Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

Reconciling Copying and Trailing for Constraint Programming Systems

submitted by

Raphael Maurice Reischuk

on October 1, 2008

Supervisor

Prof. Dr. Gert Smolka

Advisor

Dipl.-Inf. Guido Tack

Reviewers

Prof. Dr. Gert Smolka
Prof. Dr.-Ing. Holger Hermanns

Online version.
http://www.ps.uni-sb.de/~raphael

Acknowledgements

This thesis has benefited greatly from the support of many people, some of whom I would sincerely like to thank here.

To begin with, I am deeply grateful to both Guido Tack and Professor Gert Smolka for offering me such an interesting topic of investigation.

Guido, as the advisor for all my thesis work throughout the past year, deserves special recognition for his always highly competent remarks and suggestions and particular praise for his openness and his calm and friendly manner which allowed him to convey everything most graciously. Thank you very much!

Gert Smolka has had a very positive influence on me from the very beginning of my studies. His amiable disposition and motivational strength led me to gain teaching experience and to encounter a variety of interesting work by assisting in several projects. His continued inspiration along the way as well as his generous hospitality by providing me with a comfortable place to work in his group were of immeasurable value.

I would also like to thank Holger Hermanns who has been my mentor during my last three years in the honor's program for bachelor students. Not only was he able to provide useful hints and advice, but he also gave me the chance to develop deeper teaching skills by entrusting me with leading tutor duties.

There are two other people who had a positive influence on my studies, as well. Thanks for your support and the nice conversations we had, Wolfgang Paul and Michael Backes.

Furthermore, I would like to thank my fellow students, Georg Neis and Markus Rabe, for reading my thesis and offering helpful advice based on their own recent experience.

Finally, but first in my heart, my parents are due my deep gratitude for their continued moral and financial support throughout my studies, the former being of much greater importance. The broad education that I was able to enjoy while growing up has proven invaluable.

Abstract

Backtracking search is a well-known problem solving technique, which has proven to be successful in particular for constraint solvers. Traditionally, there are two techniques for backtracking: recomputation with copying and trailing. These approaches are basically dual: upon backtracking, recomputation redoes some steps in the search, while trailing undoes some steps. Both techniques have particular advantages and disadvantages. For instance, while recomputation with copying easily scales to concurrent systems, trailing can be significantly more efficient for specific types of variables, such as Boolean variables, and enables advanced techniques like conflict clause learning.

In this thesis, we investigate how trailing and copying can be combined in a single system, in a best-of-both-worlds approach. We present a preliminary design for incorporating trailing in Gecode, a constraint solver based on recomputation and copying. We discuss which additions and modifications to the current design of Gecode are necessary to support trailing.

The results show that the hybrid recomputation and trailing system considerably outperforms the original recomputation-only system for particular problem classes.

Contents

1 Introduction **1**

1.1 Constraint Programming . . . 1
1.2 Restoration Techniques . . . 2
1.3 Structure of the Thesis . . . 3
1.4 Contributions . . . 4

2 Constraint Programming **5**

2.1 Constraint Programming . . . 5
2.1.1 Search . . . 6
2.1.2 Propagation . . . 6
2.2 Further Reading . . . 8

3 Restoration Techniques **9**

3.1 Copying with Recomputation . . . 10
3.1.1 Full Copying . . . 10
3.1.2 Full Recomputation . . . 11
3.1.3 Copying with Recomputation . . . 11
3.1.4 Batch Recomputation . . . 13
3.2 Trailing . . . 14
3.3 Comparing Copying and Trailing . . . 15

4 Reconciling Copying and Trailing **17**

4.1 Information in Copying Systems . . . 18
4.2 Information in Trailing Systems . . . 19
4.3 Reconciling Copying and Trailing . . . 20
4.4 The Restoration Process . . . 25
4.4.1 Full Copying . . . 26
4.4.2 Full Recomputation . . . 26
4.4.3 Fixed Recomputation . . . 28
4.4.4 Adaptive Recomputation . . . 28
4.4.5 Last Alternative Optimization . . . 30

4.4.6 Conclusion 30

5 Constraint Propagation for SAT Problems **33**
5.1 Watched Literals 33
5.2 Conflict Clause Learning 37
5.2.1 Implication Graphs 37
5.2.2 Finding the 1-UIP 39
5.2.3 Non-chronological Backjumping 42
5.2.4 Setting the Watches for Learnt Clauses 43
5.2.5 Advantages of Shared Propagators 43
5.2.6 Interaction with Non-Clause Propagators 44
5.2.7 Interaction with Integer Variables 44
5.2.8 Conflict Clause Learning vs. Systematic Search 45

6 Implementation and Evaluation **47**
6.1 The Gecode Library 47
6.2 Modules 48
6.2.1 SharedController 48
6.2.2 VarImp 50
6.2.3 GlobalVarInfo 50
6.2.4 SharedPropagator 51
6.3 Experimental Results 51
6.3.1 Overhead for Non-trailed Variables 51
6.3.2 Overhead with Copied Propagators 53
6.3.3 Gain with Shared Propagators 54
6.3.4 Gain with Clause Learning 54
6.4 Summary 55

7 Conclusions **57**

Chapter 1

Introduction

This chapter gives an introduction to constraint programming, briefly characterizes restoration techniques, and outlines the structure of this thesis.

1.1 Constraint Programming

Evolved from logic programming, *constraint programming* (CP) is a programming paradigm to address combinatorial and optimization problems, so-called *constraint satisfaction problems* (CSP). A CSP p consists of relations over a finite set of variables. These relations are expressed in terms of *constraints*. p formalizes the typically $\mathcal{NP}$-complete problem of finding an assignment for each variable such that all constraints of p are satisfied.

Example 1.1 (Sudoku) The well-known constraint satisfaction problem Sudoku, of which a precursor was first mentioned by Leonhard Euler (1707 - 1783) as *quadratis magicis* [10], became popular only around 1986 in Japan. The task is to fill a quadratic grid of size 9×9 where the constraints are the following:

- Each 3×3 sub-grid contains each digit from 1 to 9 exactly once.
- Each row contains each digit from 1 to 9 exactly once.
- Each column contains each digit from 1 to 9 exactly once.

The number of possibilities of filling a blank 9×9 Sudoku grid is greater than $6.67 \cdot 10^{21}$ [11]. The minimum number of given fields that yield a unique solution is unknown, an open problem in mathematics. The smallest known number is 17.

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

There are many ways of solving this CSP. If you do it by hand take a pencil rather than a pen. You might need to correct some of your guesses. $*$

The *variables* of a CSP have an initial domain that is truncated during the process of solving the CSP. In this thesis we focus on finite domain constraint programming, i.e., the initial domains of all variables are finite. Depending on the programmer's model, the initial domain of the 81 variables in Example 1.1 could be $\{1, \ldots, 9\}$. During solving the CSP each domain will be truncated to exactly one value satisfying all constraints, a *solution* of the CSP.

A CSP *solver* finds solution(s) of the CSP or shows that no solution exists. Since the power of constraint propagation does not always suffice to determine a solution directly, a solver has to guess some steps towards a solution, i.e. it has to explore a search space. Since these choices might be wrong a solver needs some backtracking strategies in order to restore previous variable domains. In the following we present three well-established restoration techniques.

1.2 Restoration Techniques

There are three main strategies for undoing wrong choices in the exploration of a search space. Let s_i be a state in which the solver has to guess the next step since propagation has computed a fixed point. Assume that after x constraint propagation steps a failure in state s_{i+x} is detected.

[Copying] A simple backtracking technique is *copying*: a copy of state s_i is stored in memory before making a choice. After detecting a failure in state s_{i+x} the previous state s_i is reconstructed by just replacing the current state s_{i+x} by s_i from memory. Copying is costly in runtime and memory consumption.

[Recomputation] After detecting a failure, the previous state s_i is recomputed from the initial state s_0. For this purpose, a description for each choice made between the states s_0 and s_i is stored. This top-down method can be done quite efficiently (based on fixed points and *batch recomputation*, as discussed in Section 3.1.4).

[Trailing] The solver remembers an undo action for each choice and each propagation step and restores the state s_i afterwards in bottom-up direction. The modifications of the variable domains can be stored at a global place, more precisely on a *trail*, a stack-like data structure.

A more detailed explanation and comparison of these techniques is given in Chapter 3.

1.3 Structure of the Thesis

This thesis presents an architecture that combines the three restoration techniques mentioned above. Each technique has certain advantages for particular problem types that a hybrid solver can benefit from. We explain the interaction between variables and propagators in the restoration phase for such a hybrid solver.

- Chapter 2 gives an introduction to constraint programming along with the underlying theory.
- In Chapter 3, we explain the restoration techniques mentioned in Section 1.2 and discuss some fundamental differences between copying with recomputation and trailing.
- Chapter 4 contains the main part of the thesis. We present an architecture and the restoration techniques for a hybrid system benefiting from the advantages of copying and trailing systems.
- Chapter 5 recapitulates two powerful methods for solving SAT problems. These methods can be implemented efficiently in a hybrid solver using the architecture presented in Chapter 4.
- Chapter 6 presents a concrete implementation of a hybrid solver with trailed Boolean variables and other copied variable types. Furthermore, we present some experimental results comparing our prototype to the copying-based solver Gecode [25] and to MiniSat [9], a trailing-based solver.

- The thesis is concluded by Chapter 7 in which we formulate open questions and mention directions of possible future research.

Terms occurring for the first time are highlighted either in **boldface** or *italic*. Terms closely related to the focus of this thesis appear in boldface, terms of common usage are expressed in italics without further explanation.

1.4 Contributions

Copying and trailing constraint solvers have existed for a long time but only in their pure versions. This thesis presents an architecture and a corresponding implementation to reconcile both systems. The interaction between different restoration techniques is investigated in detail.

Our hybrid solver benefits from the particular features of trailing based SAT solvers (conflict clause learning, non-chronological backjumping) as well as from the efficiency of copying with recomputation for integer and set variables.

The experimental results for our prototype at the end of Chapter 6 demonstrate that we are able to solve a certain class of constraint problems with an enormous speedup compared to the pure solvers. Furthermore, they suggest the increase of efficiency for hybrid constraint problems consisting of an integer part and a Boolean part.

Chapter 2

Constraint Programming

The task of reconciling the restoration techniques *copying* and *trailing* requires knowledge concerning the representation of variables, about propagator scheduling and some more insights in the field of constraint programming. This chapter comprehends an introduction to constraint programming.

2.1 Constraint Programming

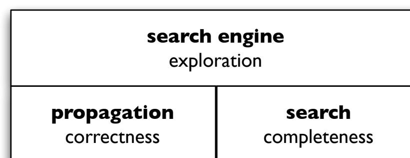
Constraint Programming is a technique to address combinatorial and optimization problems that are modelled as *constraint satisfaction problems* (CSPs). A **constraint satisfaction problem** consists of *variables* with an initial domain and *constraints* on these variables. A **constraint solver** finds solutions of a CSP or shows that no solution exists.

A **variable** v is initialized with a domain $\mathsf{Dom}\ v$ of possible values. A *Boolean variable* v_B has the initial domain $\mathsf{Dom}\ v_B \subseteq \mathbb{B} = \{0, 1\}$, an *integer variable* v_Z has the domain $\mathsf{Dom}\ v_Z \subseteq \mathbb{Z}$, and a *set variable* $v_{\mathcal{S}}$ has the initial domain $\mathsf{Dom}\ v_{\mathcal{S}} \subseteq \wp(\mathcal{S})$, where $\wp(\mathcal{S})$ denotes the power set of some set $\mathcal{S}$. In the scope of this thesis we are only concerned with finite domain constraint programming, i.e. the initial domains are finite. During the process of finding a solution, the initial domains are narrowed. A variable v is **assigned** if $|\mathsf{Dom}\ v| = 1$.

In a **solution** of a CSP each variable is assigned and all constraints are fulfilled. In a **failure** there is a variable u without support for any value, i.e. $|\mathsf{Dom}\ u| = 0$. Finding valid assignments for each variable is the task of a **search engine** which consists of two parts:

- **propagation** excludes values that cannot be part of any solution (correctness). These inference steps are implemented by *propagators* according to the constraints of a CSP.

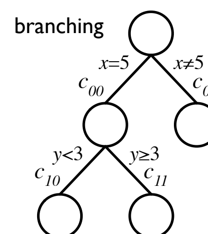
- **search** on the other hand enumerates all possible assignments (completeness). Whenever propagation has computed a fixed point, search splits variable domains and propagation is executed again.



The search engine ensures a solver is complete since all possible assignments are enumerated by search. The correctness of a solver is guaranteed by propagation: values in the domain of a variable that cannot be part of a solution are discarded. Furthermore, all assignments are checked whether they are valid. If an assignment is not valid, propagation reports failure.

The search engine explores a *search tree*. A vertex in such a tree represents a fixed point computed by propagation. The edges denote the alternatives done by search.

2.1.1 Search



An all-solution solver must guarantee that the complete search tree is traversed. The alternatives of search are exhaustive, i.e. the union over all leaves of the search tree contain all solutions. In general, a **branching** selects a variable x and splits its domain into two non-empty parts. Such a modification is called **decision**. Each decision D for a level L corresponds to a constraint c_{LD} that is added in the corresponding subtree. In a binary search tree with two alternatives the branching can for instance set $x = 5$ on the left branch and $x \neq 5$ on the right.

2.1.2 Propagation

Since splitting the domains does not exclude any values, search considers all possible assignments. Hence it is obviously correct, but requires an exponential number of steps. The more efficient and usually used procedure in constraint solvers is the use of propagation. Each constraint c of the CSP is implemented by a *propagator* p_c. A **propagator** p_c from the set of all propagators P modifies variable domains by excluding values that cannot be part of any solution because of the constraint c. In case all variables are assigned, p_c must check if the assignment satisfies the constraint c.

p_c has access to its dependent variables $vars(p_c)$ in order to modify and query their domains. Correspondingly, p_c is **subscribed** to each variable $v \in vars(p_c)$. Whenever $\mathsf{Dom}\ v$ changes, all subscribed propagators are activated which we call

scheduling. Relations between variables and subscribed propagators are called **dependencies**.

Variable domains together with propagators, dependencies and branchings determine the state t of a solver. They are collected in a **space** s_t. The propagators are organized in queues and executed one by one until none of them can contribute further changes. Such a state is a **fixed point**. We will call it **stable** in the following. The filled boxes in Figure 2.1 represent the stable states, the only sinks in the graph.

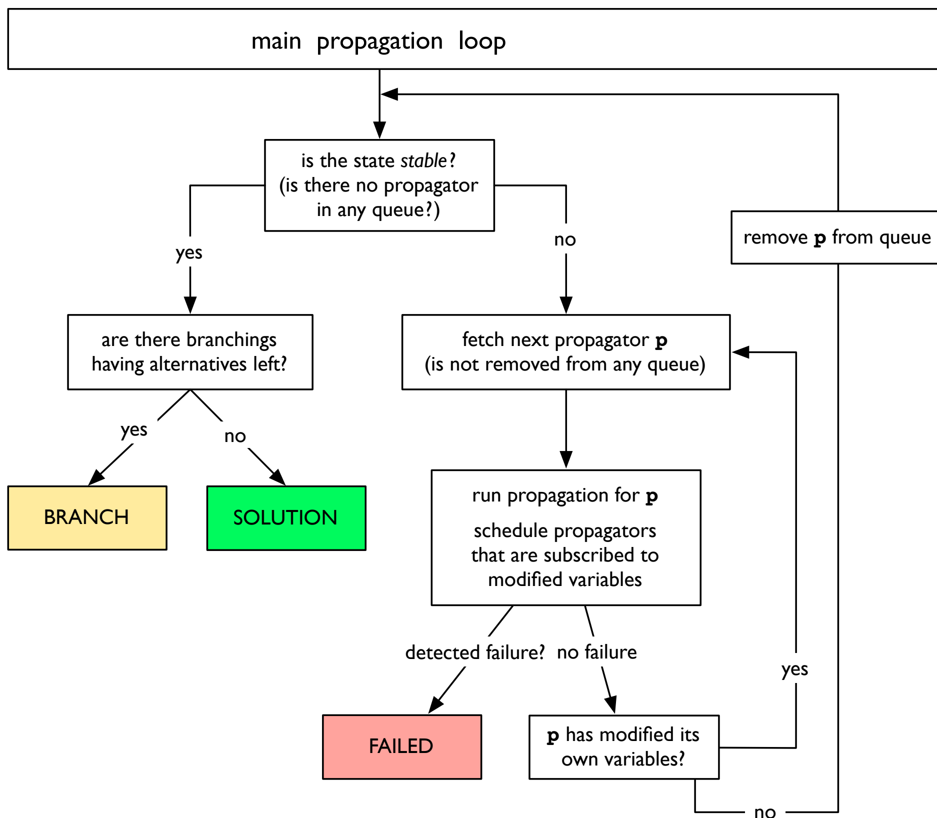


Figure 2.1: Main propagation loop

As a space represents a state, we extend the meaning of stable for spaces in the straightforward way.

A stable state has exactly one of three properties: SOLUTION, FAILURE or BRANCH. The first two are defined as for a CSP. **BRANCH** indicates that at least one variable of the state could not not be assigned by propagation. This property plays a role in CSPs in which propagation alone is insufficient in order to find a solution. In most cases, inference is not powerful enough, the defined constraints

do not contain enough knowledge about the problem; think of Sudoku boards with exactly one solution that you cannot solve without guessing.

2.2 Further Reading

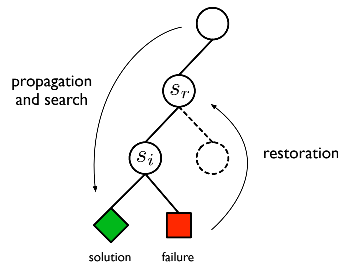
The *Handbook of Constraint Programming* (2006) edited by Francesca Rossi, Peter van Beek and Toby Walsh gives a comprehensive overview into the field of constraint programming [21]. The book contains 26 chapters written by 45 leading researchers in the constraint programming area.

Krzysztof R. Apt wrote a textbook titled *Principles of Constraint Programming* in which he provides many examples that illustrate the usefulness and versatility of the constraint programming approach [2]. The book contains exercises as well as extensive historical and bibliographic notes.

Chapter 3

Restoration Techniques

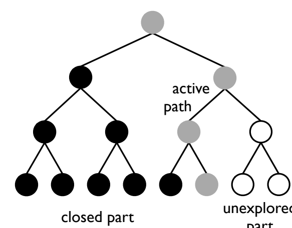
This chapter introduces different restoration techniques and discusses their advantages and disadvantages — the first step in order to reconcile the techniques. Restoration is a fundamental element in constraint solvers, the counterpart to propagation and search. The two approaches work in opposite directions: while a search tree is explored *downwards* by propagation and search, it is restored *upwards* when the search engine arrives at a leaf (solution or failure).



The above figure depicts a binary search tree illustrating a scenario in which propagation and depth-first search have found a solution (diamond). Each node represents a stable state. Since all alternatives of state s_i are explored, restoration has to undo the failure by jumping over state s_i up to state s_r, from where propagation and search can continue.

During exploration, a search tree is dynamically constructed and destructed since it is not possible to have access to all nodes at every time: storing all nodes in memory is not feasible. The set of nodes of the search tree is divided into three disjoint subsets: the set of **live nodes**, nodes that must be available for exploration, the set of **closed nodes**, nodes for which exploration is done, and the set of **unexplored nodes**, nodes that have not been touched by the search engine.

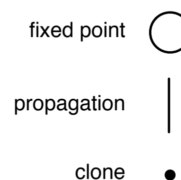
In a depth-first search engine for example, the live nodes are on a single path, the **active path**. All nodes explored before the nodes on the active path, are closed nodes. All nodes to the right of the active path are unexplored. Each unexplored node has a live node as ancestor.



A search engine has exactly one state, captured in the space the search engine currently operates on. The operation of switching between live nodes is called **restoration**, as the search engine's state is restored to a previous one.

There are two main restoration techniques, *copying with recomputation* and *trailing*. The two approaches are basically dual: recomputation *redoes* some steps in the search, while trailing *undoes* some steps. After presenting both techniques we mention some interesting extensions and recapitulate the structural differences between these techniques.

Notation Nodes in the following search trees represent stable states (Section 2.1.2). Hence, the properties *live*, *closed* and *unexplored* will also be used for states. The edges between two nodes represent decisions and subsequent propagation steps. If during exploration or recomputation a stable state is stored in memory, a so-called **clone** of a space, the corresponding node is marked with a small black circle.

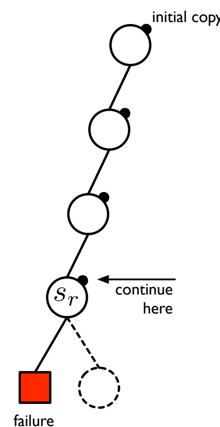


3.1 Copying with Recomputation

Copying with recomputation is a synthesis of two techniques that we explain separately in their pristine versions.

3.1.1 Full Copying

A simple restoration technique is *full copying*. A clone of a space for each live state is stored in memory before search modifies the state. After a failure, restoration can simply pick the right copy from memory and exploration of the search space can continue.



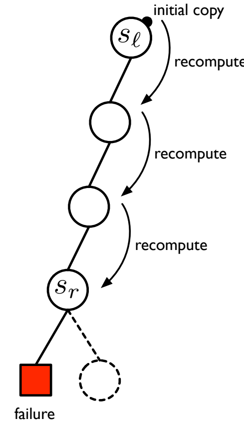
This technique is costly in memory *and* runtime: storing each state can be quite memory-expensive in case many variables and propagators are involved. Schulte showed that on the other hand, memory allocation and

deallocation for spaces with many variables can perform worse concerning runtime than recomputing spaces [22].

An advantage of full copying is the ability to explore a search tree concurrently, in parallel or in breadth-first order. Every live node is directly available and can be used to start exploration of corresponding subtrees.

3.1.2 Full Recomputation

Instead of having a space for each node stored in memory, a state can be recomputed on demand. Beginning at an initial clone s_ℓ in the root node, the previous computations to a demanded state s_r are redone: choose the right alternative and compute the fixed point.



This technique decreases memory requirements compared to full copying. Besides a path to state s_r only the initial copy has to be stored. Hence, full recomputation trades space for time. In particular, its space requisition for clones is independent of the search tree depth.

3.1.3 Copying with Recomputation

Thinking of deep search trees that are typical for solving large constraint problems, full copying clearly requires too much memory and full recomputation will last too long.[1]

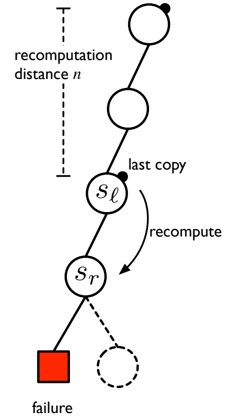
A combination of both has turned out to work quite well: *copying with recomputation* switches between copying and recomputation by copying certain nodes during exploration. Recomputation can thus start from the last copy on the path to the root node. The first constraint system using copying with recomputation was the Mozart programming system in 1991 [26]. Interestingly, pure copying was first used in the DPLL algorithm, already in 1962 [7]. Copying is implemented using techniques first developed for *garbage collection*, a technique presented by Cheney in 1970 [5] (see Jones and Lins [15] for further explanation).

In a synthesis of copying and recomputation it is important to find the best ratio between both techniques. Copying is pessimistic assuming that each choice

[1] A binary search tree of height n with 2^n leaves needs n recomputation steps for each leaf, this gives $n \cdot 2^n$ recomputation steps for the whole tree to be explored. The number of edges in the tree is $2^{n+1} - 2$ which corresponds to the number of computation steps for exploring the search tree with full copying, about 2^n computation steps less than full recomputation needs. This is a factor of roughly $n/2$.

is likely to be wrong, while recomputation is optimistic assuming each decision is right. The following sections show which aspects matter in an efficient synthesis of both techniques.

Fixed Recomputation



If the distance between two copies in the search tree is a fixed number n, the strategy is called *fixed recomputation* with recomputation distance n. The figure to the right shows a scenario for $n = 2$. The case of $n = 1$ is equivalent to full copying, and for a search tree of height h, $n > h$ corresponds to full recomputation.

Recomputation will thus find a copy in every n-th level in the search tree. This technique decreases the number of stored copies by a factor of n. Schulte's experiments [22] suggests that a value for n around 10 performs well in practice.

The overhead in the number of exploration steps for fixed recomputation compared to full copying is $\frac{n2^{n-1}}{2^n-1}$ [23]. For $n = 2$ this is 1.3, and approximately $\frac{n}{2}$ for $n \geq 5$.

Adaptive Recomputation

While fixed recomputation can save time compared to copying, a wrong choice of the (fixed) recomputation distance can annihilate this gain. In order to overcome this problem, the computation distance is computed *adaptively*: during recomputation of a node j from a node i, an additional copy is created at the middle of the path from i to j.

It is important to understand that the shape of the search tree depends inter alia on the constraint problem. In particular, this can lead to an irregular distribution of exploration and recomputation steps within a certain region of the search tree: if exploration exhibits a failed node it is quite likely that not only a single node is failed; often the entire subtree is failed. Moreover, it is unlikely that only the last choice in exploration was wrong. This suggests that a failed node should cause further exploration to be more pessimistic by placing more copies.

As an adaptive distance causes recomputation to place more copies on the path, the recomputation distance decreases which performs better concerning runtime. Concerning memory however, adaptive recomputation cannot guarantee to outperform copying.

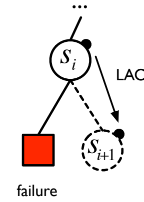
Schulte showed that the obtained speedup is almost independent of the initial

choice of the recomputation distance [23]. This implies that adaptability is the most distinguished feature of adaptive recomputation, in particular, if there is no knowledge about the problem to be solved.

If however there is some knowledge and the computation distance is carefully chosen by hand, Schulte demonstrated that adaptive recomputation performs almost as well as fixed recomputation does.

Last Alternative Optimization (LAO)

If all but one alternative of a live state s_i have been explored, the exploration steps of the remaining alternative will always perform the recomputation step from s_i to s_{i+1} first. This happens for all alternatives of s_{i+1}. Recomputation can thus perform this step exactly once and take node s_{i+1} as last live node. The copy of s_i is modified to represent the state s_{i+1}.

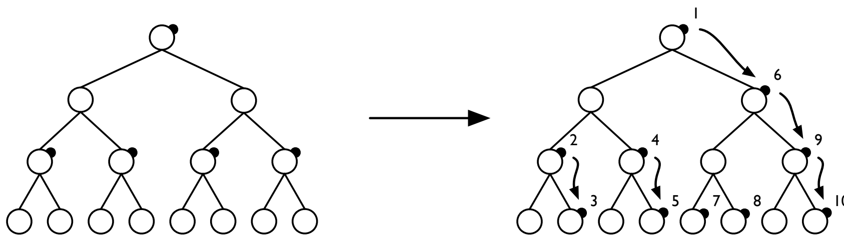


For example, the right alternative in depth-first exploration of a binary search tree can be taken for further exploration without leaving a copy of state s_i.

In particular, this optimization is interesting when the exploration of state s_{i+1} ends up in searching a subtree with root node s_{i+1} instead of exploring a single node only.

Using LAO in full recomputation saves approximately 2^h exploration steps for a binary search tree of height h [23]. The rightmost path in the tree has $h+1$ nodes and thus requires h exploration steps. A left subtree of a node at height i on this path requires $i2^{i-1}$ exploration steps. Thus a tree of of height h requires $h + \sum_{i=0}^{h} i2^{i-1} = 1 + h + (h-1)2^h$ exploration steps. Without LAO we would have h steps for each leaf, thus $2^h \cdot h$ steps in total.

Using LAO in fixed recomputation with recomputation distance $n = 2$ with binary search trees transforms the left tree into the right one:



3.1.4 Batch Recomputation

Upon recomputing a path in the search tree, a fixed point is computed for each decision. Recall that each decision adds a constraint to the corresponding subtree (Section 2.1.1). To determine the variable v that is modified by the decision a

fixed point has to be computed. Storing a reference of v and the corresponding modification enables recomputation to avoid computing all the intermediate fixed points.

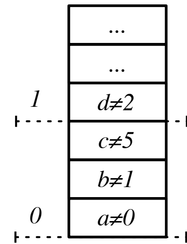
Since recomputation will never lead to a failure, it can be improved by accumulating the added decision constraints along the recomputation path and invoking propagation for computing the fixed point only once. This technique is called *batch recomputation*, presented by Choi et al. in 2001 [6].

3.2 Trailing

Trailing is a restoration technique, evolved from logic programming, that most constraint solvers, in particular SAT solvers use in order to restore previous states. Trailing first appeared, invented for Prolog, in 1983 described by Warren [27] (see Ait-Kaci [1] for a more accessible presentation).

The idea behind trailing is to record changes between two nodes such that they can be undone later. In contrast to copying, trailing is optimistic assuming that only few choices are wrong.

The term 'trailing' is traced back to the fact that common implementations of the trailing technique use a *trail* on which the modifications are stored. The trail is a stack with separators for each decision level. Modifications between two decision levels are listed between the corresponding separators.



Untrailing for the example to the right means adding value 2 to the domain of variable d. For level 0 the values $5, 1, 0$ have to be added to the domains of c, b and a, respectively.

Usability and Costs

An advantage of trailing is founded in its memory consumption. Instead of storing complete states of the solver in memory, only modifications between states are recorded. This clearly needs less memory. If however there are many modifications between two nodes, restoration via trailing can perform worse than copying concerning runtime. Concerning memory, if the entire state is changed, trailing might require more memory than copying.

Furthermore, not only a great number of modifications is problematic. If, for example, propagation removes one element from a hash table, the modification can easily be stored. But restoration might have to reorganize the hash table. The switch between successor nodes thus can be very costly; the switch via copying

in contrast is cheap.

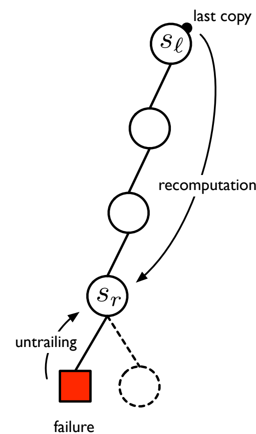
There is a disadvantage in the fact that only one node is immediately available for exploration. Therefore, parallel and concurrent search is not directly possible in trailing systems. Switching between different nodes is expensive as it may require several restoration and recomputation steps [19].

The optimistic assumption that little will change is obviously not optimal for constraint problems with strong propagation. Large differences between two nodes are hard to handle. Think of set variables, in which almost all of the elements are removed from the domain by propagation between two decision levels. These modifications require a huge representation on the trail.

However, modifications of Boolean variables can be stored easily on a trail: first, the Boolean domain $\mathbb{B}$ is small, meaning that modifications are small as well. Second, each Boolean variable is modified at most once on a path from the root node to a leaf. Moreover, these single modifications are always assignments of the variables. Restoration thus means resetting the variable domain. Therefore, it suffices to store only references to the modified variables.

3.3 Comparing Copying and Trailing

Since trailing stores changes only, a structural difference to copying with recomputation is the different direction of restoration. Conceptually, the search engine of a trailing-based solver moves the current state upwards during restoration by undoing the changes. The current state is never discarded. The search engine of a copying-based solver on the other hand discards the current state, duplicates the last clone, sets the duplicate to be the current state and performs recomputation downwards. As copying is based on garbage collection, it benefits from obtaining smaller spaces after the copying operation. The implementations of copying-based solvers use this technique for example on updating the dependencies between propagators and variables.



The main difference in expressiveness lies in the number of nodes that are simultaneously available for exploration. Using copying, every live state can be made directly available over time by storing a clone in memory; using trailing, only one state can be explored at a time.

Since creating full copies of the states might take more time than recording the differences between successor nodes only, it is not a priori clear whether copying is competitive to trailing. However, Schulte showed that copying is competitive for

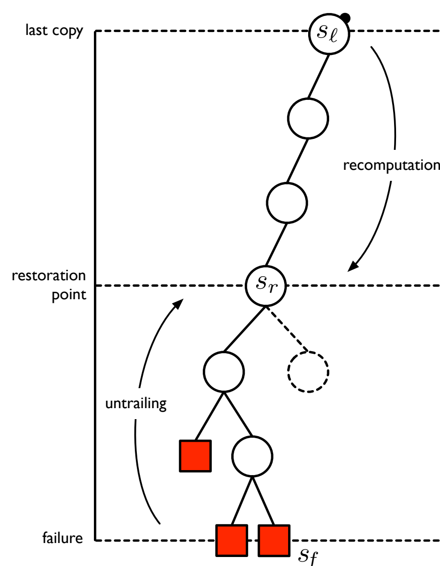
most problems [23]; for deep search trees one needs recomputation to decrease memory consumption. Schulte also provides a detailed comparison of runtime and memory requirements for selected examples. However, for Boolean problems, SAT solvers using trailing are much more efficient than pure copying solvers.

Talking about implementation issues, one substantial difference is noticeable: while copying is independent of operations that modify the nodes of a search tree, trailing has to investigate exactly these modifications. In the restoration phase it must have operations to invert the effects of the previous operations. Hence, trailing is more involved than copying: copying is only concerned with data structures, while trailing has to know the operations. The data structures (and so the operations) used in the solver thus depend in a fundamental way on the restoration technique.

Chapter 4

Reconciling Copying and Trailing

In Chapter 3 we have seen the structural difference between recomputation (downward restoration) and trailing (upward restoration). The figure to the right recapitulates the scenario: the three important levels of the search tree are marked by dashed lines: the failure level (bottom), the last copy level (top) and the restoration point (middle).



The task of the restoration process is to restore the state s_r from failure s_f. Trailing does this by simply *untrailing* the modifications stored on the trail up to level r. Recomputation on the other hand takes a clone from the last copy from level ℓ and *redoes* the exploration steps. Both techniques meet in the middle in state s_r.

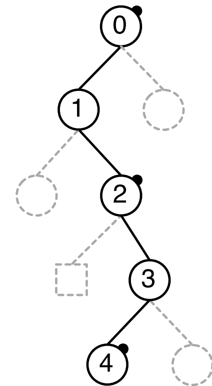
Before understanding the difficulties of coordinating copying and trailing we have to take a look at how information is managed in the different systems. In this context, *information* means variable domains, propagators, dependencies and other possible values that determine the state of a solver.

Since in depth-first search there is only a single path available from the root node to the current node, both systems use a *stack* as central restoration data structure. The stacks have one **separator entry** for each node on the current path of the search tree. The stack of a copying system is referred to as *recomputation stack*, while the stack of a trailing system is called *trail*. We will use the names interchangeably.

4.1 Information in Copying Systems

Information in copying systems is local information. The solver only operates on local copies of variable domains and propagators.

The Recomputation Stack The stack of a copying and recomputation system consists of separator entries only. The figure to the right shows a path in a depth-first search tree with its corresponding stack.



Up to three values are stored in a stack entry. The first value describes which alternative leads to the current node. The alternatives (numbered beginning with 0) thus represent a path from the root node to the current node.

The second value is a *branching description* containing the constraints that have been added by branchings in the exploration phase. Recomputation will later add exactly these constraints to the corresponding spaces of the search engine.

recomputation stack

node	alternative, description	clone
0	(0, desc)	(clone)
1	(1, desc)	
2	(1, desc)	(clone)
3	(0, desc)	
4		(clone)

Level 4 of the search tree contains neither a branching alternative nor a branching description since search has not yet explored the tree beyond level 4. All previous levels must contain both alternative and description.

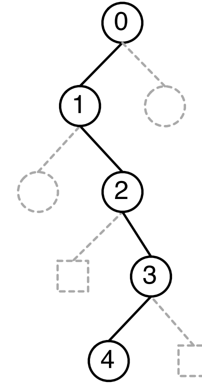
The third value may store a clone of a stable space. A clone is available in a stack entry whenever exploration or recomputation have left a copy of a certain node in memory.

Recall that the nodes of a search tree represent stable states (c.f. page 10). This is the reason why there is a stack entry only for the nodes, but not for the edges: in contrast to trailing, copying with recomputation does not care about the differences between two states.

Variables and Propagators In a copying system all variables and propagators are copied. At any time, the variable domains and the propagators are *locally* available. The dependencies between variables and propagators are also locally stored. They are mutually updated in each copying step.

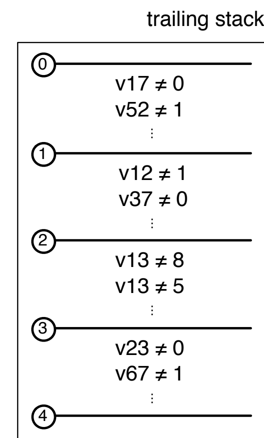
4.2 Information in Trailing Systems

Information in trailing systems is global information. The solver always operates on single global instances of variable domains and propagators.



Trailing records changes between two nodes, in particular modifications of variable domains. The state of a solver is modified *in-place*, without copying. Hence, the domains must be stored globally. Otherwise, if the domains are copied and collected in a space, trailing would have to memorize in which spaces the domains of a variable are nested. Furthermore, it would have to update all these nodes. Clearly, this is too expensive. We call these global domain variables *trailed*.

The Trailing Stack Since trailing does not store copies of spaces in memory, the separator entries of the stack only contain the level number, no spaces. The changes between two nodes at level k and $k+1$ are recorded on the stack *between* the separator entries k and $k+1$. In level 2 for instance, values 8 and 5 have been excluded from the domain of variable v_{13}. Restoration simply undoes the modifications. For v_{13} this means adding the values 5 and 8 to the domain.



For a Boolean variable v_i it suffices to store a reference to v_i since there can only be exactly one modification of $\mathsf{Dom}\ v_i$ on the trail. Restoration in this case simply resets $\mathsf{Dom}\ v_i$ to the initial domain $\mathbb{B}$.

Decisions appear before propagation steps on the trail. There is no need to separate them since both are assignments or modifications that are considered equal upon restoration.

Variables and Propagators Trailed variable domains are modified by non-copied propagators of which only a single global instance exists. No copying is necessary.

The dependencies between variables and propagators are also globally stored. Since modifications of the dependencies are not trailed, undoing them is not directly possible in the restoration phase. Hence, such modifications must be valid over time.

4.3 Reconciling Copying and Trailing

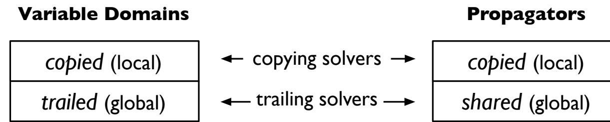
A hybrid solver has to support two variable types, one from the copying system, one from the trailing system. Accordingly, it must support two propagator types. In contrast to the propagators, the variables differ in *two* aspects: concerning domain and concerning dependencies. The first differentiation we consider is towards the domain.

Definition 4.1 (Variable Domains) A **trailed variable** v has exactly one global domain $\mathsf{Dom}\ v$. There exist no copies of $\mathsf{Dom}\ v$. A **copied variable** u however can have several (possibly modified) copies of its initial domain. $*$

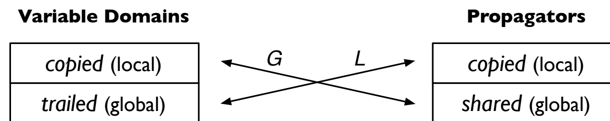
Before talking about the second differentiation for variables, we introduce the different propagator types from both systems.

Definition 4.2 (Propagator Types) Each propagator in a hybrid system has exactly one type. Propagator p is **copied** if it is stored in a local space. Hence, p can have a state that has to be copied with a space. A propagator q is **shared** if there exists exactly one global instance of q. In particular, q may not have a state that has to be restored upon restoration. $*$

There are four ways of combining the variable domain types with the propagator types introduced above. Two ways are well-known from the original solvers,



and two *new* combinations arise:



Comparison In copying solvers, propagators and variable domains are copied, hence locally available. In trailing solvers, there exist only single instances of propagators and variable domains, hence they are globally available. Global information is cheap concerning memory consumption. Each modification of the state (variable domains, states of propagators) has to be explicitly trailed for restoration. Access to variable domains is efficient since references to the domains are persistent.

Local information on the other hand is expensive in memory consumption as it has to be copied. But it has the advantage that all information is restored implicitly by picking a copy of a state from memory. A propagator in a copying-only system can hence have additional states that will be restored automatically.

The new combinations of local and global information are denoted by G and L. G is a combination of local domains and global propagators. L represents variables with global domain, managed by local propagators. We will discuss these types below.

		propagators	
		copied	*shared*
domain	*copied*	default for copying solvers	G
	trailed	L	default for trailing solvers

Figure 4.1: Local vs. global information

The second differentiation for variables is according to the dependencies between propagators and variables.

Definition 4.3 (Variable Dependencies) A variable v has exactly one of the following dependency types:

- **local** Only copied propagators can subscribe to v.
- **global** Only shared propagators can subscribe to v.
- **hybrid** Both copied and shared propagators can subscribe to v. *

Besides the global domain, variables can have global dependencies. These global dependencies are stored at a global location (in the same way as global domains for trailed variables). Whenever a trailed variable domain is modified, all shared propagators in the global dependency list are scheduled.

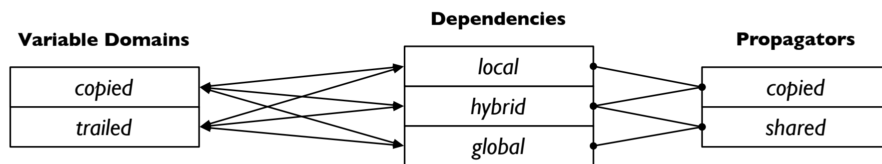
Copying solvers use copied local variables, i.e. the domains and the dependencies are locally available. Only copied propagators are used. Trailing solvers use trailed global variables, i.e. the domain is stored globally, the dependencies are global and only shared propagators have access to the variables.

Figure 4.2 shows the complete classification for variables. The vertical classification distinguishes the domain, the horizontal axis differentiates the dependencies for a variable. In the following, we will discuss the six different types.

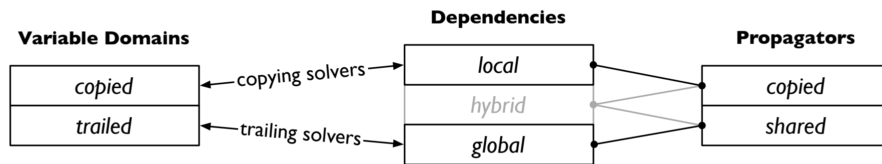
		dependencies		
		local	*hybrid*	*global*
domain	*copied*	**default for copying solvers**	H_1	G
	trailed	L	$\boldsymbol{H_2}$	**default for trailing solvers**

Figure 4.2: All variable types

Each variable type corresponds to an edge with an arrow on the left hand side of the figure below. The edges on the right hand side are fixed by definition.

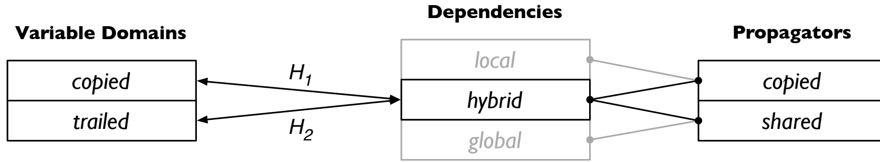


Default Types Two of the six types must naturally be supported by a hybrid solver. Copied local variables and trailed global variables are referred to as *default types*.



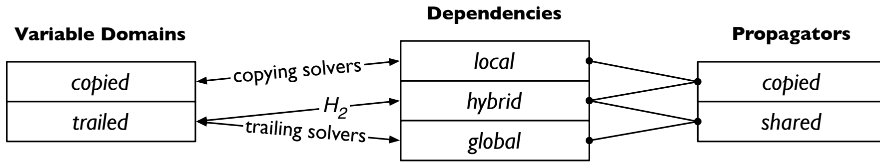
Hybrid Types The power of a hybrid solver consists in reconciling the original solvers. If no communication between the local part and the global part occurs, the efficiency of a hybrid solver is not higher than the efficiency of the individual solvers. Hence, communication between the original parts must be possible.

The figure above shows a tripartite graph with *two* connected components. Clearly, no communication between the necessary default variable types is possible. In order to allow communication between the default types, we introduce the two *hybrid variable types* H_1 and H_2, the communication types:



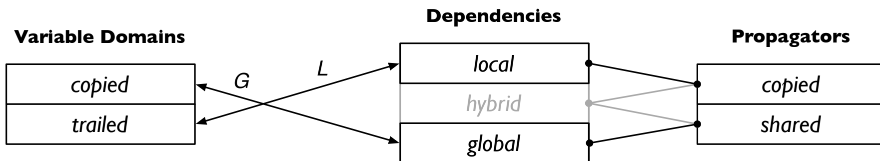
H_1 is a copied variable which both propagator types can operate on. Its dependencies are hybrid, hence it has two dependency lists, one for each propagator type. The list for shared propagators is stored at a global location, while the list for copied propagators is locally available. H_2 differs from H_1 as its domain is global, hence a trailed variable.

The figure above shows a tripartite graph with only *one* connected component, thus communication between different variable types is possible. It even suffices to have either H_1 or H_2 in a hybrid solver. We decided to implement H_2 only. Again, we obtain a tripartite graph with *one* connected component:



Remaining Types The variable types G and L are not implemented. We have no indication that a hybrid solver would benefit from their contribution.

Type G is neither more effective, nor more efficient. As there is only a single instance of a shared propagator p, it cannot easily operate on copied variable domains. p would have to be updated each time a variable domain $\mathsf{Dom}\ v$ is copied or restored. It cannot have access by a direct reference to v. An indirection through some local index structure is necessary in order to find the adequate copy of v. Furthermore, modifications that shall be persistent over time would have to be updated in all copies of v. The difference to type H_1 is a local dependency list, which does not cause a significant overhead since local information is already maintained with the local domain.



For type L the list of dependencies is locally stored. If the domain is small, e.g. a Boolean domain $\mathbb{B}$, it can be stored locally as well. This would hardly decrease

efficiency since the variable is copied anyway. This type corresponds to the first default variable.

Even for large domains, L is not necessary. Adding global dependencies to L (which then is type H_2) does not cause a significant overhead since there is already global information available, in terms of the global domain.

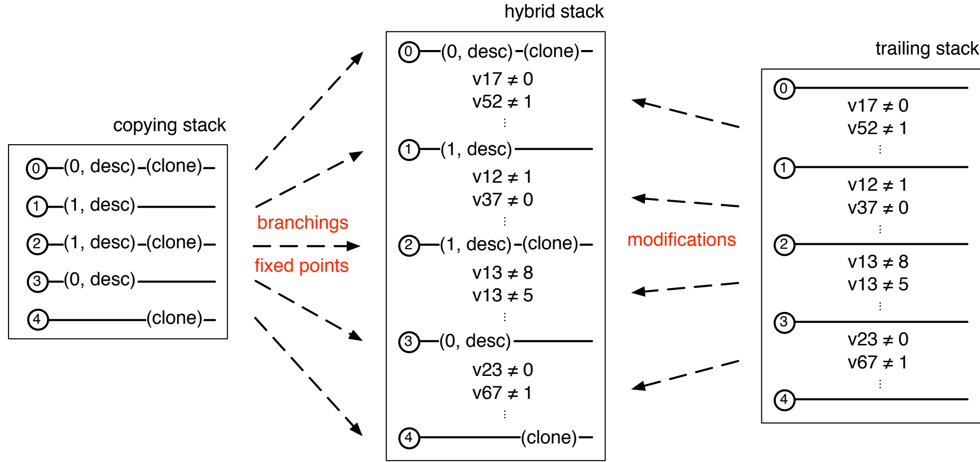
Summarizing, a hybrid constraint solver has to deal with two kinds of propagators (copied, shared) and three kinds of variables, two from the original systems (copied local, trailed global) and one new hybrid type H_2 (trailed hybrid).

Spaces Revisited In Section 2.1.2 we defined a space as a collection for variable domains, propagators and branchings. At this point, we have to refine the notion of a space. From now on, a space only stores local information: copied variables, copied dependencies, copied propagators and branchings. In particular, a space does no longer represent the complete state of a solver. Trailed variables, global dependencies and shared propagators are excluded. The search engine operates on exactly one space, the current space, and on the global information, namely trailed variables, global dependencies and shared propagators. Several clones of spaces representing the copied part of the state of a solver can be stored in memory. The global part of the state is never copied.

	state of a solver	
information:	local part	global part
storage:	collected in a space several clones of a space in memory	available at a single persistent location never copied
restoration:	**replace** actual space by previous one	**undo** the modifications successively

If a state is stable then a space is stable. The converse direction does not hold.

A Hybrid Stack We use a synthesis of the stacks from the copying- and trailing-only systems to model the stack for the depth-first exploration of the hybrid system as depicted on the next page.



The branching alternatives, the branching descriptions and the fixed points from the copying system are stored together with the modifications from the trailing system on a single synthesised stack. The separator entries contain the local copied information in terms of spaces. The global trailed information is placed between two separator entries.

4.4 The Restoration Process

In this section we explain the interaction of the different restoration techniques in copying-based solvers with trailing. Conceptually, copied variables are restored by replacing the current space by a previous one, followed by some recomputation steps. Trailed variables on the other hand are restored in-place via untrailing.

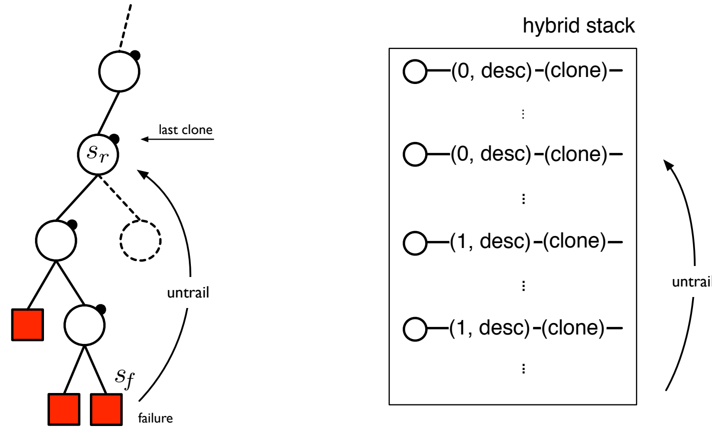
Note that we perform not only the steps of the restoration process (upwards). We let search perform one more decision step (downwards), namely the next alternative that search has to perform from the restored state to continue exploration. The last decision step is denoted by a dashed edge in the following search trees.

Notation In the following we denote the state to be restored by s_r, the last state for which a clone has been stored in memory by s_ℓ, the state at a leaf (failure or solution) from which restoration starts by s_f, c.f. page 17. We use r, ℓ and f to refer to the corresponding levels of the search tree.

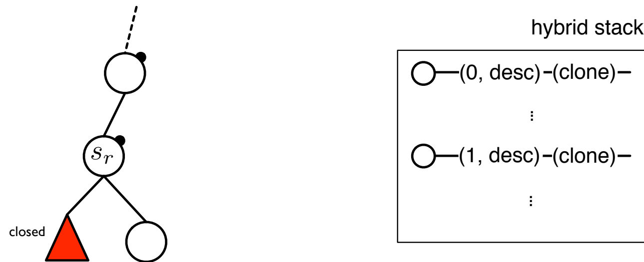
The restoration process obtains level r by looking at the entries of the hybrid stack: s_r is the first entry (looking upwards) of which not all alternatives are explored yet.

4.4.1 Full Copying

In full copying, there is a space for each live state available in memory, in particular $s_\ell = s_r$. Thus, the copied variables are restored by duplicating the corresponding clone s_ℓ and setting it to be the current space of the search engine. Trailed variables are restored by untrailing the modifications between state s_f and state s_r.



No recomputation is necessary. The active path ends in state s_r and every node in the left subtree is marked closed. Exploration commits to alternative 1 and new fixed point for the right subtree is computed.

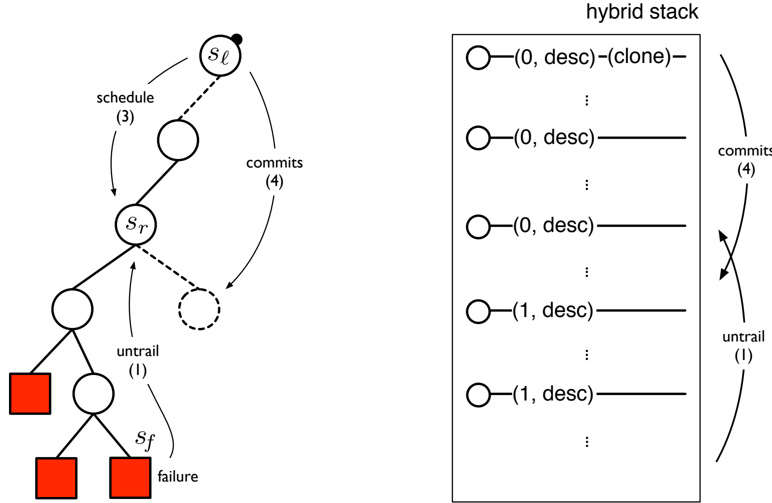


4.4.2 Full Recomputation

For full recomputation, the trailed variables are restored by untrailing the modifications between s_f and s_r (1) as above. Since there is only one clone available at the root node of the search tree, a copy of space s_ℓ will replace the current space of the search engine (2). Hereafter, the copied propagators depending on trailed variables between spaces s_ℓ and s_r must be scheduled explicitly in the new current space of the search engine (3). More precisely, for each variable

v of type H_2 on the stack between level ℓ and level r, the copied propagators $\{\, p \in \mathsf{P} \mid p$ is copied $\wedge\, v \in vars(p)\}$ listed in the local dependency list of v are scheduled for execution.

Note that in the recomputation process (4) the copied propagators are not scheduled automatically since the modifications of trailed variables between s_ℓ and s_r are not part of the recomputation. The trailed variable domains are not modified since after untrailing (1) they are in state s_r.



This explicit scheduling of copied propagators that depend on trailed variables would not be necessary if the restoration process untrailed the trailed domains to state s_ℓ. Then, recomputation would modify both, the trailed and copied domains, and all propagators would be scheduled. But clearly, this is much more expensive than untrailing to s_r and scheduling the remaining levels explicitly.

An important insight concerning efficiency is the downward direction of the explicit scheduling. The propagators have to be scheduled in the same order as they were executed in the exploration phase. Recall that scheduling is a consequence of variable domain modifications. Think of a propagator p_c implementing the constraint $c : x+y = 5$ and variable domains $\mathsf{Dom}\ x = \{2,3\}$ and $\mathsf{Dom}\ y = \{2,3\}$. If in the exploration phase another propagator q modifies x such that $\mathsf{Dom}\ x = \{2\}$, then p_c will be scheduled and set $\mathsf{Dom}\ y = \{3\}$. The reversed order (p_c, q) however will re-schedule p_c after q for a second run, since the first execution of p_c cannot modify the initial domains. Hence, the reversed order of explicit scheduling will result in the same fixed points, but more executions of the propagators are necessary.

After the explicit propagator scheduling, the current space of the search engine is modified downwards to represent the copied part of state s_r. For this purpose,

the constraints corresponding to the alternatives on the recomputation stack are added (4). These commits will not yield a failure since exactly the same computations down to state s_r are performed as before.

The fact that copied propagators are scheduled in the current space (3) *before* the commits are done (4) does not result in different fixed points compared to the exploration phase. This is due to batch recomputation: exactly one fixed point s_r is computed.

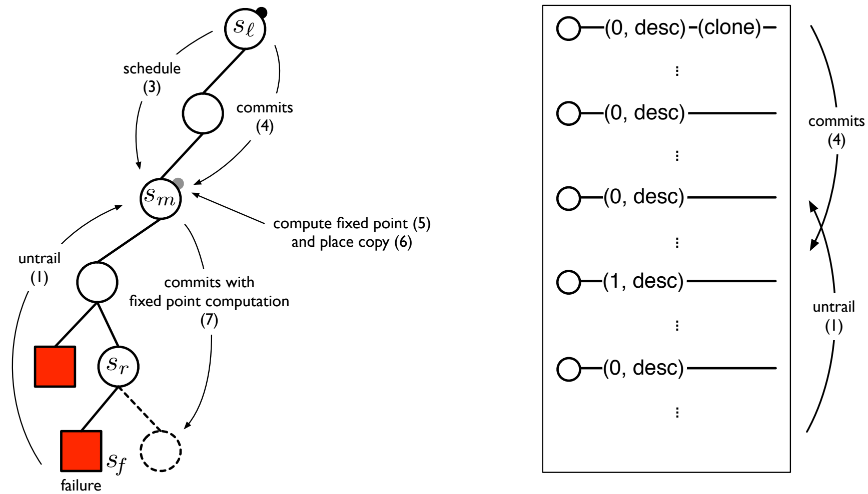
4.4.3 Fixed Recomputation

Fixed recomputation with recomputation distance n behaves nearly as full recomputation does when it comes to restoration. The last clone will be at most n levels above, while in full recomputation it is in the root level. Everything else is the same: untrailing to level s_r, duplicating the clone from level s_ℓ, scheduling the copied propagators, performing the commits, computing a single fixed point.

Note that no additional clone will be stored during recomputation. Only the last fixed point might have to be stored if the distance to the last clone has become n. This is the task of the search engine that computes the fixed point.

4.4.4 Adaptive Recomputation

Adaptive recomputation contains the most complicated interaction between copying and trailing since a space between state s_ℓ and state s_r might be stored during the restoration process.



In our example, states s_f and s_r are only one level apart. However, the complete restoration process requires more than just a few steps.

Before untrailing, we have to compute the middle m of the path from state s_ℓ to state s_r. Since we will store a space for state s_m, we need to compute a fixed point before. This means that all variable domains must be in state s_m. Clearly, the fixed point would be different if the trailed domains were in a later state. Thus, we have to untrail the trailed variable domains to level m (1), not only to level r.

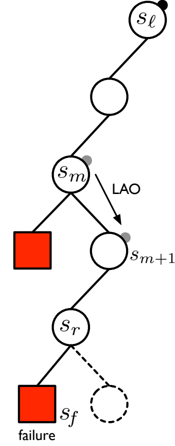
After untrailing, we set a copy of the last space of level ℓ to be the current space of the search engine (2) and schedule the copied propagators explicitely (3), in the same way as for full recomputation (Section 4.4.2). Recomputation performs the commits to transform the current space of the search engine into state s_m (4).

The difference to other restoration variants is the computation of a fixed point for level m (5). The resulting stable space is stored in memory (6). Both trailed variable domains and copied domains are at the same level m now. In order to reach state s_r, the decision constraints, available on the stack between s_m and s_r, must be added to the current space of the search engine. It is important to see that batch recomputation does not work in this case: if only one single fixed point was computed, the modifications of trailed variable domains in the computation of the single fixed point would appear on the stack between level s_{r-1} and s_r. If later on, the nodes in these levels become closed, the modifications will get lost. Hence recomputation would compute different fixed points.

A way of still benefiting from batch recomputation is the following approach: remember the modifications that are stored on the stack. Untrail the modifications as before (1), compute an intermediate fixed point (5), and do the commits (7) without computing a fixed point for each added constraint. To set the trailed variables to state s_r, replay the modifications stored on the stack. The stack hence is the same as before untrailing, the trailed variable domains have the same values. Thus computation of a single fixed point is correct.

The recomputation steps from m to s_r can be considered as re-exploration rather than recomputation. Computation of a fixed point for each level is the same as in the exploration phase. The only difference is that the branching descriptions are already available on the stack.

Last Alternative Optimization for Adaptive Recomputation

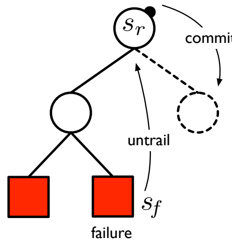


On computing the middle m between s_ℓ and s_r a last alternative optimization may be performed. Consider the example to the right. All but one of the alternatives of the state s_m for the computed middle m have been explored. Thus, recomputation would always start by recomputing the step from state s_m to s_{m+1}. On placing a copy upon recomputation, the rightmost branches are skipped. In the example, the copy of a space for state s_{m+1} would be placed in memory.

This is a cheap optimization with high impact. Fewer untrailing steps have to be performed (1). Fewer commits and fewer fixed point computations have to be done (7).

4.4.5 Last Alternative Optimization

The scenario for general last alternative optimization is special since there is a copy of the space for state s_r available in memory, hence $r = \ell$. As for full copying, the global information is untrailed from state s_f to state s_r. The space for s_r will replace the local part of s_r.



No recomputation and no scheduling is necessary. Exploration commits to the last alternative and a fixed point for the right subtree is computed.

In contrast to full copying, the space stored for state s_r is picked up from memory, it does not remain there.

4.4.6 Conclusion

All hybrid restoration techniques for the interaction between copying and trailing are recapitulated in Figure 4.3.

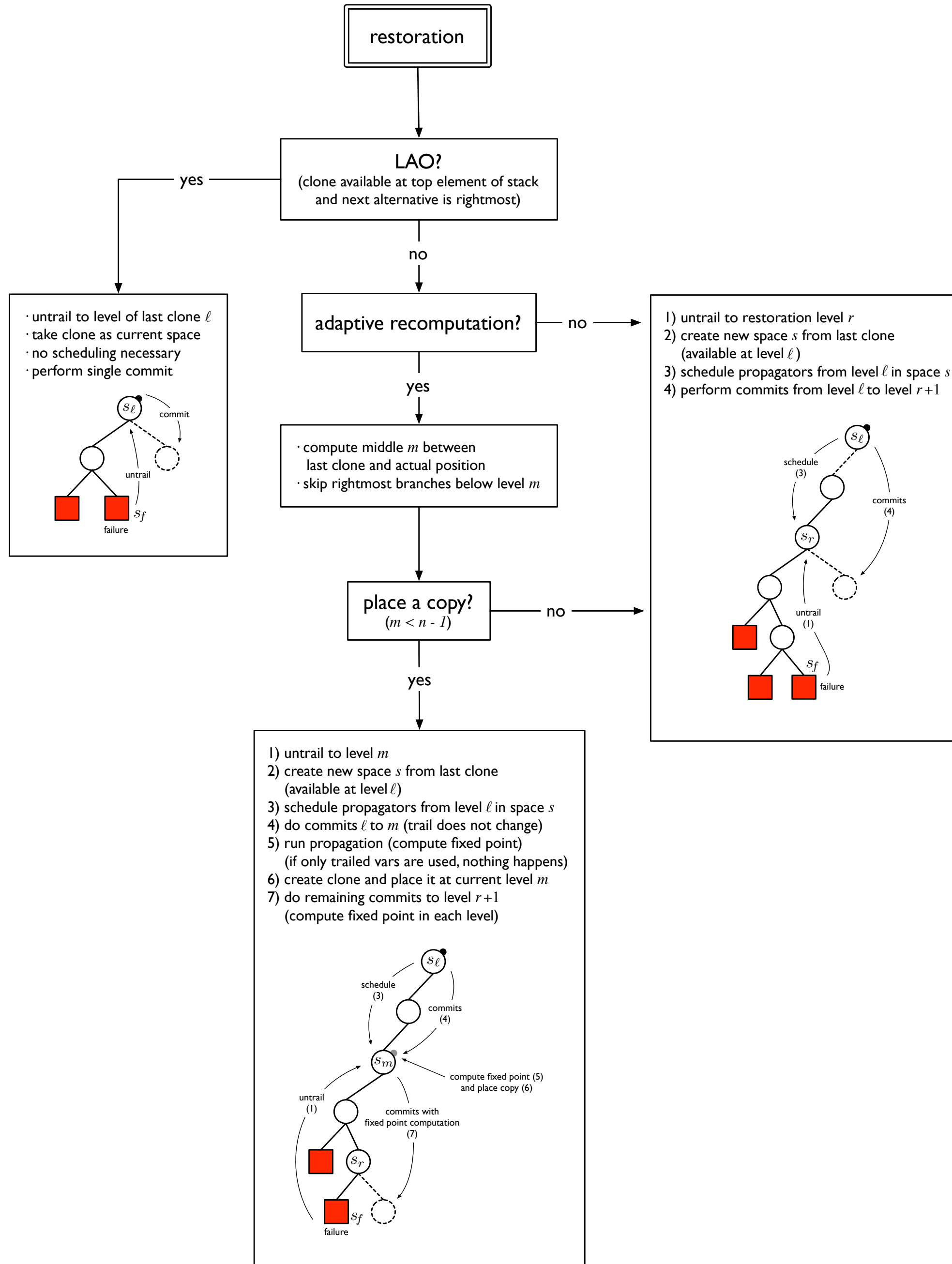


Figure 4.3: Interaction between recomputation and trailing

Chapter 5

Constraint Propagation for SAT Problems

This chapter recapitulates techniques for solving the Boolean satisfiability problem (SAT). Our hybrid restoration techniques, presented in the previous chapter, increase efficiency for solving Boolean problems. In order to further benefit from our architecture we use two techniques for constraint propagation of SAT problems. First, the concept of *watched literals* is presented: work of propagators is reduced by not scheduling propagators too often. Second, *conflict clause learning* is discussed, a technique that is only feasible in trailing systems with shared propagators.

The Boolean Satisfiability Problem Satisfiability is the problem of determining if the variables of a given Boolean formula can be assigned values that satisfy the formula. These formulae are typically represented in conjunctive normal form (CNF), a conjunction of *clauses*. A **clause** is a disjunction of *literals*. Each **literal** represents a Boolean variable with sign, i.e. a variable that can be positive or negative. Based on double negation, De Morgan's laws and the distributive law, each Boolean formula can be transformed into CNF. A formula is **satisfied** if and only if all clauses are satisfied. A clause is satisfied if and only if at least one literal has the value 1.

5.1 Watched Literals

As we have seen in Section 2.1.2, propagators are scheduled only if the domain of a dependent variable has been modified. What happens if a propagator has a large number of dependent variables that may cause the propagator to be scheduled in cases it cannot narrow any variable domain? The concept of *watched literals*

addresses this problem by not subscribing to all dependent variables. It was first introduced in 2001 by Moskewicz et al. [17] and is a major reason for dramatic improvements in SAT solvers in this decade. However, copying-based solvers may use this technique as well, as Gent et al. have shown in 2006 [12].

To illustrate the effect, we investigate propagation for the Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ with n variables in conjunctive normal form (CNF), for instance

$$f(x_1, x_2, x_3, x_4) \;=\; (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee \overline{x_4})$$

where $\overline{x} = 1 - x$ denotes the negation of a Boolean variable x.

The goal is finding an assignment for each variable $x_1, \ldots, x_n$ such that the function $f(x_1, \ldots, x_n)$ evaluates to 1 or to show that no such assignment exists. Each clause c_i of the conjunction is implemented by an individual propagator p_i since an assignment satisfying the conjunction must satisfy each individual clause (in which identical variables are assigned the same values). In particular, each clause must contain one literal that is assigned the value 1.

Such a clause propagator could thus be subscribed to all variables of its clause. This means that every modification of any of the variables of the clause will cause the propagator to be rescheduled, which is futile dissipation.

Example 5.1 Consider the first clause $(x_1 \vee x_2 \vee \overline{x_3})$ of function f above. Assume all variables to be unassigned. Then, if the value 1 is assigned to variable x_1, the clause is satisfied. Propagation cannot set any other variable since every value is allowed for x_2 and x_3. This example shows that propagation is performed only if a partial assignment of the variables *forces* at least one other variable to take a specific value. If variable x_1 is set to 0 then, in order to satisfy the clause, either x_2 must be set to 1 *or* x_3 must be set to 0. Again, propagation cannot assign any variable. $*$

How many variables of a clause should a clause propagator be subscribed to? It suffices to subscribe the propagator to only two literals of an n-ary clause for $n \geq 2$: only if all but one of its literals have been assigned, propagation may assign the last literal. For this purpose it suffices to take special consideration of two literals of the clause, the so-called **watched literals**. We will maintain the following invariant: both watched literals are not assigned the value 0.

If a watched literal l has been assigned the value 0, a new unassigned literal is sought. If another unassigned literal is found, it will replace l as new watch. This step is part (b) of the 'Rule for the Elimination of One-Literal Clauses' of the Davis-Putnam Algorithm presented in 1960 [8]. If however no other unassigned literal exists, the propagator will perform **unit propagation**, i.e., the only unassigned literal, the second watch, is assigned a value satisfying the clause.

Further, the new watched literal is not required to be unassigned. An assignment that satisfies the clause is fine as well. If, after some backtracking,

restoration and propagation steps, the value does no longer make the clause true, the propagator will be rescheduled automatically due to the new assignment.

Initially, the watched literals are chosen randomly since the solver has no idea in which order the variables of a clause will become assigned.

The following example shows two typical runs for the procedure of finding new watched literals.

Example 5.2 Consider the second clause $(\overline{x_1} \vee \overline{x_2} \vee x_3 \vee x_4)$ of function f represented in Figure 5.1. Two scenarios for clause propagation from empty domains to complete assignments are shown; one for the worst case (left hand side), and another for the best case.

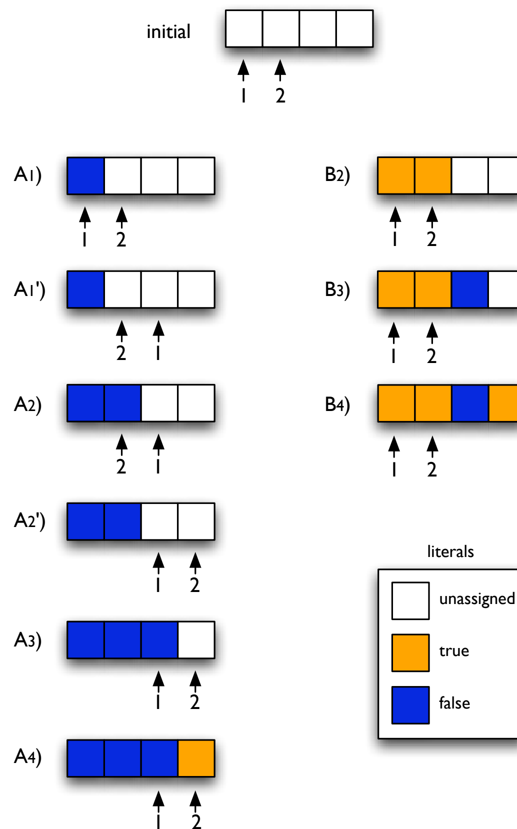


Figure 5.1: Finding new watched literals

In the first step (A_1) the propagator is called with watched literal x_1 set to 0. We have to find a new literal ($A_1{}'$). Next, watched literal x_2 is set to 0 (A_2), which causes the second watch to be moved ($A_2{}'$). Now watched literal x_3 has been set to 0 (A_3) and the propagator is scheduled again. Since this assignment does not satisfy the clause, again a new watched literal is sought. But as there

cannot be found any other unassigned literal, the literal at the second watch must be assigned the value 1 (A_4).

The scenario A represents the worst case. With each assignment of a variable the propagator is scheduled. If x_3 or x_4 were assigned before x_1 and x_2, the propagator would have been scheduled only twice. This is the best case.

Another interesting best case scenario is depicted on the right hand side. The propagator is called with watched literals x_1 and x_2 set to 1 (B_2), which directly satisfies the clause. Both literals are satisfied. This means that assignments of all other variables (B_3) and (B_4) do not force propagation. Satisfying the clause is already achieved by the watches. $*$

Summarizing, the advantage of watched literals is the reduction of work when no propagation is possible, at best case in scheduling a propagator only if propagation can narrow at least one variable domain. Furthermore, if the watches are set to literals that do not change their domain in-between some backtracking and recomputation interval, the corresponding propagator will not be scheduled at all.

Are Clause Propagators Backtrack-safe?

As two watches are stored with each clause propagator, these propagators have a state. Usually, a state must be restored upon backtracking. However, if a state is guaranteed to be valid after backtracking, it is called *backtrack-safe*. In an implementation with non-copied propagators this is an important issue, as we will see.

The state of a clause propagator is determined by the watches pointing to two literals. After backtracking, these watches must still satisfy the invariant of pointing either to unassigned literals or to literals satisfying the clause. Two cases are of interest:

- an unassigned literal does not change upon backtracking. The invariant is kept.
- a literal assigned 1 can become unassigned by backtracking, which still maintains the invariant. In case of recomputation (Section 3.1.2) the literal will become 1 again, also maintaining the invariant.

Hence, the invariant is always maintained. Clause propagation based on watched literals is backtrack-safe. In this case, restoring Boolean variable domains has constant complexity since the watches do not have to be changed.

5.2 Conflict Clause Learning

The idea behind **conflict clause learning** is to analyze the reason for a failure during search and learn from the observed conflict. Adding new constraints shall prevent similar conflicts in future search. Furthermore, it enables the search engine to backtrack in a non-chronological way to higher levels of the search tree, which often results in pruning large portions of the search space. The technique was first presented in 1996 by Silva and Sakallah [24] and extended in 2001 by Zhang and Madigan [28]. See also Beame et al. [4] for further reading.

The constraints learned this way are implied clauses that are obtained by *resolution steps* according to the resolution rule:

$$\frac{A \vee x \qquad B \vee \overline{x}}{A \vee B} \quad \textsf{resolution}$$

The result of applying the resolution rule is called *resolvent*. Resolution has been used in the Davis-Putnam Algorithm [8] under the name 'Rule for Eliminating Atomic Formulas'.

The *immediate* reason for a conflict are two clauses containing the same variable, one in positive, the other in negative form. Let A and B represent arbitrary disjunctions, then $c_A \equiv A \vee x$ and $c_B \equiv B \vee \overline{x}$ are clauses that involve a conflict for variable x as soon as both A and B are not satisfiable. The implications $\overline{A} \Rightarrow x$ and $\overline{B} \Rightarrow \overline{x}$ can be resolved to $A \vee B$. Repeating this procedure may resolve further variables until short clauses are learned.

Complete resolution solves a SAT problem with exponential complexity. Since this is too expensive, conflict clause learning performs resolution steps only when propagation and search have found a failure: it computes the resolvent that would have avoided the failure if it was added before.

The learned clause could exactly represent the assignments in the failure. If, for instance, the assignment x, $\overline{y}$, z gives a failure, the learned clause could be

$$\neg(x \wedge \overline{y} \wedge z) \equiv (\overline{x} \vee y \vee \overline{z})$$

Because of the disjoint branching alternatives, this situation will never reappear. Thus, this learned clause is correct, but it does not prune the search tree in the future. Learned clauses with more impact for future search can be found using implication graphs.

5.2.1 Implication Graphs

Each assignment of a variable domain due to search establishes a new **decision level** from which a sequence of propagation steps is initiated, which again may initiate further propagation steps. A graphical representation of these implications is a so-called **implication graph**, a directed acyclic graph whose vertices

are assignments of variables. Its edges represent the clauses that are responsible for the assignments.

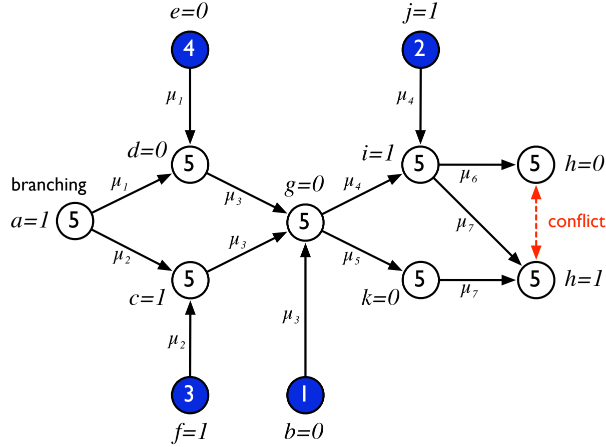


Figure 5.2: Implication graph with conflict

Figure 5.2 depicts an implication graph with decision variable a and conflict variable h. It will serve as an example for further explanations. The decision level for each assignment is written inside the vertices. The 7 corresponding clauses are

$$\mu_1 = \overline{a} \vee \overline{d} \vee e \qquad \mu_3 = b \vee \overline{c} \vee d \vee \overline{g} \qquad \mu_5 = g \vee \overline{k} \qquad \mu_7 = h \vee \overline{i} \vee k.$$
$$\mu_2 = \overline{a} \vee c \vee \overline{f} \qquad \mu_4 = g \vee i \vee \overline{j} \qquad \mu_6 = \overline{h} \vee \overline{i}$$

In decision level 5, search has set variable a to 1. As a consequence, propagation for clauses μ_1 and μ_2 assigns variables d and c.

Each non-decision variable v has incoming edges (at least one) representing the clause that is responsible for the assignment of v. An n-ary clause is represented by $n-1$ edges. Note that assignments from earlier decision levels are considered as decisions for the current decision level.

A vertex in the implication graph that is on every path from the decision variable to the conflict variable is called **unique implication point** (UIP). Intuitively, a UIP is the single reason that implies a conflict in the current decision level. In Figure 5.2 there are two UIPs, the vertices for variable a and variable g.

A **cut** in the implication graph splits the graph into two disjoint sets of vertices. The left-hand side of a cut containing the decision variable and assignments from earlier levels is referred to as **reason side**, while the right-hand side containing the conflict is called **conflict side**. Two such cuts are shown in Figure 5.3.

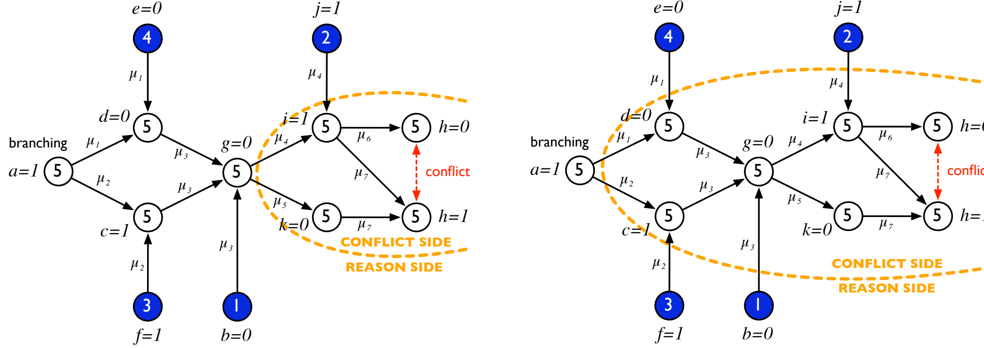


Figure 5.3: Two possible cuts

Cuts thus separate between conflict and reason. The idea is to learn variables from the reason side that have an influence on the conflict. In the left cut, the assignments of variables g and j are sufficient to imply a conflict on variable h. In the right cut, variables a, b, e, f and i provoke the conflict. Hence, the assignments $g = 0$ and $j = 1$ will always end up in a conflict, independent from the assignments of other variables.

From this conflict one may learn the clause

$$\neg(\overline{g} \wedge j) \quad \equiv \quad (g \vee \overline{j})$$

or respectively the clause

$$\neg(a \wedge \overline{b} \wedge \overline{e} \wedge f \wedge j) \quad \equiv \quad (\overline{a} \vee b \vee e \vee \overline{f} \vee \overline{j}).$$

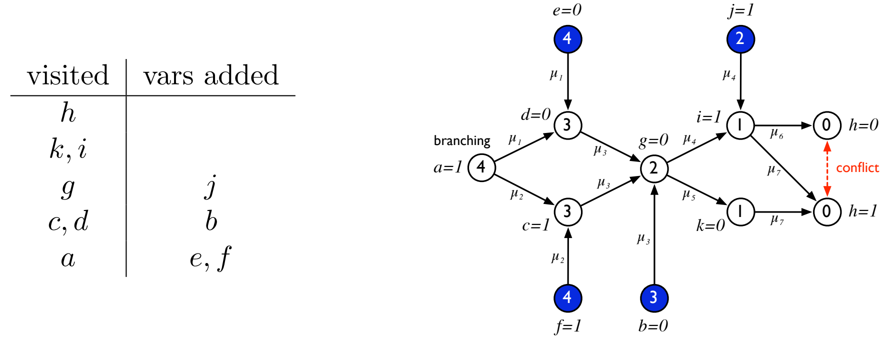
In general, a learned clause for a given cut consists of all variables (vertices) at the reason side, that have an outgoing edge into the conflict side crossing the cut. In order to choose the cut whose clause has the strongest impact for further conflict elimination, we have to consider the first UIP. The **first UIP** (1-UIP) is as close as possible to the conflict. As we will see later, the clauses with the strongest impact are learned from cuts that cross in front of the 1-UIP, but leave it on the reason side. The 1-UIP in the graph of Figure 5.2 is the vertex for variable g, the 1-UIP cut is the left cut.

5.2.2 Finding the 1-UIP

The implication graph is constructed backwards beginning at the two vertices for the conflict variable. A breadth-first search is performed in reversed order of assigning the variables. Whenever a vertex has no incoming edges, the associated variable is directly added to the conflict clause. All other vertices are to be explored. The procedure stops, when breadth-first search has reached a level with a single vertex, the 1-UIP.

Assuming the order of assignment to be $a, d, c, g, i, k, h = 0$ and $h = 1$, BFS explores the variables in the order listed below. Vertex g is recognized as 1-UIP, the second UIP is a. The figure on the right hand side denotes the order of visited vertices inside the nodes.

visited	vars added
h	
k, i	
g	j
c, d	b
a	e, f



The Algorithm

Figure 5.4 shows an algorithm that analyzes a conflict and generates the resulting conflict clause. It is basically taken from the current implementation of MiniSat 2.0 [9]. Whenever a clause propagator p has reported failure and the current trail level (decision level) is not the root level 0, the `analyzeConflict()` procedure is called. In decision level 0, propagator p tells the search engine directly that no solution exists.

The procedure uses an implicit agenda consisting of the array `seen[]`, the trail `trail[]` and the counter `agendaSize`. Each assigned variable `v` appears at exactly one position on the trail. If `v` is on the agenda, `seen[v]` is true. The function `level(v)` returns the level in which variable `v` was assigned by propagation or search. If `level(v)`$= 0$, then `v` was assigned by unit propagation in the root level.

In the following we explain each part of the algorithm separately. Lines 3 to 5 serve as setup for the helper variables:

```
3    int trailIndex := currentTrailHeight
4    int var := -1
5    int agendaSize := 0
```

The variable `trailIndex` contains the current position in the trail; initially one position above the topmost entry. Except for the first run, the variable `var` contains the index of the variable for which the reason is traced. `agendaSize` stores the number of variables that must still be picked up from the trail.

The main loop (7 to 35) performs the breadth-first search in an analogous way as in the implication graph. The loop is divided into two parts: the first part implements the exploration of the implication graph given a clause `conflictClause`, while the second part picks the next clause from the agenda.

In the first complete execution of the for-loop (11), the variable `var` has a special

```
1  procedure analyzeConflict(conflictClause) {
2
3    int trailIndex := currentTrailHeight
4    int var := -1
5    int agendaSize := 0
6
7    do {
8
9      // TRACE REASON FOR CONFLICT CLAUSE
10
11     foreach variable v in conflictClause {
12
13       if (v = var or seen[v] or level(v) = 0)
14         continue
15
16       seen[v] := true
17
18       if (level(v) = currentLevel)
19         agendaSize := agendaSize + 1
20       else
21         learnLiteral(¬v)
22     }
23
24     // SELECT NEXT VARIABLE AND NEXT CLAUSE
25
26     do {
27       trailIndex := trailIndex - 1
28     } while (not seen[trail[trailIndex]])
29
30     var := trail[trailIndex]
31     conflictClause := reason(var)
32     seen[var] := false
33     agendaSize := agendaSize - 1
34
35   } while(agendaSize > 0)
36
37   learnLiteral(¬v)
38 }
```

Figure 5.4: Finding the first UIP via breadth-first search

meaning. Except for the first time, `var` prevents adding `v` to the agenda again (13, 14). Furthermore, these lines anticipate the re-examination of a variable. Last, if `level(v)` is zero, then `v` must not be contained in the learned clause, as it is set by unit propagation in the beginning (level 0) and can thus be considered as a constant.

```
11      foreach variable v in conflictClause {
12
13        if (v = var or seen[v] or level(v) = 0)
14          continue
15
16        seen[v] := true
17
18        if (level(v) = currentLevel)
19          agendaSize := agendaSize + 1
20        else
21          learnLiteral(¬v)
22      }
```

All other variables `v` in the conflict clause are marked to be seen (16). Either they appear on the agenda (19), or they are learned (21) and shall thus not be learned twice.

Each variable outside the decision level must be added to the conflict clause directly (21), as the 1-UIP cut puts it on the reason side. Variables on the conflict side cannot be learned, they are put on the agenda (16, 19).

The second part first finds the next variable on the agenda by traversing the trail downwards,

```
26      do {
27        trailIndex := trailIndex - 1
28      } while (not seen[trail[trailIndex]])
```

assigns the next variable `var` and the next clause `conflictClause` to look at,

```
30      var := trail[trailIndex]
31      conflictClause := reason(var)
```

and removes the variable from the agenda.

```
32      seen[var] := false
33      agendaSize := agendaSize - 1
```

The last element on the agenda is the 1-UIP and must finally be learned.

```
37    learnLiteral(¬v)
```

5.2.3 Non-chronological Backjumping

The backjump level is the uppermost level of the search tree in which the learned clause can propagate, i.e. as soon as $n-1$ literals of an n-ary clause are assigned.

It is calculated as

$$bjLevel = \mathsf{max}\,\{\,\text{level}(l) \mid l \in C \setminus \{u\}\}$$

where C is the learned clause, $l \in C$ are the literals of C and $u \in C$ is the 1-UIP.

The only learned literal l with $\text{level}(l) = currentLevel$ is the 1-UIP u (line 37). Therefore, $bjLevel < \text{level}(u) = currentLevel$, i.e., all $n-1$ other literals were assigned at earlier decision levels. Hence, at the maximum of these levels, $n-1$ literals are assigned and so the learned clause can propagate.

The 1-UIP cut on page 39 creates the learned clause $(g \vee \overline{j})$ with $bjLevel = 2$. Since no learned clause from other UIP cuts contains fewer literals, the backjump level of other clauses will be at least the backjump level for the 1-UIP clause. The learned clause from the right cut in Figure 5.3 for instance has $bjLevel = 4$. Clearly, a backjump to a higher level (smaller integer) prunes a greater part of the search tree.

A proof for the optimality of the 1-UIP concerning the backjump level was presented by Audemard et al. in 2007 [3].

5.2.4 Setting the Watches for Learnt Clauses

On setting up a problem, the literals on which watches are set can be chosen randomly. Initially, the solver has no idea, in which order the variables of a clause will become assigned.

For learned clauses a correct choice of the watched literals is necessary to guarantee correctness. Assume a learned clause to have $bjLevel = n$. Further, assume both watches point to literals that are assigned in level $l < n$. After backjumping there will be no modification of the watched literals, the propagator for the learned clause will never be scheduled. Thus, the idea of preventing similar failures will be completely futile. If the solver does not remember its choices, it can even end up in an endless loop.

The watches must hence point to the 1-UIP literal since it will be assigned directly in the backjump level. The reason for this assignment must be caused by the second watch, which must hence point to a literal with the next lowest level, the backjump level.

5.2.5 Advantages of Shared Propagators

Conflict clause learning adds one clause for each conflict. Depending on the problem, the number of learned clauses hence can be exponential in the number of Boolean variables. The learned clauses have to be implemented via shared propagators: adding shared propagators does not cause any overhead on copying a space. However, adding copied propagators would slow down each branching

step in which a space is copied. For such a large number of learned clauses, using copied propagators is not feasible at all. The number of learned clauses even grows so fast, that current implementations of SAT solvers dispose global learned clauses if their impact is not high enough [9].

Another deprecative aspect of copied propagators is that learned clauses implemented by copied propagators would have to be added to every live space on the active path in order to be globally available over time.

5.2.6 Interaction with Non-Clause Propagators

Conflict clause learning traces implications in order to generate new constraints. The investigated constraints must be available in terms of logical implications. Clauses can easily be transformed into implications: the reason for an assignment of a Boolean variable due to clause propagator p_c are the assignments of all other variables of the clause p_c.

Assignments of Boolean variables due to other propagator types may have to be considered as decisions. A simple propagator $x \lor y \Leftrightarrow z$ cannot completely be rewritten as implication. If variable x has become assigned, the reason is an assignment of z and $\overline{y}$. The implication is $z \land \overline{y} \Rightarrow x$. If however variable z has become assigned, the reason is an assignment of either x or y or both. No implication is obtained in that case.

Conflict clause learning hence requires *and* generates clauses. While typical SAT solvers increase efficiency dramatically using clause learning, a hybrid solver cannot always benefit from this technique.

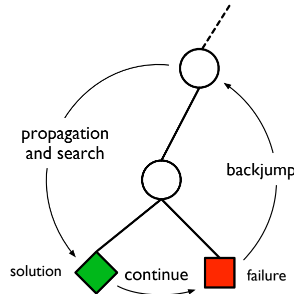
5.2.7 Interaction with Integer Variables

So far, we considered Boolean variables only. An interesting topic for future work is the interaction of clause learning with integer variables. Trailing-based solvers treat solutions as failures in order to continue search for further solutions. This avoids exploration of the same path in the search more than once. If the constraint problem contains integer variables, it is not possible to learn from the assignments of the Boolean variables only.

Consider an example in which the assignments of a solution are x, $\overline{y}$ and z. The integer variable a is assigned the value 5. Another solution might be x, $\overline{y}$, z and $a = 7$. Learning the clause $\overline{x} \lor y \lor \overline{z}$ is wrong since search must explore this path again. A correct learned clause would have to contain the integer variables as well: $\overline{x} \lor y \lor \overline{z} \lor a \neq 7$. These clauses are hard to handle in practice.

If no clause is learned for a solution, the solver could end up in an endless loop: exploration continues after restoration with the next alternative, it comes to failure, a clause is learned and the solver jumps back non-chronologically.

Since the learned clause has no effect for the previous solution and the branching descriptions have been popped from the stack, the previous solution is computed again.



Turning off the non-chronological backjumping after the first solution could avoid this loop, but decrease efficiency considerably. If there is at least one assignment of an integer variable (treated as decision) in the current decision level, then non-chronological backjumping is not possible at all (c.f. Section 5.2.3).

For a failure however, clause learning for Boolean variables regardless the integer variables is correct. This is ensured by the propagators combining Boolean variables and integer variables.

Nevertheless, it would be interesting to obtain reasons for the assignments of integer variables and find some constraints that avoid similar conflicts in the future. In 2007, Ohrimenko, Stuckey and Codish presented a dynamic translation of constraint problems into SAT problems [18]. This approach is lazy in that clauses are generated based on inferences that propagators perform on the original constraint problem; hence this is no static translation into SAT.

5.2.8 Conflict Clause Learning vs. Systematic Search

While the DPLL algorithm [7], invented in 1962, still serves as basis for most efficient complete SAT solvers, the integration of clause learning with non-chronological backjumping changed the structure of modern SAT solvers dramatically.

DPLL performs systematic search in nearly the same way as finite domain constraint solvers do: search considers all possible assignments of variables, propagation may skip assignments that are no solutions. Because of non-chronological backjumping, modern SAT solvers on the other hand *repair* wrong decisions of the search engine by learning clauses that ban search from assigning wrong values. This structural difference may be one reason for the fact that reconciling copying and trailing with clause learning and backjumping for integer problems is not easily doable, if at all.

The notion of repairing only the actual wrong decision is supported by Pipatsrisawat and Darwiche [20]. They propose a caching scheme for erased decisions

(due to backjumping) that are assumed to be correct. Whenever search selects a variable v that has previously been assigned and erased afterwards, v is assigned the previous value again. This lightweight scheme is in particular efficient if the problem consist of independent sets of clauses that share no variables, so-called *independent components*. If a solution for a component c is found, a decision concerning another component d can be corrected without recomputing the solution for c.

Chapter 6

Implementation and Evaluation

This chapter describes the interesting aspects of an implementation of the architecture for a hybrid constraint solver reconciling copying and trailing. Besides the hybrid stack (presented in Section 4.3) we have to implement trailed variable domains, global dependencies and shared propagators. The chapter concludes with experimental results comparing our hybrid solver to pure copying and to pure trailing solvers.

Our implementation is integrated in the C++ constraint programming library Gecode [25] presented below. We only mention the major changes that are necessary to support trailing in Gecode.

6.1 The Gecode Library

Due to its modular design the Gecode library consists of several independent components. At its heart there is a generic kernel around which several extensible modules (like search, constraints, variables) are placed. The most important ones are listed below.

[Kernel] Gecode's kernel provides an extensive programming interface that allows the user to easily construct new variable domains, propagators, branchings and search engines.

[Search] Search in Gecode is based on copying with recomputation. The delivered search engines allow search for some solutions, optimization search (branch-and-bound) and limited discrepancy search. Gecode can be extended to further search engines providing for instance parallel search.

[Branching] A branching specifies which variables a search engine selects in order to split their domains.

[Propagator] Gecode's propagators implement the counterpart to search, namely inference. Each constraint of a CSP is represented by at least one propagator.

Gecode comes with propagators for finite domain constraints like arithmetic and Boolean constraints. Furthermore there are distinct, count, element and table constraints, as well as constraints on regular expressions and set variables.

[Space] A space serves as container for variable domains, propagators and branchings, hence it describes the local part of the state of the solver.

6.2 Modules

This section presents some modules we have implemented in order to support trailed variables and shared propagators. The global part of the solver's state (c.f. Section 4.3) is represented by the `SharedController` module and instances of the `SharedPropagator` module. Variables are implemented via the `VarImp` module, global domains and global dependencies via the `GlobalVarInfo` module. Shared propagators are represented by instances of the `SharedPropagator` module.

6.2.1 SharedController

The `SharedController` module is responsible for maintaining trailed variables. The current space and all copied spaces contain exactly one `SharedController` instance. Global trailed variables are stored in the root space instance, hybrid trailed variables are stored in copied instances.

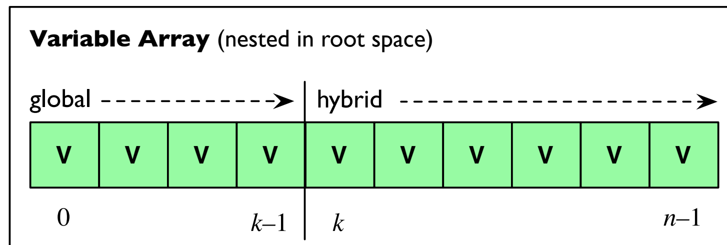
Attributes

- **int** `n`
  Total number of trailed variables.
- **int** `k`
  Number of *global* trailed variables. The number of *hybrid* trailed variables hence is `n` − `k`.
- **int** `k'`
  Total number of global trailed variables for the current `SharedController` instance. For the root space's instance, `k'` = `k`. For all copied `SharedController` instances, `k'` = 0.
- `VarImp* varArray`
  An array maintaining the trailed variables. Trailed global variables are only

stored in the `SharedController` instance of the root space. Trailed hybrid variables are contained in every `SharedController` instance. The details for the array are explained in the following.

Methods

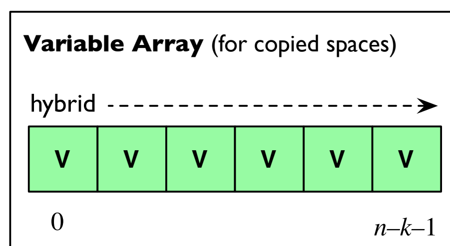
- `addVariable(VarImp v, bool global)`
  sets a trailed variable `v` to be maintained by the `SharedController`. The flag `global` indicates that `v` has only global dependencies. In this case `v` is stored in the global part of the variable array. Hence `v` obtains a unique index between 0 and $k-1$:

  The size of the global part of the array must be determined in advance. On setting up the problem's model, the number of global variables `k` must be known in advance. Global variables can only be added to the `SharedController` instance of the root space.

  The size of the hybrid part `n` − `k` may vary during exploration of the search space. This is important since propagators shall be able to add hybrid variables at any time.

- `SharedController* copy()`
  The copy operation creates a new `SharedController` instance. This operation is performed whenever a space is copied. Each space contains exactly one `SharedController` instance. Except for the root space, no instance contains global trailed variables. Upon copying the root instance, the hybrid part is copied to the left with initial index 0:

k' is set to 0 in the new instance since no global variables are contained in the array of the new instance.

- VarImp getVar(int idx)
  If a variable v with index $idx < k$ is accessed, the corresponding variable is looked up at position idx in the array of the SharedController instance of the root space. If $idx \geq k$, the variable is sought in the local array at position $idx - k + k'$. In the root space, we have $k' = k$, hence the correct position for hybrid variables is ensured. Thus, on looking up a variable in the array, no case distinction concerning the current space is necessary.

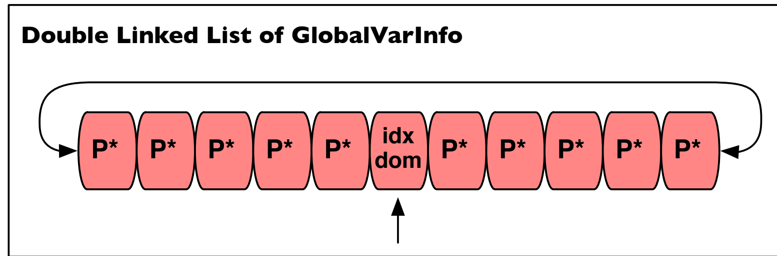
6.2.2 VarImp

The VarImp module maintains information for variables, the dependency information and the domain information. The classification of variable v is implemented according to Section 4.3:

- copied local
  The domain and the dependencies are stored locally. There is no access to a GlobalVarInfo structure. v is not maintained by the SharedController as it is not trailed.
- trailed global
  The domain and the SharedController array index is globally stored in the GlobalVarInfo structure. There are no local dependencies, v is not copied and hence in the global part of the SharedController array.
- trailed hybrid (H_2)
  The domain and the SharedController array index are globally stored in the GlobalVarInfo structure. There may be local dependencies for v, hence v is stored in the local part of the SharedController array.

6.2.3 GlobalVarInfo

The GlobalVarInfo structure for a trailed variable v is a double linked list. The head element from which the list is accessed contains the unique index idx indicating the position in the SharedController array and the domain dom. All other elements in the list are references to SharedPropagators. These are the shared propagators subscribed to v.



Each trailed variable `v` has access to exactly one `GlobalVarInfo` structure, stored at a global location. A copy `v'` of `v` has the same reference to a `GlobalVarInfo` structure as `v` has.

6.2.4 SharedPropagator

An instance of `SharedPropagator` implements a non-copied propagator. Its implementation is based on the implementation of Gecode propagators. The propagator queues (c.f. Section 2.1.2) for Gecode propagators and shared propagators are the same.

A shared propagator has access to a dependent variable `v` via the unique index of `v` in the `SharedController` array. It maintains a list of indices, one index for each dependent variable.

6.3 Experimental Results

This section provides experimental results comparing our architecture to standard copying systems and pure trailing systems for different benchmark categories.

All benchmarks were executed on an Intel Pentium 4 CPU 2.8 GHz, 1 GB RAM. Runtimes are the average of 10 runs, with a coefficient of deviation significantly less than 1% for all benchmarks. The examples are taken from the Gecode example suite, available and explained on the Gecode webpage [25].

In the following, we first analyze the overhead our hybrid solver produces for disadvantageous problems, second we look at the speedup our solver yields for the intended class of problems. Note that the results for the hybrid solver are obtained from a non-optimized prototype. A careful analysis could yield an optimization reducing runtime and memory usage.

6.3.1 Overhead for Non-trailed Variables

The following results concern the overhead between the copying-based solver Gecode in the current version 2.2.0 and our hybrid implementation on top of it.

Table 6.1 lists the execution statistics for several examples not containing trailed variables. The examples only use copied local variables. Hence, our solver cannot contribute any new feature but it has to maintain the trailing machinery.

This clearly causes an overhead which is represented in the third column *runtime hybrid*; 100% denotes no overhead. The second column states the runtime of Gecode. The column *failures* characterize the size of the search tree. For both solvers, propagation is the same. Hence, the explored search trees for each example are equal.

example	*runtime Gecode in s*	*runtime hybrid in %*	*prop. steps* $\cdot 10^6$	*failures*	*solutions*
all-interval-15	19.179	110.77	36.2	831 284	2 sol.
golomb-ruler-11	22.575	100.55	78.8	321 419	best sol.
graph-color-1	46.560	106.31	28.6	287 256	
queens-91	76.914	106.75	39.2	5 032 158	
queens-200	3.019	112.75	1.2	146 866	
TSP-2	1.349	112.62	0.6	35 311	best sol.

Table 6.1: Overhead for non-trailed variables

Brief Problem Description The first problem *all-interval-n* (CSPLib, prob007 [13]) searches for permutations $(x_1, x_2, \ldots, x_n)$ of the numbers $\{1, 2, \ldots, n\}$ such that $(|x_1 - x_2|, |x_2 - x_3|, \ldots, |x_{n-1} - x_n|)$ is a permutation of $\{1, \ldots, n-1\}$. The optimization problem *golomb-ruler-m* (CSPLib, prob006) seeks a set of m integers $0 = a_1 < a_2 < \ldots < a_m$ such that all the $m(m-1)/2$ differences $a_j - a_i$ with $1 \leq i < j \leq m$ are pairwise distinct. The optimal solution for a given m has the minimal ruler length, i.e. a_m is minimal. The graph-coloring problem *graph-color* finds a coloring of the vertices of a graph such that no two adjacent vertices share the same color. The graph in *graph-color-1* has 200 vertices. *Queens-k* addresses the problem of placing k queens on a $k \times k$ chess board such that no queen can attack any other. The optimization problem *traveling salesman* searches for the least-cost round-trip route that visits each city exactly once and returns to the starting city. TSP-2 contains 17 cities.

The overhead for the examples in Table 6.1 amounts to at most 12% in case there is rather little propagation per overall time. This implies more copying or more expensive propagation steps. More copying suggests that copying provokes overhead. Recall that each copied space has to create its `SharedController` instance (which is useless if there are no trailed variables).

6.3.2 Overhead with Copied Propagators

The examples in this section are hybrid problems consisting of trailed and non-trailed variables. All propagators are copied. Hence the complete advantages of our hybrid solver are not exploited.

Again, we compare Gecode to our hybrid prototype. The number of trailed local variables (type L) is listed in the second column. The experiments are executed using full copying (Section 3.1.1) as restoration technique.

example	*variables*	*runtime Gecode in s*	*runtime hybrid in %*	*memory Gecode in MB*	*memory hybrid in %*	*prop. steps* $\cdot 10^3$	*failures*
knights-20	8 202	0.475	169.2	126.3	138.3	126	2
knights-30	35 568	3.063	176.0	685.5	140.3	517	40
nonogram	625	1.054	101.7	1.7	100.0	101	3 242
pentom-4	924	9.923	113.8	3.1	100.0	1 711	28 078
perf-square-0	5 544	0.201	128.0	4.3	129.8	675	150
perf-square-1	5 764	0.838	148.0	5.2	129.6	1 286	1 122
perf-square-4	16 284	1.756	129.3	13.4	135.9	7 388	372

Table 6.2: Performance of hybrid problems

Brief Problem Description *Knights-n* is the problem of visiting each field of an $n \times n$ chess board exactly once by performing knights moves only. Start and end field must be the same. A *nonogram* (CSPLib [13], prob012) is a puzzle composed of a matrix of markers. For each row/column there is a specification on the number of groups of markers and their length. This example uses trailed variables only. *Pentominoes* places all pieces of a puzzle onto a grid without overlapping. The pieces may all be rotated and flipped freely. For the benchmarks we search for 10 solutions. The *perfect-square* placement problem (CSPLib, prob009) is to pack a set of squares with given integer sizes into a bigger square, again without overlapping.

Our hybrid prototype uses more memory than Gecode does: we need memory for the implementation of the trailing stack, only small variable domains are trailed and all dependencies are still copied. The greater the amount of memory overhead, the greater the overhead in runtime. This fact shows that too much copying decreases runtime considerably. As our solver supports non-copied propagators and non-copied dependencies we investigate the performance for sharing in the next section.

6.3.3 Gain with Shared Propagators

The examples for trailed variables, shared propagators and shared dependencies (taken from SATLIB [14]) are used for all the remaining benchmark suites.

example	*variables*	*runtime Gecode in s*	*runtime hybrid in %*	*prop. steps Gecode $\cdot 10^6$*	*prop. steps hybrid in %*	*failures*
dubois-20	60	40.3	69.3	141.4	111.2	3 145 727
flat-200-1	600	28.6	71.1	73.8	194.0	167 629
hanoi-4	718	126.2	67.9	296.6	204.8	888 431
ramsey-4-13	78	1 156.7	30.8	2 738.5	63.2	14 546 239

Table 6.3: Gain with shared propagators

Even if we have to perform more propagation steps (for the first three examples) in Table 6.3, our hybrid prototype is about 30% faster than Gecode. For less propagation we increase the speed-up even more. Additionally, we use only about one forth of the memory that Gecode uses.

This result shows that a trailing architecture can be integrated in a copying system saving memory and runtime. We think the gain with shared propagators is more relevant for most problems than the overhead for copied propagators.

6.3.4 Gain with Clause Learning

Since our hybrid solver supports conflict clause learning (Section 5.2), the last results can be further improved as presented in Table 6.4.

example	*time Gecode in s*	*time hybrid in %*	*prop. Gecode $\cdot 10^6$*	*prop. hybrid in %*	*fail. Gecode $\cdot 10^3$*	*fail. hybrid in %*	*mem. Gecode in KB*	*mem. hyb. in %*
dubois-20	40.3	0.004	157.3	0.005	3 146	0.001	196	39
flat-200-1	28.6	0.570	73.8	1.338	168	0.362	2 181	472
hanoi-4	126.2	0.189	296.6	0.559	888	0.078	1 924	1 333
ramsey-4-13	1 156.7	4.727	2 738.5	4.995	14 546	0.214	2 053	21 313

Table 6.4: Performance of SAT problems

Our prototype is much faster compared to a copying system without clause learning. The gain in the number of propagation steps is proportional to the runtime gain. The number of failures is decreased even more. However, the memory consumption is much greater than for Gecode. As mentioned in Section 5.2.5, learned clauses may produce a significant overhead in memory. For the example *ramsey*-4-13, more than 31 000 clauses are learned. To overcome this problem, MiniSat disposes some of the learned clauses (c.f. Table 6.5).

Comparison to Pure SAT Solvers

To get an impression of the performance of our hybrid prototype compared to state-of-the-art SAT solvers, we compare the four examples with the performance of MiniSat [9].

example	*runtime MiniSat in s*	*runtime hybrid in %*	*prop. steps MiniSat*	*prop. steps hybrid in %*	*memory MiniSat*	*memory hybrid in %*
dubois-20	0.008	21.9	13 614	51.1	690	2.0
flat-200-1	0.023	708.3	76 752	1 287.2	541	259.4
hanoi-4	0.079	301.0	160 449	1 033.8	2 655	606.6
ramsey-4-13	0.003	1 822 333.3	512	26 717 419.0	81	11 394.8

Table 6.5: Comparison to MiniSat

In general, we perform more propagation steps than MiniSat does, which results in an overhead of runtime. This overhead for our solver is justified by the fact that SAT solvers not only use conflict clause learning and backjumping, but also VSIDS (Variable State Independent Decaying Sum). VSIDS maintains a score for each literal of a variable for selecting variables in the branching phase. We have not yet implemented this heuristic.

Since our prototype does not yet dispose learned clauses, we clearly require more memory than MiniSat does. Furthermore, our clauses need much more memory for clause representation than MiniSat needs. Summarizing, the memory overhead and the overhead in propagation steps is nearly proportional to the overhead in runtime.

6.4 Summary

Our solver benefits significantly from the integration of trailing in a copying-based solver if we use shared propagators. The runtime can be reduced down to 30%. The combination of copying with trailing permits to implement clause learning. This way we can achieve a more dramatic speedup.

On the other hand, the runtime overhead can be kept small. For problems in which no part of the hybrid architecture is effectively used, our prototype causes an overhead in runtime of up to 12%. The overhead may increase significantly when copied propagators with trailed variables (type L) are used.

Using copying for large variable domains (integer and set variables) together with trailing for Boolean variables and shared propagators seems to outperform pure copying solvers as well as pure SAT solvers: pure copying solvers without shared propagators cannot implement advanced solving techniques like clause learning and non-chronological backjumping. However, Gecode, as one of the

most efficient solvers for integer constraint problems, demonstrates that copying is competitive to trailing for integer and set domains. Pure SAT solvers cannot handle integer domains. Every static translation from integer domain constraint problems looses structure information, which may cause an exponential overhead with respect to the number of constraints.

Chapter 7

Conclusions

Trailing based state-of-the-art SAT solvers are efficient in solving pure SAT problems. However, a static translation of integer problems into SAT causes an enormous overhead. Copying based integer constraint solvers (such as Gecode) on the other hand are efficient for integer and set problems, not for SAT problems.

Our hybrid solver reconciles both worlds by benefiting from their particular advantages. We have carefully investigated the interaction between the different restoration techniques, *copying* and *trailing*. Our resulting prototype solves particular problems considerably faster than the original solver. These results suggest that we are able to solve hybrid constraint problems consisting of Boolean and integer variables much faster than other efficient solvers.

Future Work

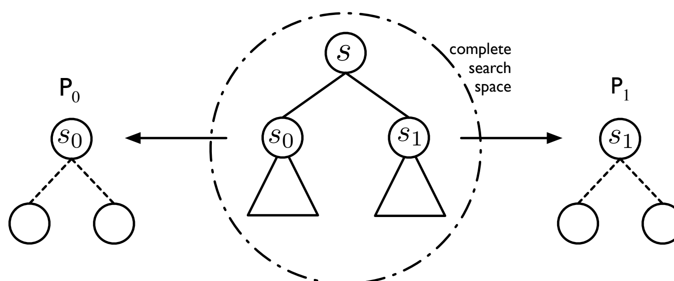
The encouraging results of our prototype are ample motivation to continue working on improving our preliminary implementation. There are various approaches of optimizing our hybrid solver, which we mention in the following.

Lazy Clause Geneartion The lazy clause generation approach of Ohrimenko, Stuckey and Codish [18] mentioned in Section 5.2.7 reconciles clause based SAT solvers and constraint solvers with integer propagators. The task of solving the problem is basically done by the SAT solver. Our system reconciles both worlds by implementing the restoration techniques of both solvers in a copying solver. One could hence think of further reconciling these different approaches.

SAT Techniques There are various techniques from the area of SAT solving that could be supported by our solver, but have not yet been implemented.

To mention only a few, there is the *Variable State Independent Decaying Sum* (VSIDS) decision heuristic presented by Moskewicz et al. in 20001 [17], a caching scheme for branching decisions by Pipatsrisawat and Darwiche (2007) [20], and an improvement of implication graphs presented by Audemard et al. in 2007 [3].

Parallel Computing In order to explore parts of the search tree concurrently, parallel computing requires the ability of communicating states between at least two search engines, two solvers or two machines. These states represent root nodes of subtrees to be explored as starting points for partial exploration of the search space. Our system captures the local part of the state of the search engine in a space, which can easily be stored in memory. The global part (trailed domains, modifications on the trail, global dependencies and shared propagators) requires copying and storing methods. It is then possible to search subtrees of the search tree in parallel by transmitting entire states to several instances of the search engine.



Open Research Questions The technique of *conflict clause learning* presented in Chapter 5 is only understood for Boolean variables and their assignments due to clause propagators since their implications can easily be obtained. Is it possible to integrate non-clause propagators that yield an implication only in some cases (c.f. Section 5.2.6)? Can we extend clause learning to integer variables? Are there efficient implementations of clauses containing integer variables? These *hybrid* clauses may help to further reconcile both systems.

Bibliography

[1] Hassan Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, USA, 1991.

[2] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[3] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. A Generalized Framework for Conflict Analysis. In Marques-Silva and Sakallah [16], pages 21–27.

[4] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Understanding the Power of Clause Learning. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 1194–1201. Morgan Kaufmann, 2003.

[5] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[6] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for State Restoration in Tree Search. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2001.

[7] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[8] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[9] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications*

of Satisfiability Testing - SAT 2003, 6th International Conference, Santa Margherita Ligure, Italy, May 5-8, 2003, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[10] Leonhard Euler. De quadratis magicis (On magic squares), 1782. Available at http://arxiv.org/abs/math/0408230v6.

[11] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible Sudoku grids, 2005. Available at http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf.

[12] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched Literals for Constraint Propagation in Minion. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2006.

[13] Ian P. Gent and Toby Walsh. CSPLib: a problem library for constraints, 2008. Available at http://www.csplib.org.

[14] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT, 2008. Available at http://www.satlib.org.

[15] Richard Jones and Rafael Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[16] João Marques-Silva and Karem A. Sakallah, editors. *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007*, volume 4501 of *Lecture Notes in Computer Science*. Springer, 2007.

[17] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001.

[18] Olga Ohrimenko, Peter Stuckey, and Michael Codish. Propagation = Lazy Clause Generation. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*. Springer, 2007.

[19] Laurent Perron. Search Procedures and Parallelism in Constraint Programming. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming - CP '99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1999.

[20] Knot Pipatsrisawat and Adnan Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In Marques-Silva and Sakallah [16], pages 294–299.

[21] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, 2006.

[22] Christian Schulte. Comparing Trailing and Copying for Constraint Programming. In Danny De Schreye, editor, *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*, pages 275–289. The MIT Press, 1999.

[23] Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302 of *Lecture Notes in Computer Science*. Springer, 2002.

[24] João P. Marques Silva and Karem A. Sakallah. GRASP—A New Search Algorithm for Satisfiability. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, November 10-14, 1996, San Jose, CA, USA*, pages 220–227. ACM and IEEE Computer Society, 1996.

[25] The Gecode Team. Gecode: generic constraint development environment, 2008. Available at http://www.gecode.org.

[26] The Mozart Consortium. The Mozart programming system. Available at http://www.mozart-oz.org, 2008.

[27] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, CA, USA, 1983.

[28] Lintao Zhang and Conor F. Madigan. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *International Conference on Computer-Aided Design, November 4-8, 2001, San Jose, CA, USA*, pages 279–285. ACM, 2001.