an inconsistent domain:

- domain wipeout (DWO);
- assignment and pruning to same value; and
- out of range assignment, i.e. $x \leftarrow a$ and $a \notin \text{initdom}(x)$.

The latter two types of failure involve assignments. It would be possible to sidestep the latter types of failure by transforming them into a DWO. This would be achieved by pruning all but assigned value instead of assigning it directly, but the initial cut is smaller when assignments are allowed.

For a domain wipeout in variable x, the initial cut is the negative of the MHAV constraint, i.e. $\{x \not\leftarrow a : x \in \text{initdom}(x)\}$. This constraint is guaranteed to yield a new and valid firstUIP constraint when Algorithm 5 is applied to it, as I will shortly show, but a preliminary definition and lemma is needed first to make it easier to prove.

Definition 3.1. A constraint propagator C is said to be *subsumed* by another constraint propagator D if D will perform a superset of the propagation that C will irrespective of the domain state.

Lemma 3.2. *(due to* [**Rya02**]*) Apart from a propagator corresponding to the initial cut, and assuming that constraints are propagated in strict order of when they become*

able to propagate, no propagator corresponding to a cut created by Algorithm 5 is subsumed by another constraint propagator already posted.

PROOF. Suppose that a new cut is subsumed by another constraint C already posted. It was built by resolving together two existing constraints denoted $\{x, A\}$ and $\{\neg x, B\}$ in this proof. Suppose without loss of generality (w.l.o.g.) that literal x was true before literal $\neg x$ was forced. Consider the solver state immediately before $\neg x$ is forced. All literals in the new cut are false and hence the corresponding constraint would have propagated in this state, had it been posted. Hence constraint C should have propagated to cause the failure before $\neg x$ was forced. This is a contradiction because that didn't happen and hence there is no C that subsumes the new constraint. □

Now the following lemma shows that a new constraint that is not subsumed by any other will be obtained.

Lemma 3.3. *A new firstUIP constraint is produced by applying Algorithm 5 (page 39) to* $\{x \nleftarrow a : x \in initdom(x)\}$ *when variable* x *has a DWO.*

PROOF. First, there must be at least two (dis-)assignments at the current depth. This is because if there were 0 (dis-)assignments at the current depth then the solver would have had a DWO at the previous depth. If there were 1, then it would have unit propagated the single (dis-)assignment not already assigned at the previous depth, and the conflict would have been avoided. Hence there are at least 2.

Hence Algorithm 5 will iterate at least once, since the loop condition will initially be true, and by Lemma 3.2 there will be a new constraint learned not subsumed by any other. □

If an assignment $x \leftarrow a$ and disassignment $x \nleftarrow a$ occur contemporaneously, there is a choice of which cut to begin with. The easy way is to use $x \leftarrow a$ as a justification for pruning any remaining values in $\mathrm{dom}(x)$, and then using the MHAV nogood as the initial cut. However a smaller initial cut is available: the cut $C = \{x \leftarrow a, x \nleftarrow a\}$. However the proof of Lemma 3.3 does not show that firstUIP will create a new constraint from C. The problem is that either $x \leftarrow a$ or $c \nleftarrow a$ may be from an

earlier depth. In this case Algorithm 5 would terminate immediately and learn a redundant constraint. To get around this a mandatory resolution is applied to C before Algorithm 5 is used.

Lemma 3.4. *Algorithm 5 generates a new firstUIP constraint from* $\{x \leftarrow a, x \nleftarrow a\}$*, except that the most recent out of* $x \leftarrow a$ *and* $x \nleftarrow a$ *should have already been replaced by its explanation.*

PROOF. Suppose w.l.o.g. that $depth(x \nleftarrow a) > depth(x \leftarrow a)$ and hence the starting cut is $C = \{x \leftarrow a\} \cup E$ where E is the explanation for $x \nleftarrow a$. E must contain at least one (dis-)assignment from the current depth, otherwise the propagation should have happened at an earlier depth. Hence $C = \{x \leftarrow a\} \cup E$ contains at least one (dis-)assignment from the current depth. Hence Algorithm 5 iterates at least once and, and by Lemma 3.2 there will be a new constraint learned not subsumed by any other. □

Finally the initial cut for $x \leftarrow a$ s.t. $a \notin \text{initdom}(x)$ is simply the explanation for $x \leftarrow a$.

Lemma 3.5. *Let* E *be the explanation for* $x \leftarrow a$*. A new firstUIP constraint is produced by applying Algorithm 5 to* E *when* $x \leftarrow a$ *is an out of range assignment.*

PROOF. Similar to proof of Lemma 3.3. □

Hence I have shown how to start the conflict analysis process for both lazy and eager learning, using various different starting points for failure, and proved that each one results in a valid new constraint.

3.4.5.3. *Minimising learned constraints.* When a (dis-)assignment is already false at the root node, i.e. at depth $0.i$ for some i, it will be false throughout search. Such a (dis-)assignment can simply be removed from any learned disjunction in which it appears. It is better for the solver to perform this optimisation, to avoid complicating each and every explainer[4]. This optimisation does not affect the level of propagation obtained, because the zero-level (dis-)assignment would never be watched by the

[4]basically, adding throughout the code conditions that $depth(dis) > 1.0$ before including a (dis-)assignment in a new explanation

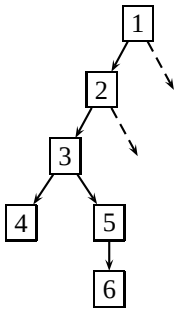


FIGURE 3.3. Nodes in a search tree

disjunction propagator. It can be implemented using stored depths: if $depth < 1.0$ then remove.

Example 3.2. *Suppose that* $x \leftarrow a \vee y \leftarrow b \vee z \nleftarrow c$ *is a learned constraint. Suppose that* $y \nleftarrow b$ *at the root node, due to propagation. Then* $y \nleftarrow b$ *and* $x \leftarrow a \vee y \leftarrow b \vee z \nleftarrow c$ *can be resolved to obtain* $x \leftarrow a \vee z \nleftarrow c$.

3.4.5.4. *Adding constraints during search.* Adding a constraint during search is complicated by *incremental propagation*. A propagator is *incremental* when it need only be notified of new (dis-)assignments, as opposed to laboriously checking every time which values have been removed. In practice, propagators perform a *full propagation* first, and thereafter perform incrementally.

The issue with adding such propagators during search is that, in minion, propagators are only notifed when values are removed, and not when they are restored back into domains[5]; the solver only "knows" that such values exist because it saw them earlier in the branch and set its internal state accordingly. When the solver backtracks beyond the point where the constraint was added, a propagator may behave wrongly because its state is not set up appropriately; that is, the state is not *backtrack stable*. It would be infeasible to notify propagators of new values on backtrack, because the solver would have to compare the old and new states and check what had changed.

The solution I have chosen is that each propagator added during search will be full propagated once at every node on the path from where it was added to the root, so that at each it can prune any values that are inconsistent with it and set up its

[5]if BT memory is trailed individual restorations could be notified efficiently, though most solvers do not do this

internal state appropriately. For example, a constraint added at node 4 in Figure 3.3 will be fully propagated at nodes 4, 3, 2 and 1. It will not be fully propagated at nodes 5 or 6, for example, because it will incrementally propagate relative to the state at node 3. This techique is general as it supports any propagator without code changes. SAT solvers use a different technique that works *only* for clauses, since the technique sets up the internal state backtrack-stably [**ES03**], avoiding any extra work after addition.

3.4.5.5. *Complexity of learning.* The space complexity of storing new constraints at every conflict is polynomial in the number of nodes searched, which is worst case exponential in the size of the problem instance. Hence the associated time complexity is exponential. It is also necessary to maintain consistency on the constraints, which is worst case polynomial time *at each node* for the constraints considered in this thesis. The effect of learning constraints can currently only be analysed satisfactorily by seeing if search time is less on individual instances. This is because learning can save a superpolynomial amount of time by reducing the search space [**Kat09**] but also waste an exponential amount of time creating and propagating them. There is no existing general analytic method for deciding if there will be an overall saving.

To begin to theoretically quantify the cost of learning, I will now analyse how long it takes to build a new constraint after a conflict. Algorithm 5 gives pseudocode for implementing the firstUIP algorithm. Algorithm 6 is a concrete algorithm specifying specific data structures and algorithms. Set c is implemented using a red black tree $curr\_d$ for events e s.t. $depth(e) \geq cd$ and a hash set $earlier$ for events f s.t. $depth(f) < cd$ (see [**CLRS01a**] for discussion of these data structures and associated algorithms).

Algorithm FIRST-UIP-CUT-CONCRETE

A1	**let** c be the set of events directly causing the initial failure
A2	$distribute(c)$
A3	**while** $curr\_d$ has ≥ 2 members
A3.1	**let** $deepest = curr\_d.max()$
A3.2	$curr\_d.delete\_max()$
A3.3	$distribute(expl(deepest))$
A4	**return** $curr\_d \cup earlier$

Algorithm 6: Concrete implementation of find firstUIP cut

Name	Definition	Lower bound	Upper bound
v	$\|V\|$	n.a.	n.a.
d	$\|D\|$	n.a.	n.a.
I	Size of initial cut	2	$O(d)$
R	Number of iterations	1	$O(vd)$
T	Time to produce all expl'ns	$O(R)$	$R.O(vd)$
S	Overall size of all expl'ns	$O(R)$	$O(T)$

TABLE 3.1. Parameters for time complexity analysis

The subroutine *distribute*(), used in Algorithm 6, takes a set of events E and distributes them into either *curr_d* or *earlier* as appropriate. Analyses will be in terms of various parameters of interest shown in Table 3.1. The bounds are mostly straightforward, however the worst case for T is based on the worst individual explanation algorithm used in this thesis, which is $O(vd)$ time in the worst case. In the following analyses, I will assume that a perfect hashing function is used, so that adding to the hash table is worst case $O(1)$.

Theorem 3.6. *The worst case time complexity of Algorithm 6 is* $O(v^2d^2\log(vd))$.

PROOF. The complexity is

$$
\begin{aligned}
& O(I\log I) && \text{(1, to add initial cut)}\\
+\,& O(R) && \text{(2, to evaluate loop condition)}\\
+\,& O(T) && \text{(3, to produce all explanations)}\\
+\,& O(S\log S) && \text{(4, to add and remove events from } curr\_d\text{)}\\
+\,& O(S) && \text{(5, to add events to } earlier\text{)}\\
+\,& O(R) && \text{(6, to create new constraint at the end)}
\end{aligned}
$$

Line (1) takes care of A1 and A2 in the code. The rationale is that in *distribute*() each event is either added to *curr_d* or *earlier*. I am assuming that adding to *earlier* is constant time, since it is a hash set. Adding to *curr_d* is worst case $\sum_{i=1}^{|I|} O(log(i))$ time, since each addition to a red black tree is logarithmic in the size of the tree. This is equal to $O(I.\log I)$ (see [**CLRS01a**]). Hence overall A1 and A2 are $O(I.\log I)$ time.

Line (2) is clear since the size of *curr_d* can be obtained in $O(1)$ time. Line (3) is true by definition. Line (4) records the time needed to add each of R events to *curr_d*

(A3.3) and to remove all but one (A3.1 and A3.2). Adding at most S disassignments is worst case $O(S \log S)$ time but the removals are worst case $O(R \log R)$, since (dis-)assignments may be added multiple times but can only be removed once since the data structure is a set. Hence overall line (4) is worst case $O(S. \log S)$ time. Line (5) is because the events produced can each be added in $O(1)$ time at most once to *earlier* during *distribute*(). Finally in (6) the contents of $curr_d$ and *earlier* are collected together and returned (A4). This is linear time because I assume that the hash set can be iterated in $O(R + c)$ where c is a constant (which might be large).

By replacing the various parts of the sum with their worst case values from Table 3.1 it becomes $O(d. \log d + vd + v^2d^2 + v^2d^2 \log v^2d^2 + v^2d^2 + vd)$ which is $O(v^2d^2 \log(vd))$. □

This upper bound is slightly uninstructive, because the worst case for producing all explanations is fairly crude. The following corollary makes it simpler to interpret:

Corollary 3.7. *The worst case time complexity of Algorithm 6 is* $O(T \log T)$.

Proof. The sum in the proof of Theorem 3.6 can be manipulated as follows using the upper bounds found in Table 3.1:

$$\begin{aligned} & O(I \log I) + O(R) + O(T) + O(S \log S) + O(S) + O(R) \\ =& O(T \log T) + O(T) + O(T) + O(T \log T) + O(T) + O(T) \\ =& O(T \log T) \end{aligned}$$

□

Hence the above implementation of the firstUIP procedure adds a logarithmic factor over the cost of producing the explanations alone. Note that by substituting T's worst case of $O(v^2d^2)$ into Corollary 3.7 the same bound as Theorem 3.6 is obtained, namely $O(v^2d^2 \log(vd))$.

3.4.5.6. *Testing explanations and learned constraints.* I have shown that algorithms that produce explanations and new constraints are intricate. This raises the question of how to improve confidence that an implementation is correct. This can be done by treating both explanations and new constraints as *implied constraints*.

Definition 3.2. An *implied constraint* c of a CSP (V, D, C) is a constraint such that $(V, D, C \cup \{c\})$ has the same set of solutions as (V, D, C).

Hence, learned constraints should always be implied, because Lemma 2.2 shows that they should only remove branches from search with no solution. An explanation E for an event e can also be treated as an implied constraint $E \Rightarrow e$, since it represents something the propagator is forced to do by the semantics of the problem.

An implied constraint can be tested by posting its negation into the problem it is implied by, provided it has at least one solution, and verifying that a reference solver finds no solution. If no solution is found, it must be implied. If the problem has no solutions to begin with, any constraint is trivially an implied constraint and we cannot use this framework to test it.

Fact 3.1. If (V, D, C) has at least one solution and c is an implied constraint of (V, D, C) then $(V, D, C \cup \{\neg c\})$ has no solutions.

Proof. By definition, every solution of (V, D, C) satisfies c, hence every solution does not satisfy $\neg c$. Therefore $(V, D, C \cup \{\neg c\})$ has no solutions. □

My implementation tests a run of learning minion by first checking that there is a solution to the problem. If not, the instance is unsuitable for testing. If so, learning minion runs and prints out the negative of each learned constraint in minion input format. Then each one is spliced into the original problem in turn and solved using stock minion as a reference solver. If there is no solution, then the learned constraint must have been a valid implied constraint.

As usual with testing, this routine can only prove the presence of faults and not the absence, but it is automatic and methodical and I have used it to find many faults in propagators with minimal effort. Once the routine is complete, all the explanations and learned constraints in the instance under test are known to be correct if stock minion is. For typical instances stock minion can test 10s of implied constraints per second. However most instances use many thousands of explanations and so there would rarely be time to test all available instances.

3.5. Lazy explainers

In eager learning, explanation is done immediately, whereas in lazy learning only the explanation record is stored and the bulk of the work is delayed. It is not possible to say a priori that lazy or eager is objectively better, in the sense that less overall work is necessary: Lazy can be better because it avoids doing unnecessary work, but eager can be better because if most of the explanations are needed eventually it may be more efficient to build them during propagation and store them immediately. Hence I will analyse the time complexity of both eager and lazy explanation, in order to estimate the break-even point for lazy learning, in terms of the proportion of explanations that are eventually used. In some cases, it will be better in practice to use eager evaluation, and this is accomodated by the lazy learning framework. Finally experiments will show that lazy learning is overwhelmingly better than eager learning in practice, showing that it is an easy decision to use lazy learning wherever possible.

In the following analysis, it will not always be necessary to describe the constraint propagator, because explanation algorithms can often be completely decoupled from propagation. However where the two routines share state it may be necessary to describe both. Some of the described explanation algorithms are novel, but others are based on existing eager routines and the novelty is centred on adaption, analysis and experimentation. In the following sections, I will give appropriate credit wherever published explanations are exploited.

3.5.1. Explanations for clauses. Clauses were defined in Definition 2.4. To explain clauses, it is sufficient to store the minimal record for each propagation. Before explaining why, it is necessary to define *unit propagation* which is the consistency level used to propagate clauses:

Definition 3.3. When all but one (dis-)assignment e_i in a clause $e_1 \vee e_2 \vee \ldots \vee e_k$ are false, *unit propagation* will set e_i to be true.

Example 3.3. *Suppose that* $a \nleftarrow 1$, $b \nleftarrow 2$ *and* $c \nleftarrow 3$, *then the propagator for the clause* $a \leftarrow 1 \vee b \leftarrow 2 \vee c \leftarrow 3 \vee d \leftarrow 4$ *will set* $d \leftarrow 4$, *as the remaining disjuncts are all false. This is necessary because at least one disjunct must be true to satisfy the clause.*

Now suppose later the explanation is needed: the (dis-)assignment was by unit propagation and hence it can be inferred that all but the propagated (dis-)assignment e_i was false at the time, and the explanation consists of the (dis-)assignments of the clause excluding e_i.

Example 3.4. *Following from the last example, when the explanation is requested* $d \leftarrow 4$ *is removed and then the clause is negated, to obtain* $\{a \nleftarrow 1, b \nleftarrow 2, c \nleftarrow 3\}$.

This form of lazy learning is very familiar because it is what SAT solvers do [**MSS96**]. It is natural for SAT solvers to do lazy learning, but we will show that it is also possible and advantageous for CP solvers. For clauses, lazy is certain to be better than eager, because eager would simply replicate the clause and *possibly* use it later.

3.5.2. Explanations for ordering constraints. Inequality and lexicographical ordering constraints are used in modelling for basic expression of problems and both are notable for their use in symmetry breaking [**FHK+06**] which is a widely used CSP modelling technique [**GPP06**].

3.5.2.1. *Inequality constraints.* Suppose that constraint $v_1 < v_2$ causes pruning $v_1 \nleftarrow a$; it is sufficient to store a minimal record to lazily construct the explanation. The value a is pruned if and only if all values in v_2 greater than a are removed, since these are the potential supports for a. Hence explanation $\{v_2 \nleftarrow a+1, \ldots, v_2 \nleftarrow \max(d_2)\}$ can be computed when required. Explanations for prunings to v_2 are similar: if b is pruned from v_2 then the explanation is $\{v_1 \nleftarrow \min(d_1), \ldots, v_1 \nleftarrow b-1\}$.

Explanations for $v_1 \leq v_2$ are very similar: in the above example the explanation only needs to have $v_2 \nleftarrow a$ added to it, since $a \in dom(v_1)$ is only unsupported once v_2 can no longer be greater than or equal to a.

3.5.2.2. *Entailment of inequality constraints.* To explain the lexicographical ordering constraint, it is necessary to be able to produce explanations for the entailment of $x \geq y$ and $x > y$, i.e. explain why the constraint must be true in all possible remaining assignments to x and y. An eager algorithm for this problem is described

in [**Kat09**][6]. The reason $x \geq y$ (or $x > y$) is entailed is that $\exists k$ s.t. $\min(dom(x)) \geq k$ and $k \geq \max(dom(y))$ (respectively $\min(dom(x)) \geq k$ and $k > \max(dom(y))$). To produce an explanation such a k must be found and the following disassignments added to the nogood:

- $\forall a < k \wedge a \in \text{initdom}(x)$, add $x \not\leftarrow a$; and
- $\forall b > k \wedge b \in \text{initdom}(y)$, add $y \not\leftarrow b$.

This set of disassignments justifies that $\min(dom(x)) \geq \max(dom(y))$ (resp. $\min(dom(x)) > \max(dom(y))$.

To reduce the size of the nogoods, the explanation algorithm I will describe will find k such that as few disassignments as possible are included, and also so that the maximum depth of any disassignment is minimised. The following example demonstrates a situation where there are several choices, one of which has a maximum depth much larger than the other:

Example 3.5. *Suppose that* $initdom(x) = initdom(y) = \{1, 2, 3, 4, 5\}$. *Suppose that values in* x *and* y*'s domains are pruned at the following depths:*

value	*depth pruned from* x	*depth pruned from* y
5	*na*=not applicable	*1.2*
4	*na*	*1.7*
3	*1.2*	*10.2*
2	*1.1*	*na*
1	*2.1*	*na*

The two choices of explanation for the entailment of $x \geq y$ *are*

$$\{x \not\leftarrow 1, x \not\leftarrow 2, y \not\leftarrow 4, y \not\leftarrow 5\}$$

and

$$\{x \not\leftarrow 1, y \not\leftarrow 3, y \not\leftarrow 4, y \not\leftarrow 5\}$$

corresponding to $k = 3$ *and* $k = 2$ *respectively. Based on the table of pruning depths, the maximum prunings in each are 2.1 and 10.2 respectively. The former is more*

[6]Note that the explanation described in [**Kat09**] is slightly different from this one, it includes all prunings to x less than $\min(dom(x))$ and all prunings to y greater than $\max(dom(y))$, which is non-optimal when $\min(dom(x)) > \max(dom(y)) + 1$.

attractive because the maximum depth of a (dis-)assignment in the explanation is smaller.

The following algorithm finds optimal (minimum size and depth) explanations for the entailment of $x \geq y$ ($x > y$) lazily:

(1) Let $initdom(x)$ be the initial domain of x and $initdom(y)$ the starting domain of y.
(2) Begin with a pointer X at $\min(initdom(x))$ and a pointer Y at $\max(initdom(y))$.
(3) If $X \geq Y$, stop. (For $x > y$, if $X > Y$, stop.)
(4) Choose X or Y depending on which value is pruned in its respective domain and, if both are pruned, choose the one pruned earliest.
(5) Add the chosen disassignment to the candidate explanation, e.g. for X add $x \nleftarrow X$.
(6) If X was picked increment X; if Y was picked decrement Y.
(7) Return to step 3.

Assuming that $\min(dom(x)) \geq \max(dom(y))$ this algorithm will terminate and the maximum depth of a disassignment in the explanation will be the minimum possible (the depth will be *minimaximal*).

The explanation contains exactly $\max\{0, \max(initdom(y)) - \min(initdom(x)) - 1\}$ disassignments[7] ($\max\{0, \max(initdom(y)) - \min(initdom(x))\}$ for $x > y$), which can easily be shown to be the fewest possible. The algorithm can be implemented with optimal time complexity, i.e. time proportional to the size of the result.

In order to prove that this greedy algorithm finds an optimal solution consisting of a set of disassignments with minimaximal depth, it is necessary to prove that it has *optimal substructure* and the *greedy choice property*. Optimal substructure means that having picked a particular disassignment at either position X or Y, the best way to proceed is to combine it with an optimal solution to the resultant subproblem, i.e. pick the remaining set of disassignments with minimaximal depth. The greedy choice property is that a greedy choice at any stage is a part of an overall optimal solution,

[7] 0 is present in case $\min(initdom(x)) \geq \max(initdom(y))$ at the beginning, in which case the explanation is $\emptyset$

i.e. the disassignment with least depth at positions X and Y is a part of an optimal solution.

Theorem 3.8. *The greedy algorithm for explaining the entailment of $x \geq y$ produces an explanation with minimaximal depth.*

Proof. For a contradiction, suppose w.l.o.g. that at the current iteration $x \nleftarrow X$ is chosen but $depth(x \nleftarrow X) > depth(y \nleftarrow Y)$. If the algorithm picks $y \nleftarrow Y$ before the end of the algorithm there is no problem, for the greedy choice was used eventually. Now assume that $y \nleftarrow Y$ is never picked and consider a complete solution: the explanation will consist of $x \nleftarrow X$, $x \nleftarrow X+1$ and so on until $x \nleftarrow Y-1$. However the explanation is still valid and the depth at least as good if $y \nleftarrow Y$ is substituted for $x \nleftarrow X$. Hence the greedy choice property is valid for this problem.

Once the greedy choice is made, the problem reduces to finding an optimal solution for the remaining subproblem. This is because by minimising the maximum depth for the subproblem, the overall depth including the greedy choice is at least as good. By induction on the number of choices made, making the greedy choice at every step produces an optimal solution. □

The following example demonstrates the algorithm:

Example 3.6. *The aim in this example is to find an explanation for why $x > y$. Suppose that $dom(x) = \{4\}$ and $dom(y) = \{2\}$. Suppose the solver is currently at depth 4.12, and assuming that the domains were $\{1, 2, 3, 4\}$ to begin with, this means the following prunings must already have happened, and the depth at which they are supposed to have occurred is given in the appropriate cell:*

value	*depth pruned from x*	*depth pruned from y*
4	*na*	*1.7*
3	*4.6*	*3.2*
2	*4.8*	*na*
1	*4.7*	*4.3*

Following the algorithm, prunings are added to the explanation in the following order: $y_3 \nleftarrow 4, y_3 \nleftarrow 3, x_3 \nleftarrow 1, x_3 \nleftarrow 2$. The size of the explanation is $\max\{0, \min(initdom(Y)) - \max(initdom(x))\} = \max(0, 4-0) = 4$ *and $k = 3$.*

3.5.2.3. *Lexicographic ordering constraint.* The lexicographical ordering constraint is a generalisation of inequality to vectors; it keeps two vectors in "dictionary" order.

Definition 3.4. Given two n-vectors $x =< x_0, \ldots, x_{n-1} >$ and $y =< y_0, \ldots, y_{n-1} >$, $x <_{\text{lex}} y$ holds if and only if $\exists k \in [0, \ldots, n-1]$ s.t. $\forall j \in [0, \ldots, k-1]$, $x_j = y_j$, and $x_k < y_k$. $x \leq_{\text{lex}} y$ holds iff $x <_{\text{lex}} y$ or $x = y$.

Example 3.7. *Let a string of digits be a shorthand for a vector, e.g.* 0000 *as a shorthand for* $< 0, 0, 0, 0 >$. *According to Definition 3.4:* $0000 <_{lex} 0000$ *is false,* $0000 \leq_{lex} 0000$ *is true,* $0100 <_{lex} 1000$ *is true and* $1101 \leq_{lex} 1100$ *is false.*

GAC propagators for $\leq_{\text{lex}}$ and $<_{\text{lex}}$ were described in [**FHK$^+$06**], and [**Kat09**] describes how to produce explanations for this constraint eagerly. In order to do so *lazily* it will be necessary to make use of stored (dis-)assignment depths in order to reconstruct the domain state at the time when the pruning was made. However this does not add an overhead to the explanation process. In this thesis I will concentrate on explanations for $<_{\text{lex}}$ but $\leq_{\text{lex}}$ is very similar.

First I must describe how the GAC propagation algorithm works [**FHK$^+$06**]. The propagation algorithm maintains two values α and β. α is the maximum index such that vectors of variables $< x_0, \ldots, x_{\alpha-1} >$ and $< y_0, \ldots, y_{\alpha-1} >$ are assigned and equal. β is the minimum index such that $x_\beta, \ldots, x_n \geq_{\text{lex}} y_\beta, \ldots, y_n$ is entailed, i.e. vectors $x_\beta, \ldots, x_n$ and $y_\beta, \ldots, y_n$ definitely violate constraint $<_{\text{lex}}$. At indices β and above there are zero or more indices i where $x_i \geq y_i$, followed by a single index where $x_i > y_i$ (or the end of the vector). This is because the $\geq$ indices don't satisfy the constraint, and the $<$ index definitely violates it. Knowing α and β at all times allows GAC to be enforced very easily.

Example 3.8. *Suppose the domains are as follows:*

i	*0*	*1*	*2*	*3*
x	$\{2\}$	$\{1\}$	$\{3\}$	$\{4\}$
y	$\{2\}$	$\{1\}$	$\{3, 4\}$	$\{2\}$

$\alpha = 2$ *because the pairs for* $i = 0$ *and* $i = 1$ *are assigned and equal, but not the pair for* $i = 2$. $\beta = 3$ *because* $4 \not<_{lex} 2$.

In the propagation algorithm, there are 2 cases where action needs to be taken and these are sufficient to enforce GAC (proved in [**FHK+06**]):

(1) If $\beta > \alpha+1$ then propagate to ensure $x_\alpha \leq y_\alpha$ (to be referred to subsequently as "rule 1"). This is done because if $x_\alpha > y_\alpha$ then the whole constraint would be violated because the vectors would be equal up to position α and x would be larger at $\alpha + 1$.
(2) If $\beta = \alpha+1$ then propagate to ensure $x_\alpha < y_\alpha$ (as described in Section 3.5.2.1 above), e.g. the domains in Example 3.8 ("rule 2"). This is done because position α is the only remaining position where x_i can be set less than y_i, so it must be done to satisfy the constraint.

As shown in [**Kat09**], eager explanations for these propagation rules can be built from three generic parts, namely

- explanations for inequality propagation (see Section 3.5.2.1),
- an explanation for why α has its current value (called E_α) and
- an explanation for why β has its current value (called E_β).

The explanation for a pruning by rule 1 (above) is the standard explanation for inequality *plus* E_α to explain why the inequality is being propagated in the first place. It is not necessary to include E_β because rule 1 is enforced irrespective of the value of β (rule 2 is the special case when $\beta = \alpha + 1$). The explanation for a pruning by rule 2 is the standard explanation for inequality plus $E_\alpha \cup E_\beta$.

In order to do the same thing lazily, the algorithms must be amended slightly. For rules 1 and 2, the value of α is known lazily because it is the same as the index of the variable pruned. For rule 2, β can be inferred from α because it equals $\alpha + 1$.

E_α can be built lazily very easily, i.e. when required build

$$\{x_i \leftarrow val : i \in [0 \ldots, \alpha - 1], dom(x_i) = \{val\}\}$$
$$\cup \{y_i \leftarrow val : i \in [0 \ldots, \alpha - 1], dom(y_i) = \{val\}\}$$

that is, collect the assignments to the vectors up to index α. This is the only possible explanation and it can be built with optimal efficiency lazily. It is the same as in [**Kat09**] except built lazily.

E_β from [**Kat09**] can also be built lazily. However stored pruning depths must be taken into account. E_β from [**Kat09**] is the union of the following parts:

- Find index B such that $\forall i \in [\beta, \ldots, B-1]$, $x_i \geq y_i$ and, if $B < n-1$, $x_B > y_B$.
- For all $i \in [\beta, \ldots, B-1]$, explain why $x_i \geq y_i$.
- If $B < n-1$, explain why $x_B > y_B$.

The following algorithm builds the explanation lazily based on explanations for entailment described in Section 3.5.2.2: for each position i starting at β if $x_i > y_i$ is entailed add explanation for entailment of $x_i > y_i$ and stop, otherwise $x_i \geq y_i$ must be entailed so add explanation for entailment of $x_i \geq y_i$.

In the following Example the various parts described above are used to give a complete example of a lazy explanation for $x <_{\text{lex}} y$.

Example 3.9. *The domain state in Example 3.8 is such that* $3 = \alpha + 1 = \beta = 3$. *Hence the solver will enforce consistency on* $x_2 < y_2$, *resulting in* $y_2 \nleftarrow 3$ *by rule 2. Assuming the domains were all* $\{1, 2, 3, 4\}$ *to begin with, the explanation, once built, is as follows.*

$$\{x_2 \nleftarrow 1, x_2 \nleftarrow 2\} \qquad \text{(for } x_2 < y_2 \text{ pruning)}$$
$$\cup \{x_0 \leftarrow 2, y_0 \leftarrow 2, x_1 \leftarrow 1, y_1 \leftarrow 1\} \qquad \text{(for } \alpha\text{)}$$
$$\cup \{x_3 \nleftarrow 1, x_3 \nleftarrow 2, y_3 \nleftarrow 3, y_3 \nleftarrow 4\} \qquad \text{(for } \beta \text{ from Example 3.6)}$$

As shown in [**Kat09**], storing explanations eagerly is an overhead over normal propagation. However with lazy explanations the worst case is not necessarily reached. Furthermore the worst case time complexity of lazy explanations is the same as eager explanations, since both can be implemented optimally, in the sense that a constant number of operations are needed for each (dis-)assignment in the built explanation. Hence building lazy explanations for lexicographical ordering cannot be asymptotically worse, but the best case can be zero additional cost over storing the record.

3.5.3. Explanations for table. The extensional or "table" constraint is an important part of a constraint library. The user lists the allowed tuples[8]. Hence it can mimic any other constraint, or be used to express an arbitrary relation in a straightforward way where in many cases it would be awkward to express otherwise, e.g. the relation "married to", $\{(tom, sally), (bob, marie), (sean, jenny)\}$.

Before continuing, I will define "trie" and describe how a trie can be used to represent strings.

Definition 3.5. A *trie* is a tree in which every edge is labelled by a letter from an alphabet Σ. A trie *contains* a string $S = s_1 s_2 \ldots s_k$ s.t. each $s_i \in \Sigma$ when it contains a path whose edges are labelled by $s_1, s_2, \ldots, s_k$ from the root to a leaf node.

I will not attempt to describe how tries are built and processed. Tries are described in [**CLRS01b**] and many other textbooks.

A trie can be used to store tuples by treating them as strings whose characters are the components of the tuples in order.

Example 3.10. *The trie at the top of Figure 3.4 contains the following tuples:*

$$\{(d=1,a=0,b=1,c=1),(d=1,a=0,b=2,c=2),(d=1,a=0,b=2,c=3),$$
$$(d=1,a=2,b=2,c=1),(d=1,a=2,b=2,c=4),(d=1,a=2,b=3,c=5)\}$$

Let *varval* be a shorthand name for "variable/value pair". For example $x = a$ is a varval consisting of a variable x and a value $a \in dom(x)$. Assume an implementation of table where tuples are stored as an array of tries [**GJMN07**], one per variable, so that all tuples involving a particular varval are readily accessible, as illustrated in Figure 3.4.

Example 3.11. *The trie at the top of Figure 3.4 represents a set of tuples all of which contain varval* $d = 1$.

I will say that a varval $x = a$ is *pruned* when $x \nleftarrow a$. A tuple is *valid* when none of its component varvals are pruned. The propagator works by ensuring that each

[8] it is also possible to list the disallowed tuples, though in this thesis I do not consider that possibility

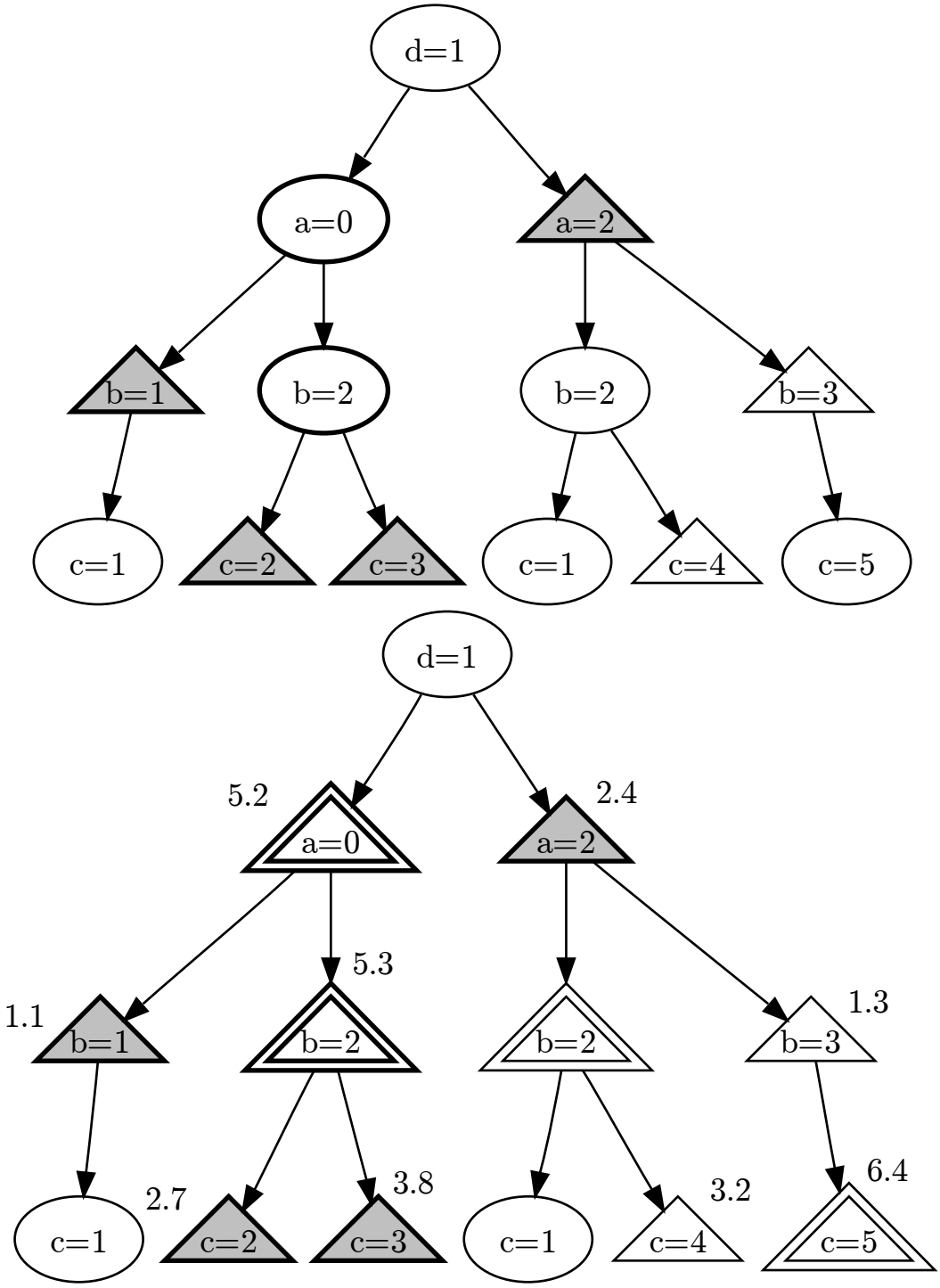


FIGURE 3.4. (top) Trie with pruned values shown as triangles, greyed nodes are those included in the explanation and nodes visited in the traversal are bold. (bottom) Same trie but values pruned between the original pruning (at depth 3.9) and the explanation being produced are in double triangles. Pruning depths are shown: permissible prunings have depth < 3.9, disallowed prunings have depth > 3.9.

varval $v_i = a$ s.t. $a \in d_i$ has at least one support, i.e. there exists at least one valid tuple containing $v_i = a$. If any component of the support is pruned either a new support can be found in the trie, or $v_i = a$ is pruned.

Such a constraint will prune the varval $v_i = a$ if and only if every tuple containing $v_i = a$ has at least one component varval pruned. A pruning $v_i \nleftarrow a$ is a *cover* for tuple t iff $v_i = a$ is a component of t. Hence the explanation for $v_i \nleftarrow a$ must be a set containing *at least one* cover for each tuple containing $v_i = a$. I will now describe explanations for GAC-schema [**BR97**] using Katsirelos' naïve scheme [**Kat09**] which was arguably the most successful of the techniques he tried. The algorithm simply picks any pruned component from each tuple. The application of this algorithm to create explanations (eager and lazy) for table constraints represented as tries is novel.

Algorithm TRIE-EXPLAIN(e, n)

```
      where e is the explanation being built
      where n is the current node in the trie
A1    let x = a be the label of the current node
A2    if x ↚ a
A2.1      return e ∪ {x ↚ a}
A2.2  else
A2.3      e ←TRIE-EXPLAIN(e, n.left)
A2.4      return TRIE-EXPLAIN(e, n.right)
```

Algorithm 7: Building explanation for table constraint eagerly

This can easily be implemented with tries: perform an inorder traversal of the trie but whenever a node corresponding to a pruned varval is visited add the corresponding pruning to the set and don't recurse any further. This is given as Algorithm 7. Each pruning covers all the tuples beneath the point in the trie when it was added.

Example 3.12. *Figure 3.4 (top) illustrates this process: when an explanation for* $d \leftarrow 1$ *is required, the traversal produces* $\{b \nleftarrow 1, c \nleftarrow 2, c \nleftarrow 3, a \nleftarrow 2\}$. *Note that* $b \nleftarrow 3$ *and* $c \nleftarrow 4$ *are not included in the traversal because all supports are covered without them.*

Lazily, the algorithm sees the same trie, but there are *at least* as many pruned values. By applying the same traversal Property 2.1 (page 25) may not be satisfied, for later additional prunings could be wrongly used when they could have had no effect on the earlier propagation. Instead, the algorithm is adapted to add to the

Algorithm TRIE-EXPLAIN-LAZY$(e, n, maxdepth)$

...	...
A2'	**if** $x \nleftarrow a$ **and** $depth(x \nleftarrow a) \leq maxdepth$
...	...

Algorithm 8: Building explanation for table constraint lazily

set only values that were made *at that time*; i.e. to explain a pruning at depth $a.b$, consider only nodes for varvals pruned at a depth less than $a.b$. Only one line from Algorithm 7 needs to change and the change is given as Algorithm 8.

Example 3.13. *Such a situation is illustrated in Figure 3.4 (bottom) where the double lined triangular nodes are not used because they occured after the original pruning at depth 3.9, though they would be included by Algorithm 7. Instead they are skipped and the same explanation as the previous example is obtained (the shaded nodes are included).*

An explanation can be built eagerly with no increase in big-O asymptotic time complexity compared to normal propagation, since propagators must traverse the entire trie prior to doing each and every pruning, and could build an explanation during that traversal. However the propagation stage would be slower because of the requirement to create a vector of (dis-)assignments during the traversal. Lazily, no action is required during propagation, except to store a minimal record, and then during lazy explanation one extra trie traversal is incurred. Hence in order to achieve a speedup, lazy learning is relying on the efficiency of propagation being improved enough to compensate for performing additional traversals later.

3.5.4. Explanations for constraints enforcing less than GAC. Propagators for the $z = x \times y$ constraint often enforce a level of consistency below GAC, since enforcing GAC is related to integer factorisation for which no polynomial time algorithms are currently available. Propagation weaker than GAC, for $z = x \times y$ and other constraints, can be one of many defined levels of consistency [**Bes06**], e.g. bounds(Z) consistency, or an ad-hoc level of consistency that doesn't correspond to any published consistency level. Minion's $z = x \times y$ propagator, on which the experiments in §3.6 are based, enforces an ad-hoc level of consistency.

The overall motivation for discussing explanations for $z = x \times y$ is that it enforces less than GAC and the literature has not discussed the ramifications of this up until now. I will not present algorithms for either propagating or explaining $z = x \times y$ but instead illustrate the issues with an example:

Example 3.14. *Suppose* $dom(x) = \{2, 4\}$, $dom(y) = \{2\}$ *and* $dom(z) = \{4, 5, 6, 7, 8\}$. *The minion propagator is only able to detect inconsistencies in the upper and lower bounds of the domain of each variable. For these domains,* $5, 6, 7 \in dom(z)$ *are all inconsistent, but the propagator is unable to detect this. However, if* $8 \in dom(z)$ *is subsequently removed, the propagator may now iteratively remove 7, 6 and 5.*

This situation creates an anomaly with the explanation, where the explanation suggests that the value is actually removed long before the propagator does so, as I will now explain: Suppose that the initial domains were $dom(x) = \{2, 3, 4\}$, $dom(y) = \{2\}$ *and* $dom(z) = \{4, 5, 6, 7, 8\}$. $6 \in dom(z)$ *is initially supported by* $3 \in dom(x)$ *and* $2 \in dom(y)$. *Suppose that disassignment* $x \not\leftarrow 3$ *is carried out by another constraint at depth* 1.1*, it is sufficient to ensure* $z \not\leftarrow 6$, *i.e.* $\{x \not\leftarrow 3\}$ *is an explanation for* $z \not\leftarrow 6$. *However* $z \not\leftarrow 6$ *cannot be carried out until either* $\min(dom(z)) = 6$ *or* $\max(dom(z)) = 6$, *which can happen at an arbitrary depth* > 1.1. *The explanations for* $z \not\leftarrow 5$ *and* $z \not\leftarrow 7$ *are even more extreme, being* $\{\}$ *in both cases, since* $5 \in dom(z)$ *and* $7 \in dom(z)$ *are unsupported in the initial domains.*

Since consistency is a property of propagation and not explanation, this problem is not unique to lazy explanations and can also happen when explanations are computed eagerly. It is proved in [**NOT06**] (Theorem 5.2) that such explanations[9] may mean that there is no UIP in the implication graph, however the constraint derived by Algorithm 5 is still valid. This is very useful because it allows explanations to be made as small and precise as possible, even if the propagator which emits them is weak. This will result in smaller learned constraints which propagate more strongly, in general.

[9] they are called "too late" explanations in [**NOT06**]

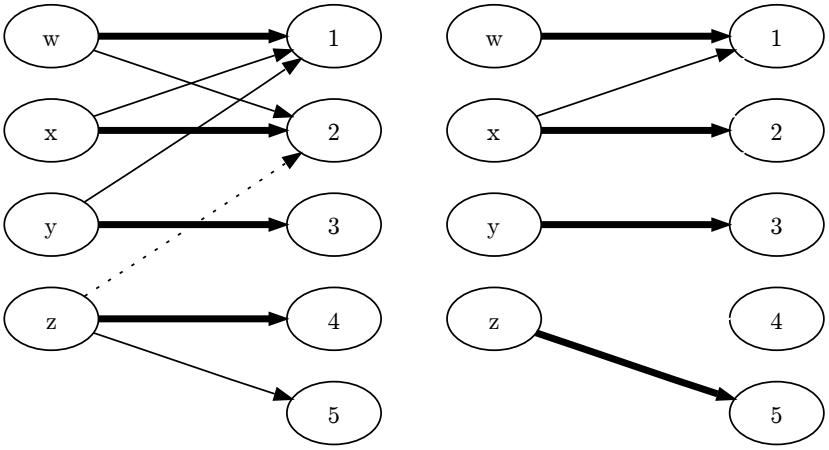


FIGURE 3.5. (left) Variable value graph at time of original pruning (right) Same graph at time of explanation

3.5.4.1. *Relationship with minimality.* It is worth discussing the relationship between consistency level and minimality. When the propagator is weak, but the explanation is strong, the explanation is minimal w.r.t. a different, stronger propagator.

Another issue is that the definition of minimality used in this thesis, and in [**Kat09**], is defined w.r.t. a single propagator. It is also possible to minimise an explanation w.r.t. propagators for *all* the constraints and implied constraints of a problem. For example suppose that $\{e_1, e_2, e_3\}$ is a valid explanation for event e caused by propagator P. It may be that $\{e_1, e_2\}$ is a minimal explanation w.r.t. propagator P, since $e_1 \wedge e_2$ are sufficient for P to infer e_3, meaning e_3 can be removed. However perhaps e_1 is sufficient for the entire set of propagators to infer both e_2 and e_3 when executed to a fixpoint, or singleton AC is applied, etc. In this case just $\{e_1\}$ is also a valid explanation and even smaller. A highly efficient form of this is used in SAT solvers, where conflict clauses are minimised using the entire set of clauses, e.g. [**SB09**].

3.5.5. Explanations for alldifferent. The alldifferent (alldiff) constraint (see [**GMN08**] for a review) ensures that the variables in its scope take distinct values.

Example 3.15. *For example, consider the* variable value graph *in Figure 3.5 (left), where there are 4 variables and 5 values. The current domains are illustrated by having a edge from variable var to value a whenever* $a \in dom(var)$*. A possible satisfying*

assignment is $w \leftarrow 2$, $x \leftarrow 1$, $y \leftarrow 3$ *and* $z \leftarrow 5$. *Another valid matching is shown by bold lines.*

In the following, let r denote the size of an alldiff's scope and d the size of the largest domain. Régin's algorithm for enforcing GAC consistency on alldiff [**Rég94**] is algorithmically complex but the principle is relatively simple. Suppose consistency is being enforced on alldiff constraint c. The central idea is to find *Hall sets* which are sets $S \subseteq scope(c)$ of k variables such that $|\bigcup^{s \in S} \mathrm{dom}(s)| = k$, i.e. sets of k variables whose *combined domain* contains exactly k values. Any valid assignment to these k variables must use all k values because each needs a distinct value. Hence the values in the combined domain of any Hall set cannot be used by variables $v \in scope(c) \setminus S$ and can be pruned. It turns out that GAC can be enforced by finding all Hall sets and pruning appropriately. This is the central idea of Régin's algorithm, illustrated by the following example.

Example 3.16. *In Figure 3.5 (left),* $\{w, x, y\}$ *is a Hall set, since the combined domain is* $\{1, 2, 3\}$. *Unsupported value* $2 \in dom(z)$ *is shown with a dotted line, it is unsupported because 2 is used by the aforementioned Hall set.*

In order to find Hall sets, Régin's algorithm first creates a maximum matching (size 4 matching shown with bold lines in the figure) in $O(r^{1.5}d)$ time and then uses Tarjan's algorithm to find Hall sets in $O(rd)$ time.

Algorithm EXPLAIN-ALLDIFF-EAGER(*hallSet*)

```
A1      expl ← {}
A2      values ← {}
A3      for v ∈ hallSet
A3.1        values ← values ∪ {currentMatching[v])}
A4      for v ∈ hallSet
A4.1        for val ∈ initdom(v) \ values
A4.1.1          expl ← expl ∪ {v ↚ val}
A5      return expl
```

Algorithm 9: Routine to eagerly explain alldiff pruning $x \nleftarrow a$

[**Kat09**] describes how to produce explanations eagerly for alldiff pruning $x \nleftarrow a$. This algorithm is very simple, but only works when the Hall set is known, such as during propagation. Hence it is ideal for eager explanation. It is reproduced as

Algorithm 9. The explanation consists of all disassignments to the variables in the Hall set for values outwith the combined domain. The combined domain can easily be found as it consists of the values assigned to each variable by the maximum matching that is maintained throughout the algorithm. The disassignments in the explanation ensure that the Hall set has a combined domain of no more than one value for each variable.

I will describe two techniques for producing an explanation for $v \nleftarrow a$ lazily, based on two different ways to obtain the Hall set that was earlier used to justify the pruning. Each has the following form:

(1) The alldiff propagator maintains a maximum matching as domains are narrowed. The most recent complete matching would have been valid when the pruning was performed: this is because earlier in the branch, the variable value graph had additional edges in it, but a matching remains correct when additional edges are added to the underlying graph. For example, notice that the matching in Figure 3.5 (right) is also valid for Figure 3.5 (left). Hence the current matching can be used to find the values consumed by any earlier Hall set.

(2) Find the Hall set that earlier consumed the pruned value, by some appropriate method to be discussed.

(3) The explanation is the conjunction of all the prunings from variables in the Hall set (except the values in the combined domain), ruling out prunings that happened after $x \nleftarrow a$. See Algorithm 10 for the details.

Algorithm EXPLAIN-ALLDIFF-LAZY($hallSet, maxDepth$)

...	...
A4'	**for** $v \in hallSet$
A4.1'	**for** $val \in \text{initdom}(v) \setminus values$
A4.1.1'	**if** $depth(x \nleftarrow val) \leq maxDepth)$
A4.1.1.1'	$expl \leftarrow expl \cup \{v \nleftarrow val\}$
...	...

Algorithm 10: Routine to lazily explain alldiff pruning $x \nleftarrow a$, based on Algorithm 9

But how can step 2 be implemented? The first and easiest technique is to just store the variables in the Hall set when the pruning is made. When the explanation

is needed later, Algorithm 10 can be invoked on the stored Hall set. Notice that the record stored for this propagation is *not* minimal.

Hence an upfront cost of $O(|S|) = O(r)$ is incurred, in order to store the known Hall set. Later on, to recover the explanation the cost is $O(rd)$ to run Algorithm 10. Hence overall the worst case time to produce an explanation is $O(rd)$ per value, but the best case is $O(|S|) = O(r)$ when it is not used. I will now describe a different approach to lazy explanations that has a superior constant time best case and an identical worst case time, albeit with a larger constant factor. It is not possible to say which is better in general as it depends on how many explanations are finally requested, so I will finally provide an empirical comparison on practical instances.

Algorithm FIND-HALL-SET-LAZY($maxDepth, a$)

```
A1        loop
A1.1        S ← findNextHallSet(maxDepth)
A1.2        for s ∈ S
A1.2.1        if currentMatching[s] = a
A1.2.1.1        return S
```

Algorithm 11: Routine to find Hall set that involved value a lazily

This alternative technique uses a minimal record. The approach used to find the Hall set when required is to re-run Tarjan's algorithm on demand, using the earlier domain state reconstructed by inspecting depths. Hence the Hall sets used earlier during propagation are re-discovered. In my implementation (Algorithm 11) when explaining $x \nleftarrow a$, immediately after Tarjan's algorithm returns a Hall set S (line A1.1) it is checked to see if value a is matched to one of the variables in S in the current matching (lines A1.2-A1.2.1), for if it does S is the required Hall set responsible for $x \nleftarrow a$ and it is returned (line A1.2.1.1). Hence Tarjan's algorithm is run only until the Hall set is found, this optimisation does not affect the worst case time complexity because the required Hall set may be the last to be found. The algorithm must terminate because the Hall set exists and will be found.

The upfront time complexity is $O(1)$ per pruning to store the minimal record. When the explanation is required the worse case time complexity is $O(rd)$ to run Tarjan's algorithm, plus worst case $O(r)$ to determine which of the Hall sets consumes value a, followed by Algorithm 10 in $O(rd)$ time. Hence the overall worst case time

complexity is $O(rd)$. However compared with the previous algorithm, the constant factor will be larger (since an additional stage is required where Tarjan's algorithm is executed).

3.5.5.1. *Experimental evaluation.* Alldifferent is probably the most important of all the global constraints, on the basis that it occurs frequently and naturally in constraint models and consequently if a solver has any global constraints usually alldifferent is one of them. Furthermore, there has recently been considerable interest in adding an alldifferent theory to SMT solvers [**Nie09, BM10**]. For this reason I will now compare the three different variants of alldifferent explanation described above, namely

eager: fully eager (Algorithm 9)

very lazy: recompute Hall sets and matching lazily (Algorithm 10 supplied with Hall set computed by Algorithm 11)

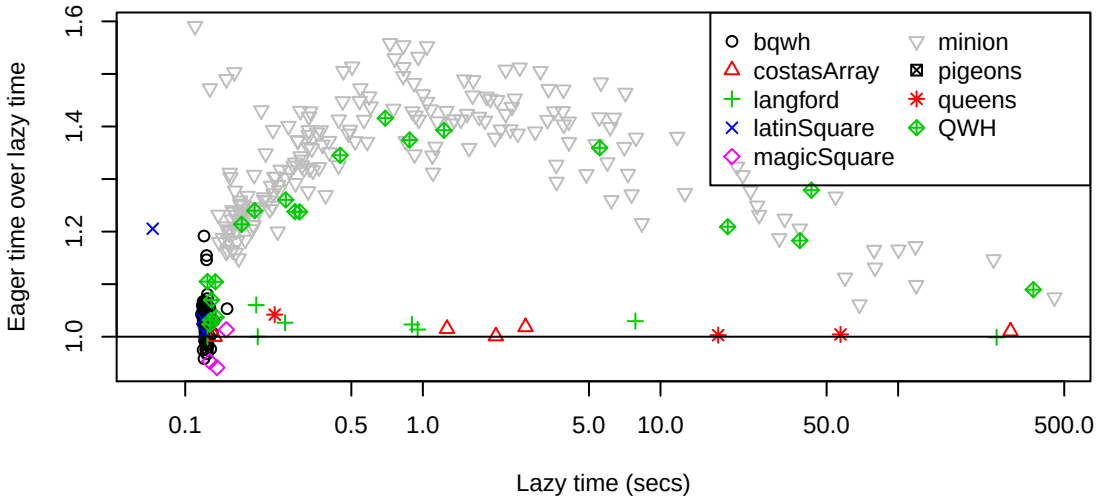
lazy: store Hall set eagerly but use matching lazily (Algorithm 10 supplied with Hall set stored earlier)

Methodology. Each of the 161 instances was solved by the g-learning version of minion (see §3.4.1) to find the first solution five times with a 10 minute timeout, over 3 Linux machines each with 2 Intel Xeon cores at 2.4 GHz and 2GB of memory, running kernel version 2.6.18 SMP. Parameters to each run were identical, and the minimum time for each is used in the analysis, in order to approximate the run time in perfect conditions (i.e. with no system noise) as closely as possible. Each instance was run on its own core, each with 1GB of memory. Minion was compiled statically (`-static`) using g++ version 4.4.3 with flag `-O3`.

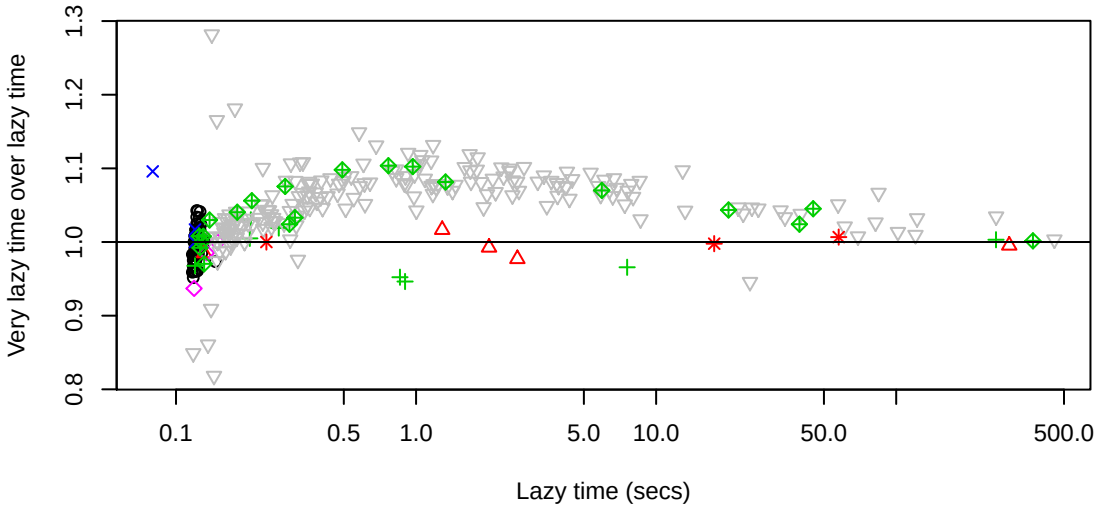
The instances used are

- all the instances from [**GMN08**] that my solver is compatible with[10],
- all the instances from [**BM10**] (the problem class is called "minion"), and
- assorted other benchmarks that contain alldiff constraints.

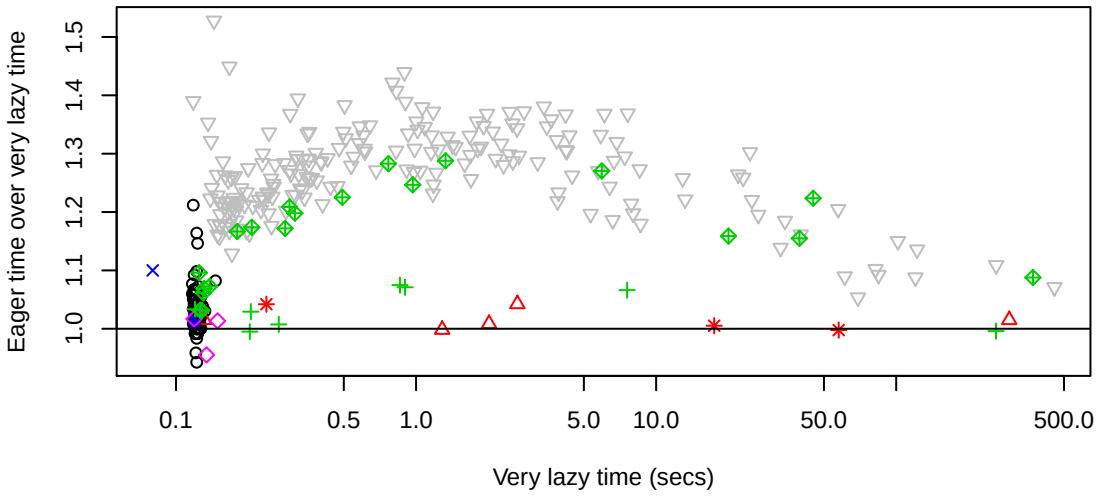
[10]that is, those using a subset of the following constraints: alldifferent, table, negative table, watched OR [**JMNP10**], lexicographic ordering, sum$\leq$, sum$<$, weightedsum$\leq$, weightedsum$<$, $x \leq y + c$, $\neq$, $x \leftarrow c$, $x \nleftarrow c$, $\lfloor x/y \rfloor = z$, $x \mod y = z$ and $x \times y = z$



(a) Lazy vs. eager



(b) Lazy vs. very lazy



(c) Very lazy vs. very lazy

FIGURE 3.6. Comparisons between variants of alldiff explanation: each point is a single instance

Problem class	Lazy vs eager		Lazy vs very lazy		Very lazy vs eager	
	Win	Lose	Win	Lose	Win	Lose
bqwh	77	15	48	44	81	11
costasArray	4	1	1	4	4	1
langford	7	1	4	4	6	2
latinSquare	5	0	5	0	5	0
magicSquare	1	2	1	2	2	1
minion	195	0	185	10	195	0
pigeons	19	0	19	0	19	0
queens	6	0	4	2	5	1
QWH	20	0	18	2	20	0
	334	19	285	68	337	16

TABLE 3.2. Speedups per variant summarised by problem class

Discussion. These results presage those of §3.6. Figure 3.6(a) compares the performance of variants of the solver, where the alldiff constraint is respectively lazy and eager, but for all other constraints the lazy explainer is used. These results show that, other than instances that finish in a very short time and are hence subject to high randomness, the lazy variant is always at least as good and up to 1.6 times better. The results in the first group of columns in Table 3.2 show that lazy wins on all problem classes handsomely.

Next I compare the lazy and very lazy variants of the alldiff explainer. Figure 3.6(b) shows that the majority of instances are faster lazily than very lazily. Hence the additional time spent re-running Tarjan's algorithm is greater than the time saved by avoiding storing the Hall set at propagation time. There are some instances that are faster "very lazily", but they are few and the improvement is not large. The second group of columns in Table 3.2 shows that lazy solves the majority of instances fastest for 5 problem classes, whereas very lazy is faster for costasArray. The bqwh category can be disregarded as these instances are solved very easily by all solver variants. Hence lazy is better on most classes but very lazy proves useful for a few.

Figure 3.6(c) and the third group of columns in Table 3.2 confirm that very lazy is still better than eager.

All 3 variants solve the same number of instances within the 10 minute timeout. Eager solves them in a total of 3383 seconds, very lazy in 3060 and lazy in 3002.

Hence, on these instances, the best lazy variant represents a 12.7% improvement over eager, which is currently the standard technique.

[**BM10**] includes a comparison between minion (with no learning) and the *argosat* SMT solver incorporating the first published theory for alldifferent. Results in this paper based on randomly generated sudoku instances show that argosat solves 194 instances within a 120 second time limit in an average time of 8.8 seconds per solved instance. Standard minion with no learning solves 174 instances in an average time of 10s per instance. On a different computer with a slightly slower clock rate, minion learning with lazy alldiff explanations solves 193 instances within the 120 second time limit, in an average time of 6 seconds per solved instance. Hence on these instances my learning solver is competitive with the only SMT solver with a theory for alldiff, even without the benefit of restarts or any sort of memory bounding technique.

3.5.6. Explanations for arbitrary propagators. I have now described how to apply the lazy approach to a variety of constraints. Katsirelos' GAC-Generic-Nogood [**Kat09**] is a procedure for finding explanations for an arbitrary propagator for constraint c with arbitrary implementation: the explanation of a disassignment $x \nleftarrow a$ (or assignment $x \leftarrow a$) is just the set of all prunings from variables $v \in \text{scope}(c) \setminus \{x\}$. It can easily be evaluated lazily by including only prunings that were made before the propagation happened. The existence of such a procedure proves that an explanation can always be produced lazily, although by specialising for each propagator as described above smaller explanations will be obtained, usually more quickly. Algorithm 12 shows how to obtain a lazy generic nogood. This procedure runs in $O(rd)$ time, which is the same worst case asymptotic complexity as Katsirelos' GAC-Generic-Nogood.

3.6. Experiments

I evaluated the effectiveness of lazy explanations in the minion-lazy solver, using the implementation decisions described in §3.4 and explanation algorithms described in detail in this chapter.

3.6.1. Other explanations used in the experiments. In the following experiments, several additional constraints are used whose explanation routines are neither

Algorithm LAZY-GENERIC-EXPLAIN($maxDepth, c$)

A1	**let** x be the variable whose dis-assignment is to be explained
A2	$expl \leftarrow \{\}$
A3	**for** $v \in \text{scope}(c) \setminus \{x\}$
A3.1	**for** $a \in \text{initdom}(v) \setminus \text{dom}(v)$
A3.1.1	**if** $depth(x \nleftarrow a) \leq maxDepth$
A3.1.1.1	$expl \leftarrow expl \cup \{x \nleftarrow a\}$
A4	**return** $expl$

Algorithm 12: Routine to lazily explain arbitrary (dis-)assignment by propagator for constraint c

published elsewhere nor described in this thesis. I do not describe them in detail because they are less interesting than or similar to those already described. Instead I give brief notes that will be helpful for reproducing the experiments.

Sum and weighted sum. Sum constraints are of the format: $\sum_{i=0}^{k-1} c_i x_i \leq c_k$ where each c_i is a constant and x_i a variable. Example 17 of [**OSC09**] describes how to propagate and explain this constraint. My propagator is the same, and the explanation routine is similar in essence except that it includes only enough disassignments to the lower bounds of the x_i's to ensure the sum minimally exceeds c_k.

Integer divide and modulo. The propagator for $\lfloor x/y \rfloor = z$ (integer divide) is very simple: wait until x and y are assigned (to a and b respectively) and then set z to $\lfloor a/b \rfloor$. The explanation for this assignment is just $\{x \leftarrow a, y \leftarrow b\}$ and laziness is straightforward.

Finally, the routine for producing explanations for $x \bmod y = z$ (modulo) is very simple. For example, for disassignment $x \nleftarrow a$ it iterates over pairs of values b and c such that $a \bmod b = c$, i.e. every pair of supports for $a \in dom(x)$, and in each pair adds to the explanation either $y \nleftarrow b$ or $z \nleftarrow c$, whichever was true at the time of pruning.

3.6.2. Experimental methodology. The experimental methodology used is identical to the experiments on alldiff described in §3.5.5.1, except that here an extended set of benchmarks have been used. There are a total of 2028 consisting of:

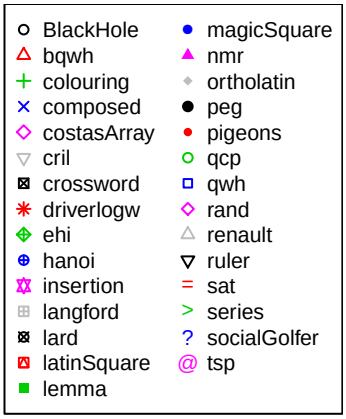


FIGURE 3.7. Legend for Figures 3.8 and 3.9.

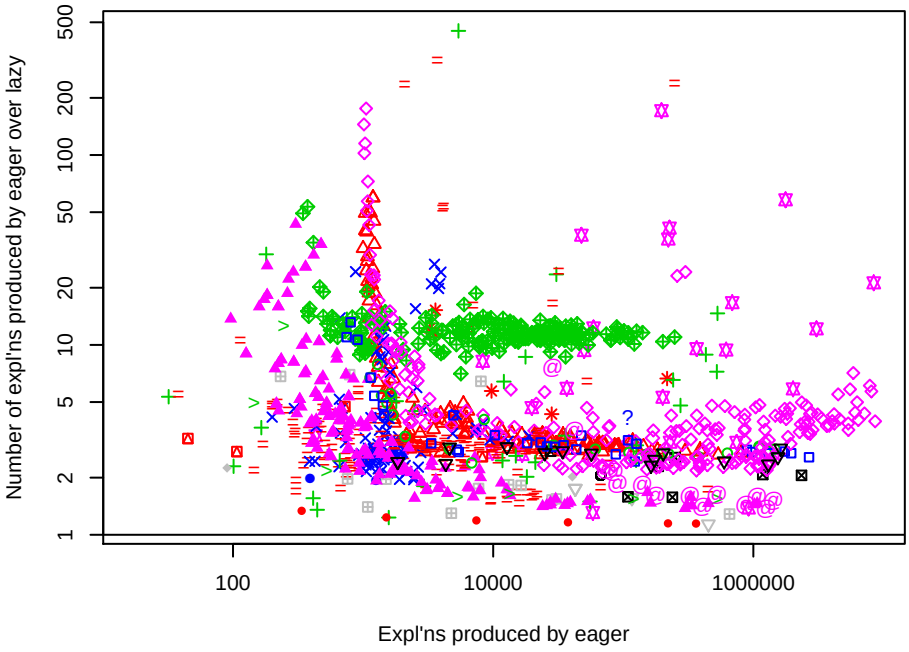


FIGURE 3.8. Scatterplot showing comparison of number of explanations produced by minion-lazy versus minion-eager, fewer for instances above the line. Legend is shown in Figure 3.7.

- all the instances from [**Lec**] that my solver is compatible with[11], once they are converted to minion format using tailor [**Ren10**] release 0.3.2, and
- all the compatible benchmarks I could find out of those used for testing and benchmarking minion internally.

[11]that is, those using a subset of the following constraints: alldifferent, table, negative table, watched OR [**JMNP10**], lexicographic ordering, sum$\leq$, sum$<$, weightedsum$\leq$, weightedsum$<$, $x \leq y + c$, $\neq$, $x \leftarrow c$, $x \nleftarrow c$, $\lfloor x/y \rfloor = z$, $x \mod y = z$ and $x \times y = z$

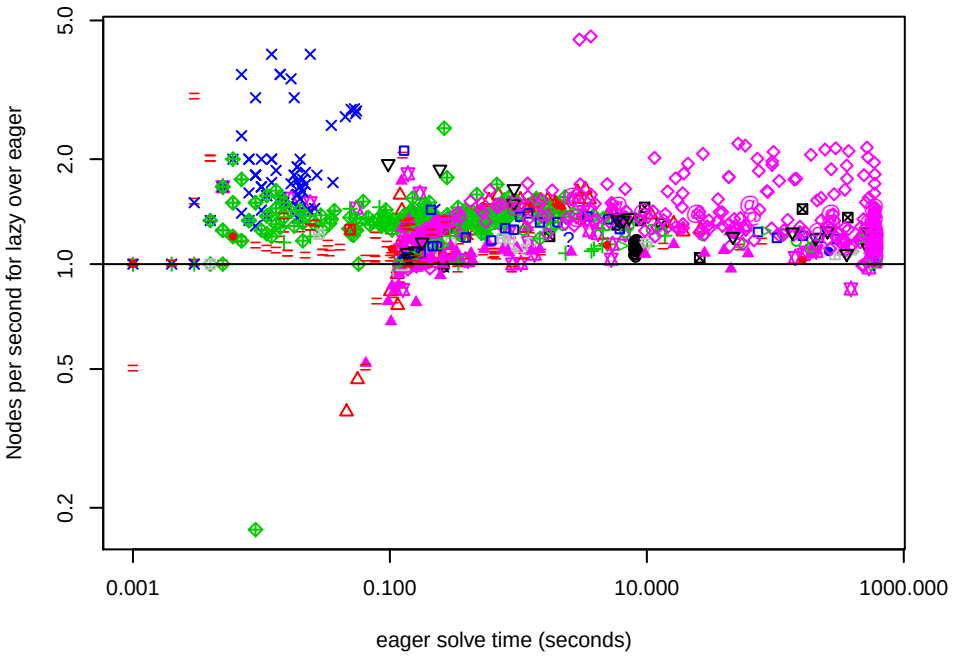


FIGURE 3.9. Scatterplot comparing nodes per second for minion-lazy versus minion-eager, more for instances above the line. Legend shown in Figure 3.7.

3.6.3. Results. Now to evaluate the subject of this chapter: are lazy explanations effective in reducing the runtime of the g-learning framework? The answer is yes. Figure 3.8 is a plot which shows the ratio of explanations produced using an eager solver to explanations produced using a lazy solver, for instances where neither solver timed out[12]. It shows a reduction in number of explanations generated in all cases, up to a factor of 500 reduction. This proves that the rationale behind lazy learning is correct—many explanations are never used and hence should not be calculated. For example a point with y-axis 20 needed just 1/20th of the explanations.

Next Figure 3.9 confirms that, on the whole, time is saved by using lazy explanations: lazy explanations can double the solver's search speed and also its overall search time, since the number of nodes searched is not affected at all. Note that this speedup is the whole solver, not just the learning engine. This is particularly significant because the solver spends only part of its time computing explanations. In fact, on some instances the maximum possible speedup is approached, i.e. time to

[12]when both solvers timed out, the search space could be slightly different as the timeout is not perfectly accurate

Problem class	Tot. inst.	Sig. inst.	Lazy wins	Draws	Eager wins
BlackHole	47	20	13	7	0
bqwh	192	22	22	0	0
colouring	51	8	6	2	0
costasArray	11	4	4	0	0
cril	10	5	5	0	0
crossword	12	8	8	0	0
driverlogw	7	1	1	0	0
ehi	200	15	15	0	0
insertion	73	16	9	7	0
langford	28	2	1	1	0
nmr	192	12	3	9	0
pigeons	39	2	1	1	0
qcp	40	13	13	0	0
qwh	40	12	12	0	0
rand	633	153	150	3	0
ruler	29	7	7	0	0
sat	166	6	4	2	0
series	25	2	2	0	0
socialGolfer	12	1	1	0	0
tsp	30	12	12	0	0

TABLE 3.3. Success of lazy learning against eager learning by problem class. Classes composed, hanoi, lard, latinSquare, lemma, magicSquare, ortholatin, peg and renault contained no sigificant instances and are omitted.

generate explanations approaches 0. In other solvers where the overhead of learning is different the speed increase may differ, but I think both eager and lazy have good implementations so the comparison is fair. The eager solver completes 1,318 instances in 437,541.9 seconds, whereas the lazy solver completes 12 more instances in 9138.1 seconds *less* time. However, lazy learning is detrimental to a small number of instances. Table 3.3 gives a detailed breakdown of how different problem classes are affected by this implementation decision. The columns of the table are respectively:

- a count of how many of each problem class was included (Tot. inst.);
- count of how many *significant instances* there were of that class (Sig. inst.), these are instances where both solvers took over 1 second to solve, to rule out noise as a source of speedups in easy instances, and where at least one instance completed search, so that two timeouts are not being compared;

- a count of how many times lazy was at least 10% faster on a significant instance (Lazy wins);
- a count of how many times neither solver was faster by at least 10% on a significant instance (Draws); and
- a count of how many times eager was faster on a significant instance (Eager wins).

The results show that lazy isn't significantly beaten by eager on any of the instances under test, and that, with the exception of 32 instances where the solvers draw, lazy wins overwhelmingly.

3.7. Conclusions

I have introduced *lazy explanations* for constraint propagation, in which explanations are computed as needed, rather than stored eagerly. This approach conveys the twin advantages, confirmed experimentally, of reducing storage requirements and avoiding wasted effort for explanations that are never used.

This chapter answered two hypotheses from Chapter 1:

Hypothesis 1. In a constraint learning CSP solver solving practical CSPs, most of the explanations stored are never used to build constraints during learning.

Hypothesis 2. The asymptotic time complexity of computing each explanation lazily is no worse than eager computation, or the practical CPU time to compute each lazy explanation for practical CSPs is no worse.

Hypothesis 1 was resolved by means of a comprehensive empirical evaluation, using benchmarks from 29 classes of problem. The number of explanations that were actually used during g-nogood learning CSP search using both eager and lazy learning was counted. The results (summarised in Figure 3.8) showed that, for all instances, using lazy explanation reduces the number of explanations needed, usually at least halving the number needed and sometimes reducing it by a factor of 500.

Also as part of the empirical evaluation, Table 3.3 summarises an experiment comparing time to first solution for g-nogood learning using eager and lazy learning on the same 29 problem classes: the lazy variant has never been known to lose by

10% to the eager variant for an instance that takes over a second to solve, whereas the lazy variant routinely beats the eager variant by well over 10%.

Hypothesis 2 was answered positively in §3.5, where I showed that lazy explainers for common constraints are no worse in terms of asymptotic time complexity than eager explainers. However there is a possibility that lazy explainers will have a larger constant factor than eager explainers so it is not automatic that computation time will be less in all cases. However the empirical results in Figure 3.9 show that a handful of instances are slowed down slightly by the use of lazy explanations (though not by more than 10%) and most are speeded up.

Chapter 4

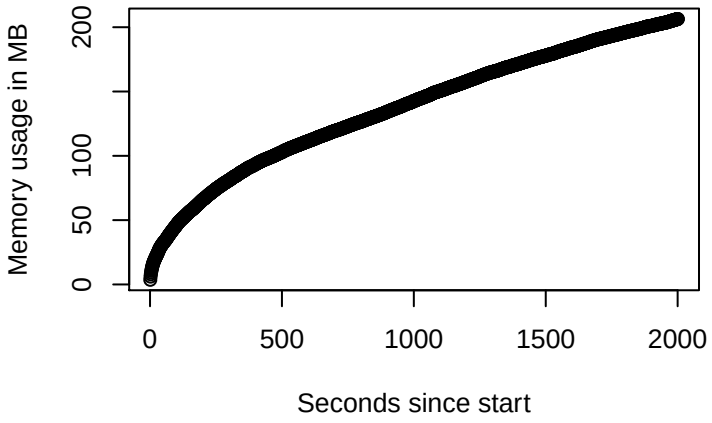
Bounding learning

> ...it is necessary that the reasoner should be able to utilize all the facts which have come to his knowledge; and this in itself implies, as you will readily see, a possession of all knowledge [which] is a somewhat rare accomplishment. It is not so impossible, however, that a man should possess all knowledge which is likely to be useful to him in his work, and this I have endeavored in my case to do.
>
> Sherlock Holmes
> *The Five Orange Pips*
> by Arthur Conan-Doyle

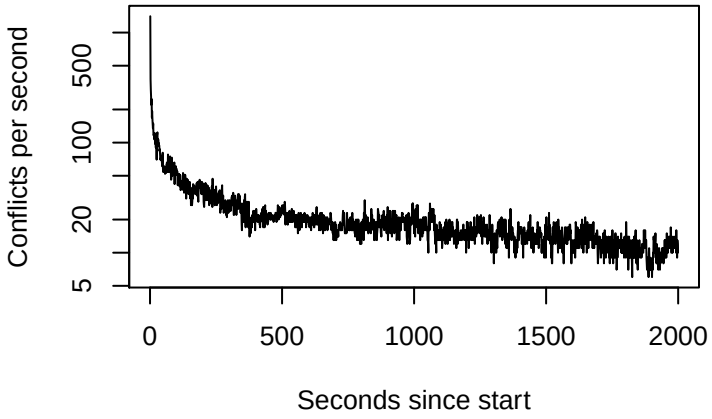
> It is of the highest importance in the art of detection to be able to recognise out of a number of facts which are incidental and which vital. Otherwise your energy and attention must be dissipated instead of being concentrated.
>
> Sherlock Holmes
> *The Reigate Squires*
> by Arthur Conan-Doyle

4.1. Introduction

g-learning is extremely effective on some types of benchmark, but its overheads can dominate on others. First, there is an overhead associated with instrumenting constraint propagators to store explanations, which are needed to produce the new constraints. This problem is mitigated by *lazy learning* described in Chapter 3, which reduces the overhead by producing explanations more efficiently. However the new constraints must still be propagated and this slows the solver down. Second, g-learning



(a) Memory usage each second throughout search



(b) Conflicts per second throughout search

FIGURE 4.1. Analysis of efficiency of unbounded solver over time

was originally described as *unrestricted learning* [**KB03**], where learned constraints are kept forever, but in an exponential search tree this results in exponential memory usage. In my experience this causes g-learning solvers to run out of RAM on commonly available systems within an hour, and so that the rate at which the solver can branch falls dramatically as it spends time propagating many learned constraints and swapping memory to and from the hard disk.

To illustrate the memory problem and drop in productivity I present Figure 4.1 showing what can happen. Figure 4.1(a) shows the growth of memory usage for a particular instance (`latinSquare-dg-8_all.xml.minion`) using unbounded learning. See §A.2 for details of how this data was collected. In just over 30 mins, the memory usage grows to over 200Mb, on a machine with 1GB per core. Another example is that instance `ruler-67-12-a3.xml.minion.conv.minion` consumes 800MB

in just over 30, none of which is reclaimed. The effect on conflict rate over time for `latinSquare-dg-8_all.xml.minion` is depicted in Figure 4.1(b). This shows that the solver's ability to find dead ends reduces over time and this is primarily caused by increased time spent enforcing consistency on learned constraints. A second effect is that memory access is less efficient: there are so many constraints that they can no longer all be cached and so the efficiency of propagation falls. Eventually disk will have to be used to store constraints, which is unacceptably slow. Hence the solver increasingly suffers from an increased quantity of less efficient propagation.

Although Figure 4.1(b) depicts a falling *conflict rate*, the same problem exists for *node rate*: more learning avoids wrong decisions but reduces the node rate. As the figure shows, the solver is slowed down so much that even if for the rest of search solutions were very easy to find, the node rate is so low that they would be found exceedingly slowly.

The contributions of this chapter are to analyse this memory issue by conducting an empirical investigation into the overheads introduced by unbounded constraint learning in CSP. This is the first such published study in either CSP or SAT. I obtain two significant results. The first is that a small percentage of learnt constraints do most propagation. While this is conventional wisdom, it has not previously been the subject of empirical study. I think it is important to verify and make precise folklore results, for until evidence exists and is published it is unverifiable and acts as a barrier for entry to new researchers, who may not yet be aware of folk knowledge. Second, I show that even constraints that do no effective propagation can incur significant time overheads. This contradicts conventional wisdom which suggests that watched literal propagators have lower overheads when not in use. This result shows why it is important to experiment on "known" results, because they are not always entirely correct. Finally, by implementing forgetting, I confirm that it can significantly improve the performance of modern learning CSP solvers, contradicting some previous research.

4.2. Context

As stated in [**AS09**], there exist few empirical studies into the effectiveness of modern conflict driven clause learning (CDCL) solvers. Although there exist many *techniques* that undoubtedly speed up such solvers, there is a lack of concrete knowledge about what underlies their success.

The fact that unrestricted learning is impractical has been understood for at least 20 years [**Dec90**]. One way to cope with this is to store constraints more efficiently than as a set of vectors of literals, e.g. for example by storing nogoods in an automaton [**RCJ06**], but this does not remove the fact storage space still grows unless the set of constraints happens to be generalisable. A second method is to bound learning at the time constraints are created, by suppressing constraints that take up too much space. *Size-bounded learning* ensures that learned constraints consist of at most k disjuncts and was introduced by Dechter and Frost [**Dec90, FD94**] in the context of s-learning CSP solvers; and used by Bayardo and Schrag [**BS97**] and Marques-Silva and Sakallah [**MSS96**] for SAT solvers.

A third method of reducing overheads is to *forget* (i.e. remove) constraints some time after they were learnt by a heuristic method. Forgetting constraints after adding them is, to the best of my belief, used universally in CDCL SAT solvers, e.g. [**BS97, ES03, GN07**]. *Relevance-bounded learning* introduced by [**BS97**] ensures that a constraint is removed once at least k of its disjuncts are no longer set, since for the constraint to unit propagate again $k-1$ of them must be set in a unique way. Hence for larger k the constraint is removed once the chance of propagation in the future diminishes sufficiently. Modern SAT solvers use *activity based* heuristics, e.g. [**ES03**], that remove constraints used less often according to subtle algorithms that count propagations but weight recent ones higher, so that clauses that propagate recently are most favoured, followed by constraints that propagated earlier, followed by those that have propagated little. I will describe some such algorithms in detail in §4.4.1.

4.3. Experiments on clause effectiveness

The following experiments analyse the overheads of unbounded constraint learning, showing that a small proportion of all learned constraints typically do the vast majority of all useful propagation and that they take a small proportion of overall time to do so.

4.3.1. Methodology. In the following experiments each instance was run once with a limit of 10 minutes search time. The reason why they were not run multiple times was that in this experiment the counts are important and variation in time due to system noise is not significant. They ran over three Linux machines with 8 Xeon E5430 cores @ 2.66GHz and 8GB memory. Lazy learning and the dom/wdeg [**BHLS04**] variable ordering heuristic were used throughout. These experiments involve the same instances used in §3.6.

4.3.2. Few clauses typically do most propagation. Received wisdom states that a small number of learned constraints do the majority of propagation in learning solvers, yet I am aware of no published evidence substantiating this view. The fact that constraint forgetting techniques are effective in learning solvers is consistent with the belief: if few constraints dominate most can be thrown away without harming search. However constraint forgetting in some form is a positive necessity to avoid running out of memory, so it would still benefit the solver even if individual constraints were comparably effective. Irrespective, the effect must be quantified, and understanding the effect quantitatively might help to design effective forgetting strategies.

4.3.2.1. *Procedure.* Measuring effectiveness of an individual constraint is more difficult in a learning solver than in a standard backtracking solver, because the learning procedure combines constraints together. Hence a constraint may do little propagation itself, but its child constraints may do a lot. Hence the influence of a constraint may be wide. This is a subtle issue and I have not attempted to measure it. Rather I will be measuring only the direct effects of individual constraints (their propagation), and not their "influence" (their own propagation and propagation by constraints derived from them during learning).

Therefore, in this section, the number of propagations is used as a measure of the effectiveness of a learnt constraint. This choice is not immediate, so I will now discuss why it was chosen. The problem is that propagations are not necessarily beneficial if they remove values but do not contribute to domain wipeouts or other failures. To get around this issue, as part of its clause forgetting system (see §4.4.1) minisat [**ES03**] measures the number of times a constraint has been identified as part of the reason for a failure. Hence, I did consider using the number of propagations that lead to failure as a measure of constraint effectiveness, rather than raw number of propagations. However, the correlation coefficient between propagation count and count of involvement in conflicts is 0.96. The procedure for an experiment computing this correlation is described in §A.1. In other words each propagation is roughly equally likely to be involved in a conflict. Hence the following results should apply almost equally to propagations resulting in failure. The advantage of using the total number of propagations is that it is more easily defined and less coupled with learning.

For efficiency reasons, solvers do not collect this data by default. In order to carry out these experiments my solver was amended to print out a short message whenever a constraint propagated, giving the unique constraint number and the node at which the propagation occurred. These data were then analysed externally with the aid of a statistical package. Although this slows the solver down, the experiment is fair because counts are not affected.

Note that the later a constraint is posted, the less time it has to propagate. Hence the number of raw propagations carried out by each constraint are not directly comparable. To get around this the propagation counts are each over the same number of nodes. Specifically, only constraints learned during the first 50% of nodes after the constraint is learned are included, and for each such constraint the number of propagations are counted only over the following 50% of nodes, so that every count is over the same number of nodes. For example, if the problem is solved in 9999 nodes, constraints learned between nodes 1 and 5000 are included, and the constraint learned at node 278 is counted from nodes 278 to 5277.

4.3.2.2. *Results and analysis.* For instance for `latinSquare-dg-8_all.xml.minion` I exhibit a graph that I will later show is representative of other instances. The upper

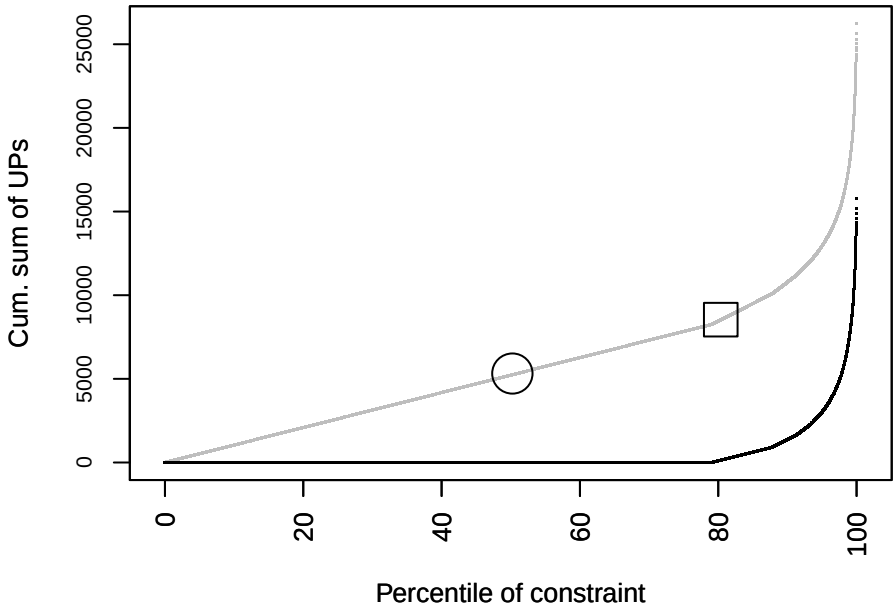
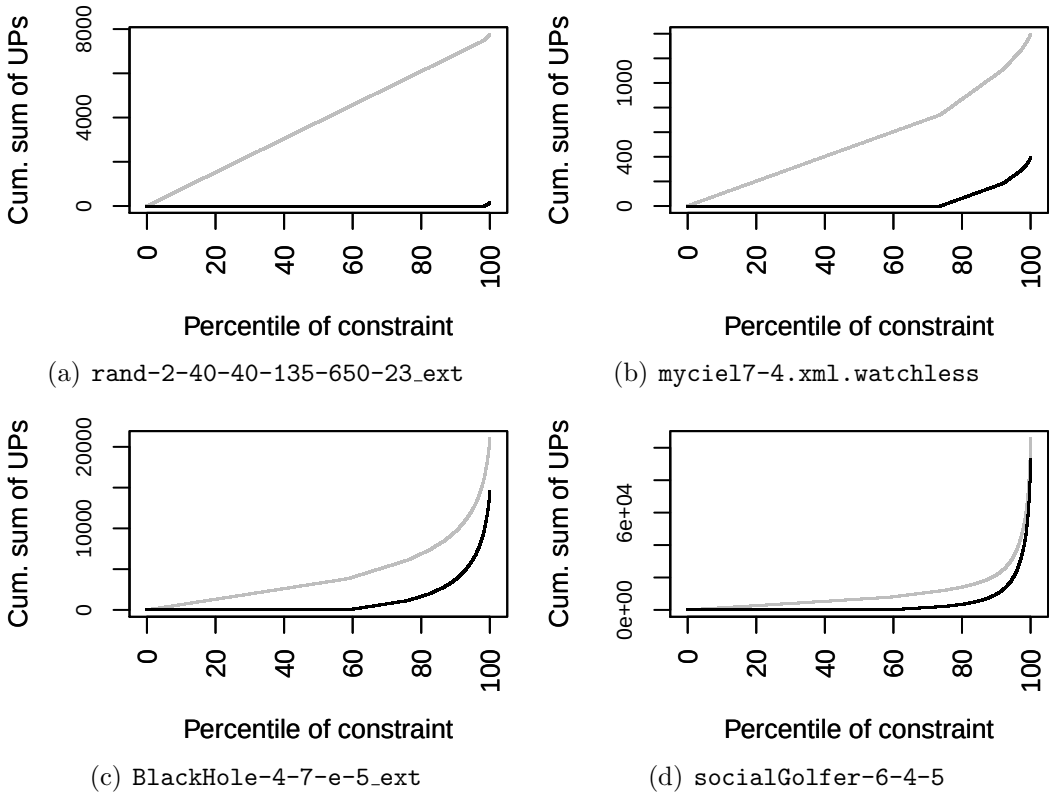


FIGURE 4.2. What proportion of constraints are responsible for what propagation? – single instance (`latinSquare-dg-8_all.xml.minion`)

curve in Figure 4.2 shows what proportion of the best constraints are responsible for what proportion of all unit propagations (UPs)[1]. By "best" I mean doing the most propagations. Each point is an individual constraint. The x-axis is the percentile of the constraint's propagation. The y-axis is the number of propagations accounted for by that constraint and those with a lower percentile. For example, the circled point on the x-axis is the median (50th percentile) constraint by propagation count: it is the 5223th constraint, out of 10446. The total propagation count for all 5223 constraints is exactly 5223 out of a total of 26220 for all constraints, i.e. 20% of the total. Hence the bottom 50% of constraints account for just 20% of all propagation. The slope is shallow until the 80th percentile constraint (marked by a small square), after which it steepens dramatically. Hence the top 20% of constraints do a lot more work than the rest. This agrees with the hypothesis that a minority of constraints do most propagation.

I noted in §2.6.1 at Example 2.16 (page 40) that each constraint is guaranteed to propagate at least once. This first propagation has the effect of a right branch, so does not contribute effectively since the solver would have done this anyway. Hence I now

[1] a unit propagation can cause either an assignment or a disassignment, depending on the unit literal



(a) `rand-2-40-40-135-650-23_ext` (b) `myciel7-4.xml.watchless`

(c) `BlackHole-4-7-e-5_ext` (d) `socialGolfer-6-4-5`

FIGURE 4.3. What proportion of constraints are responsible for what propagation? – multiple instances

report results with these ineffective propagations deleted. In the black (lower) curve in Figure 4.2 the same graph is shown with 1 subtracted from the propagation count of each constraint. Here the curve is zero until the 80% percentile, meaning that the worst 80% of constraints contribute no additional propagation after the right branch, i.e. just one propagation each: just 20% of constraints do *all* useful propagation and 10% do almost all.

In Figure 4.3, a further 4 randomly selected instances are displayed in the same style as Figure 4.2. These graphs are broadly consistent with my observations for Figure 4.2. Specifically, the black curve in each each remains at zero until at least the 60th percentile, showing that at least 60% of constraints are doing no useful propagation. The graphs vary in their other features. Figures 4.3(a) and 4.3(b) are for instances where learning is quite ineffectual, as evidenced by the fact that the black curves are far separated from the gray curves, meaning few clauses propagate

P	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1%	0.01	0.01	0.01	0.04	0.03	2.04
5%	0.01	0.02	0.04	0.09	0.09	2.04
10%	0.01	0.05	0.08	0.19	0.18	3.64
15%	0.01	0.09	0.13	0.31	0.31	3.91
20%	0.01	0.12	0.19	0.46	0.47	5.46
25%	0.01	0.17	0.27	0.64	0.68	6.80
30%	0.01	0.23	0.35	0.86	0.92	8.24
35%	0.01	0.30	0.46	1.11	1.22	9.69
40%	0.01	0.37	0.58	1.40	1.58	11.13
45%	0.01	0.47	0.72	1.73	1.99	12.57
50%	0.01	0.57	0.86	2.11	2.51	14.02
55%	0.02	0.67	1.00	2.56	3.22	16.33
60%	0.02	0.78	1.18	3.07	3.93	18.76
65%	0.02	0.89	1.34	3.65	4.86	21.27
70%	0.02	0.99	1.51	4.34	6.09	24.39
75%	0.02	1.09	1.70	5.15	7.56	27.51
80%	0.02	1.19	1.89	6.15	9.50	30.83
85%	0.02	1.32	2.08	7.40	11.75	37.07
90%	0.02	1.44	2.27	9.11	15.37	43.32
95%	0.02	1.55	2.48	11.68	21.88	50.00
100%	0.02	1.65	2.71	16.03	37.06	69.89

TABLE 4.1. What proportion of constraints are responsible for what propagation? – all instances

more than once. In Figures 4.3(c) and 4.3(d) the curves are quite close, meaning the better clauses are contributing quite a lot of additional propagation.

The previous results focus on specific instances, so I will now expand analysis to all 949 instances from the test set that cannot be solved within 1000 nodes of search. This is done to ensure that a trend has a chance to establish: to analyse only a few constraints might be less meaningful. In Table 4.1 for each chosen percentage P, I give what percentage of the best constraints are needed to account for P% of overall non-branching propagation[2]. These results show that usually a small proportion of the best constraints perform a disproportionate amount of propagation. For example 10% of all propagation is performed by a median of just 0.08% of constraints, and 100% by a median of just 2.71% and a maximum of 69.89%. Hence the behaviour

[2]It may seem anomalous that some entries exceed P%, since the best P% constraints must do *at least* P% of propagations. This apparent anomaly is because there may be no integer number of constraints doing P% of propagation, so it is necessary to overcount.

described above for a single benchmark is robust over many instances: the best few constraints overwhelmingly perform most non-branching propagation. If anything, the above sample instance understates the effect, since it required about 20% instead of the median of 2.71% of constraints to do all propagations.

4.3.2.3. *Conclusion.* I have shown empirically that the best constraints are responsible for much of the propagation and thus search space reduction.

4.3.3. Clauses have high time as well as space costs. Unit propagation by *watched literals* [**MMZ+01**] is designed to reduce the amount of time spent propagating infrequently propagating constraints, by the possibility of watches migrating to inactive literals that do not trigger and cost nothing to propagate. Before describing the experiment, I will first briefly outline how watched literal propagation works.

Recall from Definition 3.3, that unit propagation (UP) is a way of propagating clauses. Watched literals are an efficient implementation of UP, first described in [**MMZ+01**]. The idea is to *watch* a pair of variables, that are not set to false. Provided that such variables exist, the clause must be satisfiable, and unit propagation needn't happen yet. Suppose that one of these variables is set to false: if another non-false variable can be found then the propagation watches it instead, otherwise the single non-false variable has to be unit propagated to true immediately to avoid the constraint being unsatisfied. The empirical evidence suggests that since the propagator only cares about assignments to two variables it is efficient compared to other unit propagators that watch all assignments (e.g. ones that count false assignments). If the watched variables are set to 1 early in search then the clause will essentially be zero cost until the solver backtracks beyond that point, because it will never be triggered on those variables.

Hence perhaps these weak constraints do not cost much time, if space is available to store them, since there is a possibility of infrequently propagating constraints doing little work. Hence the next question is: do constraints which do not propagate cost significant time as well as space?

4.3.3.1. *Procedure.* The minimum amount of time to process a single domain event with a watched literal propagator can be of the order of a handful of machine instructions, taking nanoseconds to run, during which time the system clock may not tick.

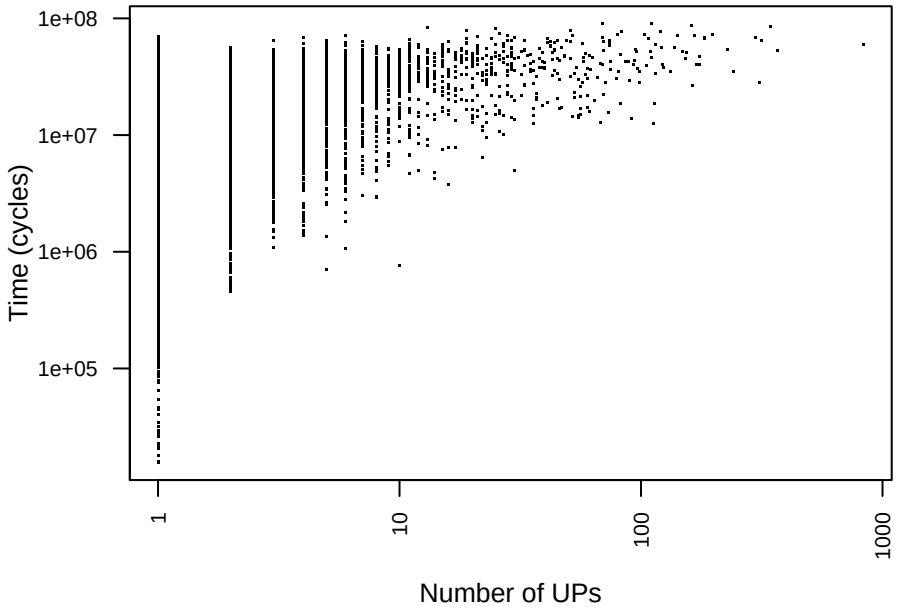
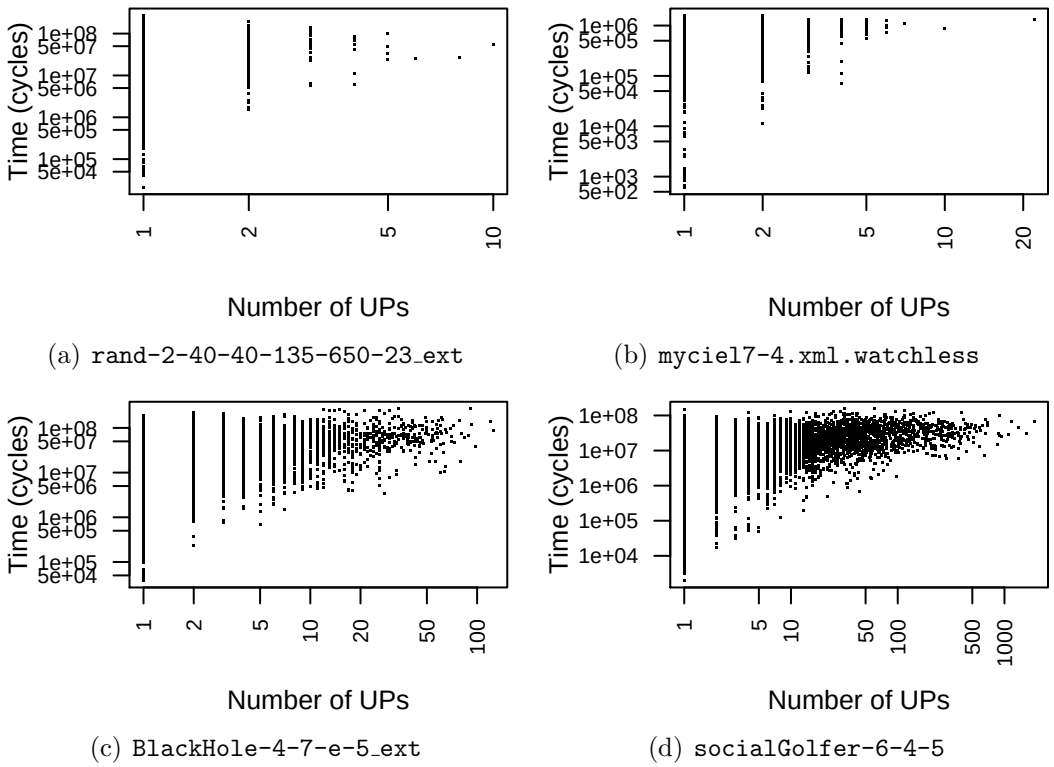


FIGURE 4.4. How much time does propagation take? – single instance (`latinSquare-dg-8_all.xml.minion`)

Hence, to obtain nano-scale timings, the solver keeps a running total of the number of processor clock ticks as recorded by the RDTSC register specific to Intel processors [**Int00**]. A processor clock tick is the smallest measure of time appropriate to a processor, being the time it takes to advance the instruction pipeline by one stage. Each of these occupies $1/(2.66 \times 10^9)$ seconds, since I used a 2.66 GHz Xeon E5430. The overhead of collecting data is very low, taking only one assembly instruction to get the number, and a few more cycles to add it to the running total.

At the end of search, all the cycle counts are printed out and analysed externally with the aid of a statistical package.

4.3.3.2. *Results and analysis.* How does time spent correlate with unit propagations performed? Figure 4.4 is a scatterplot for the single instance used in §4.3.2.2. Each point represents a single constraint. The x-axis gives the number of unit propagations (including the right-branching initial one), and the y-axis the total number of processor cycles used to propagate it during the entire search. First, and unsurprisingly, as an individual constraint propagates more, it often requires more time to do so. What may be surprising is that the *worst case* for constraints is roughly constant, and independent of the number of propagations. That is, individual constraints which



(a) `rand-2-40-40-135-650-23_ext` (b) `myciel7-4.xml.watchless`

(c) `BlackHole-4-7-e-5_ext` (d) `socialGolfer-6-4-5`

FIGURE 4.5. How much time does propagation take? – multiple instances

do no effective propagation can take a similar amount of time to propagate as individual constraints which propagate almost 1000 times. For this sample instance, 74% of propagation time is occupied with constraints that never propagate again after the first time. This suggests that learnt constraints can lead to significant time overhead without doing any useful propagation.

In Figure 4.5 a further 4 randomly chosen (the same as in Figure 4.3) graphs are displayed in the style of Figure 4.4. These exhibit a similar behaviour to that observed in Figure 4.4, with a range of different amounts of propagation: the worst case cost of the least propagating constraints is quite similar to the cost of the constraints that propagate most often. That is, the poorest propagating constraints are a major overhead.

Table 4.2 extends the study to the 1,923 instances out of the full set of 2,050 where at least one constraint is learned. Each row is a chosen percentage $R\%$ of the total non-branching propagations, and the columns are summary statistics for what % of the overall propagation time the best constraints take to achieve $R\%$ of all

R	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1%	0.00	0.02	0.17	6.12	3.32	100.00
5%	0.00	0.05	0.33	6.17	3.32	100.00
10%	0.00	0.11	0.62	6.30	3.52	100.00
15%	0.00	0.18	0.95	6.50	3.82	100.00
20%	0.00	0.26	1.38	6.79	4.38	100.00
25%	0.00	0.35	1.88	7.12	5.11	100.00
30%	0.00	0.45	2.31	7.52	5.82	100.00
35%	0.00	0.54	2.85	8.07	6.82	100.00
40%	0.00	0.63	3.38	8.46	7.75	100.00
45%	0.00	0.71	4.03	9.01	9.10	100.00
50%	0.00	0.79	4.54	9.46	9.97	100.00
55%	0.00	0.91	5.38	10.50	11.67	100.00
60%	0.00	1.04	6.08	11.16	13.32	100.00
65%	0.00	1.20	6.87	11.97	15.10	100.00
70%	0.00	1.38	7.99	13.06	17.73	100.00
75%	0.00	1.58	9.06	14.00	19.62	100.00
80%	0.00	1.78	10.07	15.27	22.59	100.00
85%	0.00	2.03	11.35	16.78	25.91	100.00
90%	0.00	2.29	12.56	18.55	30.03	100.00
95%	0.00	2.59	14.31	20.76	34.05	100.00
100%	0.00	2.89	15.23	24.01	41.02	100.00

TABLE 4.2. How much time does propagation take?–all instances

propagation. A constraint is "better" than another if it does more propagations per second of time spent propagating. For example, the third row says that the median over all instances is that 10% of all non-branching propagation can be done in just 0.62% of the time taken by the best available constraints. Using the most efficient constraints, all non-branching propagation can be achieved in a mean of less than a quarter of the time of using all constraints. All other time spent is completely wasted since it leads to no effective propagation.

4.3.3.3. *Conclusion.* The results in §4.3.3 show that learnt constraints which do no propagation contribute significantly to the time overhead of the solver. This is significant in that it shows that useless clauses can be very costly on an individual basis. Conversely, it had often been assumed that non-propagating constraints would not take a lot of time to process because the watches could migrate to "silent literals" that do not trigger often. Hence I have shown that this appears not to be the case

and the experiments of §4.3.4 confirm that the time is spent searching for watched literals.

4.3.4. Where is the time spent? The experiments of this section raise the question of what exactly the solver is doing while propagating clauses, especially when they are propagating infrequently. It is worth verifying that indeed the time is spent moving watched literals.

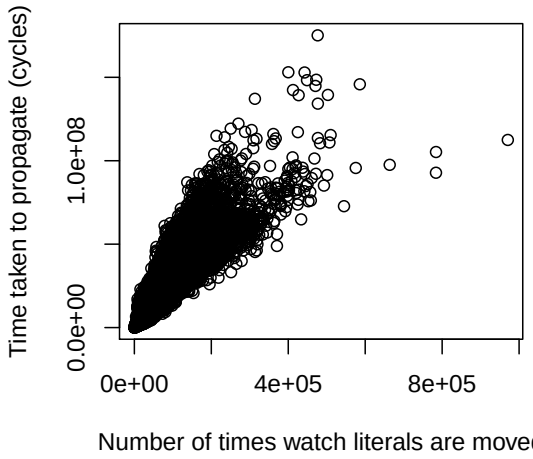
Recall that the watched literals (WLs) propagation algorithm works by detecting when WLs have become set to false, and then searching through the rest of the literals attempting to find one that is unset to replace the false WL. This involves looping over the literals until either an unset literal is found or all have been checked.

In Figure 4.6 is a plot for each of the example CSPs dealt with in the previous sections (the instance names are in the plot labels). Each point is a single constraint. The x-axis gives the number of literals inspected while search for new watched literals for the single constraint. The y-axis gives the amount of time spent propagating the constraint in total (in cycles which are 1×10^{-9} seconds each). The graphs demonstrate that the number of checks while searching for a new WL is roughly proportional to the overall propagation time as expected. The captions on individual plots in Figure 4.6 provide the correlation between these quantities which are between 0.683 and 0.976 for these instances.

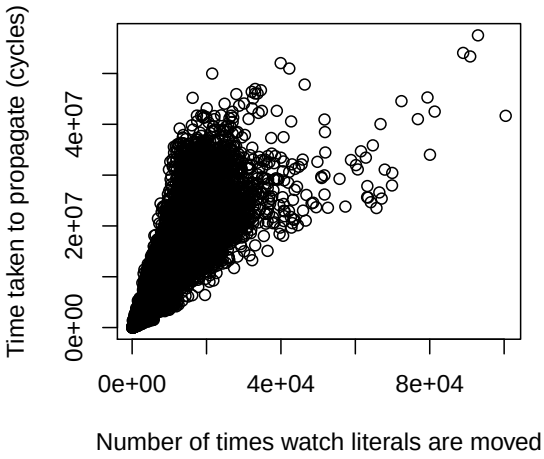
These plots are consistent with the understanding that constraints that take a lot of time to propagate move their watches more than constraints that take little time to propagate, on average.

4.4. Clause forgetting

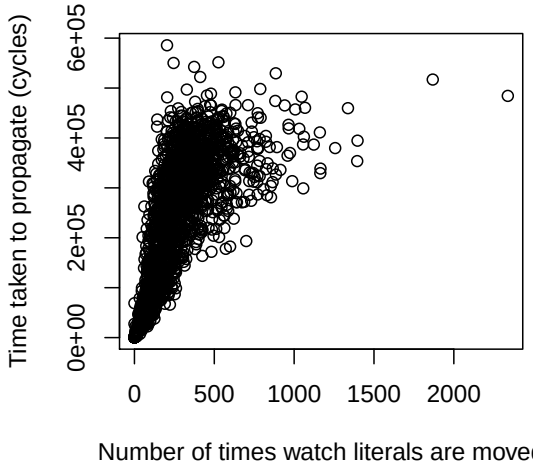
The above results suggest that, if picked carefully, the solver can often remove constraints to save a lot of time at only a small cost in search size. As described in §4.2, this is not a new idea in either constraints or SAT. Indeed Katsirelos and Bacchus have implemented relevance bounded learning for a g-learning solver in [**KB03**]. They report poor results showing that relevance bounding with $k = 3$ leads to more timeouts and slower solution time. However a very small number of similar problems are tried so results are inconclusive.



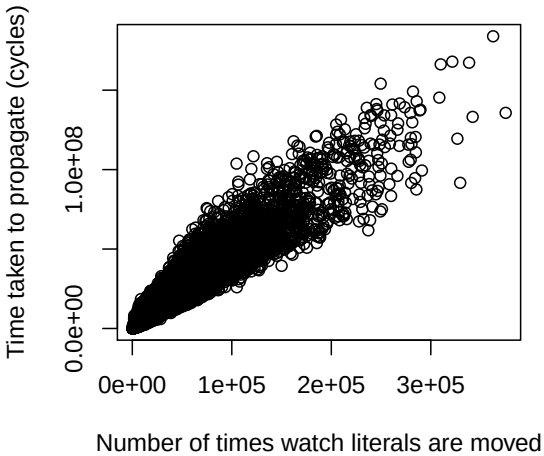
(a) `BlackHole-4-7-e-5_ext`, correlation=0.895



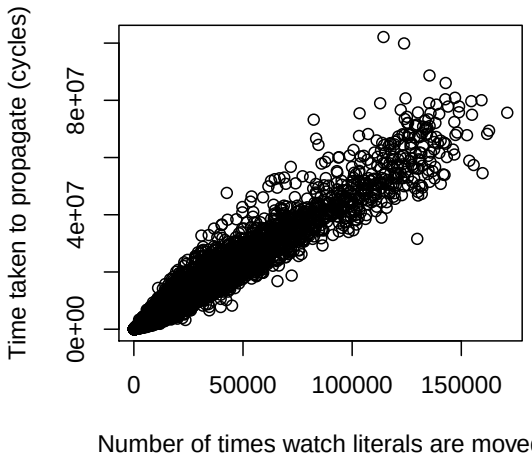
(b) `latinSquare-dg-8_all.xml`, correlation=0.862



(c) `myciel7-4.xml.watchless`, correlation=0.683



(d) `rand-2-40-40-135-650-23_ext`, correlation=0.929



(e) `socialGolfer-6-4-5`, correlation=0.976

FIGURE 4.6. Number of watched literal movements against propagation time for several instances

In this section, I try a range of well-known existing strategies for forgetting learned constraints, applying them for the first time either to CSP or to a g-learning CSP solver.

4.4.1. Context. Size-bounded and relevance-bounded learning, described above in §4.2, have been applied successfully to the CSP in the past, but using a s-learning solver. Firstly, since size-bounded learning was last tried [**Dec90**], algorithms for propagating disjunctions have progressed significantly with the introduction of watched literal propagation [**MMZ$^+$01**], meaning that learned constraints are faster to propagate. Hence the technique may no longer be useful and, if it is useful, the optimal choice of parameters will probably have changed as long clauses become less burdensome. Secondly, the learning algorithms applied have fundamentally changed with the advent of g-nogood learning. Katsirelos has shown [**Kat09**] that the properties of clauses change as a result of g-learning, for example the average clause length can reduce. This also motivates the re-evaluation of existing forgetting strategies. Finally, theoretical results [**Joh10, BSJ10**] from SAT show that there is an exponential separation between solvers using size-bounded learning and learning unrestricted on length, meaning that the former may need exponentially more search than the latter on particular problems. This means that size-bounded learning is theoretically discredited, but it remains to see how it performs in practice.

Recently there have been a collection of new forgetting heuristics in SAT solvers, which are based on activity. Using activity-based heuristics the clauses that are least used for conflict analysis are removed when the solver needs to free space to learn new clauses. As well as guessing which clauses are least beneficial, new strategies also decide *how many* to keep. This is a difficult trade off, because keeping more increases propagation time, but throwing them away reduces inference power. The best choice is problem dependent. In this chapter, the class of activity based heuristics are represented by the strategy used in the minisat solver [**ES03**], which I will call the *minisat* strategy.

The strategy has 3 main components:

activity: each clause has an activity score, which is incremented by 1 each time it is used as an explanation in the firstUIP procedure

decay: periodically, activities are reduced, so that clauses that have been active recently are prioritised

forgetting: just before the scores are decayed each time, half of all constraints are removed with a couple of exceptions:

- those that have unit propagated in the current branch of search are kept,
- those with scores below a fixed threshold are removed first even if the target of removing half has already been reached, and
- binary and unary clauses are always kept.

In order to implement this algorithm the frequency of decay & forgetting and the divisor for decay must be supplied. The threshold below which all clauses are removed is simply 1 over the size of the clause database because that is the default in [**ES03**].

4.4.2. Experimental evaluation. I will describe an experiment to test the effectiveness of the forgetting strategies from the literature described above.

4.4.2.1. *Implementing constraint forgetting.* As mentioned in §4.3.2.2 each learned constraint propagates at least once and this is necessary for the completeness of g-learning. Hence when implementing bounded learning, my solver propagates it once anyway even if the constraint is going to be discarded immediately.

In my implementation, currently unit clauses, a.k.a. *locked* clauses[3], can be slated for deletion meaning that they are not propagated any more, but the memory cannot be freed until it is no longer unit.

In my solver, restarts are not used, and hence it is easy to prove that deleting clauses is safe (i.e. the solver is still complete), provided that they are not locked.

Theorem 4.1. *Non-locked learned clauses can be removed from a clause driven constraint learning (CDCL) solver at any time without impacting completeness.*

Proof. The completeness of CDCL (without forgetting) is reliant on the fact that when the solver is at decision depth i, there are more literals inferred than the previous time it was at depth i. This is ensured because the clause learned after a backjump is an asserting clause and must infer a new literal. If non-locked clauses

[3]nomenclature due to [**ES03**]

are allowed to be removed, this property still holds, because the clauses removed do not contribute to the count of inferred literals at any current decision depth. □

Recall that for k relevance bounding, the solver must remove the constraint when k literals become unset for the first time. My implementation works as follows: when the constraint is created the literals are sorted by descending depth at which they became false[4] and the k'th depth is selected. Suppose this is depth i. When the solver backtracks beyond depth i, exactly k literals will have become unset. The constraint is therefore pushed into a stack associated with depth i. When the solver backtracks beyond depth i, constraints scheduled for deletion then can be popped off the stack one by one in $O(1)$ time each and deleted. This implementation has little runtime overhead above normal propagation because there is exactly $O(1)$ work for each constraint to decide when to delete it, once it has been added to the correct stack.

The implementation of size-bounded learning and the minisat strategy follow straightforwardly from the definitions given above.

4.4.2.2. *Experimental methodology.* Each of the 2028 instances was executed four times with a 10 minute timeout, over 3 Linux machines each with 2 Intel Xeon cores at 2.4 GHz and 2GB of memory each, running kernel version 2.6.18 SMP. Parameters to each run were identical, and the minimum time for each is used in the analysis, in order to approximate the run time in perfect conditions (i.e. with no system noise) as closely as possible. Each instance was run on its own core, each with 1GB of memory. Minion was compiled statically (`-static`) using g++ version 4.4.3 with flag `-O3`.

4.4.2.3. *Beauty contest.* I tried each strategy with a wide range of parameters and in Table 4.3 report a selection of the best parameters for each. The best parameters were found by testing a wide interval of possible parameters, and finding a local optimum. Close to the local optimum more parameters were tried to locate the best single value where possible (e.g. for discrete parameters). Minion with no learning is also included in the comparison under name "stock.undefined" (I refer to unchanged minion version 0.9 as "stock" minion). In the table, the strategies are abbreviated to name.parameter, except minisat which is abbreviated to minisat.interval.decayfactor.

[4]this information is available from the learning subsystem

The "Beauty Contest" columns give both the number of instances solved and the total amount of time spent. Hence an instance that times out does not count towards instances solved and costs 600 seconds. The best strategy is that which solved the most instances, taking into account overall time to break ties. In the table the best strategies are listed first. Finally first and third quartiles and median nodes per second (NPS) are given. These statistics show the increase or decrease in search speed. A solver with forgetting should have a higher search speed because it has fewer constraints to propagate. The 'Search measures' columns give measures of what effect each strategy has compared to unbounded learning. This is a measure of how effective search is compared to unbounded learning, as opposed to how fast. The columns are as follows:

the number of instances the variants and unbounded both complete: The number of instances being compared in the following two statistics.

what factor additional nodes the strategy needs on those instances: The smaller the number[5], the less propagation is lost as a result of forgetting.

speedup factor: e.g. speedup factor of 2 means that the strategy takes half the time to solve the all the instances. Note that because only instances completed by both are included, there are no timeouts in the total.

The aim is to maximise nodes per second, while keeping the node increase as little as possible.

4.4.2.4. *Analysis of results.* In these results, most of the strategies for forgetting clauses improve over unbounded learning (none.undefined in Table 4.3) in terms of both instances solved and overall time. There is an overall increase in the number of instances solved: provided that the increased node rate compensates for the increase in the number of nodes searched, there will be a net win. There is an apparent paradox because for some strategies that beat unbounded learning, e.g. size.2, the number of nodes increases more than the node rate in the "search measures" section. However this is not a problem, because "beauty contest" is based on all instances, whereas "search measures" is based only on instances that didn't timeout. Hence the

[5]constraint forgetting could occasionally lead to *less* search, as in backjumping [**Pro93a**], so a number under 1 is possible in principle

Strategy	Beauty contest					Search measures		
	Instances	Time	1st Q NPS	Median NPS	3st Q NPS	Instances	Nodes inc.	Speedup
stock.undefined	1667	248598.9	403.9	1353.0	10390.0	1312	129.6	6.7
relevance.6	1641	278203.7	205.3	502.4	1257.0	1336	2.4	4.2
relevance.5	1639	277357.3	217.6	541.6	1433.0	1336	2.8	4.7
relevance.4	1639	280652.1	222.5	533.4	1549.0	1333	3.6	4.3
relevance.7	1637	278973.3	201.7	482.9	1184.0	1336	1.9	4.4
size.10	1637	280804.7	196.7	534.4	1225.0	1336	4.1	5.1
relevance.10	1636	279244.4	178.1	454.1	1021.0	1335	1.6	5.2
relevance.3	1635	280366.6	242.1	566.2	1728.0	1336	5.5	3.4
size.8	1635	281008.0	214.6	566.2	1383.0	1335	5.2	4.5
size.5	1634	283213.5	235.9	595.7	1574.0	1335	7.5	3.9
relevance.14	1631	281037.3	141.7	409.5	874.6	1334	1.3	5.6
size.12	1631	282370.3	187.6	504.2	1143.0	1335	2.1	5.5
size.13	1631	282911.4	180.1	485.7	1081.0	1335	1.8	5.5
size.14	1631	283324.7	180.1	469.2	1044.0	1335	1.6	5.7
relevance.15	1629	282680.8	136.6	404.9	865.1	1335	1.3	5.9
size.9	1629	283146.9	205.9	541.2	1298.0	1334	4.5	5.0
size.11	1629	283882.0	193.7	516.0	1170.0	1333	3.0	5.3
relevance.16	1629	284854.4	134.5	406.7	860.9	1335	1.3	5.6
size.15	1628	287587.7	176.5	463.9	1007.0	1333	1.7	4.7
relevance.13	1627	281439.7	155.0	427.0	928.2	1335	1.4	5.3
relevance.2	1625	287833.7	250.6	580.3	2006.0	1329	61.3	3.2
relevance.12	1623	284866.5	159.0	420.5	928.9	1334	1.4	5.3
size.2	1621	289421.7	257.4	604.3	2088.0	1327	21.6	3.7
relevance.17	1620	288246.0	126.1	402.2	830.4	1335	1.3	5.1
size.20	1619	295401.9	155.1	413.9	907.9	1335	1.3	4.9
relevance.20	1618	293226.9	119.2	361.1	783.1	1334	1.2	5.3
size.1	1616	294566.6	262.4	611.1	2192.0	1323	61.6	3.1
mostrecent.1	1600	302325.7	227.2	544.0	2102.0	1319	65.8	3.1
mostrecent.2	1600	305267.5	206.9	500.7	2008.0	1323	37.0	2.8
mostrecent.10	1569	326114.8	155.6	381.5	1683.0	1323	34.8	2.6
relevance.30	1555	333292.2	98.4	255.6	686.2	1335	1.2	4.1
size.30	1554	330743.5	124.0	359.9	786.2	1335	1.2	4.2
minisat.1.1	1517	349391.3	112.9	278.1	1164.0	1326	8.0	2.1
relevance.40	1501	360096.1	70.5	166.2	635.5	1335	1.1	3.3
size.40	1498	354322.2	108.1	260.1	720.8	1334	1.1	3.9
mostrecent.100	1475	386555.2	77.2	217.8	1002.0	1326	6.1	2.2
minisat.201.501	1440	410767.3	60.8	173.3	810.8	1321	2.0	2.0
minisat.201.1001	1439	411044.4	60.9	170.6	800.4	1321	2.0	2.0
minisat.201.1	1438	410130.1	60.9	174.2	805.6	1321	2.0	2.1
minisat.401.501	1419	431958.5	46.4	152.4	698.8	1319	1.8	1.9
minisat.401.1001	1417	438939.3	45.6	146.5	676.0	1320	1.8	1.7
minisat.401.1	1413	444863.3	43.8	143.5	660.1	1319	1.8	1.6
relevance.100	1404	406542.4	31.4	99.2	564.3	1330	1.0	2.0
size.100	1397	406529.6	40.5	110.5	581.3	1330	1.1	1.9
minisat.601.1001	1373	500036.1	36.8	127.9	586.7	1319	1.6	1.4
minisat.601.501	1371	502484.1	36.1	121.2	583.9	1318	1.5	1.4
mostrecent.1000	1371	559058.3	31.6	106.3	566.1	1330	1.3	1.6
minisat.601.1	1367	510004.5	35.8	126.0	581.4	1316	1.4	1.5
minisat.1.1001	1344	440553.2	22.7	100.7	585.6	1322	3.0	0.9
none.undefined	1343	440552.2	22.2	76.4	510.0	1343	1.0	1.0
minisat.1.501	1343	442209.0	22.6	97.6	574.2	1321	3.0	0.9

TABLE 4.3. Comparison of various strategies for forgetting constraints

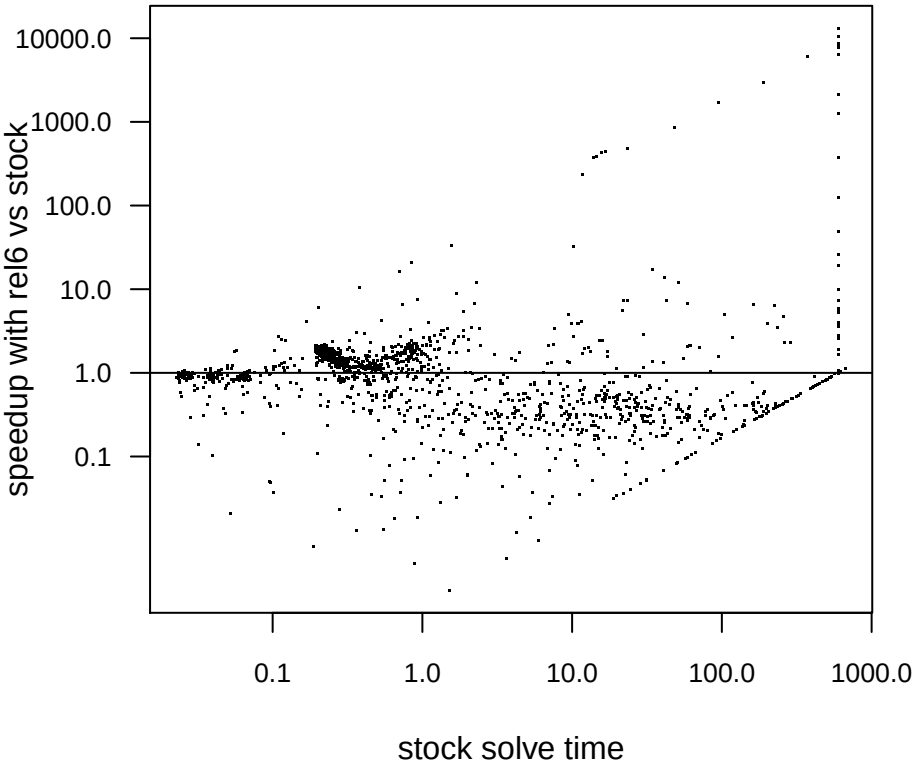


FIGURE 4.7. Graph comparing the best strategy (relevance-bounded $k = 6$) with no learning

paradox is because for these strategies, the instances that timed out were the most improved in terms of nodes and node rate. This makes sense when the instances that run the longest with unbounded learning are the most encumbered by useless clauses.

These results are interesting because contrary to [**KB03**], relevance- and size-bounded learning work well for certain choices of k. However, the results in this chapter were based on a larger set of benchmarks and a larger range of parameters were tried. Also, different implementation decisions in my solver will result in a different time-space trade off. In fact, the best strategy solves 298 more instances than unbounded learning in about 45 hours less runtime. However it still trails stock minion by 26 instances and about 8 hours of runtime. In spite of this, Figure 4.7 gives evidence that learning is still valuable and promising in specific cases. Each point is an instance, with the x-axis the runtime taken by stock minion and the y-axis is stock runtime over relevance.6 runtime; points above the line are speedups and points below are slowdowns. Whilst many instances are slowed down, speedups of up to 5 orders of magnitude are available on some types of problem. Apart from the best strategy,

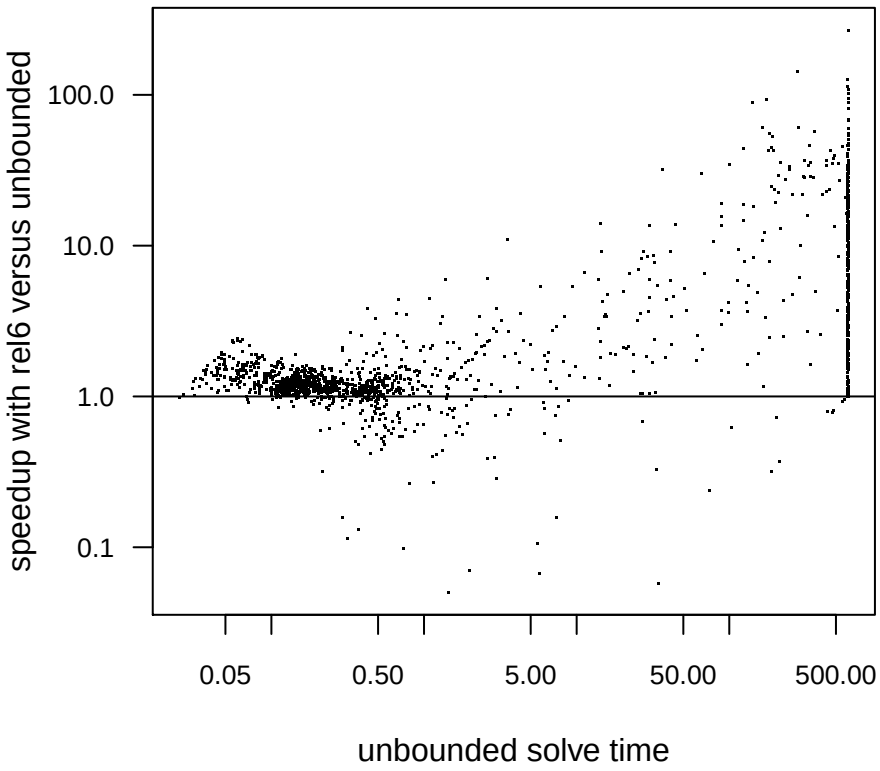
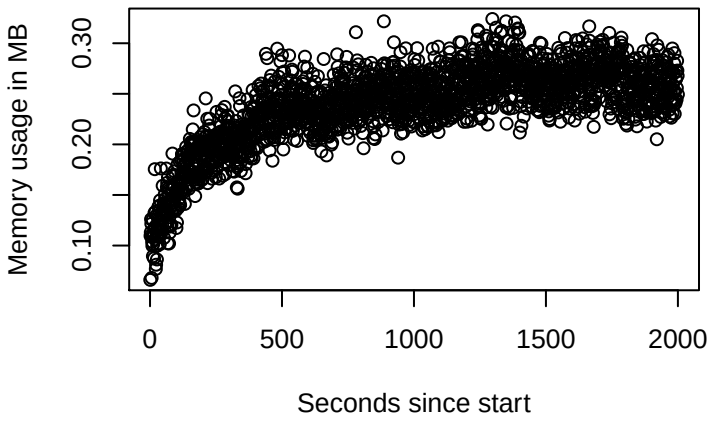


FIGURE 4.8. Graph comparing the best strategy (relevance-bounded $k = 6$) with unbounded learning

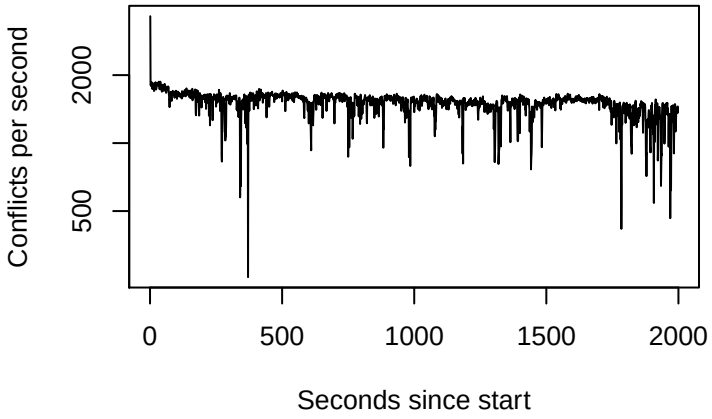
various parameters for relevance-bounded learning perform similarly to $k = 6$, as well as some size-bounded learning parameters. It seems clear that they are significantly better than unbounded learning, but not much different to each other.

The minisat strategy is not effective for any choice of parameters that I tried. However there is reason to believe that a better implementation might improve matters. Strategies 200.X, 400.X and 600.X appear to be promising because the search space increase is modest. Using a profiler, I have discovered that the reason for slowness is the amount of time taken to maintain and process the scores, and to process the constraints periodically. Hence perhaps a better implementation would turn out to perform competitively overall.

Now I will analyse the best forgetting strategy more carefully. Figure 4.8 depicts the speedup on each instance for relevance-bounded $k = 6$ compared to unbounded. It shows that most individual instances are speeded up, sometimes by two orders of magnitude, although a few are slowed down by up to an order of magnitude.



(a) Memory usage each second throughout search



(b) Conflicts per second throughout search

FIGURE 4.9. Analysis of efficiency of forgetting solver over time

In conclusion, whether to use learning remains a modelling decision, where big wins are sometimes available but sometimes it is better turned off.

Postscript Recall that in §4.1 and Figure 4.1 (page 109), I gave a detailed case analysis of how learning affects the memory usage and nodes per second, showing that memory growth is unsustainable and that the node rate drops quite quickly. I have collected the same data for the same instance using the relevance-bounded $k = 6$ strategy to see how it differs. The graphs are significantly different to Figure 4.1. Now over the first 2000 seconds of search, only around 0.3Mb is used and memory usage varies continuously as less relevant constraints are removed. The effect on node rate is also marked: the node rate rarely drops below 500 nodes per second, which is much higher than before when it dropped to around 15-20. Hence memory is conserved for other processes and search proceeds more quickly.

4.5. Conclusions

In this chapter, I have carried out the first detailed empirical study of the effectiveness and costs of individual constraints in a CDCL solver, thus resolving Hypotheses 3 and 4 from Chapter 1:

Hypothesis 3. Nogoods vary significantly in the amount of inference they do.

Hypothesis 4. Weakly propagating nogoods occupy a disproportionate amount of CPU time, relative to their level of propagation.

I found that, typically, a very small minority of constraints contribute most of the propagation added by learning. Furthermore, these best constraints cost only a small fraction of the runtime cost. These results explain why forgetting can work so well. It is obvious that forgetting is a positive necessity due to memory constraints, but this research shows that forgetting is not only necessary but also fortuitously effective because of the disparity in effectiveness between constraints.

Next I resolved Hypothesis 5 from Chapter 1:

Hypothesis 5. There are forgetting strategies that are successful in reducing the time spent solving CSPs of practical interest.

I did this by performing an empirical survey of several simple techniques for forgetting constraints in g-learning (§4.4) and found that they are extremely effective in making the learning solver more robust and efficient, contrary to some previously published evidence.

Chapter 5

c-learning

> Perhaps when a man has special knowledge and special powers like my own, it rather encourages him to seek a complex explanation when a simpler one is at hand.
>
> Sherlock Holmes
> *The Adventure of the Abbey Grange*
> by Arthur Conan-Doyle

5.1. Introduction

One possible criticism of state of the art learning in CSP, as set out in this thesis and elsewhere, is that though CSP derives its strength from powerful global constraints, CSP learning works on a SAT representation.

The idea of this chapter is to investigate how to adapt the g-learning framework to incorporate constraints more general than (dis-)assignments. This is done by means of so-called c-explanations, where c stands for "constraint". Recall that a g-explanation is a set of (dis-)assignments; a c-explananation is a set of arbitrary constraints, but I will define it rigorously later. Now I will give a quick example with various interesting features which I will point out later in this introduction.

Example 5.1. *The* element *constraint is over a vector of variables V, an index variable i and variable e, and ensures that $V[i] = e$. Suppose that e becomes assigned*

to 4 *and* 4 *is removed from dom*($V[7]$). *The propagator should detect now that* $i \nleftarrow 7$. *The best g-explanation for the pruning is just* $\{e \leftarrow 4, V[7] \nleftarrow 4\}$.

However another possible explanation is just $\{e \neq V[7]\}$, *because whenever* e *and* $V[7]$ *are not equal,* $i \nleftarrow 7$.

I will now set out some of the advantages and disadvantages of introducing constraints more general than (dis-)assignments into g-learning. These will be justified theoretically or empirically later in the chapter.

Advantages:
- The c-explanation is at least as concise, e.g. Example 5.1. This might reduce memory usage.
- a c-explanation can capture not only the condition that directly caused a (dis-)assignment, but also capture many other conditions that could have caused the (dis-)assignment. In Example 5.1, the c-explanation covers the situation for any a where $i \nleftarrow a$ because $e \leftarrow a$ and $V[7] \nleftarrow a$ simultaneously, rather than just the particular set of circumstances that led to the disassignment. This is more general and might lead to more powerful nogoods.
- As I will show later in §5.5, it is often easier to work out a good c-explanation, because the vocabulary available is higher level and often the explanation is recursively related to the definition of the constraint that emits it.
- c-explanations can be less dependent on current domain state. E.g., Example 5.1 where value 4 is eliminated from the explanation without weakening it.

Disadvantages:
- Disjunctions of literals (g-nogoods) are faster to propagate than disjunctions of constraints (c-nogoods), so there is a chance that c-nogoods will slow the solver down more if they are ineffective.

5.1.1. Expressivity of c-explanations. In his thesis, Katsirelos said "there may exist an exponential in the arity of C number of nogoods (g-nogoods) to explain the fact that C is disentailed". This shows that a single c-nogood is as expressive as an exponential number of g-nogoods. It is an elementary fact, but suggests that

c-nogoods could be very worthwhile. However, it is important that we are comparing c-nogoods against *minimal* g-nogoods, so that their full power is available. Hence in this section I will prove that a single c-nogood is as expressive as an exponential number of *minimal* g-nogoods.

A strong result can be stated on the relative expressivity of g- and c-explanations. First I must define *prime implicant*:

Definition 5.1. An *implicant* I of a Boolean formula $f(x)$ is an assignment to a subset of the input arguments of f such that the output of f must be 1. A *prime implicant* is a set minimal implicant, i.e. it can't have assignments removed from it and still be an implicant.

Prime implicants are related to minimal g-explanations in a simple way:

Lemma 5.1. A prime implicant of function f is the same as a minimal explanation for $output \leftarrow 1$ in the constraint $output = f(x)$.

Proof. g-explanations must be sufficient for the event they are explaining, and implicants must be sufficient for the output of the circuit to be true. Furthermore, minimal explanations must be setwise minimal, and prime implicants setwise minimal. □

For the parity function, there are at least 2^{n-1} different prime implicants:

Fact 5.1 (given as Proposition 6.1 in [**Weg87**])**.** The odd parity function defined as $f(x) = (\sum_i X_i) \bmod 2$ has 2^{n-1} prime implicants of length n each[1].

Such a set of prime implicants covers each possible input to f whose result is true once and only once, since each one includes an assignment to each input. By the correspondence between prime implicants and g-explanations:

Corollary 5.2. There are 2^{n-1} minimal g-explanations for $output \leftarrow 1$ for constraint $output = \text{parity}(X_1, \ldots, X_n)$.

[1] this is because all prime implicants of parity include assignments to all variables, intuitively because the parity can be changed by flipping a single input

PROOF. By Lemma 5.1 every implicant is a valid g-explanation. By Fact 5.1 there are 2^{n-1} distinct prime implicants and hence there are 2^{n-1} distinct minimal g-explanations for $output \leftarrow 1$, one per assignment to $X_1, \ldots, X_n$. □

However the c-explanation for $output \leftarrow 1$ in constraint $output = \text{parity}(f)$ is just $\text{parity}(f) = 1$, which is an extremely trivial explanation but exactly captures the required property. Hence when a failure is due to odd parity, 2^{n-1} g-nogoods are required to cover all possible reasons whereas a single c-nogood will do the job. Later, in §5.4, I will use Corollary 5.2 to show that entire search trees can be much smaller when c-explanations are used rather than g-explanations. Roughly, this is because with c-nogoods the solver can learn a small powerful constraint like $\text{parity}(f) = 1$ which can cause immediate failure and prove unsatisfiability easily, whereas using g-nogoods it is restricted to enumerating numerous weak constraints until the search space is eventually exhausted.

5.1.2. Preview of chapter. In this chapter, I will first describe similar work in constraints. Next I will give the foundational algorithms needed to implement c-learning in a CSP solver. Having done that I will prove the potential of c-learning, by showing that it is capable of solving a family of instances in polynomial time, where g-learning takes at least exponential time irrespective of the variable and value ordering used. I also show that this translates as expected to a practical improvement in search time using my c-learning solver. Next I will show how to produce c-explanations for the occurrence and all different constraints, and finish with an experiment testing c-learning on an additional problem class.

5.2. Context

The idea of generalising explanations further than g-learning has appeared several times in the constraints literature.

5.2.1. Katsirelos' c-nogoods. Katsirelos [**Kat09**] concludes his thesis by giving a very brief description of various possible techniques that use constraints more general than (dis-)assignments to explain prunings. Katsirelos presents this as the

addition of a Boolean variable v_C representing the new constraint, i.e. $v_C \leftrightarrow C$ is posted. Now v_C can be incorporated into explanations as appropriate[2].

Katsirelos describes how to use c-nogoods only in the context of logical constraints *and* and *or*. For example, consider the constraint $C_1 \vee C_2$ and suppose that C_1 is disentailed. Using delayed disjunction propagation [**HSD98**], the remaining disjunct C_2 will be propagated and suppose it causes $v \nleftarrow a$. A g-explanation for this propagation consists of a g-explanation for the disentailment of C_1 (e.g. §3.5.2.2), plus a g-explanation for $v \nleftarrow a$ by C_2. In the c-nogood, the set of literals explaining the disentailment of C_1 is replaced by the single literal v_{C_1}. No experiments testing this idea have been published [**Kat09**].

Above, in Corollary 5.2 I proved that sometimes an exponential number of *minimal* explanations are needed to show that a constraint C is disentailed, when a single c-explanation will do. In his thesis, Katsirelos said that "there may exist an exponential in the arity of C number of nogoods to explain the fact that C is disentailed". I have proved that this is still the case even when the nogoods are all *minimal*.

Compared to Katsirelos' work, my practical contributions in this chapter have been to show how this general idea can be applied to non-logical constraints, to describe a framework for it to be implemented and to complete an implementation in minion so it can be evaluated empirically. I have also progressed the theoretical understanding of this technique, by proving results about the proof complexity of c-learning versus g-learning.

5.2.2. Lazy clause generation. Lazy clause generation (LCG), which I described in §2.6.6 also generalises g-learning, by allowing nogoods to contain unary constraints like $x \leq a$ as well as (dis-)assignments. This improves the conciseness of explanations, but not their expressiveness. This is simply because if a clause contains $x \leq a$ is false, for some a, it can easily be replaced by $x \leftarrow a \vee x \leftarrow a-1 \vee x \leftarrow a-2 \ldots$. Moreover the resultant constraint propagates at least as well. Hence unlike c-learning, LCG is no more expressive than g-learning.

[2]note the obvious similarity to extended resolution [**Tse68**]

5.2.3. Caching using constraints. Learning based on constraints has been tried with some success in the context of *caching* as opposed to constraint learning [**CdlBS10**]. Caching is when the search space previously searched is stored as a set of keys, if the current part of search matches a previously searched key then the outcome can be read out of the stored cache. To some extent the distinction between learning and caching is quite artifical: learned constraints are propagated along with the other problem constraints, whereas cached keys are not propagated (see also §2.6.5). Caching relies on keys generalising the subtree in which they are found so that they can be used to avoid search elsewhere. In [**CdlBS10**], a "projected key" for each individual constraint is conjoined to form a key for the entire subtree just searched unsuccessfully. For example if the problem contains $c = \text{alldiff}(w, x, y, z)$ s.t. $w, x, y, z \in \{1, 2, 3, 4\}$ and decisions $w = 1$ and $x = 2$ then the projected key for c is $\text{alldiff}(y, z) \wedge y, z \in \{3, 4\}$. This is a key that generalises the subtree from which it is derived, because the constraints in the key are stronger than the problem constraints. The practical results in [**CdlBS10**] show that the technique can beat state of the art CSP solvers (with and without learning) on several problem classes.

5.2.4. Summary. In spite of the approaches described in this section, this chapter contains the first practical contribution towards generalising explanations beyond unary constraints, as well as fundamental algorithms and theoretical contributions towards understanding the potential of the technique.

5.3. Foundational definitions and algorithms

In this section I will introduce the framework more rigorously.

Definitions 5.1 (c-explanation and c-nogood)**.** A *c-explanation* for a solver event e is a constraint *con* that is sufficient for the solver to infer e. A *c-nogood* for (V, D, C) is a set of constraints that cannot all be satisfied in any solution.

Note 5.1. It is equally valid to think of a c-explanation as introducing a new reified constraint *con* and reification variable r such that $r \leftrightarrow con$ and then including variable r in the literals of a g-explanation[3].

[3] in this respect c-learning incorporates features of extended resolution [**Tse68**]

See Example 5.1 for a specimen of a c-explanation.

Clearly c-explanations generalise g-explanations. They can be substituted into the g-learning framework with only a few changes. However it is necessary to generalise the definition of implication graph (IG) to suit c-learning:

Definition 5.2 (c-learning implication graph)**.** An *implication graph* for the current state of variables is a directed acyclic graph where

- each node is a currently true constraintnot *necessarily entailed by any particular propagator*, and
- there is an edge from u to v iff u appears in the explanation for v. □

Recall that g-learning requires the following capability for each node in the IG:

- determine at which depth it became entailed, and
- discover the constraints that are responsible for its entailment.

It is usually relatively easy to determine if constraints are entailed: in the worst case each possible assignment could be enumerated in $O(d^r)$ time where d is the domain size and r the arity, and each can be checked for conformance to the constraint in polynomial time. Usually there is a specialised algorithm for each constraint that is efficient.

Since determining entailment is usually easy, so too is discovering the depth at which it became entailed: simply search for the first depth at which it is entailed. However it is better to use tailored algorithms for each constraint where possible, as I did in §3.5.2.2 for inequality constraints.

Discovering the constraints responsible for some propagation is done using an explanation procedure, as in g-learning. I will describe explanation algorithms for several propagators in §5.5.

Another thing to notice is that the constraint used to explain the event is *not* necessarily an existing constraint in the CSP, in fact it is quite likely not to be. This is crucially important in practice because it means that the IG *cannot* be built eagerly, while propagation is done, because many of the nodes are brand new constraints. Instead the IG must be uncovered lazily starting with the concrete events that cause failures.

Example 5.2. *Following on from Example 5.1. Suppose that at the current point in time* $e \leftarrow 4$ *and* $v[7] \nleftarrow 4$*, but the propagator for element*(V, i, e) *has not yet fired. In Example 5.1, I showed that* $\{e \neq V[7]\}$ *is a valid c-explanation for the propagation* $i \nleftarrow 7$ *that will occur. The constraint* $e \neq V[7]$ *is in fact entailed by the current domain state, but so are many other constraints*[4]*. Hence it is infeasible to build a representation of the IG eagerly, because the solver cannot anticipate what constraints will be introduced. Once the propagation* $i \nleftarrow 7$ *has occurred, the constraint* $e \neq V[7]$ *becomes concrete.*

Conversely, in g-learning, the constraints that can become involved in the IG are known at all times: it's just the set of current assignments and disassignments.

5.3.1. Required properties of c-explanations. c-explanations being used in IGs and processed to find a firstUIP cut using Algorithm 5 must conform to certain properties. Suppose explanation $\{c\}$ labels event e:

Property 5.1. The entailment depth of c may not be greater than the depth of event e.

Remark 5.2. Matches Property 2.1 for g-explanations and ensures causes precede effects, ensuring no cycles in the implication graph.

Property 5.2. Paths in the IG must be finite, i.e. c-explanations must eventually bottom out to (dis-)assignments.

Remark 5.3. This property is not necessary in g-learning, for since the edges always go from nodes with a higher to a lower decision depth, paths must be finite. In c-learning this is not automatically the case, because it would be possible for an infinite path of virtual constraints to occur with the same entailment, e.g. two equivalent constraints that each explain their own entailment using the other. An infinite path might mean a cut cannot be computed by a finite number of resolution steps.

[4] e.g. any constraint satisfied by any remaining assignment to any possible subset of the current variables

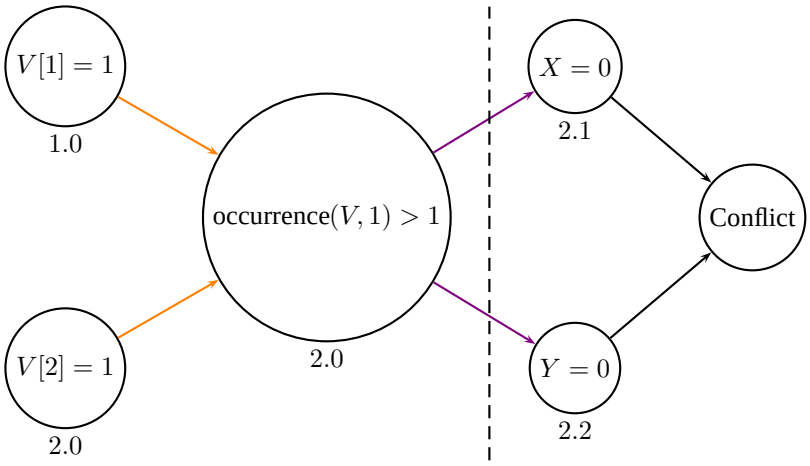


FIGURE 5.1. Implication graph for Example 5.3

5.3.2. Propagating clauses consisting of arbitrary constraints. One of the fundamental ingredients that makes nogood learning work is that the clauses learned are guaranteed to propagate on backtrack, so that progress is always made. Suppose that firstUIP cut $\{c_1, \ldots, c_k\}$ is added as nogood $(\neg c_1 \vee \ldots \vee \neg c_{k-1} \vee \neg c_k)$. By the properties of firstUIP, $c_1, \ldots, c_{k-1}$ are all disentailed when the constraint is posted. Hence c_k will be unit propagated.

My solver uses watched literals to propagate arbitrary disjunctions of constraints (*watched or*) [**JMNP10**][5]. Using watched or, each disjunct constraint must be implemented with a *complete satisfying set generators*, which means that the watched or propagator can detect as soon as it has become disentailed (see [**JMNP10**]). This means that unit propagation can happen as soon as possible.

In the case of g-learning, c_k is guaranteed to propagate, since a (dis-)assignment that is not already true or false can always propagate successfully. However I will now show that this is not the case in c-learning, by exhibiting a counterexample:

Example 5.3. *Consider the CSP consisting of variables* $V[1], V[2], X$ *and* Y *each with domain* $\{0, 1\}$ *and constraints*

- *occurrence*$(V, 1) \leq 1 \leftrightarrow X$,
- *occurrence*$(V, 1) \leq 1 \leftrightarrow Y$, *and*
- $X \vee Y$.

[5]I was involved in this research, but it does not fall within the scope of this thesis

Suppose that $V[1] \leftarrow 1$ *at depth* 1.0*. No propagation is possible by any constraint (this is less obvious for the bi-implications than for* $X \vee Y$*, but it can be verified by inspecting all possible assignments over the scope* $V[1], V[2], X$ *(similarly for* Y*) which consist of*

$$\{(0,0,1),(0,1,1),(1,0,1),(1,1,0)\}$$

for both bi-implications.

Suppose next that $V[2] \leftarrow 1$ *at depth* 2.0*. Now the left hand size of each bi-implication constraint is definitely disentailed, and the bi-implications will propagate* $X \leftarrow 0$ *and* $Y \leftarrow 0$ *respectively. Hence clause* $(X \vee Y)$ *is empty and conflict analysis will follow. The implication graph is shown in Figure 5.1. Clearly occurrence*$(V, 1) > 1$ *is a c-explanation for assignments* $X \leftarrow 0$ *and* $Y \leftarrow 0$*. The firstUIP cut is actually just* $\{$*occurrence*$(V, 1) > 1\}$*. Conflict analysis will therefore backjump to depth 0 (i.e. the root node prior to any decisions being made) and attempt to propagate the constraint occurrence*$(V, 1) \leq 1$*. The occurrence constraint cannot rule out any value and so no propagation will occur, as I set out to show.*

However the firstDecision cut is guaranteed to propagate because the disjunct that is unit will definitely be a (dis-)assignment. Hence the approach taken in c-learning is first to try the firstUIP constraint, monitoring if any propagation occurs, and if not, revoke it and add the firstDecision cut, which is guaranteed to result in some progress. Hopefully this will not be necessary very often, but it is essential for correctness. Note that the benefits of c-learning do not require that any additional propagation occurs immediately after backtrack. In fact, the more general constraints need never propagate at all: they need only be violated more often and cause the clauses to become unit more often than in the case of g-learning.

5.3.3. Common subexpression elimination. Common subexpression elimination is when the same constraint expression posted twice is replaced by a single occurrence of the expression. For example, consider the following example from [**RMGJ09**]. Expression

$$a + x \times y = b \wedge b + x \times y = t$$

might typically be flattened to

$$aux_1 = x \times y \wedge a + aux_1 = b \wedge aux_2 = x \times y \wedge b + aux_2 = t.$$

However when common subexpressions are taken into account,

$$aux_1 = x \times y \wedge a + aux_1 = b \wedge b + aux_1 = t$$

is a smaller and more strongly propagating alternative [**RMGJ09**]. See [**Ren10**] for more information.

For logical constraints like disjunction, there can be an advantage to recognising common disjuncts. The reason for this, is that, in general, there is a difference between a constraint being forced to be satisfied, and being currently entailed. For example, suppose that C is being enforced. Although C is forced to be satisfied in any solution, it is not necessarily entailed, so another constraint $\neg C \vee D$ may not become unit. Hence, disjunction propagation should be implemented to unit propagate when all but one disjunct is *either* entailed or forced to be true. I have implemented this feature in my solver for the special case described in §5.4.1.2.

5.4. Proof complexity and c-learning

I will now prove that c-learning can be significantly superior to g-learning: there is an exponential separation between the two, meaning that there exists an infinite family of instances of increasing size parameter n such that any backtracking search algorithm using g-nogood learning takes at least exponential time in n using any possible search strategy whereas there is a simple algorithm that learns c-nogoods that can solve any such problem in time polynomial in n. First some definitions are required:

Definition 5.3. The constraint $parity(X)$ ensures that $(\sum_i X_i) \equiv 1 \bmod 2$, where X is a Boolean vector. Hence $\neg parity(X)$ is just $(\sum_i X_i) \equiv 0 \bmod 2$. See Fact 5.1 on page 134 for a definition of the parity *function*.

This constraint is interesting for several reasons. The first is that until all but one of the variables is instantiated, a propagator cannot prune any values:

Lemma 5.3. *No propagator for parity(X) can remove any values until $|X|-1$ variables are instantiated.*

PROOF. Let I be the proper subset $I \subset X$ of size k that are instantiated at a particular point in search. Suppose $|X \setminus I| > 2$, i.e. fewer than $|X|-1$ variables are instantiated. Let $x \in X \setminus I$ be arbitrary and let $others = X \setminus I \setminus \{x\}$. Suppose that the sum of I is congruent to 1 (resp. 0) modulo 2. Then $0 \in \mathrm{dom}(x)$ is supported because $others$ can be assigned s.t. $\sum others \equiv 0 \bmod 2$ (resp. $\sum others \equiv 1 \bmod 2$). Also $1 \in \mathrm{dom}(x)$ is supported because $others$ can be assigned s.t. $\sum others \equiv 1 \bmod 2$ (resp. $\sum others \equiv 0 \bmod 2$). Hence 0 and 1 are supported for all uninstantiated variables if $|X \setminus I| > 2$, as required. □

The second required fact is that $parity(X)$ cannot be entailed until all $|X|$ variables are instantiated. This should be obvious from the previous lemma and its proof.

Lemma 5.4. *parity(X) cannot be entailed until $|X|$ variables are instantiated.*

I can now introduce the infinite family of problems of increasing size used to prove the result, parameterised by n:

Definition 5.4. CSP $M(n)$ consists of variable x and vector of variables X of length n, each of which has a $\{0,1\}$ domain, and constraints

$$x \leftarrow 1 \vee parity(X) \tag{1}$$

$$x \leftarrow 1 \vee \neg parity(X) \tag{2}$$

$$x \nleftarrow 1 \vee parity(X) \tag{3}$$

$$x \nleftarrow 1 \vee \neg parity(X) \tag{4}$$

Note that this problem is unsatisfiable. There are various techniques that would make this instance very easy, such as remodelling the problem by reifying $parity(X)$, and I seek to prove that c-learning is one such technique. The proof relies on the fact that it should be possible to discover when $r \leftrightarrow C$ or $r \leftrightarrow \neg C$ has already been introduced by the learning process, and to reuse r in future explanations where

possible. In practice this facility will save memory and also can be used to improve propagation (see §5.3.3). It is also necessary to prove that g-learning will necessarily find $M(n)$ hard no matter how clever it is. First I will prove that c-learning will find it easy to show that there are no solutions to $M(n)$ for any n:

Lemma 5.5. *For any given n, c-learning can prove $M(n)$ unsatisfiable in polynomial time.*

PROOF. Assign the variables in vector X so that $parity(X)$ is entailed, e.g. assignment $1, 0, 0, 0, \ldots$, then disjunctions 2 and 4 can unit propagate to cause $x \leftarrow 1$ and $x \nleftarrow 1$. Hence $\{parity(X)\}$ is the firstUIP cut for this conflict. Constraint $r \leftrightarrow parity(X)$ will be introduced, where r is a fresh variable, and the constraint $r \leftarrow 1$ learned.

Next assign vector X so that $\neg parity(X)$ is entailed, then similarly to the above $\{\neg parity(X)\}$ is the firstUIP cut. The constraint learned is $r \leftarrow 0$, since $r \leftrightarrow parity(X)$ was introduced earlier.

A conflict at the root node is guaranteed because r is forced to be both 0 and 1.

Clearly this can be implemented in polynomial time for any n. □

Finally I will prove that $G(n)$ is necessarily hard for g-learning, even when arbitrary variable and value ordering is allowed:

Lemma 5.6. *For any given n, g-learning takes exponential time to prove $M(n)$ unsatisfiable using any variable ordering.*

PROOF. Suppose that every variable in X is assigned before x. Then w.l.o.g. and by Lemma 5.4, $parity(X)$ (or $\neg parity(X)$) is entailed as soon as the last assignment is made and not before. Hence disjunctions 2 and 4 will propagate to force a conflict in variable x. The conflict analysis process must include every assignment to X, since by Lemma 5.4 all are required to ensure entailment of $parity(X)$ (or $\neg parity(X)$), without which the conflict cannot occur. This nogood rules out only the current assignment.

The case where x is assigned before X is fully assigned is only slightly more complex. By Lemma 5.3, until all but one variable x_u in X is assigned, there is no

chance of any propagation. Suppose w.l.o.g. that $x \leftarrow 1$ ($x \leftarrow 0$) when this happens. Now disjunctions 1 and 2 will unit propagate to force the remaining variable x_u to be both 0 and 1, which is required to satisfy unit implicants $parity(X)$ and $\neg parity(X)$ respectively. Hence a conflict results. The g-nogood must involve $x \leftarrow 1$ without which 1 and 2 cannot unit propagate and the entire assignment to X apart from x_u without which the parity constraints cannot propagate. This rules out only the current assignment.

Since by any possible variable and value ordering, each g-nogood only rules out one partial assignment complete except for one variable, 2^n partial assignments must be tried before the search space is exhausted and hence the algorithm takes exponential time. □

The previous lemmas combine in the obvious way to give:

Theorem 5.7. *There is an exponential separation between g-learning and c-learning.*

Recall that Theorem 5.5 takes advantage of common subexpression detection, it is an open question whether the Theorem can be proved without it. This proof does not allow for restarts during search. There is no reason to believe the result does not hold in the presence of restarts, but I have not proved it rigorously.

5.4.1. Experiments. To see the benefits of smaller search using c-learning in practice, I have implemented the parity constraint and have tried the above problem in my c-learning solver.

5.4.1.1. *Procedure.* I have run $M(n)$ for $n = 1$ to 19. The possibility of fast execution for c-learning is proved by running it according to the variable and value ordering described in Lemma 5.5. In order to demonstrate empirically that g-learning is slow I have run instances up to 19 variables 100 times each using a random variable ordering.

5.4.1.2. *Implementation.* The g-learning solver is the same as that used for the experiments in Chapter 3: basic lazy g-learning with no forgetting, learning the firstUIP cut. The explainer for parity is new to this chapter and uses minimal explanations.

The c-learning solver is based on the same solver, but uses a different explainer for watched OR [**JMNP10**]. Specifically, when a watched OR $C \vee D$ propagates D

n	**c-learn time**	**c-learn nodes**			**g-learn time**	**g-learn nodes**		
	Mean (secs)	Min	Mean	Max	Mean (secs)	Min	Mean	Max
01	0.006	1	1	1	0.006	1	1	1
02	0.006	2	2	2	0.006	3	3	3
03	0.006	3	3	3	0.006	7	7	7
04	0.006	4	4	4	0.007	15	15	15
05	0.006	5	5	5	0.007	31	31	31
06	0.006	6	6	6	0.009	63	63	63
07	0.006	7	7	7	0.013	127	127	127
08	0.007	8	8	8	0.021	255	255	255
09	0.007	9	9	9	0.041	511	511	511
10	0.007	10	10	10	0.093	1023	1023	1023
11	0.007	11	11	11	0.226	2047	2047	2047
12	0.007	12	12	12	0.589	4095	4095	4095
13	0.007	13	13	13	1.723	8191	8191	8191
14	0.007	14	14	14	5.399	16383	16383	16383
15	0.007	15	15	15	28.726	32767	32767	32767
16	0.007	16	16	16	34.970	65535	65535	65535
17	0.007	17	17	17	47.563	131071	131071	131071
18	0.007	18	18	18	117.050	262143	262143	262143
19	0.007	19	19	19	279.564	524287	524287	524287

TABLE 5.1. Comparison of c- and g-learning on parity instances

because C is disentailed, the explanation is $\neg C \cup E$ where E is the explanation for D's propagation. In order to detect when C or $\neg C$ is reintroduced by the learning system, each new constraint is added to a list when it is first posted. If the negative of an existing constraint is posted, search is stopped. This implementation is not very good and not as powerful as common subexpression detection, but does give polynomial performance and the successful experiments to follow show that the implementation suffices for present purposes.

5.4.1.3. *Results.* Table 5.1 demonstrates convincingly that c-learning is much better at $M(n)$ than g-learning. c-learning solves the problem in the same number of nodes as there are variables. g-learning uses $2^n - 1$ nodes as predicted by the proof of Lemma 5.6[6]. It is worth pointing out that no matter what the ordering used, this number does not change, again as predicted by the lemma's proof.

5.5. c-explainers

As with g-learning, much of the effort in implementing c-learning is providing small and correct explanations for each (dis-)assignment caused by a propagator. Please

[6]the proof says 2^n nodes, minion counts $2^n - 1$ because it counts nodes from 0

note that the following c-explanations are not implemented, and hence no experiments are included to compare them with the corresponding g-explanations.

5.5.1. Occurrence. The constraint $occurrence(V, i) \leq count$ ensures that there are at most $count$ occurrences of value i in vector V. In minion, i is a constant but both V and $count$ are variables. The constraint $occurrence(V, i) \geq count$ is also available in minion, however I will only describe how to derive explanations for $occurrence \leq$, since $occurrence \geq$ is symmetric.

The minion propagator for $occurrence \leq$ propagates in the following cases:

- when i is already assigned $\max(\text{dom}(count))$ times, the constraint would be failed if any more were assigned, so i is removed from all the other domains; and
- remove any values from $\text{dom}(count)$ that are smaller than the current number of assignments in V to value i.

Note that both the g- and c-explanations for $occurrence \leq$ described in the following two sections are original to this thesis.

5.5.1.1. *Explanation for* $V[idx] \nleftarrow i$. The c-explanation for this type of propagation is very simple. Suppose that $V[idx] \nleftarrow i$ by the first propagation rule above. It must be that the number of occurrences of i in V *excluding* position idx is already $\max(count)$. Hence the explanation is simply $occurrence(V[1, \ldots, idx-1, idx+1, \ldots, |V|], i) \geq count$.

A minimal g-explanation is the set of $\max(\text{dom}(count))$ assignments of variables in V to value i, unioned with the set of prunings to $count$ above $\max(\text{dom}(count))$.

The c-explanation generalises the g-explanation in a number of ways:

(1) If a different set of assignments makes the total number of i's greater than $\max(\text{dom}(count))$, the explanation will still apply, since it does not specify which variables in V are assigned.
(2) If $\max(\text{dom}(count))$ is smaller or larger elsewhere in search and the number of i's again reaches $\max(\text{dom}(count))$, the explanation will still be valid.

I will now show how many different minimal g-explanations each c-explanation covers. In the following, I assume that the domain of $count$ is entirely non-negative, for

any negative numbers would be pruned out immediately anyway. The c-explanation $\mathit{occurrence}(V[1,\ldots,\mathit{idx}-1,\mathit{idx}+1,\ldots,|V|],i) \geq \mathit{count}$ covers

$$\sum_{j=\min(\mathrm{dom}(\mathit{count}))}^{\max(\mathrm{dom}(\mathit{count}))} \binom{|V|}{j} = 2^{|\mathrm{dom}(\mathit{count})|}$$

because for each possible value for $\max(\mathrm{dom}(\mathit{count}))$, any set of that many assignments of variables in V to value i can be chosen. As shown, this sum is exponential in *count* [**Ros91**].

5.5.1.2. *Explanation for count* $\nleftarrow c$. Suppose that a propagator for *occurrence* $\leq$ has caused $\mathit{count} \nleftarrow c$. The c-explanation is $\mathit{occurrence}(V,i) \geq c+1$. This is because by the second propagation rule above, $c \in \mathrm{dom}(\mathit{count})$ is pruned when the count of i's exceeds c.

A minimal g-explanation is the set of assignments of variables in v to value i.

The c-explanation generalises the g-explanation because it captures any possible set of at least $c+1$ assignments to V.

Each c-explanation captures exactly $\binom{|V|}{c+1}$ g-explanations, that is, all the ways to set $c+1$ variables in V to i.

5.5.2. All different. In §3.5.5, I described how to produce g-explanations for the alldiff constraint. Recall that any disassignment $x \nleftarrow a$ by alldiff is forced because there exists a Hall set S of k variables s.t. $|\bigcup^{s\in S} \mathrm{dom}(s)| = k$, i.e. sets of k variables whose *combined domain* contains k values.

In lazy g-learning, to produce an explanation for $x \nleftarrow a$ the following sets are necessary:

(1) Find the Hall set whose combined domain contains a.

(2) Build an explanation which is the conjunction of all prunings from variables in the Hall set *outside* the combined domain.

In c-learning the first step remains the same, so when the explanation is to be produced lazily either the Hall set is fetched from the record stored earlier, or it is rebuilt by running Tarjan's algorithm.

However in c-learning the second step is slightly different. A possible c-explanation is $|C| = |S| \wedge a \in C$, where $C = \bigcup^{s\in S} \mathrm{dom}(s)$, where S is fixed to its value at

propagation time. Such an explanation is true whenever the combined domain has size k and contains a, by definition, so it will be true many times when the g-explanation is not. This c-explanation is novel in the sense that it has not been used before for direct reasoning, however it follows straightforwardly from Régin's alldiff propagator [**Rég94**].

In order to show how many g-explanations each c-explanation can cover I will now derive an expression for the number of possible g-explanations for a pruning $x \nleftarrow a$ by a Hall set consisting of variables $v_{s_1}, \ldots, v_{s_k}$. A particular g-explanation captures that the combined domain C is such that $|C| = |S|$ and $a \in C$. As described by Katsirelos [**Kat09**] and reproduced in §3.5.5, the minimal explanation is a set of all disassignments of values outside C from variables $v_{s_1}, \ldots, v_{s_k}$. Hence there is a unique g-explanation for each different choice of combined domain C such that $|C| = |S|$ and $a \in C$. It is fairly easy to show that there are $\binom{d-1}{|S|-1}$ such choices, since a is a forced choice and then the remaining $|S| - 1$ values for the Hall set can be chosen arbitrarily from the remaining $d-1$ values in the domains. Hence each c-explanation as described above covers $\binom{d-1}{|S|-1}$ g-explanations.

The c-explanation can be generated lazily in worst case $O(|S|)$ time, when the Hall set is stored at propagation time, or $O(rd)$ time if Tarjan's algorithm must be re-run as described in §3.5.5. Also as described in §3.5.5, it is worst case $O(rd)$ time to produce the g-explanation lazily whichever of the techniques described in that section are used. Hence if the Hall set is stored up-front, there is an asymptotic cost saving associated with building the c-explanation compared to the g-explanation.

It would also be possible to further generalise the c-explanation by removing the dependence on the Hall set S that performed the propagation. Then the c-explanation would be $\exists S \subset V$ s.t. $C = \bigcup^{s \in S} \mathrm{dom}(s)$ and $|C| = |S|$.

5.6. Experiments

Although I have only implemented c-explanations for a few constraints[7], there are enough to solve problems $M(n)$ described above in §5.4 as well as *antichain* problems

[7] watched or, parity, inequality

which are experimented on in this section. However I leave large scale evaluation such as that seen in Chapters 3 and 4 to future work.

Definition 5.5. An **anti-chain** is a set S of multisets where $\forall\{x, y\} \subseteq S.\ x \not\subseteq y \wedge y \not\subseteq x$.

In other words, the $< n, l, d >$ instance of anti-chain finds a set of n multisets with cardinality l drawn from d elements in total, such that no multiset is a subset of another. This is modelled as a CSP[8] using n arrays of variables, denoted $M_1, \ldots, M_n$, each containing l variables with domain $\{0, \ldots, d-1\}$ and the constraints $\forall i \neq j \in \{1, \ldots, n\}.\ \exists k \in \{1, \ldots, l\}.\ M_i[k] < M_j[k]$.

Each variable $M_i[v]$ represents the number of occurrences of value v in multiset i, up to a maximum of $d-1$. Each pair of rows M_i and M_j differ in at least two places: in one position k, $M_i[k] < M_j[k]$ and in another position p, $M_i[p] > M_j[p]$. This ensures that neither multiset contains the other.

The constraint $\exists i.\ M[i] < N[i]$ for arrays M and N is encoded as a watched or as follows:

$$M[0] < N[0] \vee \ldots \vee M[l] < N[l]$$

.

This problem appears quite suitable for evaluating c-learning because the watched or explanation (see §5.2.1) introduces many $<$ constraints into the implication graph. Furthermore, it is relatively easy to detect when a $<$ constraint is entailed or disentailed, so the learned constraints should be relatively efficient to propagate.

5.6.1. Experimental methodology. Each of the antichain instances was executed five times with a 10 minute timeout, over 4 Linux machines each with 2 Intel Xeon cores at 2.4 GHz and 2GB of memory, running kernel version 2.6.18 SMP. Parameters to each run were identical, and the minimum time for each is used in the analysis, in order to approximate the run time in perfect conditions (i.e. with no system noise) as closely as possible. Each instance was run on its own core, each with 1GB of memory. Minion was compiled statically (`-static`) using g++ version 4.4.3 with flag `-O3`.

[8] see [**JMNP10**] for more on this model

The g-learning solver used is as described in Chapter 3, i.e. excluding forgetting. Two different variable orderings are used and reported separately: lexicographical and dom/wdeg. The watched or propagator [**JMNP10**] is used for disjunctions. Recall that watched or is an implementation of delayed disjunction consistency: once all but one disjunct is disentailed, the remaining one is forced to propagate. The g-explanations used instantiate the scheme described in §5.2.1 by combining a g-explanation for the disentailment of all but one inequality (see §3.5.2.2) with an explanation for the propagation done by the remaining constraint. In the c-learning solver, the only difference is that the negative of the constraint itself is used to explain its disentailment.

5.6.2. Results. I will now evaluate whether c-explanations are effective in reducing the search time and nodes for antichain instances.

Table 5.2 shows the time and nodes taken to solve a selection of antichain instances. The instances were chosen to include a range of different search sizes and problem sizes. Results are given for two variable ordering heuristics (lex. ordering and dom/wdeg) and for each I provide

C nodes: c-learning total nodes

C time: c-learning total time

G nodes: g-learning total nodes

G time: g-learning total time

These results show that, for these instances, c-learning is not able to significantly reduce the space searched. Hence, the CPU time is also worse for c-learning, as expected, because the overhead of adding generalised constraints and maintaining the c- implication graph is greater. A speedup would only result due to a large decrease in nodes.

I will now supply some further runtime statistics on both solver types, in Tables 5.3 and 5.4. The former table gives statistics for dom/wdeg variable ordering and the latter for lexicographical ordering. The columns are as follows:

Median clause length: The median number of disjunctions in learned constraints.

%C UIPS: The percentage of the time that a non (dis-)assignment is the UIP.

C%: The median over all constraints of percentage of disjuncts that are not (dis-)assignments.

There is no apparent problem with the results for the latter two statistics. They show that most of the time, the UIP is a constraint rather than a (dis-)assignment, allowing for the possibility of stronger propagation. They also show that the clauses are made up primarily of constraints, allowing for better inference. The clause length statistics are more problematic, because the difference between g- and c-learning lengths is usually relatively small, although one would hope the c-learning constraints would be shorter since they are more expressive.

5.6.3. Discussion. I do not know why c-learning does not work for the antichain instances. I believe that good c-learning constraints should be significantly shorter than g-learning constraints, since they are more expressive. Extrapolating from the parity experiments in §5.4.1, c-learning appears to be powerful when long g-learning constraints can be replaced by short c-learning constraints. The fact that in these experiments, constraint length is similar is a cause for concern. I imagine that a different method for deriving cuts may be useful to achieve this, for example one that minimises cut width.

In conclusion, more needs to be done to see if the promise of the experiments in §5.4.1 extends to problems of practical interest. The fact that the technique does not work well on antichain does not imply that the theoretical results are invalid because many techniques that work well on one family of CSPs are detrimental to others. However to show that it is worthwhile to implement c-learning in other solvers, it is necessary to find more problem classes it is successful on. This may involve carefully picking some practical problems it is expected solve quickly (e.g. those described in [**CdlBS10**]) or carrying out a large scale empirical survey such as those in Chapters 3 and 4 to attempt to identify problems it is successful on.

5.7. Conclusions

In this chapter I have made practical and theoretical contributions to the understanding of c-learning. First I described how to implement this framework in a practical

solver, so that progress is guaranteed, using the new watched or propagator for disjunction [**JMNP10**]. Next, I answered an open question from the "Future work" section of [**Kat09**] in order to show Hypothesis 6 is correct:

Hypothesis 6. Using nogoods composed of arbitrary constraints, as opposed to assignments and disassignments, can significantly reduce the amount of search required to solve some CSP instances.

The proof showed that g-learning requires exponentially more search to solve a family of CSPs compared to c-learning. It used a new approach that does not rely on previous work in SAT, unlike many proofs of this type in the past. To demonstrate the practical interest of this result, I perform an experiment showing c-learning's exponential superiority over g-learning on certain contrived benchmarks. Next I described in considerable detail how to produce c-explanations for a couple of interesting constraints, precisely quantifying the difference in expressivity between g- and c-explanations. Finally I performed a short experiment testing the c-learning framework on another problem class.

Instance	**Lex ordering**				**domoverwdeg**			
	C nodes	C time	G nodes	G time	C nodes	C time	G nodes	G time
<2,2,2>	2	0.21	2	0.21	2	0.21	2	0.21
<6,4,4>	16	0.21	16	0.21	16	0.22	16	0.21
<7,3,3>	832	2.00	809	0.51	637	1.30	686	0.53
<8,3,3>	???	Time out.	14150	22.75	???	Time out.	23817	357.87
<8,3,8>	1506	45.15	1529	2.90	56	0.23	61	0.24
<8,4,5>	346	1.03	350	0.42	327	0.94	297	0.47

TABLE 5.2. Comparison of strategies for solving antichain

Instance	Median clause length (G)	Median clause length (C)	%C UIPS	C%
6-4-4	19.0	10.0	1.00	0.90
7-3-3	14.0	18.0	0.78	0.95
8-3-3	17.0	26.0	0.88	0.94
8-3-8	80.0	28.0	0.31	0.74
8-4-5	59.0	51.0	0.70	0.82

TABLE 5.3. Runtime statistics for antichain instances using wdeg ordering

Instance	Median clause length (G)	Median clause length (C)	%C UIPS	C%
6-4-4	29.0	25.0	0.60	0.76
7-3-3	16.0	24.0	0.85	0.83
8-3-3	20.0	30.0	0.80	0.89
8-3-8	80.0	63.0	0.67	0.68
8-4-5	57.0	52.5	0.72	0.85

TABLE 5.4. Runtime statistics for antichain instances using lex ordering

Chapter 6

Conclusion and future work

The previous three chapters comprise the original contribution of this thesis. The aim of this chapter is to briefly recapitulate the contributions and conclusions of each individual chapter. Then I will critically evaluate the contributions, discussing the successes and failings of the work, what its wider significance is for the field and how it advances the state of knowledge. I finish by suggesting some possible avenues for future research.

6.1. Summary

In this section I will briefly recapitulate the contributions of the thesis, referring back to the hypotheses from Chapter 1 to see if they have been resolved.

6.1.1. Lazy learning. Chapter 3 introduced lazy explanations for CSP solvers. Lazy explanations were defined to be a generalisation of normal eager explanations, where instead of storing the whole explanation at propagation time, whatever data is required to reconstruct the explanation later (using a lazy explanation algorithm) is stored. I then described an implementation framework allowing the use of lazy explanations in a g-nogood learning solver, describing how such explanations can be stored, how conflicts are handled and how to ensure the solver is complete. The next section described how to compute explanations lazily for several commonly used constraints including lexicographical ordering, table constraint and all different. During this section I gave asymptotic time complexities for each lazy explainer, showing that in each case, the asymptotic time complexity is at least as good as the equivalent

eager explanation, with the additional benefit that work may never become necessary. Once these algorithms were implemented in the minion solver, it was possible to answer the two hypotheses from Chapter 1:

Hypothesis 1. In a constraint learning CSP solver solving practical CSPs, most of the explanations stored are never used to build constraints during learning.

Hypothesis 2. The asymptotic time complexity of computing each explanation lazily is no worse than eager computation, or the practical CPU time to compute each lazy explanation for practical CSPs is no worse.

Hypothesis 1 was resolved by means of a comprehensive empirical evaluation, using benchmarks from 29 classes of problem. The number of explanations that were actually used during g-nogood learning CSP search using both eager and lazy learning was counted. The results (summarised in Figure 3.1) showed that, for all instances, using lazy explanation reduces the number of explanations needed, usually at least halving the number needed and sometimes reducing it by a factor of 500.

Also as part of the empirical evaluation, Table 3.3 summarises an experiment comparing time to first solution for g-nogood learning using eager and lazy learning on the same 29 problem classes: the lazy variant has never been known to lose by 10% to the eager variant for an instance that takes over a second to solve, whereas the lazy variant routinely beats the eager variant by well over 10%.

Hypothesis 2 was answered positively in §3.5, where I showed that lazy explainers for common constraints are no worse in terms of asymptotic time complexity than eager explainers. However there is a possibility that lazy explainers will have a larger constant factor than eager explainers so it is not automatic that computation time will be less in all cases. The results in Figure 3.9 show that a handful of instances are slowed down slightly by the use of lazy explanations (though not by more than 10%).

6.1.2. Bounding learning. Chapter 4 contained several experiments analysing learning and forgetting of constraints using g-nogood learning. The first experiment verified the first hypothesis:

Hypothesis 3. Nogoods vary significantly in the amount of inference they do.

Table 4.1 summarises the results, which show that over a large set of instances, the $k\%$ of constraints that do most propagations usually do a lot more than $k\%$ of overall propagation.

The next hypothesis was

Hypothesis 4. Weakly propagating nogoods occupy a disproportionate amount of CPU time, relative to their level of propagation.

Hypothesis 4 was answered positively in the next experiment whose results are summarised in Table 4.2, showing that considering $k\%$ of overall propagation carried out by "best" (highest propagating) constraints usually occupies significantly less than $k\%$ of the total propagation time. The converse of this is that $k\%$ of propagations by the "worst" constraints takes significantly *more* than $k\%$ of the overall time.

The final hypothesis in this section concerns the use of simple forgetting strategies from the literature:

Hypothesis 5. There are forgetting strategies that are successful in reducing the time spent solving CSPs of practical interest.

My experiments in §4.4 show that the best forgetting strategies result in significantly more instances being solved in *less* overall time than when forgetting is not used. Hence the hypothesis is resolved positively.

6.1.3. c-learning. In this section I developed a framework allowing constraints more general than disjunctions of assignments and disassignments to be learned for the first time. I showed the expressivity of a c-explanation can be exponentially better than the best possible g-explanation, and using this result proved that c-learning can be exponentially faster than g-learning, answering the following hypothesis positively:

Hypothesis 6. Using nogoods composed of arbitrary constraints, as opposed to assignments and disassignments, can significantly reduce the amount of search required to solve some CSP instances.

This hypothesis is proved in §5.4, where I proved that there exists an infinite family of instances of increasing size parameter n such that backtracking search using g-nogood learning takes at least exponential time in n using any possible search strategy whereas there is a simple algorithm that learns c-nogoods that can solve any such problem in time polynomial in n.

I also describe how c-learning can be implemented in practice. The implementation is very similar to that of g-learning, but for c-learning it appears to be essential to calculate explanations lazily and additional care is necessary to ensure completeness. c-explainers are also needed, and explanation algorithms for the occurrence and all different constraints are provided in this thesis, as well as a rigorous analysis of the expressivity of the explanations they produce. Experiments on my implementation of these ideas show that large speedups are available, but I was not able to obtain successful results on any instances of practical interest.

6.2. Critical evaluation

The three chapters of this thesis stand alone as contributions, however they are also interconnected. In terms of implementation and benefit to the solver, forgetting is orthogonal to the two others. However lazy explanations and forgetting are connected in the sense that their aim is to reduce the two largest overheads in the basic minion g-learning solver (and probably other learning solvers): generating explanations, and storing and propagating new constraints during search. Lazy explanations and c-learning are connected because it is not clear how c-learning can be implemented at all without lazy explanations, and in fact I developed lazy explanations in order to implement c-learning, but it turned out to be useful in g-learning as well.

It would be fair to say that the aim of this thesis has been to reduce the average time that a g-learning solver spends solving CSPs, and the empirical results show that I have been successful in this for my solver. An obvious question is whether this has been a useful contribution to the wider community who use different solvers and are interested in different CSPs? In the case of bounding and lazy explanations, it is impossible to say exactly what effect they would have in a different solver, where the overheads are different. For example, in a solver that is quicker at generating

explanations or that stores them more efficiently the possible gain from using lazy explanations is less. However this thesis has shown that lazy explanations are guaranteed to at least reduce the number of explanations generated and for many constraints each generation event is asymptotically at least as time-efficient. Lazy explanations have never been known to slow down an instance by much, but can improve speed considerably. Hence the available evidence suggests that using lazy explanations is a "no-brainer" and should be done in all solvers. Throughout, I have been careful to provide statistics that are not dependent on CPU speed or implementation details. In the case of forgetting, a solver that stores and propagates nogoods more efficiently would benefit less from the forgetting strategies I describe. However I have recorded what effect each heuristic has on the search space, showing that the best strategies are objectively good irrespective of implementation.

My aim has been to test out the underlying assumptions of ideas like lazy explanations and forgetting, and to do the difficult and time consuming work of tailoring each one to work with CSP solvers. Hence although some of the ideas in this thesis are related to those in SMT and SAT, this work adds huge value compared to a simple statement that "lazy explanations have been shown to work in SMT" or "forgetting is a ubiquitous technique in SAT solvers", for example. All too often, the fact that a technique works is considered sufficient discussion and the question of *why* is neglected. The danger is that the most obvious possible reason for a technique's effectiveness becomes the *de facto* explanation. However for the techniques I have introduced, I have been careful to give evidence for why they work where possible. Having a correct intuition for why existing technique works well helps researchers to make good decisions during the creative process of designing new algorithms.

The work on c-learning in this thesis provides a foundation for further investigation into learning. Although it has not yet proved to be superior to g-nogood learning in practice, I have given theoretical justification for continued experimentation. It is a practical step towards the aim of exploiting the full power of constraints in the context of learning, which is yet to be achieved satisfactorily and is an unexploited area with considerable potential.

In spite of these advances, my implementation of learning remains a risky strategy which is best turned off for efficiency reasons on certain instances[1]. This could perhaps be mitigated by using more efficient implementation techniques for certain subsystems, e.g. a SAT solver to propagate the learned constraints, in the way that lazy clause generation solvers have done [**OSC09**]. However, it seems to me that there is a place for both learning and non-learning solvers in the CP world, each suited to solving different types of problems, but this thesis has advanced the cause of learning solvers.

6.2.1. Application to other areas. There is a lot of interest in using SMT solvers (see §2.6.7 on page 57) to solve CSPs. To solve CSP instances using an SMT solver it is necessary to implement a "theory of constraints" in order to provide propagation and explanation for constraints like all different, etc. The work on lazy explanations for CSP in this thesis describes exactly how to implement a theory of constraints that produces explanations lazily, as required for the most efficient type of SMT solver, e.g. [**NOT06**]. I expect significant progress in this area over the coming years, based partly on my work.

I will finish by suggesting some future directions for research in this area.

6.3. Future work

6.3.1. Lazy explanations. When describing propagators for table constraint, I used a trie implementation (§3.5.3 on page 88). Tries are comparatively similar to another technique for storing data called an multivalued decision diagram (MDD) [**T.98**]; the main difference is that in MDDs identical subtries are merged into one to save space. Hence, it would be interesting to explore the connection between explanations for table and explanations for MDD propagation.

It would be useful to implement lazy explanations in a more efficient framework, to create an SMT theory of constraints or to integrate a lazy learning solver with a SAT solver for managing the new constraints. The aim is to resolve the question of

[1] see [**GKM+10**] for details of joint work where we were able to use machine learning to create a procedure that "guesses" whether to use lazy learning or stock minion based on the properties of the instance, and as a result solve significantly more instances than stock minion in less time

whether the lazy clause generation approach of posting all explanations as clauses is superior to lazily generating the explanations.

It is also important to integrate lazy explanations into other systems that use explanations, besides g-nogood learning. [**Jus03**] describes a wide range of applications for explanations in CSP. For example: CSP model debuggers use explanations to tell the user why values were ruled out. Lazy explanations are ideal for this because the solver can run at practically full speed until explanations are required, at which point they can be obtained. Explanations are also useful for solving the dynamic CSP, where constraints can be added or retracted from existing CSPs, for they allow the effect of individual constraints to be undone. Finally, as I described in Chapter 1, explanations are ubiquitous in other learning and backjumping algorithms and these algorithms should be reevaluated using lazy explanation techniques.

6.3.2. c-learning. As the proof of the separation between c- and g-learning does not take account of restarts during search, it would be good to extend it to cover this case. If this is possible, then it would be interesting to find if extended resolution [**Tse68**] is sufficient to solve it efficiently. If so, is it possible to find a problem that it easy for c-learning, but hard for g-learning plus extended resolution?

It is also important to extend the number of constraints for which c-explainers are available, so that the solver can be tested for a larger range of problem classes, and hopefully improved as a result.

Chapter A

Auxiliary experiments

A.1. Correlation coefficient between propagations and involvement in conflicts

2050 instances were run to a timeout of at least 600 seconds, using one solver that counts propagations and another that counts each time a constraint was resolved during conflict resolution. The solvers are otherwise identical and hence perform search identically. domoverwdeg variable ordering was used. Next a subset of the instances are chosen: those where both solvers completed search and more than 1000 nodes of search were needed. This is done to ensure that when constraints are compared, the comparison is based on the same number of nodes searched.

When the data is joined, there are 566059 pairs of counts, over 256 total instances. The correlation coefficient is 0.96.

A.2. Memory usage during search

For these experiments, malloc and free[1] were overridden so that they keep a running total of the number of bytes allocated. The total is stored in a global variable. So that the memory being freed can be removed from the total, it was necessary to add the size of the block to the start of it. The exact number of bytes allocated on the heap is not otherwise available from the operating system, by any method that I am aware of.

Each time a conflict occurred, the total memory used and the time on the system clock were printed out and the graphs drawn directly from this data.

[1] the operating system's internal memory allocation and deallocation operations

Bibliography

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.

[BB08] Anbulagan Botea and Adi Botea. Crossword puzzles as a constraint problem. In *CP '08: Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*, pages 550–554, Berlin, Heidelberg, 2008. Springer-Verlag.

[BCR10] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog 2nd edition. Technical Report T2010:07, Swedish Insitute of Computer Science, November 2010.

[Bes06] Christian Bessière. Constraint propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, pages 29–84. Elsevier, 2006.

[BHHW04] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. The tractability of global constraints. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 716–720. Springer, 2004.

[BHLS04] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI 04*, pages 482–486, August 2004.

[BKS03] Paul Beame, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1194–1201, 2003.

[BM10] Milan Banković and Filip Marić. Alldifferent constraint solver in SMT. In *8th International Workshop on Satisfiability Modulo Theories*, 2010. to appear.

[BOP03] Joshua Buresh-Oppenheim and Toniann Pitassi. The complexity of resolution refinements. In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 138, Washington, DC, USA, 2003. IEEE Computer Society.

[BR96] Christian Bessière and Jean-Charles Régin. Mac and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *CP*, pages 61–75, 1996.

[BR97] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI*, pages 398–404, 1997.

[BR01] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *IJCAI*, pages 309–315, 2001.

[BRS10] Daniel Le Berre, Olivier Roussel, and Laurent Simon. SAT competition website. `http://www.satcompetition.org`, Feb 2010.

[BS97] Roberto J. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. pages 203–208. AAAI Press, 1997.

[BSJ10] Eli Ben-Sasson and Jan Johannsen. Lower bounds for width-restricted clause learning on small width formulas. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *LNCS*, pages 16–29, 2010.

[CdlBS10] Geoffrey Chu, Maria Garcia de la Banda, and Peter J. Stuckey. Automatically exploiting subproblem equivalence in constraint programming. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *CPAIOR*, volume 6140 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2010.

[CLRS01a] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition edition, September 2001.

[CLRS01b] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[Coh07] Henri Cohen. Number theory i: Tools and diophantine equations. 2007.

[DB97] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *IJCAI (1)*, pages 412–417, 1997.

[Dec90] Rina Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[ES03] Niklas En and Niklas Srensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[FD94] Daniel Frost and Rina Dechter. Dead-end driven learning. In *AAAI-94*, volume 1, pages 294–300. AAAI Press, 1994.

[FHK+06] Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Propagation algorithms for lexicographic ordering constraints. *Artif. Intell.*, 170(10):803–834, 2006.

[FS09] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In Ian P. Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009.

[Gas09] Bill Gasarch. The 17x17 challenge. worth $289.00. this is not a joke. http://blog.computationalcomplexity.org/2009/11/17x17-challenge-worth-28900-this-is-not.html, November 2009.

[Gin93] Matthew L. Ginsberg. Dynamic backtracking. *JAIR*, 1:25–46, 1993.

[GJM06] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006.

[GJMN07] Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *AAAI*, pages 191–197, 2007.

[GJR04] Etienne Gaudin, Narendra Jussien, and Guillaume Rochart. Implementing explained global constraints. In *Constraint Propagation and Implementation, 1st International Workshop*, 2004.

[GKM+10] Ian P. Gent, Lars Kotthoff, Ian Miguel, Neil C.A. Moore, Peter Nightingale, and Karen Petrie. Learning when to use lazy learning in constraint solving. In Michael Wooldridge, editor, *European Conference on Artificial Intelligence (ECAI)*, 2010.

[GM09] Felix Geller and Ronny Morad. Method for generating an explanation of a CSP solution. United States Patent 7,523,445, April 2009.

[GMM10] I.P. Gent, I. Miguel, and N.C.A. Moore. Lazy explanations for constraint propagators. In *PADL 2010*, number 5937 in LNCS, January 2010.

[GMM11] Ian P. Gent, Ian Miguel, and Neil C.A. Moore. An empirical study of learning and forgetting constraints. In *Proceedings of 18th RCRA International Workshop on "Experimental Evaluation of Algorithms for solving problems with combinatorial explosion"*, 2011.

[GMN08] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *AIJ*, 172(18):1973–2000, 2008.

[GN07] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 155(12):1549 – 1561, 2007. SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing.

[Gor09] Andrew D. Gordon. Talk: Principles and applications of refinement types, August 2009. Presented at Summer School on Advances in Programming Languages.

[GPP06] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In P. Van Beek F. Rossi and T. Walsh, editors, *Handbook of Constraint Programming*, pages 329–376. Elsevier, 2006.

[GS00] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In Werner Horn, editor, *ECAI*, pages 599–603. IOS Press, 2000.

[GSL10] Graeme Gange, Peter J. Stuckey, and Vitaly Lagoon. Fast set bounds propagation using a bdd-sat hybrid. *J. Artif. Intell. Res. (JAIR)*, 38:307–338, 2010.

[HDT92] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, 1992.

[HE79] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proc. of the 6th IJCAI*, pages 356–364, Tokio, Japan, 1979.

[HSD98] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *J. Log. Program.*, 37(1-3):139–164, 1998.

[Hua07] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence*, pages 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[Int00] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.

[JDB00a] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *CP '02: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 249–261, London, UK, 2000. Springer-Verlag.

[JDB00b] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *CP*, number 1894 in LNCS, pages 249–261, September 2000.

[JMNP10] Christopher Jefferson, Neil C.A. Moore, Peter Nightingale, and Karen E. Petrie. Implementing logical connectives in constraint programming. *Artificial Intelligence Journal (AIJ)*, 174:1407–1420, November 2010.

[Joh10] Jan Johannsen. An exponential lower bound for width-restricted clause learning. In Oliver Kullmann, editor, *SAT*, volume 5584 of *LNCS*, pages 128–140, 2010.

[Jun01] Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, Seattle, WA, USA, August 2001.

[Jus03] Narendra Jussien. The versatility of using explanations within constraint programming. Habilitation thesis, University of Nantes, 2003.

[Kat08] George Katsirelos, December 2008. Personal correspondence.

[Kat09] George Katsirelos. *Nogood Processing in CSPs*. PhD thesis, University of Toronto, Jan 2009. `http://hdl.handle.net/1807/16737`.

[KB03] George Katsirelos and Fahiem Bacchus. Unrestricted nogood recording in CSP search. In *CP*, pages 873–877, 2003.

[KB05] George Katsirelos and Fahiem Bacchus. Generalized nogoods in csps. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 390–396. AAAI Press / The MIT Press, 2005.

[KM10] Lars Kotthoff and Neil C.A. Moore. Distributed solving through model splitting. In *3rd Workshop on Techniques for implementing Constraint Programming Systems (TRICS)*, pages 26–34, 2010.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 2008.

[Lec] Christophe Lecoutre. CSPXML benchmark repository. `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html`.

[Lec09] Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms*. Wiley-IEEE Press, 2009.

[LP96] C. Lecoutre and P. Prosser. Maintaining singleton arc consistency. In *Proceedings of 3rd International Workshop on Constraint Propagation And Implementation held with CP'06 (CPAI'06)*, pages 47–61, 1996.

[LSTV07a] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *JSAT*, 1(3-4):147–167, 2007.

[LSTV07b] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Transposition tables for constraint satisfaction. In *AAAI*, pages 243–248, 2007.

[Mac77] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. Reprinted in ¡I¿Readings in Artificial Intelligence¡/I¿, B. L. Webber and N. J. Nilsson (eds.), Tioga Publ. Col., Palo Alto, CA, pp. 69-78, 1981. [This paper was honoured in ¡I¿Artificial Intelligence¡/I¿ 59, 1-2, 1993 as one of the fifty most cited papers in the history of Artificial Intelligence.].

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC 01*, 2001.

[Moo08] Neil C.A. Moore. Learning arbitrary constraints at conflicts. In *Proceedings of the CP Doctoral Programme*, September 2008.

[Moo09] Neil C.A. Moore. Propagating equalities and disequalities. In *Proceedings of the CP Doctoral Programme*, September 2009.

[Moo11] Neil C.A. Moore. C-learning: Further generalised g-nogood learning. In Alan Frisch and Barry O'Sullivan, editors, *Proceedings of the ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, pages 103–119, 2011.

[MP08] Neil Moore and Patrick Prosser. Species trees and the ultrametric constraint. *Journal of Artificial Intelligence Research*, 32:901–938, 2008.

[MSS96] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[NB94] B. Neveu and P. Berlandier. Maintaining arc consistency through constraint retraction. In *Proc. TAI94, IEEE*, pages 426–431. Press, 1994.

[Nie09] Robert Nieuwenhuis. SAT modulo theories: Enhancing SAT with special-purpose algorithms. In *SAT*, page 1, 2009.

[NO05] R. Nieuwenhuis and A. Oliveras. Decision Procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. (Invited Paper). In G. Sutcliffe and A. Voronkov, editors, *12h International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'05*, volume 3835 of *Lecture Notes in Computer Science*, pages 23–46. Springer, 2005.

[NORCR07] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Challenges in satisfiability modulo theories. In *RTA '07: Proceedings of the 18th international conference on Term Rewriting and Applications*, pages 2–18, Berlin, Heidelberg, 2007. Springer-Verlag.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[OS08] Olga Ohrimenko and Peter J. Stuckey. Modelling for lazy clause generation. In *CATS '08: Proceedings of the fourteenth symposium on Computing: the Australasian theory*, pages 27–37, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[OSC07] Olga Ohrimenko, Peter Stuckey, and Michael Codish. Propagation = lazy clause generation. In C. Bessière, editor, *CP*, volume 4741 of *LNCS*, pages 544–558, 2007.

[OSC09] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In Joo Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[PD09] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers with restarts. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP-09)*, pages 654–668, September 2009.

[Pro93a] Patrick Prosser. Domain filtering can degrade intelligent backtracking search. In *13th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1993.

[Pro93b] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence, Volume 9, Number 3*, pages 268–299, 1993.

[Pro95] Patrick Prosser. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report Research Report/95/177, Dept. of Computer Science, University of Strathclyde, 1995.

[RCJ06] Guillaume Richaud, Hadrien Cambazard, and Narendra Jussien. Automata for nogood recording in constraint satisfaction problems. In *In CP06 Workshop on the Integration of SAT and CP techniques*, 2006.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, pages 362–367, 1994.

[Ren10] Andrea Rendl. *Effective Compilation of Constraint Models*. PhD thesis, School of Computer Science, University of St Andrews, 2010.

[RJL03] Guillaume Rochart, Narendra Jussien, and Francois Laburthe. Challenging explanations for global constraints. In *CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03)*, pages 31–43, Kinsale, Ireland, 2003.

[RM03] Igor Razgon and Amnon Meisels. Maintaining dominance consistency. In *CP*, pages 945–949, 2003.

[RMGJ09] Andrea Rendl, Ian Miguel, Ian P. Gent, and Christopher Jefferson. Automatically enhancing constraint model instances during tailoring. In Vadim Bulitko and J. Christopher Beck, editors, *SARA*. AAAI, 2009.

[Ros91] Kenneth H. Rosen. *Discrete mathematics and its applications (2nd ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1991.

[RSST09] Raphael M. Reischuk, Christian Schulte, Peter J. Stuckey, and Guido Tack. Maintaining state in propagation solvers. In Ian Gent, editor, *CP'09*, volume 5732 of *LNCS*. Springer, 2009.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[Rya02] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers. Master's thesis, Simon Fraser University, 2002.

[SB09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *SAT '09: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243, Berlin, Heidelberg, 2009. Springer-Verlag.

[SBB+04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-2004)*, 2004.

[SBF10] Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the minizinc challenge. *Constraints*, 15(3):307–316, 2010.

[SBK05] Tian Sang, Paul Beame, and Henry Kautz. Heuristics for fast exact model counting. In *In Proc. 8th International Conference on Theory and Applications of Satisfiability Testing*, pages 226–240, 2005.

[SC06] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[SF94] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI*, pages 125–129, 1994.

[SFSW09] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Why cumulative decomposition is not as bad as it sounds. In *CP'09: Proceedings of the 15th international conference on Principles and practice of constraint programming*, pages 746–761, Berlin, Heidelberg, 2009. Springer-Verlag.

[SG10] Symmetries and Lazy Clause Generation. Geoffrey chu and maria garcia de la banda and chris mears and peter stuckey. In Pierre Flener and Justin Pearson, editors, *Proceedings of the Tenth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon)*, pages 34–48, September 2010.

[Sub08] Sathia Moorthy Subbarayan. Efficient reasoning for nogoods in constraint solvers with BDDs. In *PADL*, volume 4902 of *LNCS*, pages 53–67, 2008.

[SV94] Thomas Schiex and Gérard Verfaillie. Stubborness: A possible enhancement for backjumping and nogood recording. In *ECAI*, pages 165–172, 1994.

[T.98] KAM T. Multi-valued decision diagrams : Theory and applications. *Multiple-Valued Logic*, 4(1):9–62, 1998.

[Tse68] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[vB06] Peter van Beek. Backtracking search algorithms. In P. Van Beek F. Rossi and T. Walsh, editors, *Handbook of Constraint Programming*, pages 85–134. Elsevier, 2006.

[Vil05] Petr Vilím. Computing explanations for the unary resource constraint. In *CPAIOR*, volume 3524 of *LNCS*, pages 396–409. Springer, 2005.

[Weg87] Ingo Wegener. *The complexity of Boolean functions*. Wiley-Teubner, 1987.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.