Exploring Alternative Restoration Techniques in Constraint Programming

Yong LIN

(Bachelor of Engineering, Sichuan University)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information that have been involved in the thesis.

This thesis has not been submitted for any degree in any university previously.

Yong LIN

31 March 2014

Acknowledgements

Foremost, I express my sincerest gratitude to my supervisor, Prof. Martin Henz, who has supported me throughout my study and research at National University of Singapore, for his advice, patience, enthusiasm and knowledge. I attribute the level of this thesis to his encouragement and guidance and it would not have been completed or written without him. One simply could not wish for a better or friendlier supervisor.

I would like to thank the members of my thesis committee, Prof. Roland Yap and Prof. Joxan Jaffar, for their insightful comments and enlightening questions. My thanks also goes to Prof. Christian Schulte and Dr. Guido Tack for their advice on developing our techniques. I thank Srikumar Karaikudi Subramanian for initial discussions, and I thank my fellow friends for their encouragements. Meanwhile, I acknowledge the support of the School of Computing and the university for my study and research.

Last but not the least, I offer my deepest appreciation to my family and relatives: LIN Xuefu, Zou Fangzhen, LIN Shiguo, LI Guangju, LIN Shiqiong, DING Daping, LIN Shibin and SHAO Fanfan, for supporting me throughout my life.

Table of Contents

Summary

Constraint programming is a powerful tool for solving combinatorial optimization problems in many practical applications, and constraint programming systems provide the facilities to support this tool. In such a constraint programming system, search defines the strategies to explore solutions and restoration recovers a previously visited state to continue when search encounters an inconsistency. Hence, a state-of-the-art state restoration technique is essential for an efficient constraint programming system.

In this thesis, we first investigate recollection as an alternative restoration technique; its main idea is to maintain the variables that were affected by constraint propagation to reason fix points for conducting restoration. Compared with the existing technique of copying, recollection exhibits a finer granularity; compared with recomputation, it avoids re-running the propagator filtering algorithms; and compared with the bottom-up restoration technique of trailing, recollection proceeds in a top-down manner and thus is suitable for systems that restore state in this manner.

We implemented recollection within the Gecode system in several alternatives, which are configurable through compile time flags. Our experimental evaluation reveals that recollection is able to improve runtime against recomputation on integer problems with deep search trees and intensive propagation, at the expense of moderate memory investment. An extended comparison with copying reveals that it saves both runtime and memory for some large problems with deep search trees, and previous cross-system comparison allows us to extrapolate these results to trailing-based systems.

Subsequently, we explore programming restoration granularity, which aims at providing strategies and facilities to enhance the customization of restoration in a constraint programming system. We initially implemented a prototype by integrating coarse-grained copying, finer-grained recollection and constraint-based recomputation, and this prototype uses the first search failure as a trigger to adjust the restoration technique.

To assist the switch between restoration code segments, we explicitly employ a signal in the prototype. This approach however couples tightly with a specific program and is not quite extensible when users intend to customize. To facilitate systematic programming, we propose to program the stored restoration information as an aspect, an abstraction developed in the aspect-oriented programming paradigm. Its significance is modularizing the implementation of restoration techniques and potentially providing more options to build search engines that run a wide spectrum of search algorithms while enhancing the extensibility of the constraint programming system.

List of Tables

List of Figures

Chapter 1

Introduction

Combinatorial optimization problems are ubiquitous in many application domains, including scheduling, timetabling, computational biology and software verification, to name a few. These problems are generally computationally NP-hard and their solving requires considerable expertise in optimization and software engineering. The constraint satisfaction approach to combinatorial optimization emerged from artificial intelligence (such as [22] and [20]) and programming language research (such as [35] and [7]). In such an approach, solving a combinatorial problem is to specify a set of constraints to represent the solutions, and a search procedure indicates the means to explore them. Constraint programming (CP) aims at simplifying this approach by providing rich alternatives to specify constraint and search strategies, while being efficient in performance.

In this chapter, we first define the constraint satisfaction problem in Section 1.1; in subsequent Section 1.2, constraint programming is briefly introduced with a

specific example, Sudoku and Section 1.3 specifies the main contents and organization of this thesis.

1.1 Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) is defined by a set of variables and constraints. The variables describe the objects that the problem deals with, and each variable has a non-empty domain to specify the set of value candidates it can take. Each constraint imposes on a subset of variables to specify the allowable combinations of values for the variables. In common applications, the variables of a problem are restricted to a finite set of integers[1], and a variable is *fixed* when it contains a singleton domain. For a CSP, an assignment is fixing a subset of its variables and the assignment is *consistent* if it violates none of constraints. The process of solving a CSP is fixing its variables to consistent values, and a solution to a CSP is a consistent assignment to all variables. For an optimization CSP, it also requires the solution to maximize or minimize a *cost function*.

Search is a complete method for solving a CSP, which guarantees that solutions can be found provided they exist. The brute-force search is such a complete approach: each possible value combination of all variables is enumerated to verify whether it is a solution to the problem. However, for such an approach, the number of possible value combinations is generally too large to enumerate all in a reasonable runtime consumption. Fortunately, the constraint programming community has developed techniques to reduce the search space: constraints are activated to

[1] We restrict our discussion to such finite domain integer CSPs.

eliminate inconsistent values from variable domains to reach consistency, which propagates the implications to other constraints to trigger the domain shrinking of more variables. This technique can significantly reduce the amount of search efforts, making it is possible to solve some hard problems.

1.2 Constraint Programming in a Nutshell

Constraint programming includes two phases: modeling and solving. The modeling step is to abstract a problem as a CSP and present it as a script, using the language/predicates provided by a constraint programming system (CPS); the solving phrase is to search for the solution(s) in the system. In this section, we employ the Sudoku problem as an example to briefly go though these two steps.

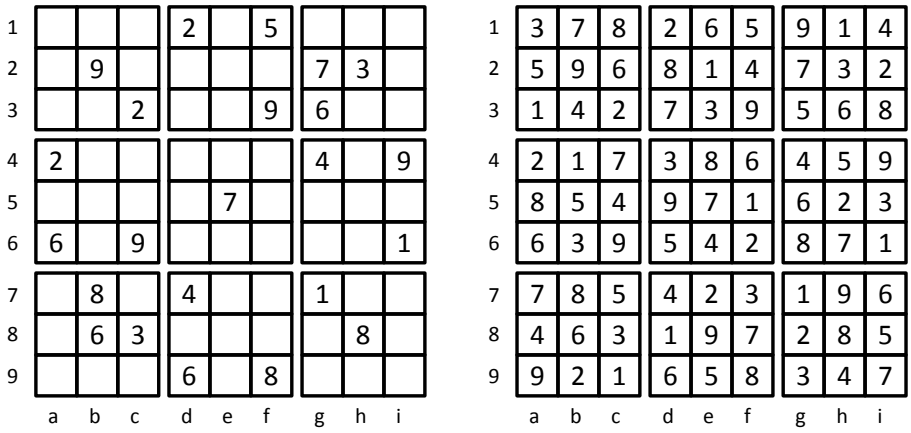

Figure 1.1: A Sudoku Puzzle and its Solution

The Sudoku problem is a puzzle to fill a 9 by 9 matrix with values from one to nine, with the objective that values appear in each row, column and major 3 by 3

block are pair-wise distinct. Usually, the matrix is partially pre-filled with values to ensure a unique solution. Figure 1.1 depicts a Sudoku puzzle setting (left) and its solution (right).

Modeling To model a Sudoku puzzle, we can declare each matrix entry as an integer variable, taking initial values from the integer set $\{1...9\}$. To enforce the rule that variables in each row, column and major block must take pair-wise distinct values, a common `all-different` constraint [25] can be utilized. As for problem decomposition (will explain in solving phrase), a typical strategy is `first fail` [13], a heuristic to take the variable with the smallest domain and then try to assign/remove one of its left values. To search for the first solution, it usually employs depth first exploration.

Describing the main idea to model the Sodoku problem, it is then straightforward to write up a model script in a constraint programming system. Figure 1.2 illustrates the script for modeling Sudoku in the Gecode system [33], an open source constraint programming library developed in C++. In Gecode, a model always inherits from the class `space` and implements the model in its constructor. Additionally, the model must implement a *copy constructor* and a *copy function* to clone fix point computation spaces. This is because Gecode is a system that bases on copying with recomputation for state restoration.

Solving The constraint programming systems solve a CSP through *inference* and *search*. The inference removes the values that cannot appear in solution from variable domains through reasoning. Let take the entry in row 1, column *i* (E_{1i})

```
class Sudoku : public Space {
   public:
      IntVarArray   entries;       /* variables for matri*/
     Sudoku(const int instance[9][9])  : entries(this,  9*9,  1, 9)    {
         Matrix <IntVarArray> m(entries, 9, 9);
       for (int i = 0; i < 9; i++)  {
            distinct( this, m.row(i) );      /* constraints for rows */
            distinct( this, m.col(i) );       /* constraints for columns*/
        }
     for( int I = 0; i < 9; i +=3 )
        for( int j = 0; j < 9; j += 3 )
           distinct( this, m.slice( i, i+3, j, j + 3 ) );  /*constraints for major blocks*/
     for( int i = 0; i < 9; i++ )
          for( int j = 0; j < 9; j++ )
              if( int v = instance[i][j] )
                 rel( this, m(i, j), IRT_EQ, v );             /* prefilled entries*/
     /* Decomposition Heuristic : first fail */
     Branch( this, entries, INT_VAR_SIZE_MIN, INT_VAR_SPLIT_MIN )
    }
    Sudoku(bool share, Sudoku & s): Space(share, s) {
          Entries.update(this, share, s.entries);      /* Constructor for cloning*/
     }
    Virtual Space * copy(bool share)    {
         return new Sudoku(share, *this) ;              /* copying during cloning */
     }
};

int main(int argc, char * argv[])  {
     int instance[9][9] =  {
                                      /* prefilled values */
       }
     Sudoku * root = new Sudoku(instance);
     Sudoku * solution =  DFS( root );  /* pass problem space to search engine*/
     std:cout << solution->entries  << std::endl;
     delete root;  delete solution;
     return 0;
}
```

Figure 1.2: Gecode Script for Modeling Sudoku Puzzle

of Figure 1.1 for an example. For E_{1i}, its original domain is $\{1...9\}$; however, the reasoning entails that values $\{3, 6, 7\}$ cannot appear in any solution since the top-right block has already assigned three entries respectively to 3, 6 and 7. In similar, the values $\{2, 5\}$ and $\{1, 9\}$ can also be removed from its domain by examining

the row 1 and column *i* respectively. Finally, E_{1i} has an updated domain {4, 8}. However, these value removals at E_{1i} can cause other entries similarly shrink their domains. This process may finally solve the problem. But most likely, it will reach a status that none of variable domains can be further shrunken while the problem is not solved, and we call such a status a *fix point*. The fix point signals the insufficiency of inference alone for solving problems, which means that search is necessary.

The search process decomposes the fix point problem into multiple disjoint subproblems so that inference can continue at each subproblem. Let continue the Sudoku problem: if E_{1i} of a fix point has a domain {4, 8}, the problem can be divided into two subproblems: one has E_{1i} assigned to {8} and the other has value 8 removed from the domain of E_{1i}. In both cases, the subproblems are further constrained and thus inference can resume. The inference and decomposition steps alternate until solutions are discovered or the problem is proven non-solvable, and the search process defines an order to visit the subproblems (typically, Depth-First-Exploration is employed to limit memory consumption).

However, inference at a subproblem may turn out to be an inconsistency. For the Sodoku problem, if the variable attempts to assign the E_{1i} with 8, it eventually will reason an inconsistency. An inconsistency signals a false search direction, and the system should *restore* the previous state that E_{1i} has a domain {4, 8} and then try the other alternative (remove 8 from E_{1i} domain). This task is fulfilled by *state restoration* (restoration for short) in a constraint programming system. Intuitively, the restoration can be accomplished by memorizing the whole puzzle setting or undoing the performed reasoning effects etc. Actually, various state restoration

techniques have been developed for building constraint programming systems. Restoration is one of the key components in constraint programming systems; it can significantly affect the system performance and architecture design.

1.3 The Thesis

This thesis is organized as follows: Chapter 1 overviews constraint programming in a nutshell and Chapter 2 recapitulates the main fundamental concepts that are referred throughout this thesis; three mainstream state restoration techniques are reviewed in Chapter 3; Chapter 4 and Chapter 5 respectively presents the idea of recollection and programming restoration granularity; Chapter 6 intensively explains the implementation issues of our developed techniques and empirical evaluation is placed in Chapter 7; lastly, Chapter 8 concludes this thesis.

Chapter 2

Constraint Programming

Constraint programming systems provide the facilities to model and solve CSPs. In this chapter, we briefly explain the techniques and terms that are referred to in constraint programming and its systems.

2.1 Basics

A constraint programming system implements variables and constraints for modeling CSPs, and it provides facilities to solve the modeled problem. For a CSP in a constraint programming system, the conjunction of its variables form a *store* to map their domains; each constraint is implemented as one or multiple *propagators*. A propagator can amplify the store by executing its built-in filtering algorithm to rule out the inconsistent variable values.

A store and its connected propagators form a *state*. Within a state, the store plays as a communication channel for its connected propagators. Specifically, a propagator computes the variable values that are consistent with its constraint, and the store reflects the computation results of the propagator. That is, a propagator entails partial information about values of variables. Furthermore, this entailed partial information may trigger the computation of other propagators that share the same variables with the immediately executed propagator, enforcing more variable values are eliminated from the store to maintain consistency. This process of scheduling propagators for execution is called *constraint propagation* (or propagation for short), which implements the constraint inference as described in Sudoku example.

During the propagation process, if the domain of any variable becomes empty, an inconsistency occurs, and an inconsistent state indicates a search failure; if propagation has all variables fixed, the state represents a solution; if propagation reaches a fix point other than a failure or solution, constraint propagation alone is not able to solve the problem. In the latter case, none of the propagators are able to further reduce the domains of the variables, and search is required to proceed.

2.2 Constraint-based Search

Constraint programming systems conduct search by splitting current fix point problem into multiple more constrained subproblems, the disjunction of which is equivalent to the original problem. This splitting task is called *branching*, a service provided by *brancher*. A brancher branches on a fix point state and then

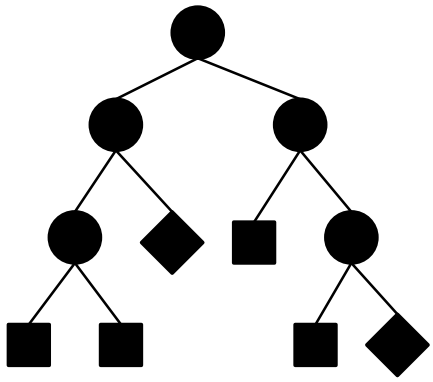

Figure 2.1: A Computation State Search Tree

generates a *choice* of the fix point. The choice contains multiple mutually exclusive constraints[1], which are respectively denoted by *alternatives*. Subsequently, one of constraints in the choice can be committed to the current fix point to lead to a further constrained state where propagation can resume. A choice is *open* if it has an uncommitted alternative; otherwise is *closed*. A state is open if its choice is open, and a state is closed if its choice is closed.

The search process can direct the constraint commitment in a certain order, such as depth-first, to visit subproblems. The constraint propagation, branching, choice commitment steps alternate and create a tree of states, the *search tree*. Search is a complete method, thus these steps continue until a solution is discovered or the problem is proved non-solvable.

In such a search tree, the *root* node is the initial problem; the *current* node is the state that search is exploring; branches are constraints (represented by choice

[1] We confine our discussion to binary choice.

alternatives); internal nodes are fix points and a leaf node is either a solved state or a search failure. Figure 2.1 illustrates a search tree, where solid circles are fix points, squares are failed states and diamonds are searched solution states. In a search tree, the branches and fix point states between the root and current node form a search *path*[2]. The path presents the set of previously committed constraints and reasoned fix point between the root and current state.

Since constraint propagation may reveal an inconsistency to signal a false search direction, search thus requires to restore a previously visited state to switch exploration direction. This service is provided by *state restoration* (restoration for short) in a constraint programming system. For restoration, one key step is to decide the state to restore since an intelligent decision is able to skip the subtrees where solutions cannot exist, and this is essential for a constraint programming system to solve problems efficiently.

Intelligent backtracking has been intensively investigated, and prominent algorithms such as dependency-directed backtracking [32], backjumping [10] and conflict-directed backjumping [24] have been proposed in the community. (For a comprehensive survey on backtracking search algorithms, please refer to [34]). Nevertheless, *chronological backtracking* [11] is the common strategy employed to construct constraint programming systems, and we stick our discussion to the chronological backtracking throughout the thesis.

[2]We focus on sequential search in this paper and therefore there is a single path.

2.3 Restoration

To achieve the restoration of a previously visited state, sufficient information should be stored as search proceeds. The information can be stored in various formats, and these formats determine the way to conduct state restoration.

Algorithm 1 Depth First Search

```
Input: State S, Stack ST
Output: Solution State
 1: while true do
 2:     Log log
 3:     switch (Propagate(S, log))
 4:     case solved:
 5:         return S
 6:     case inconsistency:
 7:         S ← Restore(S, ST)
 8:         if S = NULL then
 9:             return non-solvable
10:         end if
11:         Chunk chunk' ← getTop(ST)
12:         Choice choice' ← getChoice(chunk')
13:         Commit(S, choice', second)
14:         break
15:     case fix_point:
16:         Choice choice ← Branch(S)
17:         Commit(S, choice, first)
18:         Chunk chunk ← Record(S, log, choice)
19:         Push(ST, chunk)
20:         log ← ∅
21:     end switch
22: end while
```

Algorithm 1 summarizes previously described constraint-based search and describes a Depth-First-Search without customizing a specific state restoration technique. In this pseudo-code, the path related information is maintained explicitly using a stack *ST*. Constraint propagation of a state can be performed by calling the operation `Propagate()` (Line 3), which releases a propagation result value. The `switch` statement responds according to the propagation result in relevant code segments. Specifically, the search engine returns the solved state as a solution and it calls the method `Restore` to restore a previously visited open state

to switch search direction when propagation exhibits a failed state. If propagation reaches a fix point, the search engine branches on this state to generate a choice and then commits to the first alternative (Line 8); in the meantime, a *chunk* will be constructed by the method `Record` (Line 18) and then pushed onto the stack *ST*.

In this pseudocode, `Record` and `Restore` form a pair of abstract methods, whose implementation determines the specific restoration technique in use. In the subsequent chapter, we would review restoration techniques through describing the implementations of this abstract method pair.

Chapter 3

Existing Restoration Techniques

In CPSs, a state can be achieved by either memorization or reconstruction. States are memorized by *copying*, which clones each reasoned fix point state. State reconstruction can be achieved by *trailing* and *recomputation*. *Trailing* rolls back previous performed operations, while recomputation redoes the computation work. In this chapter, we present these three mainstream restoration techniques by defining the pair of abstract method *Record* and *Restore* respectively.

3.1 Trailing

A trailing-based constraint programming system maintains a global data structure, *trail*, to accumulate the information to *undo* the operation performed to change states. Conceptually, the undo information should describe how changes hap-

pened to states (e.g. the eliminated values). In practical implementations, the state changing operations are considered as updates of memory locations. If a memory location is updated, its address and old content image is stored onto the trail. This kind of trail is referred as Single-Value trail, which is essentially the technique used in Warren's Abstract Machine [3]. Other trail variants are Time-Stamping and Multiple-Value trail(see [2]). For a comprehensive description on implementing trail in CP systems, one can refer to [16].

In a trailing-based system, it implements the `Record` method to collect operation undo information into the trail (it is the stack *ST* in this context). To trail state changes, it is required to track constraint propagation. In Algorithm 1, a data structure *log* fulfills this task. Specifically, if propagation reasons a fix point, the content of *log* will be wrapped into a chunk and pushed onto the stack *ST*. As for restoration, the fundamental restoration idea is undoing the logged information to restore to previously accessed states and the Algorithm 2 abstracts the main process: first rolls back the operations stored in the current *log* (Line 1); subsequently, access the chunks in stack *ST* in a top-down manner and roll back the information stored in those chunks (accomplished in the **while** loop). This process iterates until it backtracks to the first state which has an open choice (the condition of the **while** loop).

The concept of trailing first appeared in the Warren's abstract instruction set [37] and was implemented in Logic Programming (LP) Prolog. Subsequently, Jaffar introduced constraint into logic programming and laid the foundation for a successor of Prolog, Constraint Logic Programming (CLP) [15]. In fact, most of today's constraint programming systems are constraint logic programming sys-

Algorithm 2 Trailing-based State Restoration

```
Input: State S, Stack ST, Log log
Output: State S
 1: undo(S, log)
 2: Chunk chunk ← getTop(ST)
 3: Choice choice ← getChoice(chunk)
 4: while choice has no uncommitted alternative do
 5:     log ← getLog(chunk)
 6:     undo(S, log)
 7:     Pop(ST)
 8:     if Size(ST) = =0 then
 9:         return NULL
10:     end if
11:     chunk ← getTop(ST)
12:     choice ← getChoice(chunk)
13: end while
14: return S
```

tems that evolved from Prolog and inherit its search facilities such as Ecl^{i}ps^e [1], cc(FD) [36], CHIP [9] and clp(FD) [6] etc; meanwhile, there are also systems that are not built on top of Prolog, like Screamer [30] (Lisp) and ILOG Solver [14], use trailing. Trailing is the dominantly used restoration technique in the community of constraint programming systems.

Trailing has demonstrated its efficiency for solving large problems with weak propagation [27]. However, trailing is concerned with operations to change state and requires to monitor the constraint propagation. This implies that the search facilities is not an orthogonal issue with the other underlying components in a trailing-based system; instead, they are tightly coupled. In such an architecture, it is is of great complexity to implement users customized search algorithms. Moreover, trailing for elaborated data structures can also become quite complex; for the exploration of multiple nodes, it should be accomplished in an interleaved manner to switch between nodes in expensive operations. This however can limit the parallel search, which is essential for solving large problems in modern computer architecture.

3.2 Copying

Copying-based strategy clones an identical state before change and maintains it in memory for direction retrieval. This method offers advantages with respect to expressiveness: multiple states of a search tree are simultaneously available in memory for further exploration, which is essential for programming parallel and users-customized search algorithms. Unlike trailing, copying is concerned with data structures rather than operations. This feature alleviates the coupling between search facilities and the rest part of a system, which potentially simplifies the design and implementation of a CP system.

Copying-based restoration defines the $\texttt{Record}_{copy}$ method to store a copy of each reasoned fix point state in created chunks; the corresponding $\texttt{Restore}_{copy}$ method is straightforward: retrieve the chunk that contains the expected open state and then return; Algorithm 3 illustrate the pseudocode of $\texttt{Restore}_{copy}$ method.

Algorithm 3 Copying-based State Restoration

Input: State *S*, Stack *ST*, Log *log* (ignored)
Output: State *S*

```
 1: delete S
 2: Chunk chunk ← getTop(ST)
 3: Choice choice ← getChoice(chunk)
 4: while choice has no uncommitted alternative do
 5:     if isEmpty(ST) then
 6:         return NULL
 7:     end if
 8:     chunk ← Pop(ST)
 9:     choice ← getChoice(chunk)
10: end while
11: chunk ← getTop(ST)
12: return S ← getState(chunk)
```

A system that features garbage collection already provides the essential functionality to support copying (garbage collection was first presented as a technique

in [4], see [17] and [8] for further explanation), on account of the intensive memory allocation and deallocation. The Mozart system [23] is designed for the programming language Oz [31]; it was the first constraint programming system that employed the copying-based state restoration scheme. In Mozart, a state is implemented as first-class *computation space* (space for short) [28], which encapsulates variables, propagators as well as branchers at that state. The system provides an operation `clone()` to duplicate a fix point space. These efforts together facilitate the programming of a search engine, and [26] presents computation space as abstractions with which users can program search engine at a high level.

Copying is more memory intensive than trailing, while its intensive memory management can introduce a factor of hurting the runtime performance. Meanwhile, main memory page fault is possible to occur as problem sizes increase, which may significantly prolong the runtime. Nevertheless, the experimental comparisons between trailing and copying have demonstrated that copying causes neither memory nor runtime issues for small and medium size problems; copying alone for large problems with deep search tree is unsuitable: a majority of runtime will be spent on garbage collection while memory requirement is prohibitive [27].

3.3 Recomputation

The idea of recomputation is straightforward: any state in the path can be computed from the root state, using the information that is stored in the path. Recomputation-based restoration implements $\texttt{Record}_{recomp}$ to store the generated choices and committed alternatives; the corresponding $\texttt{Restore}_{recomp}$ exploits the path to

conduct recomputation.

The Mozart/Oz system conducted pioneer work on recomputation; it memorizes the committed choice alternative at each fix point. Restoration then requires stepwise recomputation: first branch on root state to re-generate the choice and commit to the old alternative to propagate to next fix point; then repeat branching to generate choice and committing to old alternative to reason fix point. This process is repeated until the expected open state is restored. This naive method can be computation intensive; the subsequent batch recomputation [5] explicitly maintains the committed constraints in a global data structure. Restoration then can be implemented by consecutively committing all necessary constraints in single round then propagate to compute the open state, as illustrated in Algorithm 4.

Algorithm 4 Recomputation-based State Restoration

Input: State *S*, Stack *ST*, Log *log* (ignored)
Output: State *S*

```
1:  delete S
2:  Chunk chunk ← getTop(ST)
3:  Choice ch ← getChoice(chunk)
4:  while ch is not an open choice do
5:      Pop(ST)
6:      chunk← getTop(ST)
7:      ch← getChoice(chunk)
8:  end while
9:  S ← getRootState(ST)
10: for each chunk ∈ ST do
11:     choice ← getChoice(chunk)
12:     Commit(S, choice, oldAlternative)
13: end for
14: return Propagate(S)
```

Gecode generalizes batch recomputation by combining copying, which leads to *fixed recomputation* and *adaptive recomputation* [28]. Fix recomputation places a state copy every d exploration steps, where d is a constant value called *copying distance*; recomputation then can start from the last state copy in the path. This effort aims at weakening computation intensity. Adaptive recomputation further

extends fixed recomputation: if recomputation from S_1 to S_2 occurs, an additional state copy will be put in the middle place between S_1 and S_2 to tentatively shorten future recomputation distance. Adaptive recomputation has been demonstrated as one of the most competitive restoration technique in the community [27], and it is supported by the Mozart/Oz and Gecode systems. In addition, other techniques such as Last Alternative Optimization [12] have been introduced to optimize the performance of recomputation variants.

Since only choices/alternatives are required to store, the memory for supporting recomputation can stay almost constant, even for large problems. It however may introduce runtime cost as a result of redundant computation for state restoration. If the computation of a problem is expensive, then recomputation alone is usually not suitable for solving the problem, especially for the one with deep search tree and extensive search failures. The combination of recomputation and copying strives to balance the memory and runtime cost following a certain strategy, which is usually effective enough to configure an acceptable performance.

Chapter 4

Recollection

In this chapter, we propose an alternative restoration technique that we call *recollection* for building constraint programming systems. This technique memoizes the variables that were modified during constraint propagation; restoration then can be accomplished by updating a state at high level of the search tree downwards, using the memoized variables. Section 4.1 explains the motivation for proposing recollection and its main idea; Section 4.2 visualizes the difference between trailing and recollection; Section 4.3 and Section 4.4 respectively define the `Record` and `Restore` methods of recollection.

4.1 Motivation

In the previous chapter, we have thoroughly examined the mainstream restoration techniques: trailing, copying and recomputation. As explained, recomputation maintains branchers generated constraints/alternatives to compute from the root or other higher search level states. However, recomputation conducts redundant constraint propagation, which may generate runtime penalty as a result of intensive propagator scheduling and expensive propagators' built-in filtering algorithms.

To avoid the repetitive computation, copying clones each visited fix point state in a coarse-grained manner, while trailing records the changes between states. Recall the statement in Chapter 2 that the aim of constraint propagation is eliminating inconsistent variable values; therefore, it should be feasible to memoize the modified variable domains and use them to conduct state restoration.

4.2 Characteristics

Intuitively, both recollection and trailing intend to store the part of states for restoration, they however approach restoration in opposite directions, as illustrated in Figure 4.1. S_0 is the root state and the solid triangle is a failed subtree; a search failure is encountered at state S_f. In a depth first search strategy, state S_r should be restored to switch search to state S_n (bold line and circle). In a trailing-based system, the restoration first rolls back the changes between S_t and S_f (represented by directed line R_1) to restore S_t; subsequently, the performed operations between

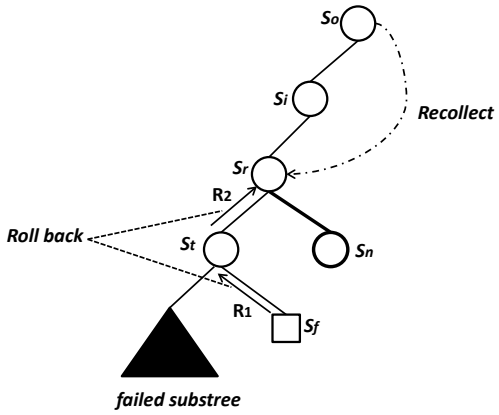

Figure 4.1: Visualization of Trailing and Recollection

S_t and S_r is further undone to finally restore S_r. This process demonstrates: (1). trailing launches restoration on the failed state and consecutively rolls back state changes until it restores the target state, which may internally go through many tentative states; (2). the step-wise restoration can be costly if the search strategy intends to jump between states within the search tree; (3). trailing proceeds in a bottom-up direction.

By contrast, recollection updates a state at higher level (the root state in this illustration) downwards to restore the state S_t, as shown in the dashed directed line. The recollection process reveals: (1). recollection conducts restoration in a single step to avoid the reconstruction of internal states; (2). recollection can achieve the jumping between states easily; (3). recollection proceeds in a top-down fashion.

4.3 The Record Method

Algorithm 5 Definition of $Restore_{recollect}$ Method

Input: State *S*, Choice *choice*, Log *log*
Output: Chunk *chunk*
1: Domain *doms* $\leftarrow \emptyset$
2: **for each** *var* $\in$ Variables(*S*) **do**
3: **if** isChanged(*var*) **then**
4: *doms* $\leftarrow$ *doms* $\cup$ recordDomain(*var*)
5: **end if**
6: **end for**
7: **return** Chunk(*choice*, *doms*)

The $\texttt{Record}_{recollect}$ of recollection implements to memorize the changed variables, and Algorithm 5 describes the process. Specifically, each chunk includes an object of *dom* class *Domain*. Variables are sequentially scanned; if a variable was updated during constraint propagation, its domain will be copied into the object *dom* (Line 4). Lastly, the *dom* will be wrapped with the generated choice to create a chunk.

4.4 The Restore Method

The $\texttt{Record}_{recollect}$ method is defined to store the changed variable only, which implies that a variable domain shall not be copied if it stays unchanged. This definition scatters the empty entries across chunks, i.e. a chunk usually does not include the domain of all variables. However, recollection aims at conducting state restoration in a single step.

To guarantee the single-step restoration, recollection should collect the correct variable domains across the stack of chunks, and we name this process as *domain collection*. Domain collection is searching the chunk stack *ST* in a top-down direction to identify the first variable domain entry in the chunks and use it to re-

store. In principle, the domain collection can be implemented in variable-centered and chunk-centered two flavors.

Algorithm 6 Definition of $Restore_{variable-centered}$ Method

```
Input: State S, Stack ST, Log log
Output: State S
 1: delete S
 2: S ← root_state
 3: Chunk chunk ← getTop(ST)
 4: Choice ch ← getChoice(chunk)
 5: while ch is not an open choice do
 6:     Pop(ST)
 7:     chunk← getTop(ST)
 8:     ch← getChoice(chunk)
 9: end while
10: for each var ∈ Variables(S) do
11:     Integer index ← Size(ST) - 1
12:     chunk ← getChunk(ST, index)   /*scan from stack top*/
13:     while Domain(var) ∉ chunk do
14:         index ← index - 1    /*move to next chunk location*/
15:         chunk ← getChunk(ST, index)
16:     end while
17:     Reconstruct(var, chunk)
18: end for
19: return S
```

Variable-Centered Collection The variable-centred approach, shown in Algorithm 6, picks one variable *var* at a time and searches the stack *ST* in a top-down direction (moving in the search tree in a bottom up direction!) for the first chunk that contains its domain (Line 7 to 10), and then reconstructs it (Line 11). In the worst case, this approach scans the entire stack for each of the M variables and thus conducts $N \times M$ chunk access operations for a restoration, where *N* is the current stack size; a fairly weak propagation problem can exhibit such worst-case behavior. On the other hand, in the presence of a strong propagation problem, only the top-most chunk (bottom-most node) may contain all variables and thus there is no need to even access any further chunks on the stack.

Algorithm 7 Definition of $Restore_{chunk-centered}$

```
Input: State S, Stack ST, Log log
Output: State S'
 1: delete S
 2: S ← root_state
 3: Chunk chunk ← getTop(ST)
 4: Choice ch ← getChoice(chunk)
 5: while ch is not an open choice do
 6:     Pop(ST)
 7:     chunk← getTop(ST)
 8:     ch← getChoice(chunk)
 9: end while
10: Integer index ← Size(ST) - 1
11: while index ≥ 0 do
12:     Chunk chunk ← getChunk(ST, index)
13:     for each var ∈ Variables(chunk) do
14:         if var has not been reconstructed then
15:             Reconstruct(var, chunk)
16:         end if
17:     end for
18:     index ← index - 1
19: end while
20: return S
```

Chunk-Centered Collect By contrast, the chunk-centered approach, depicted in Algorithm 7, scans *ST* in a top-down manner (moving bottom-up in the search tree), and keeps track of reconstructed domains. For each chunk, all memoized variables are scanned and a variable domain is reconstructed, whenever the domain of the variable has not been reconstructed yet (Lines 6–14). This query scheme accesses the stack once in a restoration, regardless whether the problem exhibits weak or strong propagation. To accelerate the variable domain checking in chunk-centered scanning, we introduced an index, which will be thoroughly explained in the chapter of discussing implementation issues. Our experimental results demonstrate that the indexed chunk-centered query generally has a slight runtime advantage over variable-centered restoration. The experiments of the next section have been conducted using the indexed chunk-centered query.

4.5 Variations

Our discussion on recollection so far assumes that a single state is maintained at the root of the search tree and that restoration will begin from scratch at the root state. Similar to recomputation, we observe that this approach incurs a significant runtime penalty. Analogous to recomputation, we extend recollection to the variants of *fixed recollection* and *adaptive recollection*, which place state copies in the search tree in the way that has been explained in chapter 3.

Chapter 5

Programming Restoration Granularity

In this chapter, we propose to program state restoration granularity to achieve customized state restoration scheme, striving for even better state restoration performance (consume even less runtime or memory). Section 5.1 gives the statistics that motivate our investigation into programming restoration granularity; Section 5.2 presents the granularity of various restoration strategy and Section 5.3 proposes to program restoration granularity as an alternative approach for developing state restoration.

5.1 Motivation

For solving a problem, the constraint propagation characteristics can evolve as search proceeds. For example, a problem can impose a strong propagation at its first search steps, but it may become rather weak propagation when search approaches the bottom part of the search tree. Similarly, search failures can happen intensively at the bottom part of the search tree in one problem, but for another problem the search failures may distribute evenly. Table 5.1 illustrates the search tree statistics of four problems that explore for the first solution. The Queens problem is modeled by either a set of disequality constraints or three global constraints of the "all-different constraints" family (denoted as Queens-S). The size of Queens problem is 200; the sizes of both Knights and Sport-League are 22.

In this table, the column *failures* counts the total number of failures during search; *first* signals the tree level where the first search failure emerges, while *peak* is the peak depth of the search tree. [1, *first*) accumulates the number of failures occurs between the root and the *first* search tree level, while [*first*, *peak*] records the number of failures between *first* and *peak*.

Problem	*failures*	*first*	*peak*	[1, *first*)	[*first, peak*]
Queens	146,838	164	200	0	146,838
Queens-S	146,838	164	200	0	146,838
Knights	19,877	386	451	0	19,877
Sport-League	1,035	62	249	5	1,030

Table 5.1: Search Tree Statistics of Problem Search Trees

From these search tree statistics, we perceive that the emergence of the first fail-

ure can be an important signal for intensive search failures. In such circumstance, if copying is employed as the restoration scheme, the space copies maintained between the root and the $first$ (exclusive) level cannot contribute while may occupy a substantial amount of memory. This observation exemplifies that an ideal restoration should be application-specific.

5.2 Restoration Granularities

To support a restoration technique, a particular format of information should be stored. This information has a certain granularity, which we call *restoration granularity*. In the following paragraphs, we respectively discuss the restoration granularities of developed state restoration techniques.

- **Copying.** Copying is coarse-grained since it stores all information of visited states; it come at an expense of substantial memory occupation. Although copying causes neither runtime nor memory issues for small and medium size problem, its potential intensive memory management may introduce runtime penalty. Nevertheless, copying can be combined with other techniques such as recomputation and recollection, which can significantly improve the their runtime performance with a reasonable memory investment.

- **Trailing.** Trailing is finer-grained than copying since it records the changes to states. Trailing has been proved efficient for the problems imposing weak propagation. However, trailing can be quite complex for complicated data structures. Moreover, its implementation requires to couple search facili-

ties with constraint propagation, which potentially increases the design and implementation complexity of a constraint programming system as well.

- **Recomputation.** Recomputation does not store any specific information with respect to visited states. Instead, it keeps the constraint commitment instructions (meta-information) that were commanded to search. This approach consumes almost constant memory, but its runtime may be dragged rather significantly if a problem is computationally expensive.

- **Recollection.** Recollection is also finer-grained than copying that it logs the changed variable domains, consuming less memory than copying; it has demonstrated to improve the runtime performance than recomputation for the problems that impose expensive computation with deep search trees.

5.3 Programmable Restoration

Most constraint programming systems employ a specific restoration technique such as trailing in constraint logic programming systems. The aim of a programmable state restoration was facilitated by the development of computation space. The computation space allows users to program the places where a space clone should be put, and it was intensively used to combine with recomputation. However, the space is a rather coarse data structure, while the recomputation stores only instructions (constraints).

As we have explicitly examined in previous section, each restoration technique has its own advantages and limitations. Trade-offs usually exist if a constraint

programming system employs a specific restoration strategy for solving all kinds of CSPs. To alleviate this trade-off, we should facilitate the customization of the restoration by users, which requires to store information of various granularities as search proceedsl; this is *programming restoration granularity*. As an initial step, we implement a prototype to address the limitation exposed in Table 5.1 (deep search with intensive failures at bottom part). The detail specification with respect to the prototype is given in chapter 6.3.1.

Chapter 6

Implementation

In this chapter, we respectively explain the key implementation issues of recollection and programming restoration granularity. In Section 6.1, we briefly overview the structures that are highly relevant to our proposed techniques in the target Gecode system; Section 6.2 presents how we address the issues to realize recollection, and Section 6.3 discusses the implementation of programming restoration granularity.

6.1 The Gecode System

The Gecode system is an open source C++ constraint solver. This section introduces the computation space and its provided key operation interfaces as well as its internal structures in Section 6.1.1; Section 6.1.2 depicts the pseudo-code of a

DFS search engine and Section 6.1.3 outlines the profile of the class Edge in the Gecode system.

6.1.1 Computation Space

A computation space encapsulates the store, propagators and branchers. It provides a `status`() method to conduct constraint propagation and returns a value of `SpaceStatus` type. Inside a space, the constraint propagation is implemented by scheduling propagators for execution. A fix point space can be cloned by `clone`(). `choice`() generates the choice of a fix point space, and its contained constraints can be committed to its spaces (or the equivalent clones) by the `commit`() method.

The propagators of a computation space inter-connect as a chain, and propagators are picked for execution to implement constraint propagation. Meanwhile, branchers are also chained, and the head brancher is always called for branching until its subscribed variables are all fixed.

Propagator. In the Gecode, a class `Propagator` has been defined to declare a set of virtual methods including `propagate`(). Every propagator is defined as a subclass of `Propagator` and should implement its filtering algorithm in `propagate`() method body. A propagator subscribes a set of variables and it can be scheduled if one of its subscribed variables has a domain change; the propagator scheduling policy can be specified by *modification event* and *propagation condition*. For a comprehensive description of the constraint propagation design

in the Gecode, please refer to [29].

Brancher. In the Gecode, a class `Brancher` has been defined as the superclass, and every brancher should inherit from it and define its reserved virtual methods. Of these methods, the `choice()` specifies the way to create a *choice* and the `commit()` injects the constraints to spaces. A brancher generally subscribes an array of variables and it will be disposed if it cannot branch any more.

6.1.2 Search Engine

Program 8 Search Engine

```
while true do
    switch (Status(space))      /* query space status */
    case fixpoint :
        Choice ch←Choice(space)      /* return solution space */
        Push(Edge(ch . . . ))
        Commit(space, ch)      /* commit a constraint to space */
    case solution:
        return space      /* return solution space */
    case failure:
        if not adjust(. . . ) then
            break      /* The problem is not solvable */
        end if
        Restore(. . . ) {
          Recomputation code      /* programmed recomputation */
        }
    end switch
end while
```

The Gecode is designed that the search engine interacts with computation space, while computation space encapsulates the implementation detail with respect to constraint propagation, branching etc. The search engine responds according to the enquired computation status: if the space turns out to be stable (a fix point), the search engine will back up relevant information and then continue searching; if the space is recognized as a solution, the search engine will directly return the

space; if the space is evaluated as a failure, the search engine will first request to adjust search direction by calling `adjust()` [1] and then enter the code segment where recomputation is defined. Program 6.1.2 illustrates a Depth-First search engine.

6.1.3 Class Edge

Program 9 Class Edge

```
class Edge    {
      Space * _space;     /* Space copy */
      Choice * _choice;    /* fix point generated Choice */
      unsigned int _alt;    /* committed choice alternative */
      vector _doms;    /* variable domains */
   public:
     Edge(Space * s, Choice * c, vector<int> _doms):    {
      _alternative = 0;     /* commit to the first alternative */
      . . .     /* other initialization statements */
     }
     . . .
   }
```

In the Gecode system, a class *Edge* is defined to track the constraints that were committed to visit fix points spaces during search. Each Edge object was created to memorize a generated choice and the committed alternatives, and all such objects are pushed onto a stack structure *Path*.

Program 9 outlines the profile of the class Edge. In this class, the *Choice* encapsulates the two constraints that are generated by branching on the current fix point, and they are respectively represented by 0 and 1. The committed constraint alternative is denoted by the integer variable *_alt*. In an Edge object, the Choice is a compulsory information whereas the space copy *_space* is optional. This flexibility enables the change of state restoration paradigm: if a space copy is

[1] It will return a Boolean **false** if another search direction is impossible.

placed in each Edge object, the system will work in a copying-based scheme; if none of Edge objects stores a space copy (except root), the system will conduct a recomputation-based restoration; hybrid scheme can be obtained by placing space copies occasionally.

6.2 Implementing Recollection

This chapter is concerned with key implementation issues with respect to recollection in the Gecode system. Section 6.2.1 describes the schemes for accessing variables within computation spaces; Section 6.2.2 explains the way to detect the changed variables during constraint propagation; Section 6.2.3 sketches the memory management policy to support recollection; an indexed domain collect scheme is illustrated in Section 6.2.4, and Section 6.2.5 describes the detail to reconstruct a variable.

6.2.1 Variable Access

The Gecode system is a layered architecture, and variables are not exposed to other components except propagation and branching. Recollection requires to access variables and thus we should first address the issue of variable access. We cope with this issue by utilizing branchers: introduce a set of virtual methods to class `Brancher` as interfaces and define these methods to implement recollection. Figure 6.1 illustrates the introduced main methods.

For the names of introduced methods, those with a prefix `logRanges` implement the variable memorization function and four approaches as have been developed. Specifically, the `unary` means the memory is linearized compares with `binary` implementation (further explained in Section 6.2.3); `sparse` means that the problem entails weak propagation and we tackle this situation to retain performance. The method `restore`() gives the interface to variable constructions. For the definitions of these virtual methods, we realize them in the template class `ViewBrancher`, a subclass of class `Brancher`.

```
virtual void logRanges_binary( *** );

virtual void logRanges_binary_sparse( *** );

virtual void logRanges_unary( *** );

virtual void logRanges_unary_sparse( *** );

virtual void restore( *** );
```

Figure 6.1: Introduced Virtual Methods for Class Brancher

We extend the class `Space` to define two methods `copyVariable_unary` and `copyVariable_binary`, which aim at coordinating the chains of branchers in spaces to copy all variables. Restoration through multiple branchers is accomplished by the method `restoreVars`() in the class `Space`. Note that all these methods are declared as virtual methods. This is because a problem is modeled in a subclass of `Space`, making use of dynamic polymorphism.

One additional issue is ensuring the completeness of memorized variables since a brancher typically subscribes a subset of the variables in a computation space. To address this issue, one can simply introduce an extra brancher or multiple branch-

ers at the tail of the original brancher chain to guarantee a complete monitor of the variables to track. Figure 6.2 partially depicts the script for modeling Sport-League problem. The Sport-League problem is modeled by three integer arrays *home*, *away* and the *game*, and the original script branches on the *game*. We extend to introduce two additional branchers to track the other variable arrays, as denoted in the figure. Note that, the additionally introduced branchers appear at the tail of the branchers chain and they affect neither the search tree shape nor fix point computation; meanwhile, they cause a negligible memory consumption.

```
class SportsLeague : public Script {
  protected:
      const int teams;      ///< number of teams
      IntVarArray home;     ///< home teams
      IntVarArray away;     ///< away teams
      IntVarArray game;     ///< game numbers

  public:
      SportsLeague(const SizeOptions & opt)  {



      branch(* this, game, *** );
    /* Extra branchers created, never branch */
      branch(* this, home, *** );
      branch(* this, away, *** );
  }
```

Figure 6.2: Variable Accessing via Extra Branchers

Overall, we implement branchers to fulfill the services of accessing variables. This scheme follows the target system layer architecture and eases the coding efforts.

6.2.2 Variable Change Detection

A key implementation issue in method $\texttt{Record}_{recollect}$ is to identif the changed variables in the process of reasoning fix points. Fortunately, our chosen implementation platform, Gecode, provides an abstraction called *advisor* [19], which facilitates our implementation of recollection significantly.

Advisors are introduced in the Gecode to inform variable changes and advise constraint propagation. An advisor belongs to a propagator and *subscribes* to a variable of its propagator; it can be defined to store domain change information that is need by propagation engine. Whenever a variable changes, the `advise`() method of the advisor's propagator is executed with the advisor as argument.

We introduce a Boolean variable `changed` into the template class `VarImp`, which defines the method `advise`(). This class `VarImp` is the superclass of `IntVarImp`, and each variable implementation is an object of `IntVarImp`. This inheritance ensures that every instantiated `IntVarImp` object will contain a Boolean member variable `changed` which is initialized to a Boolean **false** value; on the other hand, the code of `advise`() method will set the Boolean variable to a **true** value. These features assist to recognize the changed variables during constraint propagation, and all technique detail is specified in the `core.hpp` file of the source package.

6.2.3 Memory Management

The Gecode's *memory manager* is centered on spaces, but the created chunks live outside spaces. To store the copied variable domains, a proper memory management is necessary. Our prototype explored two options. The first option allocates memory incrementally; it performs a memory `new`/`delete` operation for each variable. By contrast, the second approach calculates the exact memory to occupy and then allocates memory once for a chunk. Conceptually, we take the second approach as a linearized version of the first memory management policy, as marked by dashed line in Figure 6.3 where the table of *offset* records the starting position of the relevant variable domain.

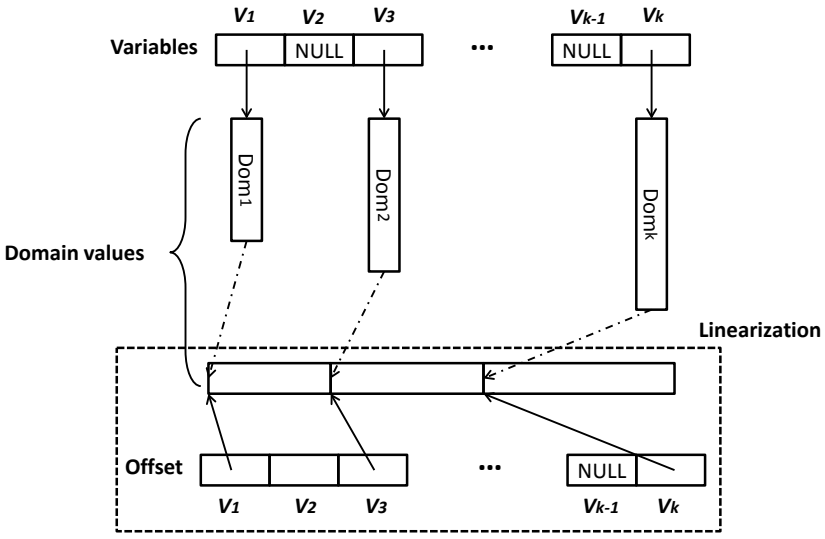

Figure 6.3: Memory Management for Recollection

Our experiments reveal that the first approach is marginally more runtime efficient for problem with weak propagation, while the second approach is more suitable

for problems of intensive variables with strong propagation. In our experimental prototype [21], we can switch between the two alternatives by setting a compile-time flag; we use the first memory policy in all experiments reported in the next section.

6.2.4 Indexed Collection

We explained both variable-centered and chunk-centered collection in Chapter 4. To facilitate the chunk-centered collection scheme, we introduce an index in this section.

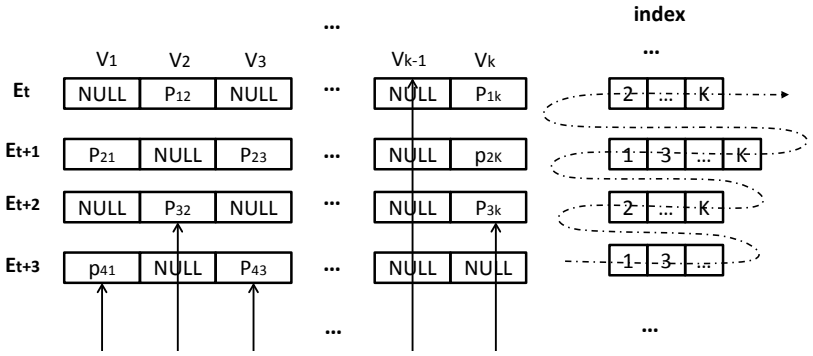

Figure 6.4: Index-based Domain Query

As shown in Figure 6.4, the directed lines visualize first appearance of variable domains across chunks. In the case that a large proportion of `NULL` domain entries exist in chunks, it is less efficient to scan chunks to collect. To accelerate the access, an index structure can be created to map the variables whose domains are stored in current chunk. Thus, the collection can be accomplished by scanning the

index instead as demonstrated by the dashed directed line.

6.2.5 Variable Reconstruction

An *interval* has a upper boundary and lower boundary values, and it represents that all values between the two boundaries are consistent with the constraints of the problem, and the domain of a variable usually consists of multiple intervals rather than a single interval. In the Gecode system, an interval is implemented as a *range* structure and multiple ranges are inter-linked as a *chain*, as illustrated in Figure 6.5. In addition to the chain structure, a variable implementation also maintains an extral range at the head of the chain (marked as `dom-info` in the illustration). This head range stores the maximum and minimum values of the integer domain, aiming at fast access during constraint propagation.

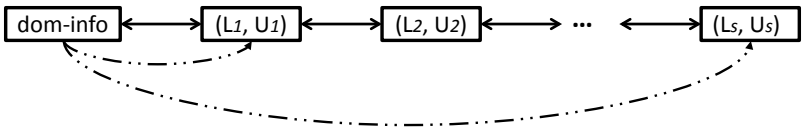

Figure 6.5: Integer Variable Implementation

The recollection requires to update variable domains by defining `Reconstruct` (invoked in Algorithms 6 and 7). Suppose a variable domain originally contains a chain of length L, whereas restoration would adjust this chain to a new length of R. To achieve such an adjustment, we have two alternatives. The first is to destroy the old chain and then rebuild. This is a straightforward solution, but it may cause intensive garbage collection, especially when dealing with long chains; therefore,

this method generally fits short and medium length chains better. The second approach is to tailor the original chains by either trimming or extending to fit in a new requirement. In our prototype, both approaches have been implemented and they are configurable through a compile-time flag. In our latter evaluations, we employ the second approach.

Apart from the way to reconstruct variable chains, a few other implementation issues should be highlighted:

- **Domain Bounds**. The range `dom-info` stores the maximum and minimum values of a variable domain; it should be updated correctly in variable reconstruction to ensure the correctness of the state restoration.

- **Single Range**. The range `dom-info` is to designed to track the global lower and upper boundary values of a variable domain; however, this range will be used to represent an entire domain instead if the variable domain contains a single range. This is an optimization that the Gecode adopts and implements, and attentions should be paid to this observation when recollection reconstructs a single range variable; otherwise, restoration space would encounter segmentation fault during the following search steps.

- **Variable Assignment**. The Gecode system employs an event-based mechanism to schedule propagators, and a propagator becomes subsumed when all its monitoring variables become assigned. The subsumed propagator will automatically be disposed from its computation space, and the system internally provides the service to cancel the subscriptions between propagators and variables. To guarantee such correct cancellations, recollection

should generate an event to schedule the subscribed propagators when a reconstructed variable turns out to be assigned. This objective is achievable by calling the method `eq()` of the variable implementation.

6.3 Programming Restoration Granularity

This section is concerned with the implementation issues of programming restoration granularity. We first implement a prototype in Section 6.3.1; Section 6.3.2 proposes to use aspect-oriented programming to build a more flexible restoration component for programming restoration granularity.

6.3.1 A Prototype

For restoration techniques, recomputation has an *optimistic* assumption [27] that few search failure will occur; by contrast, copying is *pessimistic* in a sense that every node need restoration. It is cheap to store constraints in terms of both runtime and memory, but the restoration requires to recompute; copying is memory expensive, but it avoid computation.

Table 5.1 lists the problems that have failures intensively occurred within a certain area of the search trees. To deal with the skewed failure distribution, a straightforward method is to store restoration information of difference granularity as search proceeds and then restoration adapts the stored information. The statistics clearly shows that search can keep optimistic until the occurrence of the first failure, and

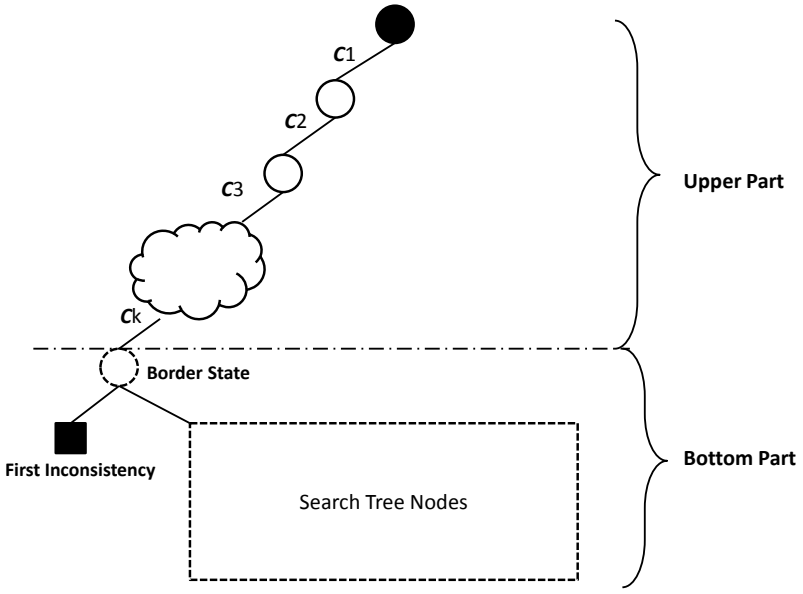

Figure 6.6: Programming Restoration Granularity Prototype

we can build a prototype of programming restoration granularity by taking serious of the emergence of the first search failure.

In the prototype, we integrate copying, recomputation and recollection, and we take the search tree level where the *first* failure emerges as a *border level*. This border level horizontally divides the search tree into *upper* and *bottom* two parts. For search in the upper part of the search tree, nothing but constraints are stored. When search proceeds beyond the border level, both copying and recollection will be activated to collaborate: the placing of a coarse-grained state copy alternates n finer-grained recollection explorations (n is configurable and takes a default value *eight*).

As a result, the system needs to switch between restoration code segments. Specif-

ically, when a state in upper part of the search tree should be restored, the system switches to recomputation. When a state in the bottom part is to restore, the system first attempts to recollect from the nearest state copy; if this effort fails, it then recomputes to restore the border state and then recollect. Intuitively, we can take an optimization measure by maintaining a border state or consolidating the recomputation and recollection.

To meet the demands of switching among restoration techniques, we naively used a signal variable *ffDepth* to track the first failure depth in the search tree. However, this signal couples tightly with a specific program. Suppose one redefines the scheme to switch restoration techniques, the code is quite likely to change, probably drastically. Therefore, it is of great significance to propose a more flexible and modular design to program restoration granularity.

6.3.2 Program as an Aspect

As clarified, restoration is actually determined by the granularity of the stored information; the information is stored at exploration steps while exploited in state restoration. This observation claims that the stored restoration information *cuts across* the two abstractions: search exploration and state restoration.

In developing applications, the occurrence of *crosscutting* abstraction is not rare; transactions, security-related operation, logging etc all exemplify crosscutting abstractions. To facilitate the programming of crosscutting abstraction, aspect-oriented programming [18](AOP) provides solutions to encapsulate a crosscutting abstraction as an *aspect*. The implementation of an aspect mainly consists of two

tasks: *advice* and *pointcut*. An advice is a means of specifying the code to run at a place, while a pointcut determines the matches of executing specified advice at the place. We propose to engineering the restoration information as an aspect in a constraint programming system, striving for a more flexible implementation of programming restoration granularity.

Chapter 7

Evaluation

This chapter conducts empirical evaluation of our proposed and implemented techniques and prototype. Section 7.1 specifies the hardware and software as well as benchmark problem details for the evaluations; Section 7.2 mainly intensively compares the performance between recomputation and recollection; Section 7.3 extends copying into comparison and Section 7.4 studies the performance of the programming restoration granularity prototype.

7.1 Configuration

We used an Intel Core 2 Quad processor PC system, running the Ubuntu operating system 11.10 with four Gigabyte main memory. We built our prototype [21] on top of the Gecode system of version 3.7.3 [33], which also served as the reference

instance for comparison. Each collected runtime[1] value is an arithmetic mean of 20 runs with a variation coefficient less than 2%; memory numbers are the peak memory occupation.

As benchmarks, we used Finite Domain Integer and Boolean problems. They were selected to cover a wide spectrum of constraints, spawn a varying number of propagators and impose different propagation intensity. They cover first, all and best (branch-and-bound) solution search. We limit to the problems included in the Gecode repository, and stick to the configuration of propagation consistency level, branching strategy etc configured in the respective problems scripts.

Problem	Sols	Propagators	Propagations	Nodes	Failures	Depth
Queens(100)	one	14,850	16,821	138	22	96
Queens-S(100)	one	3	428	138	22	96
Magic-Square(5)	one	15	2,292,251	144,471	72,227	33
Sport-League(22)	one	1,199	207,066	2,273	1,035	249
Black-Hole	one	742	986,542	5,284	2,631	47
BIBD	one	9,693	912,464	2,625	1,306	968
Knight(22)	one	1	74,610	40,184	19,877	451
Pentominoes	one	81	6998	143	64	27
Alpha	all	21	136,179	14,871	7,435	49
Langford-Num	all	37	22243	303	149	17
Golomb-Ruler(10)	optimal	39	2,760,799	39,875	19,928	33
Ind-Set	optimal	21	101,317	29,849	14,895	40

Table 7.1: Benchmark Problem Search Trees Characteristics

The set of selected benchmark problems are: the Queens problem modelled by either a quadratic number of disequality constraints or three global constraints that generalize *all-different*; the magic-square puzzle of size 5; a round tournament problem with 22 teams; the black hole patience game; Balanced Incomplete Block Design (BIBD), the knights tour problem of size 22; the Pentominoes problem; the Alpha crypto-arithmetic puzzle; the Langford's number problem with 3 by 9

[1] We take wall clock time in this work.

values and; Golomb-Ruler problem of size 10 and the problem of independent sets in graph (Ind-Set). Table 7.1 lists the characteristics of these problems, where the *propagations* are the numbers collected when using adaptive recomputation for restoration with default argument settings. For more detail information, please refer the source modeling scripts in [21], and for the original scripts, refer to the Gecode distribution [33].

7.2 Recomputation and Recollection

Adaptive recomputation generally exhibits superior performance compared with other recomputation schemes [27]. Similarly, adaptive recollection is generally the most competitive recollection variant. We therefore first focus on a direct comparison between adaptive recomputation and adaptive recollection, fixing the copying distance to *eight* in both cases.

	Recomputation		Recollection	
Problems	Time(ms)	Mem(KB)	Time (ms)	Mem(KB)
Queens(100)	16	4,301	15	4,663
Queens-S(100)	1	240	2	602
Magic-Square(5)	579	63	653	73
Sport-League(22)	352	7,710	331	7,937
Black-Hole	535	1,927	508	1998
BIBD	573	4,678	575	4784
Knights(22)	1,858	4,460	1,704	4,592
Pentominoes	20	1,158	19	1,173
Alpha	55	45	66	50
Langford-Number	13	132	13	135
Golomb-Ruler(10)	556	69	547	70
Ind-Set	58	41	68	43

Table 7.2: Comparison of Recomputation and Recollection

Table 7.2 depicts the experimental results. Neither recomputation nor recollection

can demonstrate a consistent performance advantage over all problems. Specifically, recollection hardly improves the runtime of the problems with shallow search trees and limited number of failures such as Pentomonies and Langford-Number; or even leads to an inferior runtime, as in Alpha and Magic Squares. Recollection is competitive for finite domain integer problems with deep search trees and intensive failures such as Sport-League, Golomb-Ruler and Knights. Meanwhile, it is important to note that the runtime improvement is afforded using a small amount of additional memory in these cases.

Boolean problems generally do not benefit from recollection, even though a Boolean problem would instantiate a rather deep search (BIBD) or encounter intensive failures (Ind-Set). The information contained in previously memoized singleton domains is not dense enough to compete with their recomputation via re-running the respective propagation algorithms.

We conducted the comparison with a specific copying distance *eight*, and were concerned that this choice may have skewed the results. To dispel this concern, we ran Sport-League and Knights problem in adaptive recomputation and adaptive recollection over a range of copying distances. Table 7.3 displays the runtime measurement.

	Copying Distance (d)							
	$d = 1$	$d = 5$	$d = 10$	$d = 20$	$d = 40$	$d = 80$	$d = 160$	$d = 320$
Sport(*recomp*)	337	341	350	351	355	359	360	359
Sport(*recoll*)	336	326	330	329	333	334	336	335
TimeΔ(ms)	1	15	20	22	22	25	24	24
Knights(*recomp*)	1598	1830	1856	1855	1868	1872	1855	1864
Knights(*recoll*)	1589	1695	1712	1711	1703	1697	1700	1714
Time Δ(ms)	9	135	144	144	165	175	155	150

Table 7.3: Sport and Knight run over a range of copying distances

Table 7.3 shows that adaptive recollection can adjust quickly to converge to a small runtime interval, even if the copying distance is set to a large value. This observation indicates that the configuration of copying distance is not significant, confirming and generalizing the corresponding original observation reported on adaptive recomputation. The runtime difference between recomputation and recollection initially increases as copying distance increase, and then shrinks somewhat after reaching a peak performance gap(at d=80 in both cases); afterwards, it stays almost stable with the further increase of the copying distance.

7.3 Copying and Recollection

We extend the comparison to copying-based restoration. In Gecode, copying-based restoration can be easily obtained by setting the copying distance to *one*. By contrast, a more direct comparison of trailing and recollection would require an implementation of state-of-the-art trailing within the Gecode system, which is beyond the scope of this thesis. Schulte has provided such a system-crossing comparison in [27]. By following this choice of benchmark problems, Table 7.4 attempts to give a broader view on the performance of recollection.

The table reveals that recollection consumes less memory than copying, especially for the large problems Queens-100 and Knights-18. For runtime, recollection does not outperform copying on small or medium size problems; however, it cuts the runtime almost in half on large problems (Queens-100 and Knights-18). Schulte [27] observes that copying together with adaptive recomputation can outperform trailing-based system for large problems with deep search trees

Problems	Copying		Recomputation		Recollection	
	Time(ms)	Mem(K)	Time(ms)	Mem(K)	Time(ms)	Mem(K)
Alpha	51	54	55	45	66	50
Queens(10)	26	77	34	53	37	55
Queens-S(10)	17	41	21	29	25	31
Queens(100)	**39**	**26076**	**16**	**4301**	**15**	**4663**
Queens-S(100)	3	1662	1	240	2	602
Magic-Seq	51	4358	51	4358	51	4361
Knights(18)	**31**	**11271**	**20**	**1596**	**19**	**1681**

Table 7.4: Comparison with other restoration techniques

(Queens-100 and Knights-18). Recollection further improves the runtime on the two benchmark problems Queens-100 and Knights-18 problems.

7.4 Programming Restoration Granularity

We evaluate our programmed prototype over the four problems, where we sought the motivation to program restoration granularity. Both recomputation and recollection have adaptive service enabled and set *copying distance* to *eight*.

Problems	Recomputation		Recollection		Prototype	
	Time(ms)	Mem(K)	Time(ms)	Mem(K)	Time(ms)	Mem(K)
Queens	4,330	25,748	4,578	28,238	4,601	6,244
Queens-S	2,156	1,485	2,473	3,974	2,469	542
Knights	1,858	4,460	1,704	4,592	1,744	2,333
Sport-League	352	7,710	331	7,937	339	6,109

Table 7.5: Programming Restoration Granularity Evaluation

We compare the prototype with both adaptive *recomputation* and adaptive *recollection*, and Table 7.5 depicts the evaluations results. These numbers reveal that our prototype can significantly save memory than the other two restoration alter-

natives; but for Sport-League problem, the memory saving is not as significant, which can be on account of it *first* failure comes earlier (at level 62 of a tree with peak depth 249). Meanwhile, Queens problems expose better runtime performances by adaptive recomputation. Nevertheless, it deserves to highlight that Knights problem almost halves memory consumption than the other two techniques, while marginally improves its runtime than recomputation. This promising result confirms the opportunities of programming restoration granularity to seek better performance.

Chapter 8

Conclusion

In a constraint programming system, state restoration implements the strategy to recover previously accessed state, and a state-of-the-art state restoration is essential for the performance of a constraint programming system.

In this thesis, we first proposed a restoration technique called recollection, which maintains the variables that were affected by propagation to reach fix point states for restoration. It neither rolls back performed operations as trailing does nor repeats previous computation work as recomputation does, while consuming much less memory than copying. Empirical evaluation demonstrated that recollection can be competitive for solving problems for solving problems with deep search tree and expensive constraint propagation.

Subsequently, we explored building a state restoration service through programming restoration granularity. This scheme aims at providing strategies for users

to customize the restoration facilities in a constraint programming system, and a naive prototype has been constructed. The empirical study of the prototype gave promising evidence for further exploration, and we proposed to engineering the restoration granularity using the aspect-oriented programming paradigm, striving for a more extensible and modular system.

Possible further research in this area could investigate the systematic deployment of intelligent backtracking in constraint-based search. Another avenue would be search engines that are aware of computation resource constraints (e.g. memory and power) to further extend the utility of programming restoration granularity.

Bibliography

[1] Abderrahamane Aggoun, David Chan, Abderrahamane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Bruno Perez, Emmanuel Van Rossum, Joachim Schimpf, and Periklis Andreas Tsahageas. Eclipse 3.5 user manual, 1995.

[2] Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the chip compiler system. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint logic programming*, pages 421–435. MIT Press, Cambridge, MA, USA, 1993.

[3] Hassan Aït-Kaci. Warren's abstract machine: A tutorial reconstruction. In *Logic Programming Series*, Cambridge, MA, USA, 1991. The MIT Press.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Communicatio of ACM*, 13(11):677–678, November 1970.

[5] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In *Proceedings of the 7th International Conference on*

Principles and Practice of Constraint Programming, CP '01, pages 240–255, London, UK, 2001. Springer-Verlag.

[6] Philippe Codognet and Daniel Diaz. Compiling constraints in CLP(FD). *The Journal of Logic Programming*, 27(3):185–226, 1996.

[7] Alain COLMERAUER. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[8] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[9] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier. The constraint logic programming language CHIP. In *Proceeding of the International Conference on Fifth Generation Computer Science FGCS-88*, pages 693–702, 1988.

[10] John Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*, pages 268–277, Tronoto, 1978.

[11] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12(4):516–524, 1965.

[12] Gopal Gupta and Enrico Pontelli. Last alternative optimization. In *Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 538–541, 1996.

[13] Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[14] ILOG. ILOG Solver: User manual. *version 3.2*, 1996.

[15] Joxan Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 111–119, New York, NY, USA, 1987. ACM.

[16] Joxan Jaffar and Michael J. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, pages 503–581, 1994. Special Issue: Ten Years of Logic Programming.

[17] Richard Jones and Rafael D Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, 1996.

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[19] Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In Christian Bessiere, editor, *Thirteenth International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 409–422, Providence, RI, USA, September 2007. Springer-Verlag.

[20] Jena-Lonis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial intelligence*, 10(1):29–127, 1978.

[21] Yong Lin. Prototypical implementation of recollection based on Gecode. `www.comp.nus.edu.sg/~henz/recollection`, May 2013.

[22] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[23] Mozart Consortium. The Mozart Programming System. Documentation and system available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, 2008.

[24] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[25] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, volume 94, pages 362–367, 1994.

[26] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533. Springer-Verlag, October 1997.

[27] Christian Schulte. Comparing trailing and copying for constraint programming. In *Proceedings of the 1999 International Conference on Logic Programming*, pages 275–289, Cambridge, MA, USA, 1999. Massachusetts Institute of Technology.

[28] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[29] Christian Schulte and Guido Tack. Implementing efficient propagation control. In *TRICS 2010, Third Workshop on Techniques for Implementing Constraint Programming Systems*, 2010.

[30] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic Common Lisp. *IRCS Technical Reports Series*, page 14, 1993.

[31] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Lecture Notes in Computer Science*, volume 1000, pages 324–343, Berlin, 1995. Springer-Verlag.

[32] Richard M Stallman and Gerald J Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial intelligence*, 9(2):135–196, 1977.

[33] Gecode Project Team. Generic COnstraint Development Environment. `www.gecode.org`, June 2012.

[34] Peter Van Beek. Backtracking search algorithms. *Foundations of Artificial Intelligence*, 2:85–134, 2006.

[35] Pascal Van Hentenryck. Constraint satisfaction in logic programming. Logic Programming Series, 1989.

[36] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*, 37(1-3):139–164, 1998.

[37] David HD Warren and Artificial Intelligence Center. *An abstract Prolog instruction set*, volume 309. SRI International Menlo Park, California, 1983.