MONASH University

Effective Profiling of Tree Search in Constraint Programming

by

Maxim Shishmarev

Thesis

Submitted by Maxim Shishmarev

for fulfillment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisor: Dr. Guido Tack

Associate Supervisors: Prof. Maria Garcia de la Banda, Dr. Christopher Mears

Faculty of Information Technology
Monash University

March, 2018

© Copyright

by

Maxim Shishmarev

2018

Contents

List of Tables . . . . **vi**

List of Figures . . . . **vii**

List of Abbreviations . . . . **xi**

Abstract . . . . **xii**

Acknowledgments . . . . **xv**

1 Introduction **1**

2 Background **9**

2.1 Introduction 9
2.2 Profiling in Imperative Programming 9
2.3 Constraint Satisfaction Problems 10
2.4 Tree Search 11
2.4.1 Alternative to CP Approaches 14
2.5 Modelling with MiniZinc 15
2.5.1 MiniZinc Compilation Process 17
2.6 Related Work 19
2.6.1 Brief History of Profiling of Constraint Problems 20
2.6.2 Search Tree Visualisations 22
2.6.3 Comparison of Executions 27
2.6.4 Visualisation of Domains 29
2.6.5 Constraint Visualisations 32
2.6.6 Propagation Views 36
2.6.7 Visualisation of Solutions 36
2.7 Summary 38

3 An Architecture for Profiling CP Search **39**

3.1 Requirements for Effective Profiling 39

3.2 Design Decisions . . . 41
3.2.1 The Profiling Architecture . . . 42
3.2.2 Communication Protocol . . . 44
3.3 Development and Evaluation of a Prototype . . . 46
3.3.1 The Protocol Specification . . . 46
3.3.2 Efficiency Evaluation of the Profiling System . . . 48
3.4 Solver Integrations . . . 50
3.5 Related Work . . . 53
3.6 Summary . . . 55

4 Visualisation of Search Trees . . . 57
4.1 Traditional Search Tree Visualisation . . . 58
4.1.1 The State of the Art in Search Tree Visualisation . . . 58
4.1.2 Visually Determining the Size of Subtrees . . . 59
4.2 Overview Visualisations . . . 62
4.2.1 Indented Pixel Tree Plots . . . 62
4.2.2 Icicle Tree . . . 66
4.3 Navigation . . . 68
4.4 Summary . . . 71

5 Finding Recurring Patterns in Executions . . . 73
5.1 Measuring the Similarity among Subtrees . . . 74
5.2 Algorithms for Similar Subtree Analysis . . . 76
5.2.1 Similarity by Contour . . . 76
5.2.2 Finding Identical Subtrees . . . 78
5.2.3 Filtering the Results . . . 81
5.2.4 Eliminating Subsumed Patterns . . . 82
5.3 Implementation within CP-Profiler . . . 83
5.4 Case Study . . . 86
5.5 Summary . . . 88

6 Techniques for Comparing Executions . . . 89
6.1 Introduction . . . 89
6.2 Comparison of Executions . . . 89
6.2.1 Comparison Algorithm . . . 90
6.2.2 Performance experiments . . . 92
6.3 Search Replaying . . . 94
6.3.1 The Protocol for Replaying . . . 95

6.3.2 The Algorithm for Recording Search 96
6.3.3 The Algorithm for Replaying 96
6.4 Case Studies 99
6.4.1 Case Study: The All Interval Series Problem 99
6.4.2 Case Study: The Intensity-Modulated Radiation Therapy Problem . 102
6.5 Related Work 106
6.6 Summary 107

7 Analysis of Learning Solvers 109
7.1 Background on No-Good Learning 109
7.2 The Methodology 113
7.3 Implementation 115
7.4 Case Studies 115
7.4.1 First Case Study: the Free Pizza Problem 116
7.4.2 Second Case Study: the Golomb Ruler Problem 118
7.4.3 Third Case Study: the Radiation Problem 120
7.5 Summary 124

8 Conclusions 125

List of Tables

3.1 Information sent to build a simple tree. 46
3.2 Field types in the protocol. 47
3.3 Example of a correspondence between a solver and the profiler. 48
3.4 Time taken in a worst-case scenario (no propagation). 49
3.5 Time taken for the Golomb Ruler (GR) problem. 50
3.6 Time taken for the N-Queens problem. 50

5.1 Steps performed for finding identical subtrees in Example 5.2.1. 81
5.2 Similar subtree analysis performance (by contour). 85
5.3 Similar subtree analysis performance (by identity). 85

6.1 Experimental results of merging search trees. 93
6.2 All Interval Series Problem, number of failures to find all solutions. 101
6.3 Solving the Radiation problem using Gecode, Chuffed, and Gecode with search replaying. 106

7.1 Most effective learnt clauses in the Free Pizza problem. 116
7.2 Aggregate Results for Free Pizza over a set of random instances (relative). . 119
7.3 Most effective learnt clauses for Golomb Ruler (after the first modification). 119
7.4 Most effective learnt clauses for Golomb Ruler (before the first modification).120
7.5 Golomb Ruler Results. 120
7.6 Most effective learnt clauses in `radiation`. 123
7.7 Adding a redundant constraint to the model for the Radiation problem. . . 123

List of Figures

2.1 Golomb Ruler problem illustration. 11
2.2 Search tree for solving the Golomb ruler problem (5 marks). 12
2.3 MiniZinc Compilation Process. 18
2.4 Search-tree visualisation in Oz Explorer, taken from [Schulte, 1996]. 22
2.5 Search-tree visualisation in CHIP, taken from [Simonis et al., 2000]. 23
2.6 Christmas Tree Visualisation from OPL Studio, taken from [Bracchi et al., 2001]. 24
2.7 Search tree visualisation in the APT Tool, taken from [Carro et al., 2000b]. 24
2.8 Pseudo-3D tree visualisation in Prolog IV Debugger, taken from [Bouvier, 2000]. 25
2.9 Search tree visualisation with alternating planes in CLPGUI, taken from [Fages et al., 2004]. 25
2.10 Phase-line display in CHIP, taken from [Simonis et al., 2000]. 26
2.11 Node distribution per depth (left) and a tree-map (right) in CP-Viz, taken from [Simonis et al., 2010]. 26
2.12 Traditional search tree visualisation in CP-Viz, taken from [Simonis et al., 2010]. 27
2.13 Search tree visualisation in Gist (Gecode). 27
2.14 Side-by-side tree comparison in CHIP, taken from [Simonis et al., 2000]. . . 28
2.15 Side-by-side tree comparison in CLPGUI, taken from [Fages et al., 2004]. . 28
2.16 Side-by-side tree comparison of tree-maps in CLPGUI, taken from [Fages et al., 2004]. 29
2.17 Variable stack in Grace, taken from [Meier, 1995]. 29
2.18 Variable update view in CHIP, taken from [Simonis et al., 2000]. 30
2.19 Evolution of variable domains in VIFID, taken from [Carro et al., 2000a]. . 30
2.20 Evolution of variable domains in TRIFID, taken from [Carro et al., 2000a]. 31
2.21 Evolution of variable domains in CLPGUI, taken from [Fages et al., 2004]. . 32
2.22 Incidence matrix in CHIP, taken from [Simonis et al., 2000]. 33
2.23 Constraint graph in VIFID, taken from [Carro et al., 2000a]. 33
2.24 Visualising a ternary constraint by enumerating Y (VIFID), taken from [Carro et al., 2000a]. 33

2.25 Constraint graphs in SATGraf for two different problem instances, taken from [Newsham et al., 2015]. 34
2.26 Bin-packing constraint (left) and Cycle constraint (right) visualisations in CHIP, taken from [Simonis et al., 2000]. 34
2.27 Global constraint visualisation in Comet: alldifferent (left) and bin-packing (right), taken from [Dooms et al., 2009]. 35
2.28 S-boxes of DiSCiPl showing hierarchy of constraints, taken from [Goualard et al., 2000]. 35
2.29 An example of a user-defined view for partial solutions in Oz Explorer, taken from [Schulte, 1996]. 36
2.30 Solution visualisation for the warehouse location problem, taken from [Dooms et al., 2009]. 37
2.31 The change in the objective over time in CBLS, taken from [Dooms et al., 2009]. 37

3.1 High-Level Overview of the Profiling architecture. 43
3.2 Internal Architecture of the Profiler. 43
3.3 Profiler Components for an individual execution. 44
3.4 A simple search tree. 45

4.1 Search Tree with Collapsed Subtrees. 58
4.2 Search Tree (No Collapsing). 58
4.3 Collapsing of failed subtrees: traditional (left) and proposed (right). 61
4.4 Traditional collapsing of failed subtrees (execution with restarts). 61
4.5 Proposed collapsing of failed subtrees (execution with restarts). 62
4.6 Basic Search Tree (left) and Corresponding Pixel Tree (right). 63
4.7 An example of a pixel tree view applied for search trees. 64
4.8 Lantern tree for a problem with many solutions. 64
4.9 Golomb Ruler with 11 marks, Gecode. 66
4.10 Golomb Ruler with 11 marks, Gecode. 66
4.11 Golomb Ruler with 11 marks, Gecode. 66
4.12 The Pixel Tree with a variable profile (Radiation problem instance solved with Chuffed). 67
4.13 Icicle Tree Visualisation. 67
4.14 Icicle tree color mapped using decision variables. 68
4.15 Pixel tree with a selected slice. 68
4.16 Distinct parts of the tree highlighted using the pixel tree. 69
4.17 A single subtree highlighted using the icicle tree. 69
4.18 Currently selected node is indicated on the icicle tree. 70
4.19 Using the icicle tree to navigate to a particular node in the tree. 70

5.1 Visualisation of a basic search tree. 74
5.2 Different trees of the same contour. 75
5.3 A simple tree used for demonstrating Algorithm 4. 79
5.4 Contours of underlying subtrees can differ. 82
5.5 Viewing similar subtrees. 85
5.6 Subtrees of interest highlighted on the tree visualisation. 85

6.1 Two simple search trees used for demonstrating search merging. 90
6.2 Comparison result collapsed (left) and with pentagons expanded (right). . . 91
6.3 Sorted list of pentagons (left), which allows to locate them on the tree (right). 93
6.4 Basic search tree to replay (left) and corresponding search log (right). . . . 96
6.5 Search tree as a result of solving Golomb 10 with restarts (Gecode). 99
6.6 Intensity matrix (left) and its decomposition (right) in the Radiation problem. 102
6.7 Two out of five cells are exposed to radiation using two horizontal rods. . . 102
6.8 The radiation problem solved using Chuffed. 105
6.9 The radiation problem solved using Gecode with search replaying. 105

7.1 Search tree for `freepizza` using Chuffed. The path to the highlighted node is labelled. 112
7.2 Part of the implication graph for `freepizza`. Decision literals are double boxed. 112
7.3 Model executed with learning (left) and the resulting search is replayed without learning (right). 114
7.4 Two trees are merged to pinpoint the effect of no-goods. 114
7.5 Execution time of original and improved pizza models (logarithmic scale). . 118
7.6 Chuffed backjumps in the Radiation problem. 121
7.7 Gecode's subtree repeated 240 times in the Radiation problem. 122
7.8 Gecode's tree for the Radiation problem with similar subtrees highlighted. . 122

List of Algorithms

1 Algorithm for calculating contours of subtrees [Kennedy, 1996]. 76
2 Clustering subtrees using contours. 77
3 “Less” operator for contours. 77
4 Partition refinement algorithm for subtrees. 79
5 Removing subsumed patterns 83

6 The algorithm for merging search trees. 92
7 The procedure for comparing nodes. 92
8 The algorithm for recording search. 97
9 Outline of the search engine used for replaying. 98
10 Procedure for extracting next branching from a search log. 98

List of Abbreviations

1UIP	First Unique Implication Point
CBLS	Constraint-Based Local Search
COP	Constraint Optimisation Problem
CP	Constraint Programming
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problem
GUI	Graphical User Interface
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LCG	Lazy Clause Generation
SAT	Boolean Satisfiability
TCP	Transmission Control Protocol
SMT	Satisfiability Modulo Theories
XML	Extensible Markup Language

Effective Profiling of Tree Search in Constraint Programming

Maxim Shishmarev
maxim.shishmarev@monash.edu
Monash University, 2018

Supervisor: Dr. Guido Tack
guido.tack@monash.edu
Associate Supervisors: Prof. Maria Garcia de la Banda, Dr. Christopher Mears
maria.garciadelabanda@monash.edu, chris.mears@monash.edu

Abstract

Combinatorial problems, where a set of decisions needs to be made that satisfies a set of constraints and possibly maximises/minimises an objective function, occur frequently in many areas of life, including transportation, emergency management, and energy distribution. Unfortunately, solving these problems is remarkably difficult due to the exponential number of possible sets of decisions that can be made. Modern approaches to solving Combinatorial problems first focus on describing them in a declarative modelling language to form *models*, and then solving them using general purpose *solvers* which search among all possible decisions looking for high quality (or optimal) solutions. Constraint Programming is one of the modern solving technologies that underlie these solvers, and it is the focus of this thesis.

How fast a high quality solution can be found by a solver is largely dependent on how the problem is described in the model. For all but the smallest problems, the initial models rarely yield high quality solutions in a reasonable amount of time and need to be modified. In order to know which modifications would be favourable, the modeller needs to determine what parts of the model are responsible for the slow search, modify them, and evaluate the effect of these modification. The above process is traditionally known as *profiling*.

In imperative (rather than declarative) programming, profiling tools routinely highlight the causes behind inefficiencies in an execution and, in doing so, identify the parts of the program that should be modified. In Constraint Programming, which adheres to the declarative paradigm, such profiling capabilities are not as readily available to the programmer due to its declarative nature, which makes the link between the program and its execution rather tenuous. The existing systems for profiling constraint programs work reasonably well for small problems. However, they lack the means for automatically analysing executions and effectively evaluating model modifications, something crucial when dealing with large real-world problems, when the search for high quality solutions is too long and complex to be analysed manually. Further, many existing systems are solver-dependent and do not support modern solvers and their powerful search techniques.

This work aims to overcome the limitations of the current state of the art by designing an effective profiling system and associated profiling techniques. To achieve this, in this thesis I have designed, implemented and evaluated a profiling system that (a) is efficient

and thus scales for large problems, (b) is solver-independent and thus can easily integrate different solvers, and (c) can support modern solvers and be extended to those developed in the future. In addition, I have designed and integrated several profiling techniques into this system: search-tree visualisations that improve on the state of the art, and are more suitable for large problems; a technique for automatically finding recurring patterns in large problems that are often the reason for slow executions and can aid the modeller in improving them; two complimentary techniques that allow modellers to evaluate model modifications; and a technique for analysing so-called *learning* solvers that can aid the modeller in discovering constraints present in the model that can be strengthened as well as redundant constraints that can be added to the model faster executions for both learning and traditional solvers. Together, these techniques allow modellers to find high quality solutions faster by aiding them in making effective model modifications.

Effective Profiling of Tree Search in Constraint Programming

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Maxim Shishmarev
March 23, 2018

Acknowledgments

First and foremost, I owe thanks to my supervisors, Maria, Guido, and Chris, without whom this thesis would not have been possible. In particular, I'd like to thank Maria for providing me the opportunity to study in Australia and supporting me ever since, Guido for providing a great deal of expertise in programming, and Chris for always being patient and willing to explain even the most elementary concepts with enthusiasm.

I'd also like to thank my friends and colleagues at Monash, who have always been a there to support and provide source of much-needed distraction. In no particular order, I'd like to thank: Dora, Richard, Kevin, Steve, Tom, David, Vahan. Especially, I wish to thank My for providing a tremendous amount of support, particularly during the final year of this journey.

Last but not least, I'd like to thank my parents for enabling me to realise my full potential and always believing in me, and my brother for having been a constant source of inspiration.

Maxim Shishmarev

Monash University
March 2018

Chapter 1

Introduction

Combinatorial problems, where a set of decisions needs to be made that satisfies a set of constraints, occur frequently in many areas of our lives, including transport, emergency management and energy distribution. Combinatorial problems are described using a set of variables, each with an associated domain of possible values, and a set of constraints that restricts the combinations of values the variables can take. The problem is solved when all variables are assigned to values that satisfy all constraints. Such problems are thus called constraint *satisfaction* problems. A constraint *optimisation* problem adds to this an objective function whose value is to be minimised (or maximised).

Obtaining a high quality solution, that is, one where the objective value is close to its minimum (maximum), is of great importance when, for example, one need to minimise the damage done by radiation therapy, maximise the efficiency and reliability of food transport, and minimise the cost and operational risk of a chemical processing plant. Unfortunately, solving these problems is remarkably difficult due to the exponential number of possible sets of decisions (the *search space*) that can be made.

In general, no efficient algorithm exists for solving large combinatorial problems. Modern approaches to solving such problems first focus on describing their variables, constraints and objective function (if any) in a declarative modelling language, obtaining a *model* of the problem. Then, a general purpose *solver* is used to solve the model by performing some form of intelligent systematic search in order to find solutions.

Solving Combinatorial Problems

Solving combinatorial problems through modelling and the use of general-purpose solvers has several advantages over the more traditional approach, which focuses on writing dedicated programs with hand-crafted solving algorithms for every problem. An important advantage is that the existing solvers are highly optimised and incorporate decades of solver development research. As a result, writing hand-crafted solving algorithms that can find a solution faster than a general-purpose solver is time-consuming and challenging. In contrast, modelling languages allow one to describe a problem in a relatively short period of time and solve it taking advantage of the advanced algorithms already present in existing solvers.

A second advantage is that it is easy, and therefore customary, to write models that can be used to solve a whole family of similar problems by separating a problem's description from its data. Further, if the problem description has to change even slightly, the approach taken by the hand-crafted solving algorithm might not be appropriate for the new problem,

and thus developing a whole new algorithm might be necessary. In contrast, a model can be easily modified to fit the new problem description.

Constraint Programming

There are a number of different solving technologies that underlie the general purpose solvers used for solving models, such as Mixed Integer Programming (Land et al., 1960, Dakin, 1965), Boolean Satisfiability (Davis et al., 1960, Marques Silva et al., 1996, Moskewicz et al., 2001, Biere et al., 2009), Satisfiability Modulo Theories (Nelson et al., 1979, Armando et al., 1999), and Constraint Programming (Mackworth, 1977, Rossi et al., 2008). To appropriately limit the scope of my work, this thesis focuses on Constraint Programming technology.

Constraint Programming solvers often solve problems using a combination of propagation and tree search. The propagation engine uses the constraints and the current domains of the variables to further reduce their domains. If it detects a constraint that cannot be met, or a variable with an empty domain, the problem is known to be unsatisfiable and the solver reports a failure. If all variables have a single value and all constraints are satisfied, the problem is known to be satisfiable and the solver reports a success. Otherwise, the solver begins to search.

A typical tree search process splits the problem into two or more subproblems that partition the search space. Each subproblem is the same as its “parent” but with an additional constraint, which is referred to as the *branching decision*. The propagation engine is run on each subproblem, and further splitting is done if necessary. In this way, the search moves towards subproblems that can be immediately detected as being satisfiable or unsatisfiable. This process implicitly defines a *search tree* rooted by the original problem, where each node represents a branching decision, which is used as the node’s label. In its simplest form, the tree search continues branching until either a solution or a failure is reached. In the case of a failure, the search backtracks up the tree, undoing previous decisions until it reaches a node where an alternative decision can be made. In the case of a solution, the search finishes (when a single solution is sufficient), or backtracks to search for other solutions. Additionally, if the problem is an optimisation one, a new constraint is added after finding a solution to require new solutions to be of better quality than the current one. When there is no alternative decision left in the entire tree, the search finishes.

Debugging

Transforming a problem’s description into a model is usually a relatively easy task. For hard to solve problems, however, the first model will often fail to deliver the desired outcome. It is common to distinguish between two types of unsuccessful outcomes:

- those where the solver finds solutions that do not satisfy the original problem description (or fails to find the solutions expected);
- those where the solver does not find an optimal (or close to optimal) solution within a reasonable amount of time.

The former case is associated with the correctness of the model and is the focus of *correctness debugging*. This thesis is concerned with *performance debugging*, or *profiling*, which focuses on the latter case of slow executions.

Performance Debugging

Writing an efficient model – that is, one that can be solved in a reasonable amount of time – can be very time-consuming and challenging, often requiring considerable levels of expertise. In the case where a solution cannot be found sufficiently quickly, a common approach is to find a different way of describing the problem, that is, a different model. There are, indeed, many ways to describe the same problem, and thus many possible models for a problem, each resulting in possibly very different execution times. Additionally, many solvers can follow a given search strategy if provided in the model, thus allowing the modeller to influence the search process. Therefore, changing the search strategy is another way for the modeller to affect the execution. Finally, when it comes to choosing a solver with which to execute a model, there is no clear favourite, as no single available solver consistently outperforms all other solvers. Modern modelling languages allow switching between different solvers seamlessly, providing yet another way for the modeller to influence the execution.

Trying all possible combinations of models, search strategies, and solvers is impractical. It is therefore extremely important for a modeller to know which changes will lead to more efficient executions. In particular, it is important to know:

(a) what part of the program (that is, the triplet of solver, search, and model) slows down the execution;

(b) how a particular change in the program will affect the execution.

In procedural (rather than declarative) languages, this knowledge is usually obtained through the process of profiling. In particular, the profilers routinely used in procedural languages highlight the causes (e.g., lines of code, or functions) of inefficient behaviour and, in doing so, identify for the programmer the parts of the program that should be modified. After some modification has been applied, the programmer can see its effect on the execution and decide if a different modification should be applied instead. The above process is usually repeated several times until the programmer is satisfied with the program's performance.

In Constraint Programming such profiling capabilities are not as readily available to the modeller due to the declarative nature of this style of programming, which makes the link between the program and its execution rather tenuous. As a consequence, there is no effective way to identify the causes of inefficient behaviour. Additionally, the existing profiling systems for Constraint Programming do not provide an effective way to gain real insight regarding the effect of model and search changes on the execution.

Further, the profiling of combinatorial problems comes with a number of new challenges. First, as already mentioned, it is not uncommon for a modeller to use different solvers while trying to speed up the execution. Being able to evaluate the changes in execution after a different solver is selected, requires a profiling system that supports both solvers (in general, one that is *solver-independent*). There is, however, no standard mechanism to produce the information necessary for profiling, and thus significant solver instrumentations are usually required. This is the reason why most of the existing profiling systems only targeted one solver (for example, [Schulte, 1996], [Simonis et al., 2000], [Carro et al., 2000b]). A few other systems provided means of solver-independent profiling ([Fages et al., 2004, Bracchi et al., 2001]), but have not been integrated by many solvers, as they required extensive solver instrumentation. The only system integrated by a large number of solvers (more than three) is the one described in [Simonis et al., 2010].

Second, while many of the currently available solvers are similar enough to make solver-independent profiling viable, the advancements in the area of solver development constantly lead to new functionality being supported by some of those solvers. A profiling system will therefore be more effective if it can adapt to these changes by supporting the additional functionality of these solvers, while remaining backwards compatible with more traditional ones. Most existing profiling systems that support multiple solvers have not been designed with adaptability in mind; instead, they rely on solvers to produce large volumes of information, some parts of which would often be unused by the profiler, but could become necessary for new profiling techniques.

Third, the large amount of profiling information that is often produced during the execution of large real world problems, requires exploring large trees with many variables and constraints. As a consequence, a profiling system needs to be able to (a) process large amounts of information in a reasonable amount of time, and (b) focus the programmer's attention on those parts of the execution that are likely to lead to the discovery of meaningful *insights*. The most popular way for modellers to study their program's execution is to examine their search tree using search-tree visualisations. While some existing systems are capable of visualising executions with many millions of nodes, their profiling techniques do not provide any aid in the form of automated execution analysis, which can be necessary to discover the places the modeller must focus on when dealing with large executions of multi-million node trees.

Research Question

This work aims to overcome the above challenges by addressing the following research question:

How can a profiler help a modeller find improvements to the program in a way that it:

(Q_1) is efficient, and thus scales for large real world problems;

(Q_2) is solver independent, and thus can easily integrate different solvers;

(Q_3) supports modern solver techniques and can be extended for those developed in the future;

(Q_4) provides good insight into the solving process, for example, by means of automated analyses;

(Q_5) helps the modeller evaluate the impact of changes in the model.

In these thesis I have addressed the above question by designing:

- A framework for *effective* profiling – one that is efficient (Q_1), solver-independent (Q_2), supports modern solving techniques and is extensible (Q_3), and makes it easy to incorporate different profiling techniques.
- Several profiling techniques suitable for large executions, that help a modeller gain insights regarding executions (Q_4), as well as gain a better understanding of the effect of the resulting model changes (Q_5).

Contributions

In particular, this thesis provides the following contributions.

C_1: An Effective Profiling System

A solver-independent (Q_2) framework for profiling Constraint Programming executions has been designed. The framework is suitable for profiling large executions (Q_1), and is built on top of a newly designed, lightweight, flexible and extendable protocol that supports a variety of modern solver techniques (Q_3). A working prototype of the framework, CP-Profiler, has been implemented, and its capability to integrate many solvers, support different solver techniques, and deal with large executions has been evaluated. All profiling techniques discussed below have been implemented and integrated into this framework for their evaluation.

C_2: New Search Tree Visualisations

The visual examination of a search tree is by far the most common and familiar way of profiling Constraint Programming executions currently employed by modellers. This thesis presents “lantern trees” as a way to improve the conventional way of visualising very large search trees without losing as much structural information as previous approaches. The resulting visualisation is demonstrated to be superior to those currently used in modern profiling systems in a number of case studies. Additionally, a few unconventional methods for visualising search trees have been studied, and their usefulness in the context of constraint program profiling has been demonstrated. These methods for search tree visualisation allow modellers to see search trees in a different light, potentially enabling them to discover new insights regarding executions (Q_4).

C_3: Similar Subtree Analysis

A technique capable of discovering inefficiencies in large executions has been designed – the *similar subtree analysis* – which works by automatically finding repeated patterns in the execution, something that often indicates the presence of an inefficient search. This contribution addresses the need for automated analysis that can extract insights (Q_4) from large executions, as it allows modellers to focus on potentially interesting parts of the search tree. Additionally, this technique has been demonstrated to lead to effective model modifications.

C_4: Search Merging

A technique for evaluating changes that leads to different executions (Q_5) has been designed: *search merging*. This technique allows the modeller to compare the search trees of two different executions, before and after some change in the model, solver, or search strategy, and assess the effect of that change in detail. Importantly, the comparison is performed automatically and works for large executions.

C_5: Search Replaying

A profiling technique has been designed that allows modellers to evaluate the differences in constraint propagation between two executions (Q_5): *search replaying*. This technique

works by having the two executions make the same search decisions, and thus isolates the effect of changes in constraint propagation. Additionally, it works hand-in-hand with search merging, as together they enable a more accurate comparison of two executions. Several case studies demonstrate that the two techniques can be used to find useful insights (Q_4) regarding model modifications, which could suggest further model improvements.

C_6: No-good Analysis

Recently, a new class of solvers, called *learning* solvers, has been defined. These powerful solvers learn so-called *no-goods* while they search for solutions to the problem. No-goods capture the reasons for failure and are used by learning solvers to avoid areas of the search space that will lead to similar failures. This often allows learning solvers to find solutions to hard constraint problems faster than non-learning solvers. However, for some problems, learning solvers seem to be unable to benefit from the learning and perform poorly compared to non-learning solvers. It is, therefore, of great importance to better understand the behaviour of a learning solver. This work presents a technique specifically designed for analysing the executions of such solvers (Q_3): *no-good analysis*.

The proposed technique is built on top of the search replaying and the search merging techniques, and adds extra functionality specific to learning solvers. In particular, the technique allows modellers to identify those no-goods that have the most impact on the executions, that is, that yield the largest reduction of the total amount of search.

This thesis demonstrates, by means of several case studies, how the no-good analysis can help modellers better understand the executions of learning solvers (Q_4) and aid modellers in finding effective model modifications, capable of yielding one to two orders of magnitude faster executions.

Publications

The contributions of this thesis have led to two publications. In particular, the profiling system itself (C_1) along with the profiling techniques associated with contributions C_2–C_4 have been published as [Shishmarev et al., 2016b]:

> Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda, "Visual Search Tree Profiling". *Constraints, Volume 21, 2016.*

The analysis of learning solvers (C_5) has been published as [Shishmarev et al., 2016a]:

> Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda, "Learning from Learning Solvers". *Principles and Practice of Constraint Programming - 22th International Conference, CP 2016. Proceedings.*

Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 first formally introduces combinatorial problems, and describes several techniques that underlie modern solvers with an emphasis on Constraint Programming. This is followed by an introduction to the modelling language *MiniZinc* [Nethercote et al., 2007], which is used in the later chapters to describe all problem models and their modifications. The chapter concludes by providing a survey of existing systems for profiling combinatorial problems.

Chapter 3 presents contribution C_1: the design of the solver-independent, efficient, and extensible profiling framework that is at the core of this thesis. In particular, the chapter first presents the architecture of the framework, along with the protocol that is used for communication between the profiler and the solvers. This is followed by the evaluation of the framework, in particular, its efficiency and ease of integration with new solvers. Note that this framework is later used for evaluating the profiling techniques discussed in the remaining chapters.

Chapter 4 presents contribution C_2, which aims to improve the way search trees are visualised. It starts with an overview of the traditional node-link tree visualisation, commonly used for displaying search trees in the literature. It then presents an improvement on the conventional search-tree visualisation, and demonstrates how the resulting visualisation is better suited for profiling. Finally, it presents two unconventional ways of visualising search trees, and demonstrates how they can benefit the process of profiling.

Chapter 5 presents contribution C_3: the *similar subtree analysis*, a technique for analysing the structure of execution trees. Through a case study, it demonstrates how this technique can be effective for finding inefficiencies in executions.

Chapter 6 presents contributions C_4 and C_5, associated with profiling techniques that can help modellers compare multiple executions of a single problem. First, it presents *search merging* as a technique for assessing the impact of a modification applied to a model by comparing two executions of the original and modified models. It then introduces *search replaying*, a technique for ensuring that the same search is used by multiple executions, as a complementary way of comparing executions. The chapter concludes by demonstrating through two case studies how these techniques can help modellers better understand the effect of a model modification.

Chapter 7 presents contribution C_6, associated with the analysis of learning solvers. It starts by giving a brief overview of modern learning solvers. It then presents the *no-good analysis* as an effective new technique for studying these solvers that takes advantage of the profiling techniques discussed in the previous chapter. The chapter concludes by demonstrating through three case studies how applying the no-good analysis technique can help identify culprits of inefficient behaviour in an execution, and thus lead to significant model improvements.

Finally, Chapter 8 provides my conclusions and discusses future work.

Chapter 2

Background

2.1 Introduction

This thesis investigates ways in which individual executions associated with solving constraint problems can be analysed, in order to speed up subsequent executions for the same problems. This chapter provides the background necessary to understand the remaining chapters of the thesis. In particular, it introduces constraint problems – the central concept of this work. This is followed by a brief overview of modern approaches for solving them, mainly those involving *tree search*, which are the focus of this work. Finally, the chapter surveys the existing approaches commonly used for profiling the executions of constraint problems and discusses their limitations.

Structure of the Chapter

Section 2.2 gives a brief overview of the profiling approaches that are commonly employed for programs written in imperative programming languages. Section 2.3 introduces the basic concept of constraint satisfaction problems and provides an illustrative example. Section 2.4 gives a brief overview of the modern technologies used for solving such problems with emphasis on the Constraint Programming technology, which is the focus of this work. Section 2.5 introduces the MiniZinc modelling language, which is used in this work for demonstrating different ways of modelling problems. Finally, Section 2.6 gives an overview of the state of the art in profiling of constraint programs.

2.2 Profiling in Imperative Programming

When a program execution takes longer than expected, it is important for the programmer to know which part of the execution causes the slow behaviour. This information is usually obtained by programmers through the process of *profiling*. Profiling is a very well established field in the imperative programming paradigm, where the program's code directly states how the problem should be solved. Tools that enable profiling (*profilers*) are widely available for the programming languages that adhere to this paradigm, for example, gprof [Graham et al., 1982], JProf [Shirazi, 2002], and Visual Studio [Johnson, 2012].

Most of these profilers work by gathering *profile information* and presenting it to the programmer in an aggregated form. Profile information is usually expressed in terms of a program's method calls. In particular, profilers commonly gather *call counts*, that is,

the number of times a given method is called, and *execution times*, that is, the portion of time that the program spent executing a given method. Execution times provide a general idea regarding the methods of the program that take the most time to process, and thus let the user make effective modifications to the program by focusing only on those methods. Call counts can be used for a similar purpose, but with the focus on determining the complexity classes that algorithms (represented by methods) belong to, and thus reasoning about the scalability of the algorithms. Additionally, profilers often show the programmer a *call graph* – a directed graph, where nodes represent the program's methods, and edges between two nodes N_1 and N_2 represent invocations of method N_1 by method N_2 [Ryder, 1979]. These graphs allow the programmer to better understand the flow of the program's execution.

The profiling techniques that are used in practice mainly differ in the way they gather the profile information. There are two main approaches: instrumentation-based and sampling-based. The former requires inserting special code that is responsible for gathering the profile in important regions of the program. This approach is intrusive, as it requires modifying the program, but provides accurate results. The latter approach probes the execution by interrupting it at regular intervals to determine which method is executed at each interval. This approach does not require program instrumentation, but it is less accurate than the instrumentation-based one.

This thesis deals with profiling in Constraint Programming (described further below), which adheres to the declarative, rather than imperative programming paradigm. Although there are certain similarities between profiling techniques in both paradigms (for example, they both use directed graphs to present the execution flow), profiling in Constraint Programming is more challenging. For example, counting and timing individual events in a Constraint Programming execution does not reveal a great deal of insight into the execution as these events are difficult to link to specific parts of the program. Different profiling approaches are, therefore, needed in Constraint Programming.

2.3 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) involves finding values for a given set of variables (called decision variables) that satisfies a given set of constraints imposed on them. Formally, a CSP is often represented as a tuple $\langle X, D, C\rangle$, where X is the set of decision variables, D is a set of domains of variables, and C is the set of constraints. Each domain $d_i \in D$ corresponds to exactly one variable $x_i \in X$, and represents the finite (or less often, infinite) set of values that x_i can take. A mapping $x_i = v_i$, where $\forall i,\ x_i \in X',\ X' \subseteq X$, $v_i \in d_i$, is called an assignment to X. The assignment is called *partial* if $X' \subset X$. The constraints in C further restrict the sets of values that one, two, or several variables can take in the same assignments (called, respectively, unary, binary and n-ary constraints).

A constraint $c \in C$ is said to be *satisfied* by some assignment to X if a simultaneous assignment to all variables involved in c is not prohibited by its definition, and it is said to be *violated* otherwise. Solving a CSP involves finding an assignment to X that satisfies all constraints in C. If, additionally, an expression $f(X'')$ (where $X'' \subseteq X$) is specified, and it is required to be minimised (or maximized), the problem is said to be a constraint optimisation problem (COP) and $f(X'')$ is called the *objective* function.

Example 2.3.1. Consider the constraint satisfaction problem of *Golomb ruler*. The problem requires putting m marks in the integer positions of a ruler of length n, such that the distances between any two pairs of marks are unique, and the distance between the

left-most and the right-most marks is minimal. Figure 2.1 depicts such a ruler of length 6 with 4 marks.

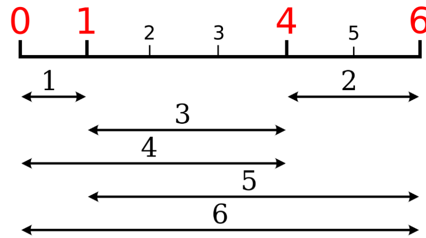

Figure 2.1: Golomb Ruler problem illustration.

Formally, the positions of marks are represented by decision variables $X = \{x_1, ..., x_m\}$ with domains $d_i = \{0, .., n\}$, $i \in 1..m$. The problem requires the following constraints C to be satisfied:

- $\forall_{i,j} : x_j > x_i$, $i, j \in 1..m$, $j > i$
- $x_j - x_i$, $1 \leq i < j \leq m$, are all distinct

The Golomb ruler problem additionally requires the objective function $f(x_1, x_n) = x_n - x_1$ to be minimised. In this notation, the solution to the problem depicted in Figure 2.1 can be described as $x_1 = 0$, $x_2 = 1$, $x_3 = 4$, and $x_4 = 6$, and the value of the objective is 6.

2.4 Tree Search

The techniques for solving constraint satisfaction problems can be divided into two groups: *complete* and *incomplete*. The difference between them lies in the way they explore the set of all possible assignments to X, or the *search space*. Incomplete techniques, such as *local search*, only partially explore the search space, relying on the search procedure to select areas with high likelihood of solutions. They can be effective for a range of problems with large search spaces. However, since incomplete techniques do not explore the search space entirely, they cannot guarantee the optimality of solutions they find or determine if the problem has any solutions in principle. On the other hand, complete techniques perform systematic search of the entire search space, and thus, given enough time, they are guaranteed to find a (optimal) solution if one exists. A common way to perform such systematic search is to use tree search. The focus of this work is the Constraint Programming (CP) solving technology, which is a popular form of tree search (Mackworth, 1977, Rossi et al., 2008). Solvers that employ this technology for solving problems are referred to as constraint solvers. The general idea behind the solving process by tree search in Constraint Programming is briefly described below.

Tree search is often represented as the traversal of a *search tree*. Figure 2.2 shows an example of a search tree that corresponds to solving the Golomb ruler problem with 5 marks (variables X_i are shown as mark_i). The tree, in this case, distinguishes between three types of nodes: blue circles, red squares, and green diamonds, which represent, respectively, intermediate states, failure states, and solution states. Solution states represent assignments to all variables in X that satisfy all constraints in C. Failure states represent assignments (or partial assignments) to X that violate at least one constraint in C. Lastly, states that are neither a failure state nor a solution state (or cannot be identified as such) are intermediate states, and referred to as branch nodes in the context of a search tree.

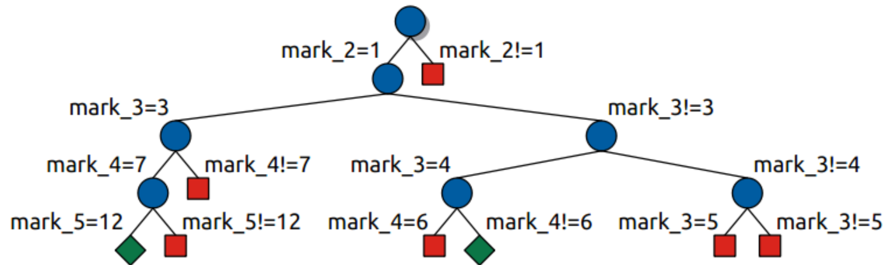

Figure 2.2: Search tree for solving the Golomb ruler problem (5 marks).

At the beginning of any execution, the tree consists of a single node, the *root* node, which represents the original problem, and thus an *empty assignment*. In Figure 2.2 the root is shown as the top-most node. At any branch node, including the root, the solver performs *propagation* – a procedure for removing values from the domains of variables that are inconsistent with at least one of the constraints in C. It works by reacting to variable domain changes, which can be caused by either a search decision or by earlier propagation, resulting in a *firing of a propagation event*. More precisely, a change in a variable's domain *awakens* all constraints involving this variable, that is, it puts these constraints in a queue for processing to determine potential inconsistencies that the domain change could cause. This procedure is repeated until no more constraints are left in the propagation queue, at which point a fixpoint in propagation is said to be reached.

Propagation can reduce the domain of a variable to a single value, thus making an implicit assignment to that variable. Whenever propagation empties the domain of some variable, that is, $\exists x_i \in X : D(x_i) = \emptyset$, the search reaches a failure. A solution state is reached by the search whenever all variables in X are assigned either by search, or implicitly by propagation. Finally, if the propagation procedure is not sufficient to identify a node as a failure or a solution, the current node becomes a branch node, and the search tree is extended with new nodes.

The nodes are added to the tree as a result of the branching process, described as follows. At any branch node the problem p is split into k subproblems by adding branching constraints b_k, such that each node $i \in 1..k$ represents an augmented problem $p \wedge b_i$ and $p \iff \vee_{i\in 1..k}(p \wedge b_i)$. The decision of selecting a branching constraint b_i at some point in the search is called a branching decision. The process of making branching decisions is often referred to as *labeling*. The distance from the root node to the node where a decision is made, that is, the number of decisions on the path, is referred to as the decision's level. Whenever a new branching decision is made, the currently examined node gets extended with k children nodes, representing the resulting subproblems. For instance, the first branching decision in the Golomb ruler example splits the problem into $k = 2$ subproblems by adding branching constraints $b_1 = (mark_2 = 1)$ and $b_2 = (mark_2 \neq 1)$.

The order in which the nodes are explored during the search depends on the *tree exploration strategy* employed by the solver. In general, there are three basic searches used for exploring trees: depth-first search, breadth-first search, and best-first search. Depth-first search aims to reach leaf nodes as soon as possible. It is the most commonly employed strategy in constraint programming (also known as "chronological backtracking"), as constraint solvers benefit from early solution/failure encounters, and it requires a polynomial amount of memory in terms of the problem size. In this strategy, left-most branches are explored first until the search reaches either a solution or a failure state. In the Golomb ruler example, the search proceeds by consecutively making three more search decisions, until it reaches a solution state. Whenever a failure node is reached, the search proceeds

by *backtracking* – a procedure that involves moving up in the tree to a node, undoing previous decisions until it reaches an ancestor node, where an alternative decision can be made. The search will discard the decisions that are not associated with the new node, potentially restoring the values in domains previously pruned due to those decisions. The search then proceeds to make further decisions, until the next failure is encountered.

A modification of depth-first search that is particularly important for this work is *backjumping*. It is often utilised by *learning solvers*, wherein reasons for past failures (called *no-goods*) during the search are analysed (see Chapter 7). Backjumping can be thought of as an improvement of chronological backtracking: rather than backtracking upon reaching a failure, this exploration strategy can perform a backjump, which involves moving multiple levels up the tree, potentially skipping unexplored nodes. (Note that the actual areas of search that these skipped nodes represent can contain solutions, and they thus might need to be explored at a later point in search.) The search then effectively makes a search decision inferred from the no-good associated with the backjump. After that, the search resumes exploring the tree in a chronological backtracking manner until the next backjump is reached.

Contrary to depth-first search, breadth-first search explores all nodes on the current depth level first before proceeding to the lower levels. This property of breadth-first search makes it encounter first solutions or failures rather late during the search, and thus it is largely unsuitable for constraint satisfaction problems, where the goal is usually to find the first solution as quickly as possible. Finally, best-first search determines exploration order based on a given evaluation function for nodes: for example, in a minimisation problem a variation of best-first search could explore nodes with the smallest lower-bound on the objective function first. A well-known example of best-first search is A* search, which is widely used for solving planning and pathfinding problems [Hart et al., 1968].

In addition to the basic search explorations techniques discussed above, there are certain meta-searches that can theoretically be used in conjunction with any such technique. One is *branch-and-bound* search, which is commonly employed for optimisation problems. In the context of CP, this strategy relies on propagation to determine the *bounds* on the objective value at every node, that is, the smallest and the largest values from the objective's domains. Additionally, it adds a constraint each time it reaches a solution that requires subsequent solutions to be of higher quality. The bounds on the objective at each node let the search determine whether the search at the node can reach a better solution than the most recent one. For example, when the objective is to be minimised (maximised), the search does not need to explore a node if its lower (upper) bound is larger (smaller) than the objective value of the current solution. When the bounds on the objective are tight, that is, the difference between the lower and the upper bound is small, the branch-and-bound strategy can avoid exploring large portions of search without the loss of any (good quality) solutions.

Another example of a meta-search is *restart-based search*, which is becoming increasingly popular in optimisation problems. In a restart-based search nodes are explored according to the selected basic search most of the time (for example, following the depth-first search order). However, at some points during the search (e.g. when a solution is reached) a *restart* is triggered, which discards all decisions on the current path and starts a new search, effectively backtracking to the root node. In doing this, restart-based search aims to avoid spending too much time in areas of search that are devoid of solutions, and thus can be more robust compared to the basic searches.

Independently of which of the above tree exploration strategy is used, many solvers are able to perform the search in *parallel*. In this case, different areas of the search space are

explored by different *search agents*, each running in its own thread. Note that each such agent can choose to perform any kind of search in the areas designated to them. However, chronological backtracking is most commonly employed.

The fact that search strategies other than chronological backtracking exist poses the following challenge to profiling tools: users might need to examine the order in which the nodes have been explored, and thus there should be a way to present this information to them. This problem will be addressed in Chapter 4, which is concerned with search tree visualisations.

Although backtracking search is often used to achieve an *exhaustive* search (that is, to explore the entire search space), this is not always necessary in practice. For example, if the problem is a CSP, finding a single solution is often sufficient, and the exploration can finish at that point. If a problem requires finding an optimal solution, the search will backtrack on solution states as well, exhausting the search space in order to prove optimality. If no solutions are found in an exhaustive search, the problem is unsatisfiable, that is, it has no solutions in principle. If the search tree in an exhaustive search does contain solutions, and the problem is an optimisation problem, the final solution is the optimal one.

In addition to a tree exploration strategy, which determines the order in which nodes are explored, the modeller can often specify a *search strategy*, which determines the branching decisions made during search. Most often, branching decisions are of the forms $X < a$, $X \leq a$, $X > a$, $X \geq a$, $X = a$, $X \neq a$, where X is a decision variable and a is a value from its domain. A search strategy is said to be *static* if the order in which the decision variables and values are selected for branching decisions is static, that is, predetermined before the search begins. Otherwise, a search strategy is said to be *dynamic*, that is, the order in which the decision variables are selected can change during the execution.

2.4.1 Alternative to CP Approaches

Other popular techniques for solving combinatorial problems that use tree-based search are Boolean Satisfiability, or SAT (Davis et al., 1960, Marques Silva et al., 1996, Moskewicz et al., 2001, Biere et al., 2009), Satisfiability Modulo Theories, or SMT (Nelson et al., 1979, Armando et al., 1999), and Mixed Integer Programming, or MIP (Land et al., 1960, Dakin, 1965). Compared to CP, they use more restrictive modelling languages which allow them to run more specialised algorithms.

SAT can be thought of as a subset of CP that supports only Boolean decision variables, and all constraints are described through *clauses*. Each clause is a disjunction of variables or their negations (together called *literals*). The conjunction of such clauses, or the formula, constitutes a problem in SAT, which is solved when an assignment to its variables is found that turns the formula into a truthful expression. The search procedure in SAT is similar to that in CP, however, its propagation is conceptually simpler as it involves applying a single rule – the *unit clause rule*. Applying this rule involves finding unit clauses, that is, clauses, where all but one literal evaluate to *false*, and assigning the remaining literal to *true*.

SMT extends the language of SAT with additional expressions over variables that have a meaning in some theory outside of SAT. One example of these expressions is those that state inequality relations between variables following arithmetic rules, which allow many constraint problems to be modeled more naturally. While SMT solvers have a SAT solver at their core, usually a specialized “theory solver” is additionally employed for the purpose

of evaluating the theory expressions with respect to the variable assignments inferred by the SAT solver.

MIP is another tree-based technique that is commonly used for solving problems with linear constraints and a linear objective function, that is, problems that can be expressed as linear algebraic expressions. MIP uses *branch-and-bound* in conjunction with best-first search as its search exploration strategy, but it is very different from CP in the way it obtains bounds on the objective. In particular, it uses the well-known Simplex method [Dantzig, 1963] to solve a linear programming relaxation (an approximation that allows all discrete variables take continuous values) of the subproblems at every node.

MIP, SAT, and SMT can often perform better than CP on some types of combinatorial problems, and it is as important to support profiling of their executions as it is of CP executions. However, this work focuses on profiling of backtracking search in CP only. Despite that, the profiling framework and many of the profiling techniques presented here are suitable for other tree-based searches, like the ones in MIP and SAT, in principle, and would only require instrumentation of corresponding solvers.

2.5 Modelling with MiniZinc

When a CSP or a COP is defined in a formal modelling language, it can be interpreted and solved by a generic solving system (solver) that supports this language. Such a specification of a problem is called a *model* for that problem, and the language is called a *constraint modelling language*.

The usual practice when modelling a problem is to separate its conceptual description (the model) from its data, so that the same model can be used to solve a whole class of similar problems. Combining a generic model with a specific data input is called instantiation, and the resulting program is called an *instance*.

Many modelling languages have been used in practice. Among the most popular ones in Constraint Programming are OPL [Van Hentenryck et al., 1999], Essence [Frisch et al., 2007], and MiniZinc [Nethercote et al., 2007]. Because of the expertise in the MiniZinc language available to me through my thesis advisers, I selected this modelling language for the purpose of describing constraint problems in this work. The following briefly describes the syntax of MiniZinc using the example of the Golomb Ruler problem introduced earlier (see Section 2.3). The corresponding model is shown in Listing 2.1.

A typical model in MiniZinc consists of the following items:

- Parameter declarations
- Decision variable declarations specifying their domains
- Constraint declarations
- Include items
- A single *solve* item

Parameter values are those that remain constant throughout the execution. The MiniZinc language requires declaring the parameter type and its name separated by the "`:`" symbol. The parameter values have to be provided either in the same line as the corresponding parameter declaration (as parameter `n` in line 2), or in a separate data file (as parameter `m` in line 1). In this case, the only user-specified parameter `m` represents the number of

```
1  int: m;
2  int: n = m*m;
3
4  array[1..m] of var 0..n: marks;
5
6  array[int] of var 0..n: differences =
7      [ marks[j] - marks[i] | i in 1..m, j in i+1..m];
8
9  constraint marks[1] = 0;
10
11 constraint forall ( i in 1..m-1 ) ( marks[i] < marks[i+1] );
12
13 include "alldifferent.mzn";
14 constraint alldifferent(differences);
15
16 % Symmetry breaking
17 constraint differences[1] < differences[(m*(m-1)) div 2];
18
19 solve :: int_search(marks, input_order, indomain_min, complete)
20   minimize marks[m];
```

Listing 2.1: *MiniZinc* model of the Golomb Ruler problem.

marks on the ruler, while parameter `n` is defined as a function of `m` and serves as an upper bound on the ruler's length.

Decision variables can be identified by the keyword `var` present in their declaration item. A single item can declare either individual decision variables, or arrays of variables, including multidimensional arrays. The format for declaring an individual decision variable and an array of decision variables is as follows:

```
var <domain>: <name>;
array[<dimensions>] of var <domain>: <name>;
```

For example, the item in line 4 of the Golomb ruler model declares an array of decision variables of length `m` called `marks` (more precisely, its index set is $1..m$), where each mark is an integer decision variable with domain of $0..n$ for each of the marks in the problem, denoting their positions on the ruler. Another array of (auxiliary) variables `differences` is declared in line 6 using an array generator expression in line 7. Generator expressions are similar to the list comprehension popularised by the Python programming language and the mathematical set notation from which they originate. In this case, the expression effectively generates an array with elements equal to the differences between the positions of any two marks. Note that the length of the array is omitted in this case (i.e. the index set is not provided), since it can be inferred from the expression.

Constraint items start with the keyword `constraint` followed by some Boolean expression, which must evaluate to "true" and usually involves at least one decision variable. In the Golomb ruler example the constraint item in line 9 declares one of the simplest constraints: it restricts the domain of a variable (the first element of the `marks` array) to value `0`, effectively assigning it to the variable. The purpose of this constraint is to fix one of the ends of the ruler in order to avoid finding equivalent solutions: for every solution $x_1 = v_1$, $x_2 = v_2$ ... $x_m = v_m$ equivalent solutions can be trivially constructed of the form $x_1 = v_1 + k$, $x_2 = v_2 + k$ ... $x_m = v_m + k$, where $k \in \mathbb{Z}$ and $(v_m + k) \in D(x_m)$. As we are looking for the shortest Golomb ruler, starting the marks at `0` is an intuitive decision.

It is a good practice to disallow such equivalent (often referred to as symmetric) solutions by adding *symmetry breaking* constraints such as the one in line 9, as this significantly reduces the search space. Another symmetry breaking constraint appears in line 17, which declares a single disequality relation, stating that the difference between the first two marks must be smaller than that of the last two marks. Note that the original definition of the problem does not require this relationship. It is easy to show, however, that for any given solution of the Golomb Ruler problem, another solution can always be trivially constructed by conceptually reflecting the ruler.

A constraint item can effectively represent a conjunction of constraints. For example, the constraint item in line 11 declares a constraint of the form `marks[i] < marks[i+1]` for each of the values of `i` in the set $1..(m-1)$.

MiniZinc additionally supports *global constraints*, which define some complex relationship among multiple variables. For example, the *alldifferent* global constraint states that no two variables from an array can hold the same value. The item in line 14 imposes such relationship on the variables of the `differences` array. Although the *alldifferent* constraint is logically equivalent to its *decomposition* (a conjunction of pairwise disequalities on the array's variables), it is often beneficial to keep the higher level structure in the model. This is due to the fact that many solvers offer propagation algorithms for global constraints that are more efficient than those used for their decomposed versions.

Include items act as placeholders for code from a different file that should be pasted during the compilation process (see Section 2.5.1), allowing multi-file models. The item in line 13, for example, pulls in a standard decomposition of the *alldifferent* constraint in case it is not supported by the target solver.

Lastly, each model in MiniZinc has to contain exactly one solve item, indicating whether the problem is a COP with an objective function, or the problem is a CSP, and thus simply requires all the constraints to be satisfied. For example, the item in line 19 specifies that the position of the final mark should be minimised, thus indicating that the problem is a COP.

Each item in a model can be annotated with some optional information that the target solver might take into account. It is common, for example, to annotate the solve item with a search annotation as shown in line 19. A solver, in this case, is instructed to perform a *complete* (exhaustive) search by assigning variables from the `marks` array in input order, that is, in order from its first element to its last. The `indomain_min` option specifies that the smallest values from the variable domains should be tried first.

2.5.1 MiniZinc Compilation Process

Users of constraint solvers (modellers) often use a high-level language to develop models. These models are then compiled into lower-level language models that solvers can understand and solve. The presence of this compilation step allows modellers to use useful abstractions and thus makes developing (complex) models easier. At the same time, however, it hinders the modeller's understanding of the underlying solving process: any information that solvers can provide about their execution is generally presented in terms of the lower-level model, which is likely to be unfamiliar to the modeller. As a consequence, the compilation process can be problematic in profiling systems, as they ultimately need to communicate the results of their analyses to the user. In order to show potential modifications that a model could undergo when compiled to a lower-level language, I demonstrate the compilation process using a popular high-level modelling language MiniZinc and its low-level counterpart FlatZinc.

A FlatZinc model is a flat list of decision variables, simple constraints, and a solve item. It does not contain loops or complex, conjoined constraints. A MiniZinc instance can be compiled down to its FlatZinc representation, or FlatZinc model, using the MiniZinc compiler.

Different solvers support different sets of constraints and variable types. For this reason, FlatZinc representations of the same MiniZinc model often differ when they are compiled for different solvers.

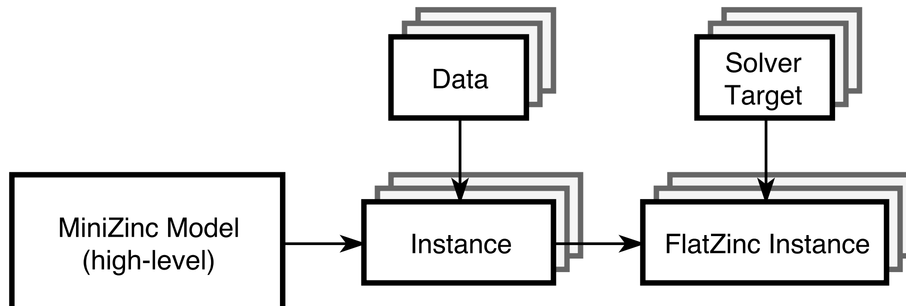

Figure 2.3: MiniZinc Compilation Process.

The compilation process is illustrated in Figure 2.3. As already mentioned, to increase the reusability of a model, its parameters are often specified in a separate data file, and a conceptual combination of them forms an *instance*. For each solver the MiniZinc compiler takes a problem's instance and creates a FlatZinc instance using the kinds of constraints and variable types the solver is designed to deal with. Note that a single MiniZinc model can thus be used to create a multitude of FlatZinc instances.

An example of a FlatZinc instance of the Golomb Ruler problem with 4 marks is shown in Figure 2.2. The model in this case is compiled for Gecode, a modern constraint solver [Schulte et al., 2016]. Note that all variables from arrays in the model have been renamed and flattened, and as a consequence, they are indistinguishable from single variables. For example, the variables `X_INTRODUCED_1_`, `X_INTRODUCED_2_`, `X_INTRODUCED_3_` (declared in lines 3 to 5) represent `mark[2]`, `mark[3]`, `mark[4]` from the MiniZinc model, respectively, as indicated by line 10. Note, however, that `mark[1]` does not have an associated variable, and it is substituted with 0, instead: its value was inferred from the constraint in line 9 from Listing 2.1 during the compilation process. Additionally, this constraint is not present in the FlatZinc instance, as it cannot affect the execution process any further. In larger examples, it is not uncommon for the compiler to additionally introduce auxiliary variables (called *introduced* variables) for expressing complex relationships between the *user-defined* variables.

For this particular model and solver, all constraints from the MiniZinc model are represented by only two kinds of constraints in the FlatZinc model. One is `int_lin_le`, which represents a linear inequality expression. For example, the expression in line 14 is equivalent to the following expression:

$$\texttt{X_INTRODUCED_1_} \leq \texttt{X_INTRODUCED_2_ - 1},$$

which states that `mark[2]` must be positioned closer to the beginning of the ruler than `mark[3]`. A similar relation for variables `mark[3]` and `mark[4]` is stated in line 15. Note that the two constraints together represent a single constraint item from the MiniZinc model (line 11 in Listing 2.1).

Being able to describe a problem once and use the same problem description with many solving technologies is a powerful capability that solver-independent languages like MiniZinc enable. The convenience of using these languages, however, comes at a price: it makes it more difficult for the modeller to reason about model executions as solvers operate on FlatZinc models, whose relation with the original model is not very clear.

Note that this fact imposes a challenge on execution profiling (which is the focus of this thesis) for the same reason: solvers can only produce profiling information in FlatZinc terms, that is, its flattened variables and constraints. For example, if a search decision is made on an introduced variable, it would require the modeller to analyse the automatically generated FlatZinc instance thoroughly in order to understand the meaning of that decision. An effective profiling system thus should be able to interpret the information provided by the solver and relate it in MiniZinc terms for the modeller.

```
1  predicate all_different_int(array [int] of var int: x);
2  array [1..2] of int: X_INTRODUCED_4_ = [1,-1];
3  var 1..16: X_INTRODUCED_1_;
4  var 0..16: X_INTRODUCED_2_;
5  var 0..16: X_INTRODUCED_3_;
6  var 0..16: X_INTRODUCED_8_ ::var_is_introduced :: is_defined_var;
7  var 0..16: X_INTRODUCED_9_ ::var_is_introduced :: is_defined_var;
8  var 0..16: X_INTRODUCED_10_ ::var_is_introduced :: is_defined_var;
9  array [1..4] of var int: mark:: output_array([1..4]) =[0,X_INTRODUCED_1_,
10   X_INTRODUCED_2_,X_INTRODUCED_3_];
11 array [1..6] of var int: differences = [X_INTRODUCED_1_,
12   X_INTRODUCED_2_,X_INTRODUCED_3_,X_INTRODUCED_8_,
13   X_INTRODUCED_9_,X_INTRODUCED_10_];
14 constraint int_lin_le(X_INTRODUCED_4_,[X_INTRODUCED_1_,X_INTRODUCED_2_],-1);
15 constraint int_lin_le(X_INTRODUCED_4_,[X_INTRODUCED_2_,X_INTRODUCED_3_],-1);
16 constraint all_different_int(differences);
17 constraint int_lin_le([1,-1,1],[X_INTRODUCED_1_,X_INTRODUCED_3_,
18   X_INTRODUCED_2_],-1);
19 constraint int_lin_eq([1,-1,-1],[X_INTRODUCED_2_,X_INTRODUCED_1_,
20   X_INTRODUCED_8_],0):: defines_var(X_INTRODUCED_8_);
21 constraint int_lin_eq([1,-1,-1],[X_INTRODUCED_3_,X_INTRODUCED_1_,
22   X_INTRODUCED_9_],0):: defines_var(X_INTRODUCED_9_);
23 constraint int_lin_eq([1,-1,-1],[X_INTRODUCED_3_,X_INTRODUCED_2_,
24   X_INTRODUCED_10_],0):: defines_var(X_INTRODUCED_10_);
25 solve :: int_search(mark,input_order,indomain_min,complete)
26   minimize X_INTRODUCED_3_;
```

Listing 2.2: *FlatZinc* model of the Golomb Ruler problem (m=4) compiled for Gecode.

2.6 Related Work

Previous work on profiling of constraint programs can be categorized as follows: search tree based visualisations, visualisation of variable domains, visualisation of constraints (including specialised visualisations for global constraints), detailed views of propagation steps, and visualisations of solutions.

This section starts by briefly outlining the evolution of profiling methodologies and introduces the associated tools. The contributions of these tools are then summarised based on the above categories. The examination of executions' search trees has been the most effective way of analysing solver executions, and it is the central theme of my work. Many search tree visualisations have been employed in the past, and I summarise them first. I then show the few existing techniques for comparing different executions

through comparing their search trees. Finally, I briefly summarise the work related to other categories, such as the visualisations of variable domains and constraints, in order to give a complete picture.

2.6.1 Brief History of Profiling of Constraint Problems

One of the earlier efforts dedicated to the profiling of constraint programs was presented in [Meier, 1995]. This work outlined some basic profiling goals and principles, such as the fact that one should be able to *compare* different models and to see the effect of a given constraint (possibly redundant) from a model. The paper additionally recognized the importance of examining the labels on the path (from the root to the current node) as a way of determining why a solution is not found quickly. The author demonstrated how these goals and principles were addressed in *Grace*, a graphical debugger for the ECL^iPS^e Constraint Programming system [Schimpf et al., 2012], which provided a basic view of domains on the path from the root to the current node, but was lacking a search tree visualisation as such.

Soon after, the Oz Explorer [Schulte, 1996] was developed, a debugger for the Oz constraint solver. It was the first system to utilise a search tree visualisation for constraint programs. The search tree was interactive and could be used to control the search, or to inspect information on variable domains for a given node. The way the tree was visualised in Oz Explorer had significant influence on subsequent profiling systems.

The success of these early profiling tools for constraint programs prompted the establishment of the DISCiPl project [Deransart et al., 2000], a large-scale European research project carried out in 1996-1999. The project aimed to develop a formal methodology for analysing constraint program executions and to build tools around it, focusing both on performance debugging (that is, profiling) and correctness debugging. It was largely influenced by the search tree visualisation of Oz Explorer, but additionally provided detailed views on domains, variables, and constraints. The project resulted in a considerable amount of new tools, roughly one per constraint language developed by the groups involved in the project, including Prolog IV Visual Debugger [Bouvier, 2000], APT Tool [Carro et al., 2000b], the profiling tool for CHIP [Simonis et al., 2000], and VIFID/TRIFID [Carro et al., 2000a].

Systems like the APT Tool, the Prolog IV Visual Debugger and the profiling tool for CHIP mainly focused on the search tree visualisations. The Prolog IV Visual Debugger, for example, presented the first pseudo-3D view for search trees, while the APT Tool experimented with adding statistical information directly to the tree, and the profiling tool for CHIP provided alternative ways for visualising search. On the other hand, VIFID/TRIFID focused on visualisation of domains of variables and their evolution during the search.

Importantly, while all of the aforementioned profiling systems developed effective techniques for profiling, those systems were tightly coupled with their host platform, and thus their usefulness was restricted to their host platform users. To some extent, this changed with OPL Studio [Bracchi et al., 2001], which was built as an IDE for the OPL language, and enabled profiling based on the tracing facilities provided in the ILOG Solver. While it provided similar profiling capabilities to those of the existing systems, such as a search tree visualisation, path visualisations and propagation views, its novelty lies in the use of a client-server architecture. In this architecture the profiler (server) and ILOG Solver (client) communicated via sockets, with the two components being only loosely coupled. This made the profiling with OPL Studio available to any system that implemented the

same XML-based protocol. Additionally, in order to aid in integrating new systems, the authors provided a C++ library that encapsulated the protocol and the communication layer.

There is no evidence of other constraint systems implementing the protocol of OPL Studio to take advantage of its profiling capabilities. However, given the high cost of developing a profiler, it became apparent that investing in research of solver-independent profiling could yield significant benefits. The same rational lead to the birth of OADymPPaC [Deransart, 2004], another large European project concerned with profiling of constraint programs, which was carried out from 2001 through 2004. The authors recognised the importance of solver-independent profiling as to: (a) encourage the sharing of debugging tools between different parts of the community, and (b) enable users to more easily switch between constraint solving systems. Indeed, one of its main contributions was `GenTra4CP` [Langevine, 2005] (later defined more formally in [Deransart, 2011]), a generic format for communicating between constraint solvers and profilers. The format is designed to capture pre-defined events about search (failure, solution, backtrack/backjump, etc) and propagation (awakening of a constraint, domain reduction, etc). A debugging/visualisation tool could choose which information to use and which to ignore, and the level of detail provided by GenTra4CP could be changed dynamically.

As part of the project, a notable debugging tool CLPGUI [Fages et al., 2004] was developed, which used the GenTra4CP protocol for communicating with constraint solvers. Similarly to the OPL Studio, it employed a client-server architecture, with its components communicating via sockets. Interestingly, the authors of CLPGUI additionally proposed a protocol complementary to GenTra4CP for controlling executions, and used it to enable a two-way communication between a solver and the debugger. These protocols, however, were very detailed and complex, making integration with new constraint solvers hard. As a consequence, GenTra4CP did not find widespread use [Deransart, 2011].

More recently, CP-Viz [Simonis et al., 2010] was proposed as another attempt at creating a system-independent profiler, which greatly simplified the integration with the solver. To achieve this, CP-Viz put some restrictions on its functionality; for example, it did not support controlling the search, viewing individual propagation steps, or displaying the constraint graph.

The system works in a *post-mortem* mode, that is, it can analyse executions only once they are finished. CP-Viz requires a solver to create an XML log of an execution with the information necessary for profiling. After the log file is completely parsed by CP-Viz, the execution can be analysed.

Many constraint systems implemented support for CP-Viz profiling, including ECL^iPS^e, G12/FD, OR-tools, and Choco. To the best of my knowledge, however, the profiling tool of CP-Viz has not been extensively used in recent years. Partly, this could be because of some conceptual limitations of the system, namely, its post-mortem nature: for example, the user did not see the progress of the execution in real time, and thus, could not determine (or guess) if it should be run until completion. More importantly, the support for comparison of executions, crucial for model improvement, was very limited in CP-Viz.

Although no profiling system for constraint programs seems to have attained widespread use, many of the techniques developed within these systems are worth mentioning. The following sections categorise and summarise the contributions to the profiling techniques.

2.6.2 Search Tree Visualisations

Possibly the earliest mention in the literature of visualisation of search trees for (relatively large) combinatorial problems was presented in [Held et al., 1971], where a variation of the branch-and-bound procedure was used to solve instances of the well-known Traveling Salesman Problem. The visualisation was not interactive and contained very little information in addition to the general tree structure, so it could only be used for giving the user a general idea about the search.

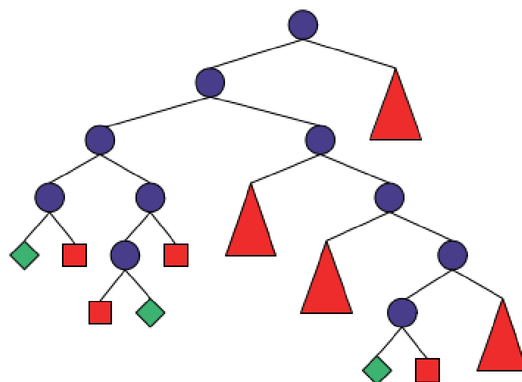

Figure 2.4: Search-tree visualisation in Oz Explorer, taken from [Schulte, 1996].

As mentioned earlier, the Oz Explorer [Schulte, 1996] was one of the first systems to utilise a search tree visualisation for constraint programs (an example is shown in Figure 2.4). The tool allowed a search tree to be built incrementally, thus visualising the search in real time. It also utilised powerful abstractions to enable visualisations of large search trees. One is that of a virtual canvas (possibly larger in dimensions than the monitor screen), where the user had an option to view only part of the canvas, navigate it by means of scrollbars, and zoom in/out when necessary. Another powerful abstraction is the use of *collapsed* nodes, which represent entire subtrees, but occupy a small space on the canvas (comparable to other, single nodes). The user had the ability to manually collapse arbitrary subtrees (and expand them back), thus controlling which parts of the tree to focus on. To aid the user in this, failed subtrees could be collapsed automatically during search.

The debugger in CHIP borrowed many ideas from Oz Explorer: it similarly supported retrieving detailed information about the solver's state by reenacting the decisions on the path to the node, as well as collapsing uninteresting subtrees into single nodes. The tool additionally allowed colour-coding the nodes with a user-defined scheme, displaying, for example, the number of alternatives in the node (see Figure 2.5). Textual information on each node could also be configured by the user to show, for example, the name of the variable labelled, or the current depth in the tree. For large, deep trees, the visualisation allowed users to combine several choices into a single node with multiple children. The trees in this case would potentially become shallower, but occupy more horizontal space.

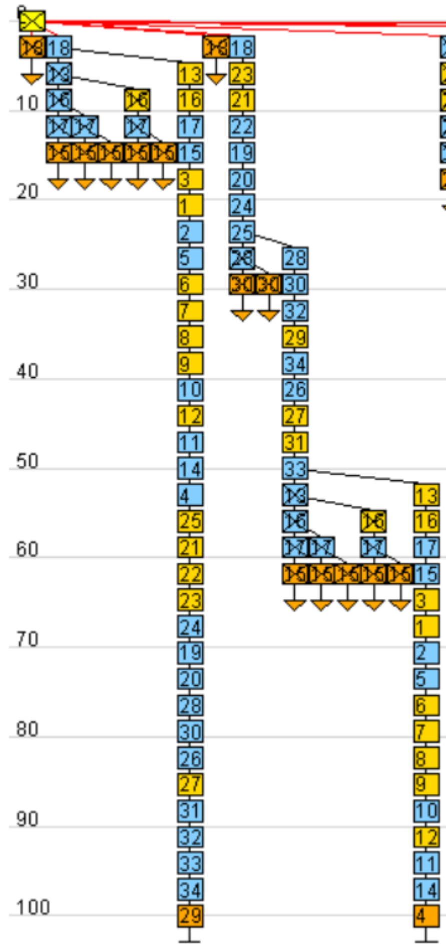

Figure 2.5: Search-tree visualisation in CHIP, taken from [Simonis et al., 2000].

A very different approach was available in OPL, where the search tree is visualised based on some statistical information about the execution, such as the number of propagation steps, or the time spent in each node. Figure 2.6 shows an example of such hybrid view (called a *Christmas Tree*) applied to a small problem. The visualisation uses a common colour scheme: blue, red, and green nodes represent decision nodes, failures, and solutions, respectively. However, different intensity levels convey additional statistical information: darker (lighter) nodes in the tree indicate that the domain reduction is large (small) at a given node. Additionally, the node's size represents the number of propagation events that fired at the node, giving an idea of how much time is spent in the corresponding part of the execution.

A somewhat similar approach was employed in the APT Tool, where nodes representing collapsed subtrees (shown as triangles in Figure 2.7) were depicted using different shades of grey based on the size of the underlying subtrees.

A few works explored the possibility to utilise the third dimension for search tree visualisations. Within the DiSCiPl project, the Prolog IV Search-Tree Visualiser extended the Prolog IV Debugger with a search tree visualisation, such as the one shown in Figure 2.8, where the third dimension is used to distinguish between different call levels.

The use of 3D was also employed in CLPGUI involving alternating planes as follows: if the depth of the tree is represented by the Z axis (with orthogonal axes X, Y, and Z), the layout procedure will position children in a line parallel to the X axis if the current decision level is odd, and parallel to the Y axis, otherwise (see Figure 2.9). Of course, a single 2D

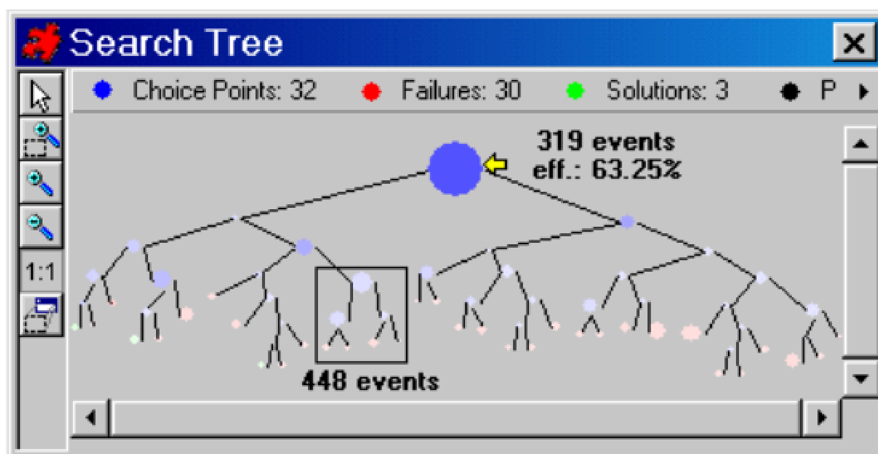

Figure 2.6: Christmas Tree Visualisation from OPL Studio, taken from [Bracchi et al., 2001].

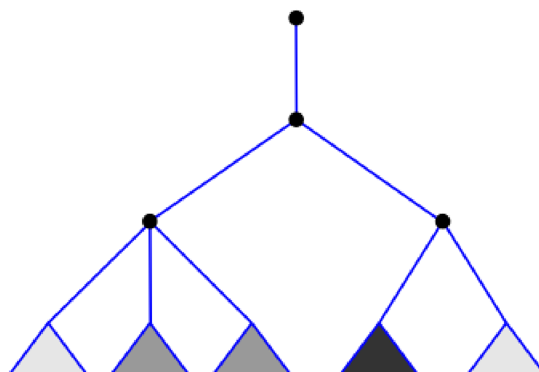

Figure 2.7: Search tree visualisation in the APT Tool, taken from [Carro et al., 2000b].

projection would be insufficient for interpreting such trees, and therefore, CLPGUI allowed the user to interactively explore the tree by rotating the view along arbitrary axes.

Radically new ways of visualising search trees have also been proposed in the literature. One example is the *phase-line display*, as proposed by Simonis et al. for CHIP. While the positions of nodes in this view are the same as they would be the traditional tree view (see Figure 2.4), there is a difference in the meaning of edges: they connect nodes assigning the same variables rather than depict parent-child relationships. Observing the phase-line display, the user could get an idea of how dynamic the search is: horizontal edges would indicate static order of variable choice, whereas crossings between edges would indicate where this order had changed. The authors suggest using the phase-line display to compare variable selection orders in different search heuristics. Figure 2.10, for example, shows two search trees in a single phase-line display, with the first and the second tree occupying, respectively, the left and the right halves of the view. In this case, the figure indicates the same variable selection order for the first several decision levels, followed by major differences as indicated by the skewed edges further down.

When trying to display the search tree for large executions, CP-Viz introduced another unconventional view for search trees, based solely on statistical information about nodes in the tree. The visualisation, which reportedly could be used to reason about the structure

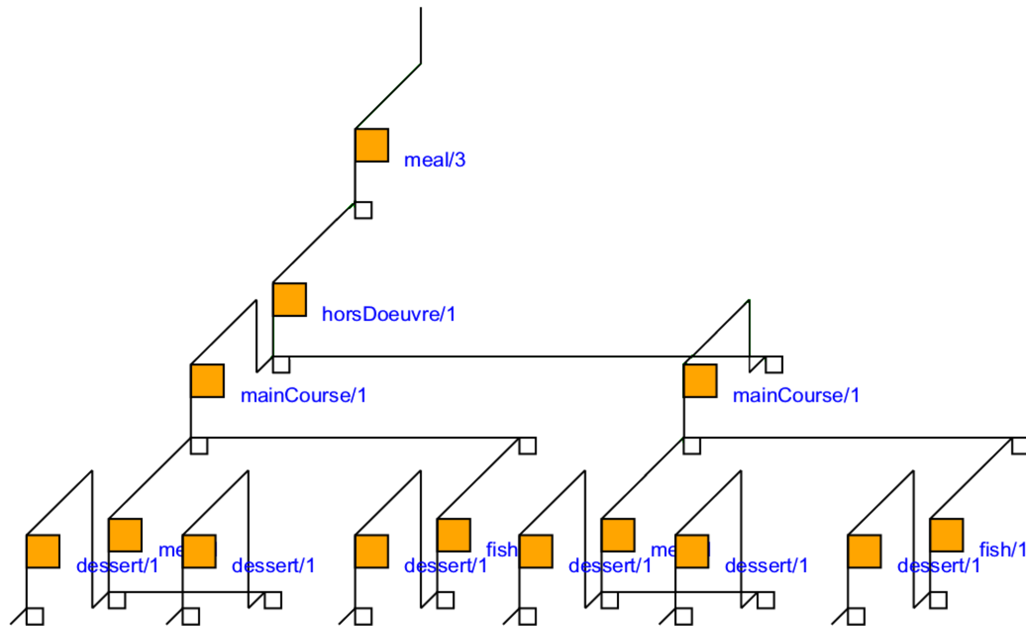

Figure 2.8: Pseudo-3D tree visualisation in Prolog IV Debugger, taken from [Bouvier, 2000].

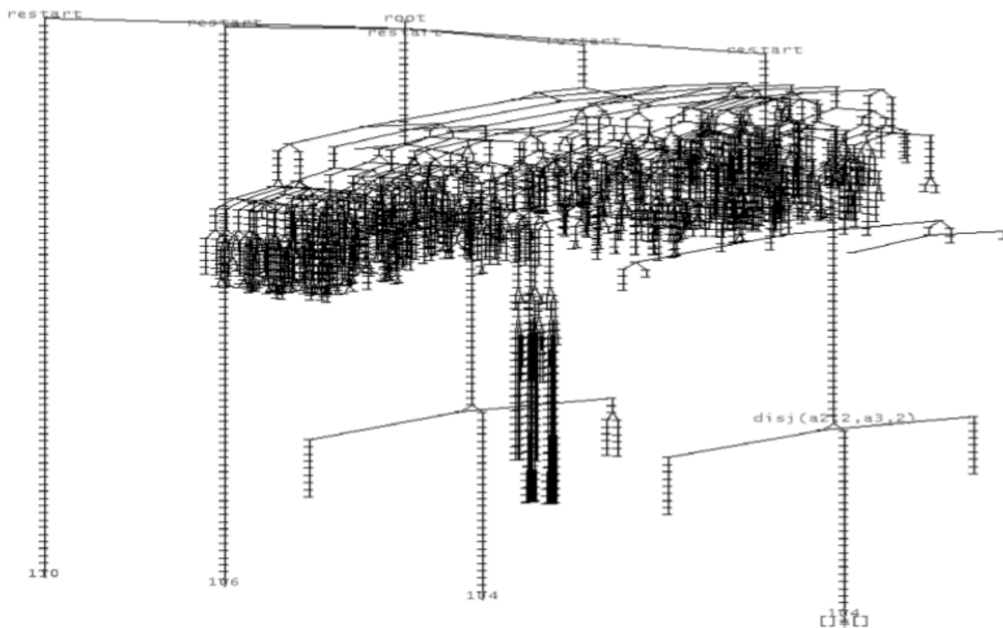

Figure 2.9: Search tree visualisation with alternating planes in CLPGUI, taken from [Fages et al., 2004].

of the tree, consisted of a graph showing the number of branch and failure nodes as a function of depth in the tree (see Figure 2.11, left). The same authors suggested using a *tree-map* as a visualisation paradigm for displaying search trees, where the nodes are shown as rectangles, and the child-parent relationship is shown through their containment (see Figure 2.11, right). In this visualisation, the size of a given subtree is represented by the area of the corresponding rectangle, and thus it is immediately apparent to the user.

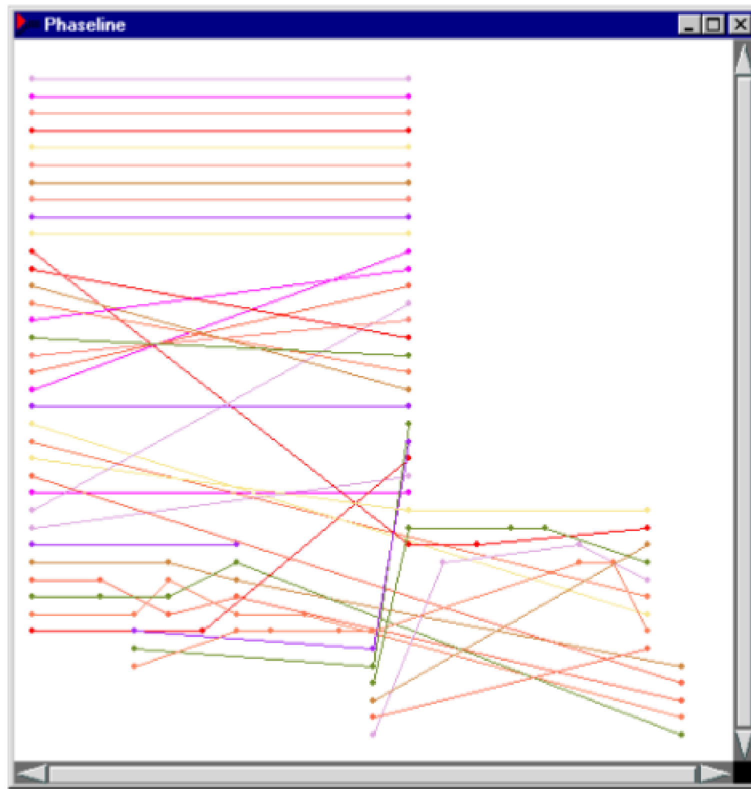

Figure 2.10: Phase-line display in CHIP, taken from [Simonis et al., 2000].

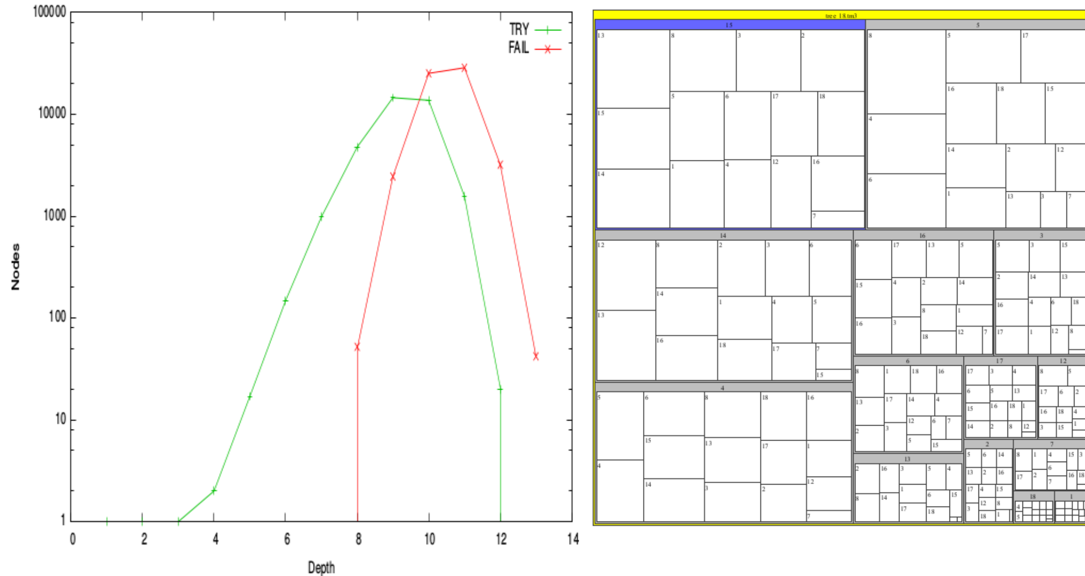

Figure 2.11: Node distribution per depth (left) and a tree-map (right) in CP-Viz, taken from [Simonis et al., 2010].

These alternative views clearly did not convey the same amount of information as the traditional node-link visualisations, and therefore, they were not intended to replace them. The relatively modern profiling tool CP-Viz, for example, still provides a search tree visualisation, resembling that of the pioneering Oz Explorer: note its triangle-shaped collapsed failed subtrees in Figure 2.12. The visualisation tool in Gecode (called Gist), which is a

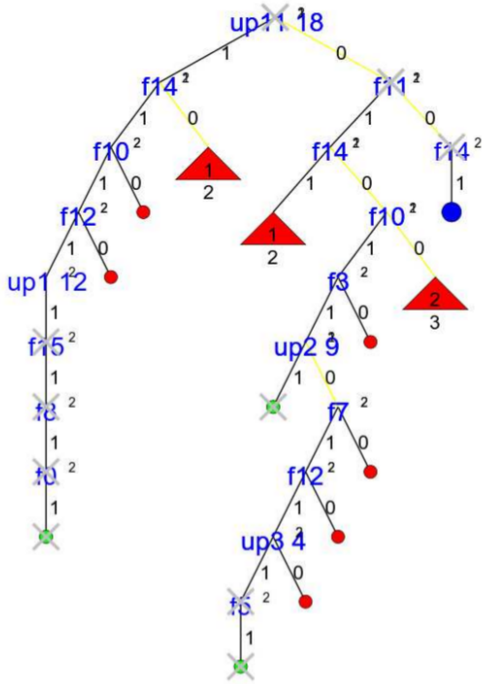

Figure 2.12: Traditional search tree visualisation in CP-Viz, taken from [Simonis et al., 2010].

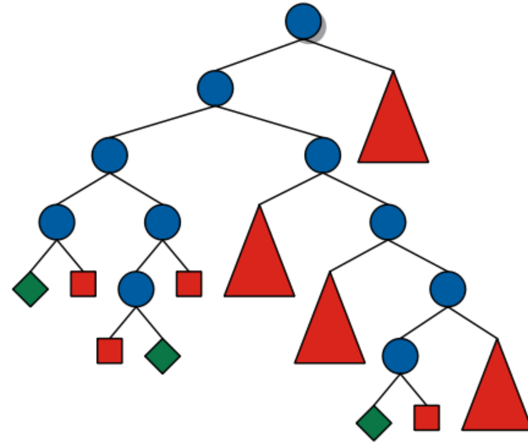

Figure 2.13: Search tree visualisation in Gist (Gecode).

modern successor of Oz Explorer, also provides search tree visualisation capabilities very similar to those of its predecessor (see Figure 2.13).

In conclusion, the traditional node-link visualisations appear to be indispensable in profiling of constraint problems, and prevail in modern profilers. Some ways of enhancing them have been proposed as well, in particular, those incorporating statistical information into the search tree visualisations. While such visualisations have clear advantages over the traditional visualisations, they have not seen wide adoption, possibly because they are less familiar for most users.

2.6.3 Comparison of Executions

The importance of comparing executions was already identified in the early work for Grace [Meier, 1995]. The system allowed connecting two solver executions in such a way that all control commands, such as the labelling of variables, would be synchronised. During such executions, differences in variable domains could be highlighted using colour, thus providing a simple means of comparing executions.

With search tree visualisations becoming more commonplace, some systems considered comparing executions by drawing their trees side-by-side. Often, the two trees would be drawn as subtrees of a larger "supertree". For example, Figure 2.14 shows the comparison of two different search strategies in CHIP applied to the same problem. Search trees could be compared in a similar way in CLPGUI as shown in Figure 2.15, where the two trees correspond to the same problem solved with and without a symmetry breaking constraint (shown as the right and the left subtrees, respectively).

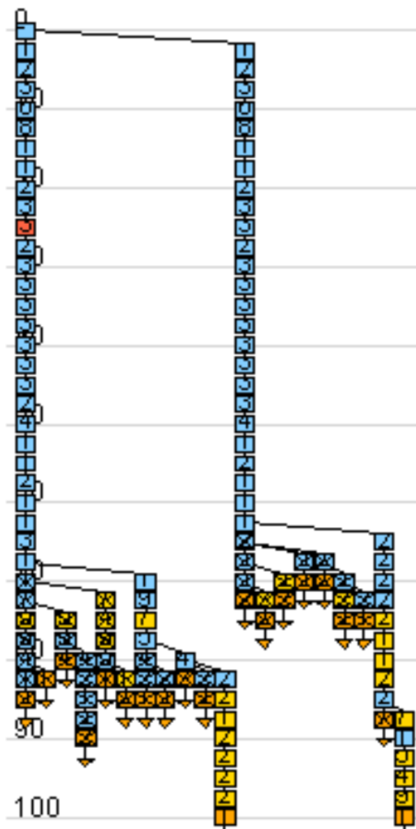

Figure 2.14: Side-by-side tree comparison in CHIP, taken from [Simonis et al., 2000].

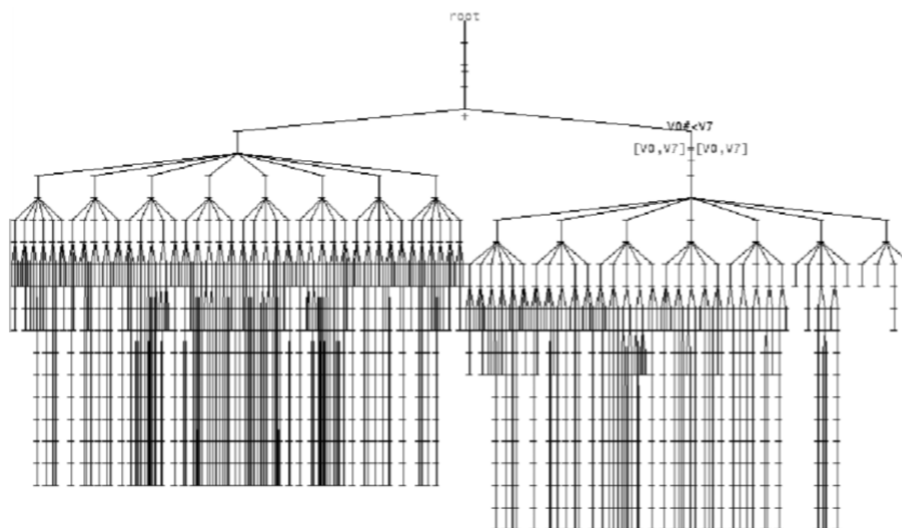

Figure 2.15: Side-by-side tree comparison in CLPGUI, taken from [Fages et al., 2004].

Interestingly, alternative search tree visualisations have been utilised for the side-by-side comparison as well. For example, as it has already been mentioned, the phase-line display of CHIP was mainly used for this purpose (see Section 2.6.2). CLPGUI also utilised its alternative viusalisation of a tree-map for side-by-side comparison (see Figure 2.16).

While novel and interesting, the above techniques for comparing executions either do not scale for large problems, or only provide a very general idea about how the two executions differ. For example, Grace allowed one to compare domains of variables in

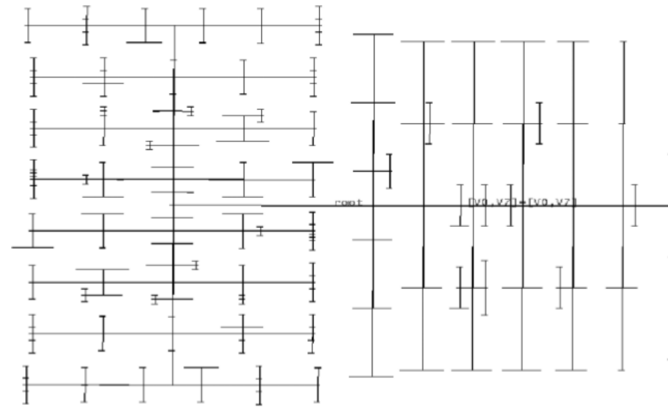

Figure 2.16: Side-by-side tree comparison of tree-maps in CLPGUI, taken from [Fages et al., 2004].

great detail, but only for a pair of nodes at a time. On the other hand, while side-by-side comparison allowed users to compare entire trees at once, only very general aspects of executions could be compared. This work investigates ways in which large executions can be compared in a higher level of detail.

2.6.4 Visualisation of Domains

Debugging for imperative languages, such as C++ or Python, usually involves inspecting the values that variables hold at a given point in the execution. In Constraint Programming, which adheres to the declarative programming paradigm, this is roughly equivalent to inspecting variable domains (which are sets of values, rather than single values).

Variable Stack

Domain	Dp	Variable
	0	312: 13.3
	1	312: 13.6
	2	312: 13.9
	3	123: 3.12
	4	231: 13.1
	5	123: 6.11
	6	231: 6.11
	7	231: 13.2
	8	231: 2.1

Figure 2.17: Variable stack in Grace, taken from [Meier, 1995].

Grace, one of the earliest debuggers for constraint programs, provided a visualisation of the *variable stack*, which displayed the following information for every variable currently labelled: the variable's domain, the current value, the values already tried, and those still to be tried (see Figure 2.17). A rectangle is used to abstract each domain, where vertical slices within the rectangle represent different values (or ranges of values), and the left-most (right-most) slices represent the smallest (greatest) values in the domain. The different kinds of values (tried, current, or remaining) are shown in different colours, which helps the user to identify the current position in the search space. This view is particularly important for *Grace* as it did not provide a tree visualisation, only showing the information relevant to the current path.

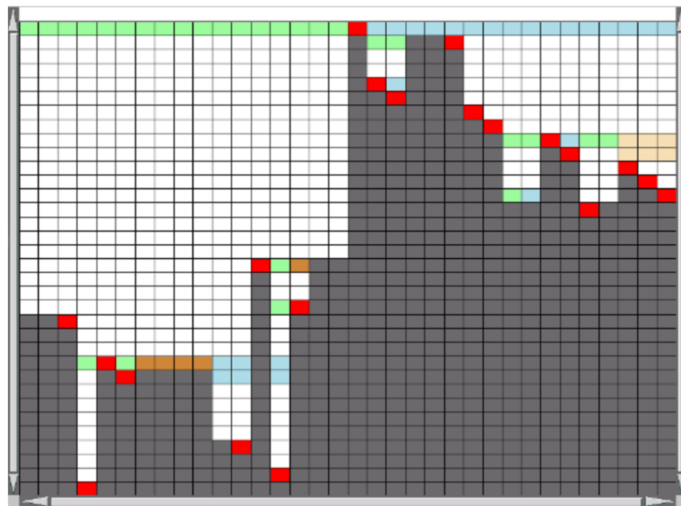

Figure 2.18: Variable update view in CHIP, taken from [Simonis et al., 2000].

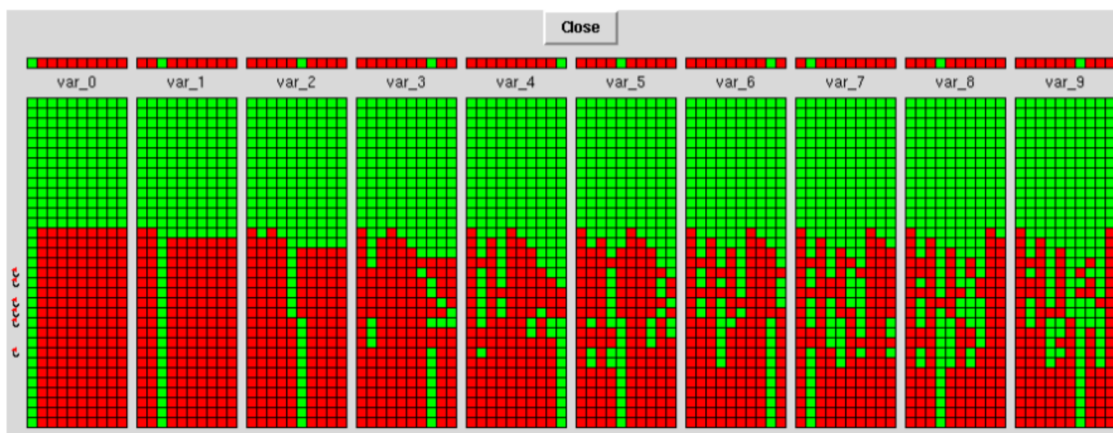

Figure 2.19: Evolution of variable domains in VIFID, taken from [Carro et al., 2000a].

In Oz Explorer, when the user double-clicked on a node, the detailed information regarding the state of variables for that node, such as the variable domains, would be dynamically recomputed and displayed to the user. This information, however, was conveyed in textual form only, and thus inspecting large domains was potentially a tedious task. Oz Explorer, however, partially compensated for the lack of domain visualisation by supporting user-defined views for (partial) solutions (see Section 2.6.7).

The CHIP system built on the ideas from Oz Explorer, and provided a more detailed matrix-based visualisation of the execution state. For the path from the root to the current node, a variable update view was available (see Figure 2.18), where all variables were shown in the horizontal axis, and the decision levels – on the vertical axis, with the depth increasing from bottom to top. This matrix used colour to show the evolution of the variable: whether it is assigned (red), its domain changed (grey), etc. When variable domains are relatively small, it is also possible to display the domain of a variable as a horizontal rectangle and coloured according to possible values. To deal with large executions, the visualiser in CHIP allowed users to annotate the source code to indicate which variables/constraints should be displayed in the corresponding views.

Carro et al. investigated techniques for visualising data evolution in constraint program executions, and demonstrated their implementation within the VIFID/TRIFID tool. Similarly to CHIP, VIFID/TRIFID uses a row of coloured cells to represent a domain of a given finite domain variable. Stacking such rows could display the evolution of a given variable. Figure 2.19 shows an example of such view for a problem with 9 variables, where rows from top to bottom show how the domain of each variable changes with each level of decision during execution. Although this view only displays evolution on the current path, the user can get a sense of the current position in the tree looking at the small curved arrows on the left, which indicate points of backtracking. For large domains the tool provided several abstractions. One is that of a *domain size*, a scalar value that represents a whole domain. Using this abstraction the authors proposed a 3D visualisation for displaying the change in the domains of variables over time: the variables and time were shown along two of the axes, while the remaining axis represented the domain size.

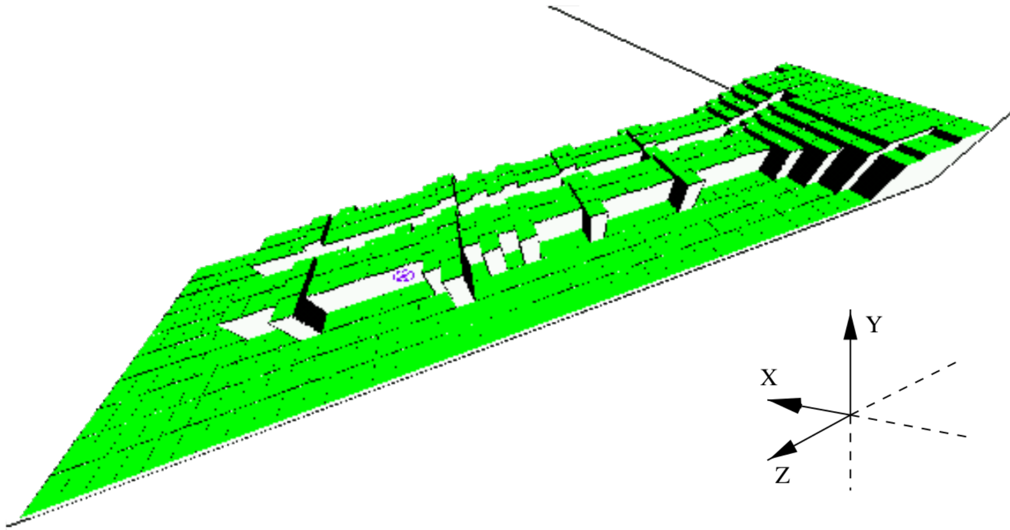

Figure 2.20: Evolution of variable domains in TRIFID, taken from [Carro et al., 2000a].

An example is shown in Figure 2.20, where variables are shown along the X axis, the time is shown along the Z axis, and the number of values left in the corresponding domains is shown along the Y axis. Although the actual values in the domains are hidden in this visualisation, it was still able to provide useful insights. The user could, for example, notice similar patterns in the domain change of some variables, and thus, conclude that the variables are highly related. A similar view (see Figure 2.21) was later integrated into CLPGUI.

Visualisation techniques for variable domains have been extensively studied, and profiling systems could benefit from many of the results. At this stage, the 2D matrix-based visualisations seem to be most prevalent and effective, however, with the advancement in 3D rendering and virtual reality technology becoming more commonplace, various 3D visualisation are likely to become as viable. While making further improvements in this area are out of scope of my work, I aim to ensure that these and future techniques for visualising variable domains can be easily integrated into my profiling framework.

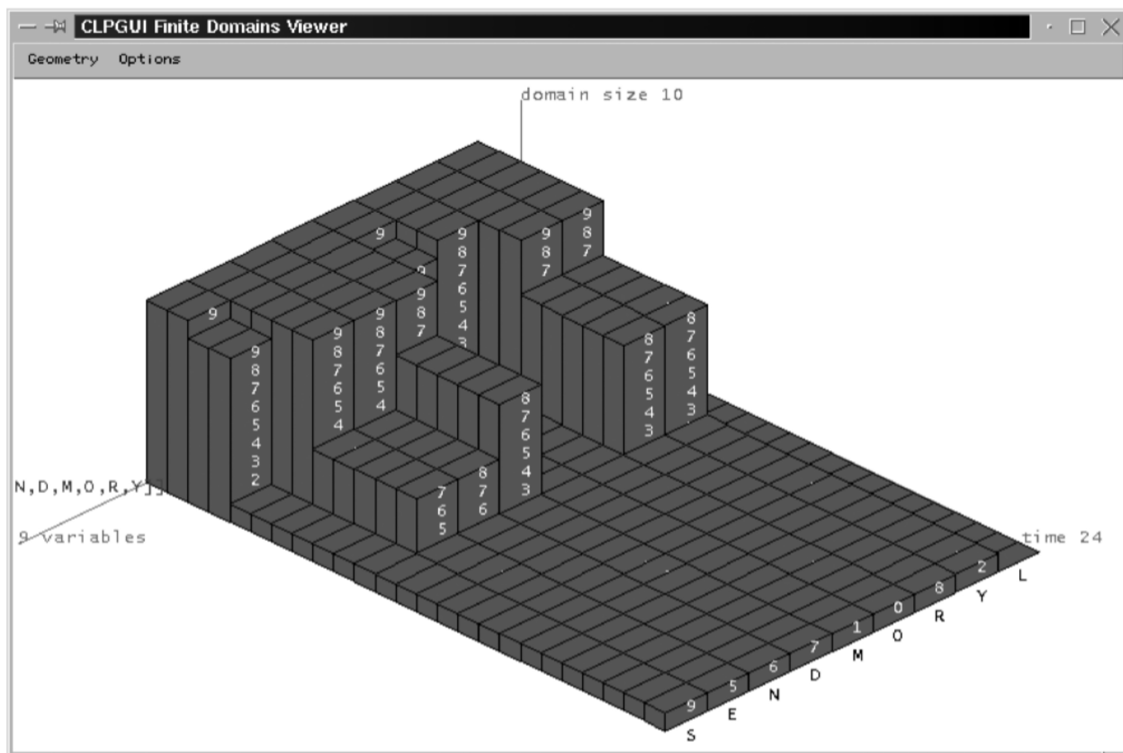

Figure 2.21: Evolution of variable domains in CLPGUI, taken from [Fages et al., 2004].

2.6.5 Constraint Visualisations

Matrix-based approaches have been the most commonly used for visualising constraints. For example, CHIP allowed users to view constraints via an incidence matrix (Figure 2.22), where the variables involved are shown along the X axis, and all constraints along the Y axis. The purpose of this view was to give the user an idea about the structure of the problem at hand. Additionally, constraints and variables could be colour coded to distinguish between, for example, different kinds of constraints, or to indicate whether a given variable has been assigned. Another matrix-based view on constraints in CHIP allowed users to see the evolution of constraints on the path from the root to the current node by representing the decision level in the tree as rows, and the constraints in the problem as columns. Different colours on the intersection of a given row and column could display the status of a constraint (whether it is woken, caused some domain update, etc.) on the corresponding level.

The work of Carro et al. for VIFID further investigated techniques for visualising constraints. A binary constraint could be visualised as a 2D-matrix, where the colour of cells indicates values that the constraint allows for the two variables involved. The authors recognise the difficulties involved in visualising n-ary constraints in the same way, and suggest a simplification that involves a series of 2D-matrices. Rows in the matrices represent variables, and columns represent values from their domains. For a selected variable, the values from its domain are evaluated (the said matrices representing each of the values), and rows for the remaining variables reflect how a given assignment changes their domains. An example of such view is displayed in Figure 2.24, where for a ternary constraint (involving variables `X`, `Y`, and `Z` with domains of 1..6), the matrices show how assigning Y to values from its domain affects other variables.

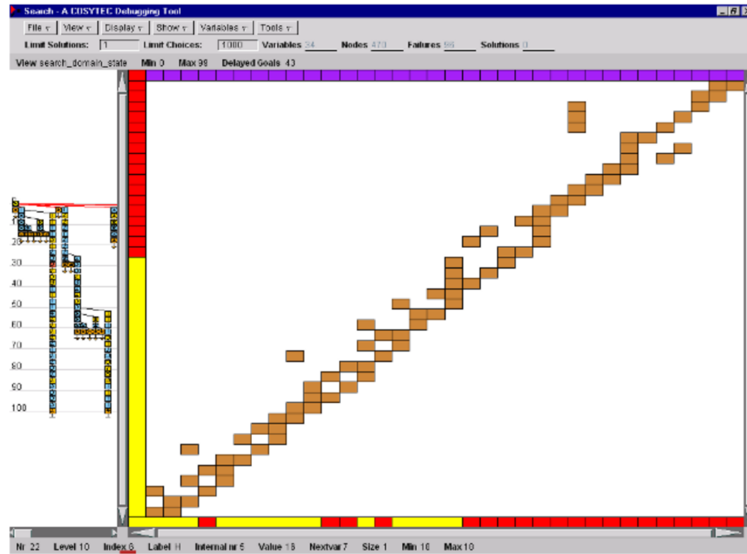

Figure 2.22: Incidence matrix in CHIP, taken from [Simonis et al., 2000].

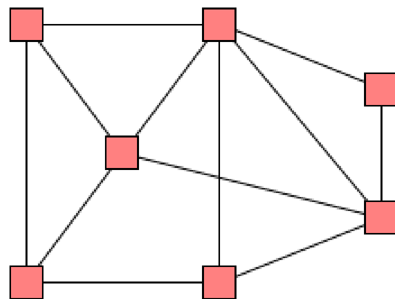

Figure 2.23: Constraint graph in VIFID, taken from [Carro et al., 2000a].

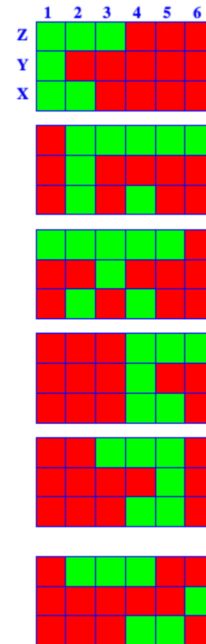

Figure 2.24: Visualising a ternary constraint by enumerating Y (VIFID), taken from [Carro et al., 2000a].

VIFID supported another way of visualising constraints: the constraint graph, where variables are represented as nodes, and the nodes are connected via an edge if their corresponding variables are connected via a constraint (an example is shown in Figure 2.23). Edges could be visually tagged with additional information (via the line thickness, colour, or length), such as: the number of times there had been propagation between the corresponding variables, or the actual number of constraints between them. The purpose of this view is to help the user find tightly connected variables in the problem, which could be used to discover good variable selection strategies.

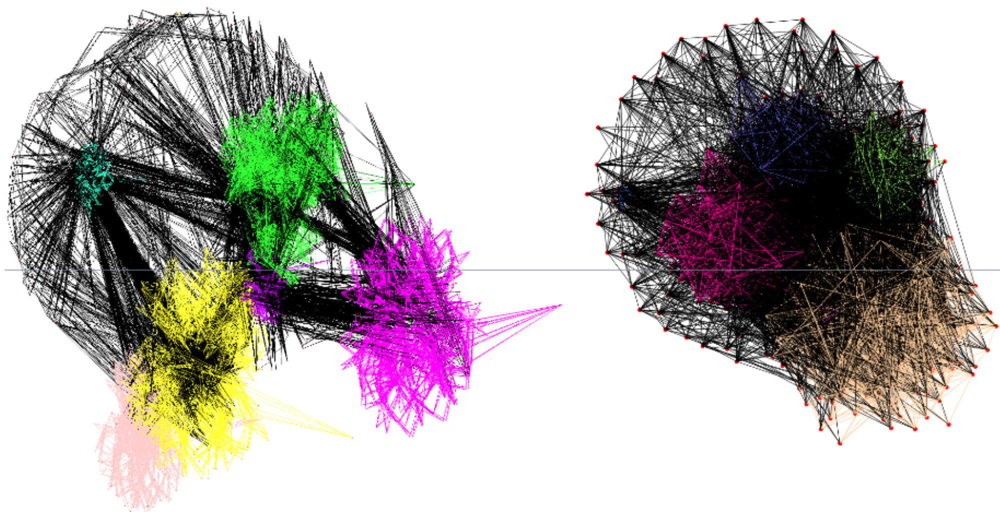

Figure 2.25: Constraint graphs in SATGraf for two different problem instances, taken from [Newsham et al., 2015].

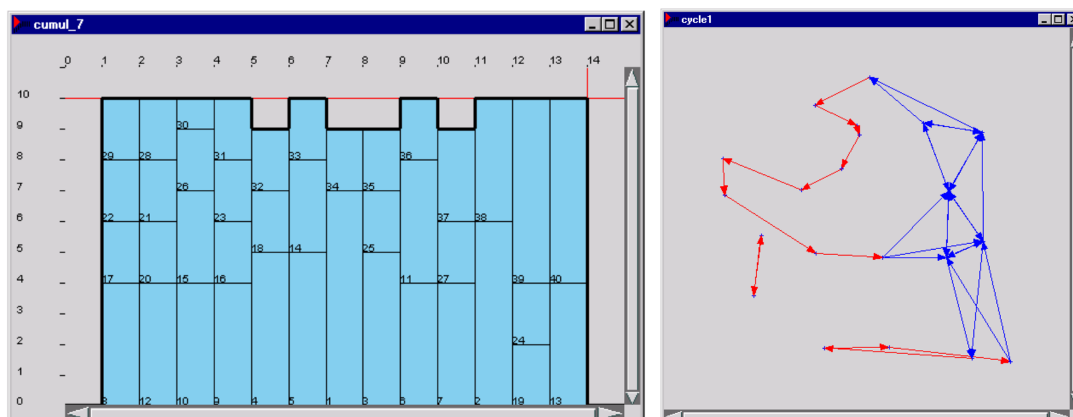

Figure 2.26: Bin-packing constraint (left) and Cycle constraint (right) visualisations in CHIP, taken from [Simonis et al., 2000].

The idea of a constraint graph view was also employed for Boolean satisfiability problems (SAT) in DPVis [Sinz et al., 2005], where clauses in a Boolean formula (along with learnt clauses) play the role of constraints. This has been inspired by studies showing that industrial problems (in contrast to randomly generated problems) tend to contain inherent structure, and modern SAT solvers can take advantage of this structure.

SATGraf [Newsham et al., 2015] built further on this idea by adding functionality to automatically identify groups of tightly connected variables, or *communities*, as well as to show the evolution of the constraint graph: how the communities morph into one another as new clauses get added, and other clauses get deleted. The authors indicate that when different communities are highlighted in different colours, as in Figure 2.25, the differences in the structure of several problems can become apparent.

Specialised visualisations have also been proposed in CHIP for problems involving global constraints, which provided the user with a set of common views. For a given constraint, the user would specify through an annotation which views should be used. For example, the 2-dimensional view shown in Figure 2.26 on the left could be used for the *bin-packing* constraint, which requires that n items of known size s_i be distributed among

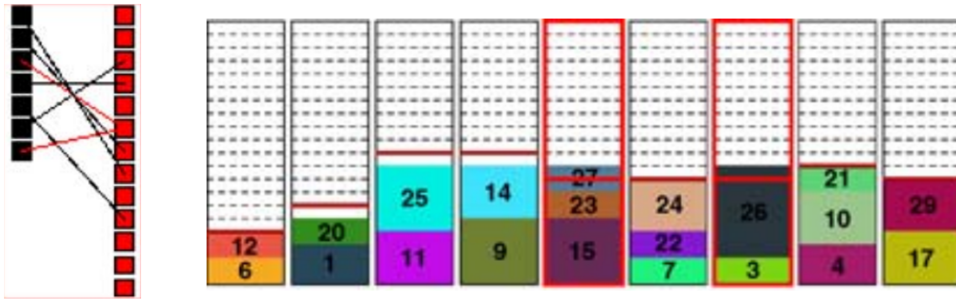

Figure 2.27: Global constraint visualisation in Comet: alldifferent (left) and bin-packing (right), taken from [Dooms et al., 2009].

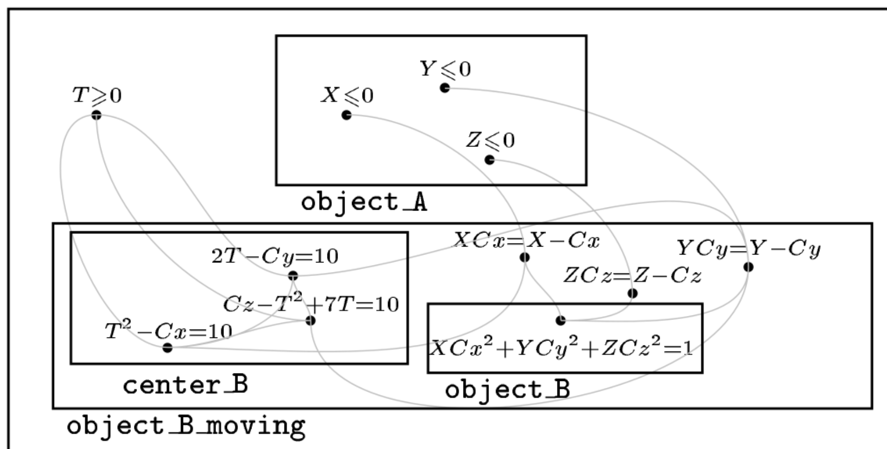

Figure 2.28: S-boxes of DiSCiPl showing hierarchy of constraints, taken from [Goualard et al., 2000].

m bins of capacity C. In this view, the m different bins are drawn along the X axis, and their capacity, as well as the amount of space occupied by the corresponding items, is shown along the Y axis. As another example, a node-link diagram would be appropriate for the *cycle* constraint, which allows one to define cycles in a directed graph.

A view similar to the one depicted in Figure 2.27 (right) has been employed in the Comet constraint system [Dooms et al., 2009], which additionally shows how the constraint is violated using rectangles outlined in red. Another view developed for Comet is well-suited for visualising the alldifferent global constraint constraint (Figure 2.27, left) by means of a bipartite graph, where each black node (variable) is connected to exactly one red node (value) representing assignments to variables. Similarly, violations of this constraint are highlighted: edges leading to the same value node are shown in red.

In order to deal with a large number of constraints and variables, the *S-box* abstraction was proposed within the DiSCiPl project [Goualard et al., 2000], which represents a group of logically related constraints. S-boxes can be shown as rectangles enclosing their corresponding constraints (see Figure 2.28). Additionally, they could contain other S-boxes, forming a hierarchy of constraints. The user could thus see the interactions between the S-boxes, and optionally zoom in on the appropriate level in the hierarchy.

In the area of constraint visualisation, matrix-based and graph-based approaches seem to be the most widely used. The existing matrix-based visualisations, for example, the incidence matrix in CHIP, can be effective for a small number a variables, but might not be viable for large problems. Graph-based approaches are mainly represented by constraint

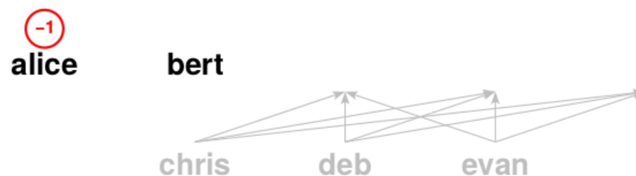

Figure 2.29: An example of a user-defined view for partial solutions in Oz Explorer, taken from [Schulte, 1996].

graphs, displaying variables as nodes and showing their constraint relationships through edges. These approaches seem to scale better for large problems (as demonstrated by SATGraf), but the insights they provide are quite coarse and do not seem to readily suggest model improvements. Contributing to the area of constraint visualisation is out of scope of my work, however, similar to the variable domain visualisations, I will aim to design a framework that could integrate these visualisations easily.

2.6.6 Propagation Views

The constraint system of CHIP introduced the idea of presenting detailed information about propagation to the user via matrix-based views. A matrix in such a view consists of rows representing propagation steps and columns for every variable involved, with cells indicating updates in variables as a result of propagation at the corresponding steps.

Enabling propagation visualisations clearly requires deep integration into the system, or a very detailed protocol, such as GenTra4CP. This level of detail is arguably too low for most users, and perhaps useful only for educational purposes or for propagation algorithm developers for correctness debugging. For these reasons, my work will not focus on displaying detailed propagation views.

2.6.7 Visualisation of Solutions

User-defined views for solutions, as well as for partial solutions, were already available in Oz Explorer. Figure 2.29, for example, shows a visualisation of a partial solution for the "Photo" problem, which requires arranging five people (Alice, Bert, Chris, Deb, and Evan) in a sequence according to their preferences. The problem in this case is modeled with five variables representing each person's position with the domains of 1..5. The visualisation in Figure 2.29 shows the assigned values for two variables (Alice and Bert), and the remaining values in the domains of unassigned variables (Chris, Deb, and Evan) using arrows. Although the same information could be conveyed as clearly in a textual form, in larger problems, visual problem-specific representation of domains is arguably more suitable than textual representation.

Visualisations of complete assignments, or solutions, have also been explored in the context of constraint-based local search (CBLS). The work in [Dooms et al., 2009] presents a compositional and extendable framework for declaratively visualising solutions. These visualisations are reminiscent of global constraint visualisations (like those available in CHIP) and often can be viewed as a combination of them. The authors used mainly two familiar paradigms: matrix-based and graph-based visualisation. For instance, the graph-based visualisation in Figure 2.30 shows a solution to the warehouse location problem, which involves finding the optimal location for warehouses given a set of customers on a

Figure 2.30: Solution visualisation for the warehouse location problem, taken from [Dooms et al., 2009].

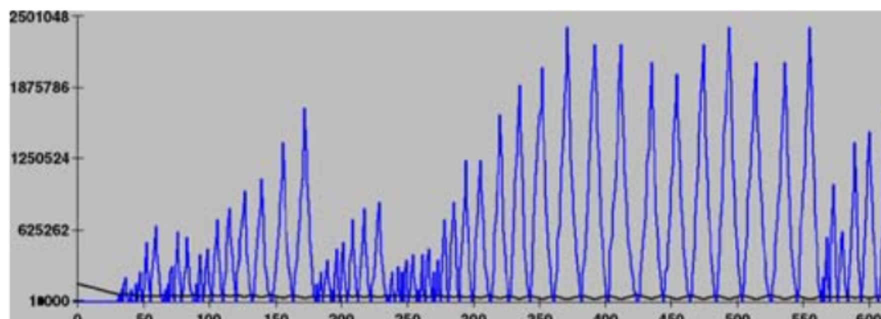

Figure 2.31: The change in the objective over time in CBLS, taken from [Dooms et al., 2009].

map. Visualisations of the objective value were also available within the same system. One example is displayed in Figure 2.31, which shows the change in the objective's value over time.

Solution visualisations are certainly valuable for both correctness and performance debugging, especially in large problems, and it is important for a profiling system to support them in principle. Contributing to this area of research falls outside the scope of my work, but I will aim to ensure that solution visualisation can be easily integrated into my framework.

2.7 Summary

Many profiling techniques proposed in the past have shown to be able to provide valuable insights into CP solver executions. However, most of these techniques have been implemented as part of solver-specific profiling systems (CHIP, Oz Explorer, APT Tool, Prolog IV Visual Debugger etc), and thus, the sharing of such profiling techniques among the systems was quite limited as it required significant amount of work. As a consequence, if the user wanted to use a specific profiling technique, they were limited to only those solvers that provided support for that technique. A few solver-independent tools, such as CLPGUI, aimed to support multiple solvers, but haven't received wide acceptance due to the complexity of their communication protocol and, as a consequence, the large amount of work required to instrument solvers.

CP-Viz proposed a much simpler protocol for debugging, which allowed it to be integrated with many solvers. However, to the best of my knowledge, it has not been extensively used in recent years. One reason for that could be some limitation of its design, namely, its post-mortem nature, which restricted users to profiling only finished executions.

More importantly, the existing profiling systems lacked support for analysing executions automatically, which appears to be necessary for large executions. In particular, one of the fundamental requirements for effective profiling: being able to compare executions before and after some model modifications, was hardly achieved.

Additionally, many modern paradigms for solving constraint problems, like the use of clause learning or restart-based search, are becoming commonplace. However, there is no mention in the literature of profiling for them.

This work aims to design a framework that is solver-independent and supports these modern solving paradigms. It then explores the possibility of analysing executions automatically to aid the user in examining large executions, and thus, addresses some of the above limitations in the state of the art.

Chapter 3

An Architecture for Profiling CP Search

The extent to which a profiling system can be useful is largely determined by the early, architectural decisions put into its design. This chapter first discusses the requirements that such a system will need to adhere to, and argue why each of them is necessary for *effective* profiling. Section 3.2 then discusses the design decisions that help achieve the said requirements as part of a complete architecture. Then, the protocol designed for communicating between the system and constraint solvers is described in detail in Section 3.2.2. Section 3.3 evaluates the system in terms of the overhead that it imposes compared to solver executions without profiling. Section 3.4 shows the solver modifications required for integrating a new solver into the system, and demonstrates the ease of such integration on the example of the popular solver Choco. Existing solver-independent profiling systems are summarised in Section 3.5 and compared with the system presented here. Finally, Section 3.6 summarises the chapter.

3.1 Requirements for Effective Profiling

The first requirement (R_1) for an *effective* profiling system is to provide a *search tree visualisation.* The most popular and familiar way of analysing executions discussed in the literature is through observing and exploring visualisations of their search trees. They are often used to control other profiling modules, such as domain or solution visualisations, by letting the user specify to which nodes to apply these visualisations. Additionally, a search tree visualisation – or more precisely the structure of the search tree – is necessary for a range of effective automated analysis techniques, which I introduce in later chapters.

The second requirement (R_2) for an *effective* profiling system is to be *efficient.* Large, real-world problems tend to require traversing search trees with many millions of nodes. It is with these hard to solve problems that profiling is often necessary. Enabling efficient profiling, that is, one that is capable of dealing with large, real world problems, is, therefore, of great importance and one of the goals that I aim to achieve in my work. It is at the same time challenging, as it involves being able to visualise, quickly navigate, and analyse large, multi-million node search trees. Some existing profiling systems, such as the search visualiser in Gecode, supported multi-million node search trees, and thus achieved this requirement.

The third requirement (R_3) for an *effective* profiling system is to be *solver-independent*. A great number of solvers are available to the users of Constraint Programming technology (for example, Gecode, Choco [Prud'homme et al., 2017], Oscar [OscaR Team, 2012], Chuffed [Chu, 2011] etc). While some of them are more suitable for a given problem than others, it is rather difficult to know beforehand which solvers will perform best (in terms of solving time, for example) for the problem at hand. Because of that, trying a variety of solvers when tackling a large problem is often part of the modelling process. To be able to analyse solver executions even when multiple solvers are used, requires solver-independent profiling. Further, some solvers with small, easy to understand code bases exist solely for learning purposes (e.g. [Hartert, 2017]). It can be beneficial to pair such solver with, for example, a search visualiser in order to improve the learning experience. However, such solvers will rarely be used for solving large, real world problems, since they cannot compete with state-of-the-art solvers such as *Gecode* or *Choco*. As a consequence, there is little incentive to develop dedicated profiling tools for them, and thus, such solvers would also benefit from a solver-independent profiler. Finally, a solver independent profiler would aid the development of new solvers, as it would save the time of developing dedicated debugging tools for them. As discussed in the previous chapter, very few existing profiling systems achieved the requirement of solver-independence. The only system that fully utilised its solver-independence is CP-Viz, which integrated many constraint solvers. Although other profiling systems, namely, OPL-Studio and CLPGUI were decoupled from the solver, they were difficult to integrate, and thus they have not been widely used with solvers other than those developed by the same groups of developers.

The fourth requirement (R_4) for an *effective* profiling system is to be able to visualise executions in real time. This is in contrast to post-mortem systems, which only allow one to visualise and analyse executions once they are finished. The only advantage of the post-mortem approach over real-time profiling is its relative ease of implementation, which is outweighed by its disadvantages. For example, post-mortem profiling does not allow the user to estimate the amount of search space already explored, and thus determine whether the execution should be run until completion or be interrupted earlier. On the contrary, real-time profiling allows the user to make such predictions (albeit not always accurately) based on the number of unexplored nodes and their depth in the tree. Additionally, when an execution is visualised in real time, the user can identify early if the search is not behaving as expected, and thus interrupt the execution and change it accordingly. For instance, the user could see that restarts are too infrequent in a restart-based execution, and change the restart policy, that is, the conditions for performing a restart during search. Among the previously developed profiling systems working in real-time are Grace, Oz Explorer, OPL Studio, and Gist (see Chapter 2).

The fifth requirement (R_5) for an *effective* profiling system is *search-independence*. In addition to using different solvers, users can often select from a variety of search tree exploration techniques. In particular, in addition to the regular depth-first search, more advanced techniques are becoming commonplace, including: *restart-based* search, *parallel* search, and *large neighbourhood search*. To be *search-independent*, that is, to be able to support modern and popular techniques, is clearly desirable for profiling. To an extent, a few existing systems meet this requirement by supporting arbitrary order for adding nodes to the tree ([Simonis et al., 2010], [Fages et al., 2004] etc).

The sixth requirement (R_6) for an effective profiling system is to be *extensible*. It is important for the profiling system to be easily adaptable to the advancements in solver design, and thus, to be able to support the new profiling techniques that they might require. For example, one modern solver technology – *Lazy Clause Generation* – is becoming increasingly popular as utilising it often results in significant performance gains on some

classes of problems. Understanding why some solver technology is effective (or ineffective) for a given model can often lead to model improvements, but is likely to require special purpose profiling. A profiling system thus needs to be *extensible*, that is, be able to adapt to new technologies in modern (or, indeed, future) solvers while remaining backwards-compatible with traditional solvers.

A solver-specific profiler can always be extended to the needs of its host solver. The challenge is in extending a solver-independent profiler, as all the supported solvers would need to be taken into account when a change, for example, in the protocol is made. In order to cover the needs of any profiling functionality that could be added in the future, systems like CLPGUI relied on the verbosity of their protocols and the ability of solvers to ignore some messages. Extending such protocols was, in theory, possible, as new types of messages could be added to them.

Finally, the effective profiling system needs to be able to *effectively compare executions* (R_7). When an execution is slow, the modeller often aims to improve its performance by either modifying the model, changing the search, or trying a different solver. Evaluating how such modifications affect the execution is necessary to know which of them to keep as well as which further modifications should be applied. Although an effect of a modification could simply be evaluated by comparing execution times, that approach requires that two executions finish in a reasonable time (and thus, might not need profiling). More importantly, the comparison of execution times is too coarse an approach that provides very little insight. As previously mentioned (see Chapter 2), the only means for comparing executions existing profiling systems provide is the visual (that is, manual) side-by-side comparison of the corresponding search trees. I show in Chapter 7 how a more fine-grained comparison can be an indispensable component of an effective profiling methodology.

In summary, I argue that *effective* profiling demands an architecture that meets the following requirements:

- (R_1) provides a search-tree visualisation;
- (R_2) efficient, i.e. it scales for large real world problems;
- (R_3) solver-independent, i.e. it can easily integrate different solvers;
- (R_4) search-independent, i.e. it supports different search exploration techniques;
- (R_5) real-time;
- (R_6) extensible;
- (R_7) supports effective comparison of executions;

3.2 Design Decisions

This section discusses architectural decisions that aim to enable *effective* profiling as defined above. In order to evaluate these decisions, and later, evaluate the profiling methodologies discussed in the following chapters, I have implemented a working prototype of the system – a set of tools capable of visualising and analysing the process of solving in Constraint Programming. I will refer to them collectively as CP-Profiler, or *the profiler* for short.

Clearly, separating the profiler from any solver executable is necessary (albeit not enough on its own) to achieve the property of being solver-independent (R_3). However,

there are further advantages to this design choice. For example, this is a more natural design choice for profiling distributed parallel executions, where the "solver" can be comprised of independent processes running across multiple computers. Additionally, this decoupling enables comparison of executions from different solvers (R_7).

The decoupling of the profiler and solvers requires some form of interprocess communication. A common approach involves writing a log (or "trace") of the solver execution to a file and letting the profiler read the file. The above approach has both advantages and disadvantages. In terms of disadvantages, writing to a file incurs unnecessary overhead, especially if its sole purpose is to enable communication rather than being used at some point in the future. Further, it often results in post-mortem behaviour, as the reading from a file would usually be performed once the trace recording is finished. In terms of advantages, it can be convenient to have the file as a persistent representation of an execution that can be replayed multiple times, or shared with other users. For example, if the execution is non-deterministic (that is, it depends on random or unpredictable events), which is the case, for example, in *parallel* search, this is the only practical way to reproduce the same results. Additionally, if there is a need to profile the same execution for a second time, reading it from a file is often much faster than than re-executing the model, and thus, a significant amount of the user's time can be saved.

The approach used in CP-Profiler is to omit the intermediary in a form of a file and communicate over sockets in a direct TCP connection between the solver and the profiling system. This approach is similar to those used in a few other existing systems, such as OPL Studio and CLPGUI. Note that the TCP stream can still be written to a file (which, in turn, can be read and re-sent to the profiler) with little effort, thus having the benefits of a persistent representation when it is desirable. CP-Profiler, for instance, provides that functionality via the "Search Logger" and the " Search Reader" tools.

3.2.1 The Profiling Architecture

The high-level architecture is shown in Figure 3.1. The profiler listens for messages from executions on a TCP socket. Solver executions are depicted on the left, where "Execution I" denotes a regular, single-threaded execution, and "Execution II" a parallel execution. (There is no real limit on the number of executions, but only two are shown here for simplicity.) To enable profiling, an execution simply needs to send relevant information (following the protocol described in Section 3.2.2) to the port on which the profiler is listening. In case of a parallel execution, the search tree is explored by multiple workers, each of them running in a separate thread or process and establishing their own connection to the profiler. Note that the profiling information could be sent to the "Search Logger" tool instead, which can record it to a file. At any point in the future, the "Search Reader" tool can be used to open the file and send the information to the profiler, mimicking the work of the solver.

As discussed earlier, the choice of a direct TCP connection encourages real-time processing of profiling information, and thus enables instant feedback as the executions could then be observed in real-time (R_5). The use of socket-based communication also allows the solver and the profiler to run on different computers, so that the profiler need not contend for resources with the solver.

Independently of how the communication is performed, a good choice of a protocol is essential. An important requirement for the protocol is to reduce the amount of bytes communicated as much as possible, since the size of each message will affect the overhead

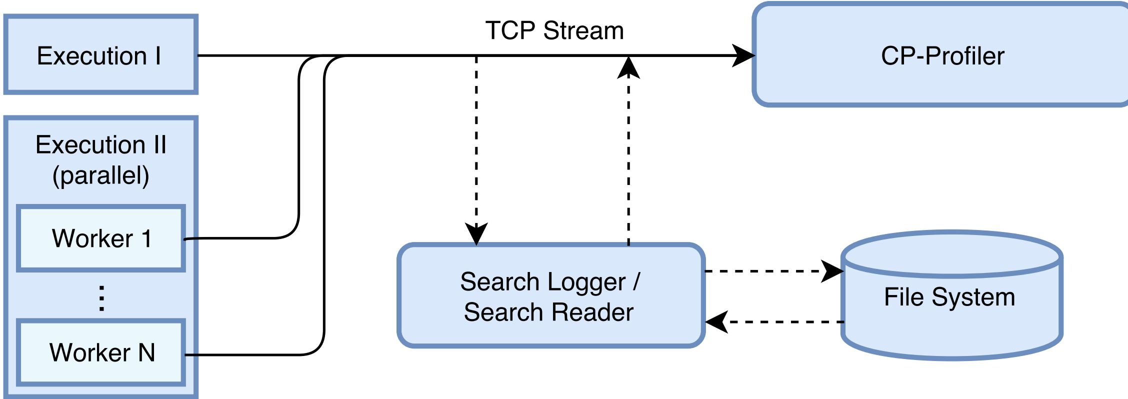

Figure 3.1: High-Level Overview of the Profiling architecture.

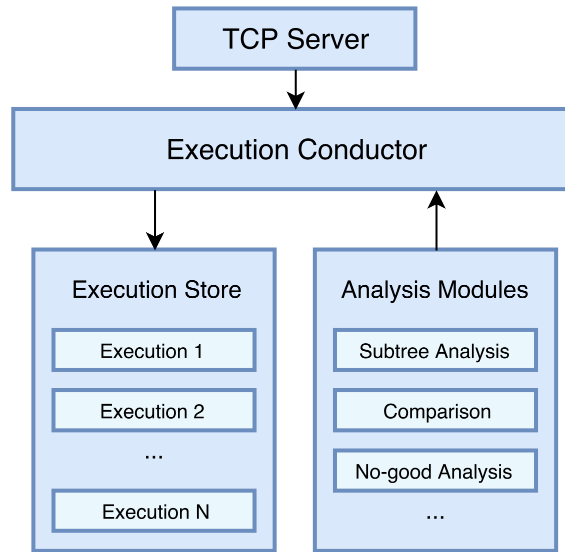

Figure 3.2: Internal Architecture of the Profiler.

incurred by the communication (R_2). This is especially important in the case of distributed search, where the bandwidth of the network can be a significant limiting factor. Additionally, when an execution is recorded to the file system, the choice of the protocol will determine the size of the resulting file. Another requirement on the protocol is to be flexible and extensible (R_6). A protocol is *flexible* if it allows optional fields to be communicated alongside the required fields. While required fields are necessary to represent the basic structure of the search tree, other information regarding the state of the solver at each search node can also be recorded, such as variable domains, the amount of domain reduction caused by constraint propagation, or the actual solution in the case of solution leaf nodes etc. A protocol is *extensible*, if it remains backwards compatible, even when it is augmented with new, optional fields. This property is important as it allows the profiler to adapt to new kinds of information that modern solvers can provide, while at the same time being compatible with more standard solvers. I designed a protocol that achieves these goals, and present it in detail in Section 3.2.2.

The architecture of the profiler itself is outlined in Figure 3.2. Different execution representations, which encapsulate necessary execution-specific information for one individual execution each, are shown as part of the "Execution Store" component. The Execution Conductor receives information that comes to the TCP Server and forwards it to the corresponding execution representation among the possibly many executions being profiled

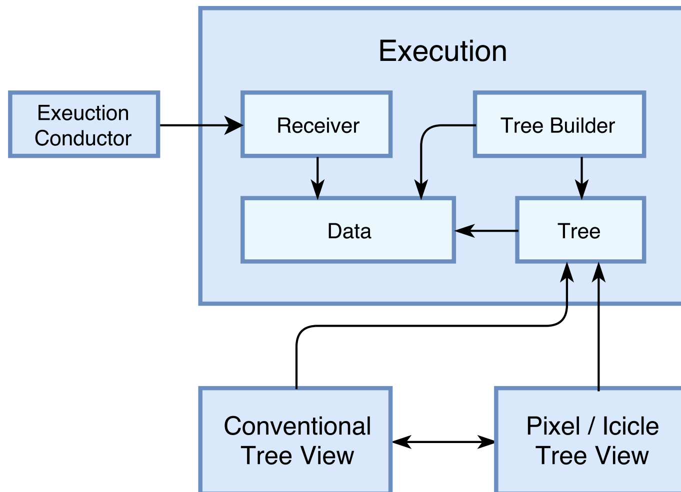

Figure 3.3: Profiler Components for an individual execution.

by the user. It is additionally responsible for determining whether a new execution representation needs to be created within the profiler by means of the *Start* message described later in Section 3.2.2. The functionality of the various analysis techniques, such as the ones discussed in the following chapters, is provided in the form of separate modules. These modules can query the conductor for the information necessary for the analysis, whether they work on a single execution or across multiple ones.

Each individual execution has four components: *Receiver*, *Builder*, *Data*, and *Tree*, whose interactions are shown in Figure 3.3. The Tree component is responsible for visualising the search tree of an execution (R_1). The job of the Receiver is to listen to all communications forwarded from the Conductor as relevant to its execution. In the process, it populates the data structures, represented as the Data component. The information stored by the Data component can be of two kinds: information necessary for constructing the tree, or additional information that can be queried at any point in time by one of the analysis modules. The information relevant for constructing the tree is constantly being observed by the Tree Builder component, which reacts to the changes and uses the newly arrived bits of information to construct the corresponding part of the tree.

Note that process of receiving information from a solver and the process of building the tree are only loosely coupled, and thus, they can be easily parallelised. Indeed, in my implementation the Receiver and the Tree Builder run in parallel to each other and to the GUI thread, thus taking advantage of modern multi-processor systems and allowing the tree visualisation to be interactive even during the solving process.

The following section describes the protocol that a solver needs to follow in order to communicate information to the profiler.

3.2.2 Communication Protocol

The proposed protocol distinguishes between the following types of messages: *Start*, *Restart*, *Done*, and *Node*.

The *Start* message is sent only once per worker in a solver execution – as the first message. In addition to instructing the profiler to create a new execution representation (if one has not been created by a different worker), it allows the solver to specify as parameters the *name* of the execution and its *execution id*, along with any other information relevant to the entire execution. The execution id is a globally unique identifier, and it is used

internally by the profiler to differentiate between different executions, while the name is used as the execution's descriptor to be presented to the user. For example, the name of the model file with a time stamp (the solver's starting time) can be used as the name. Once the execution id is obtained by the profiler, all subsequent messages communicated via the same TCP connection will be associated with the corresponding execution. Note that, although the profiler could generate a new unique identifier for every new connection (and not require solvers to do it), this approach would not work with parallel executions, which most likely require multiple connections with the profiler).

The *Restart* message is sent when a solver performs a restart, and thus, it is relevant for restart-based executions only. Its purpose is to tell the profiler that the following node messages correspond to a new tree, and thus, they should be handled accordingly.

The *Done* message simply tells the profiler that no further nodes should be expected (for example, the search was finished or interrupted by the user), and thus the execution can be finalised. This can involve, for example, releasing limited machine resources: sockets and memory are only needed until the tree is fully built. Note that, while the end of the execution can be inferred, for example, when there are no open nodes expecting children, the same does not hold for incomplete searches or executions with restarts.

Finally, the *Node* messages represent solver states right after all propagation is finished and communicate, among other things, the information directly required for building the search tree. *Node* messages have both required and optional fields. A minimal *Node* message contains (1) the node's identifier, (2) the identifier of the node's parent and (3) the node's position relative to its siblings, (4) the number of children, and (5) the node's status. The latter distinguishes, for example, nodes corresponding to failures from those corresponding to solutions, and can hold the following values (represented as numbers internally): *branch*, *solution*, *failure*, and *skipped*. It can be shown that this required information is sufficient to reconstruct a search tree.

Example 3.2.1. Consider the basic tree depicted in Figure 3.4. The information required to build this tree according to the protocol above (omitting the *Start* and the *Done* command messages) is presented in Table 3.1. The node ids (1) are unique for the execution, and thus allow the profiler to distinguish between different nodes. The `Parent Id` field (2) specifies how the node should be connected by communicating the unique id of its parent. For example, nodes labelled "Failed" and "Solution" have `Parent Id = 0`, and thus they are children of the node with `Id = 0`, which is the "Root" node in this case. The `Alternative` field (3) distinguishes between siblings, i.e. nodes with the same parent, and thus the same `Parent Id`. For the left-most child node the value of the `Alternative` field is `0`, for the second left-most it is `1` etc. Note that the "Root" node has a sentinel value of -1 for both Parent Id and Alternative to indicate that it has no parent. Finally, the `Number of Children` (4) field gives the number of children nodes, i.e. the number of alternative branching decisions made at that node. In the common case of the binary branching that number is 2 for all nodes except the leaves, where the number is 0.

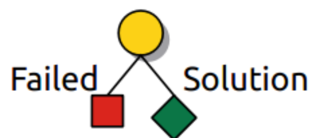

Figure 3.4: A simple search tree.

Note that knowing the number of children would not be necessary in a complete search, as it can be recovered by counting the number of nodes with a specific `Parent Id`. However,

Label	Id (1)	Parent Id (2)	Alternative (3)	Number of Children (4)	Status (5)
`Root`	`0`	`-1`	`-1`	`2`	`BRANCH`
`Failure`	`1`	`0`	`0`	`0`	`FAILED`
`Solution`	`2`	`0`	`1`	`0`	`SOLVED`

Table 3.1: Information sent to build a simple tree.

in order to accurately represent the state of the search half way through its execution (or when the execution was interrupted), it is helpful to see parts of the search space not yet explored.

It is important to note that the protocol makes it possible for nodes to be sent in any arbitrary order. The only requirement is that a node is created after its parent, which is naturally the case in any constraint solver. This property is important as it does not restrict solvers to any one search exploration strategy, thus allowing for profiling of restart-based search, parallel search and clause learning solvers that involve *backjumping* (see Chapter 7). The architecture thus meets the requirement of being search-independent R_4 defined above.

The information defined by the above protocol is sent in binary form in order to ensure the size of each message in bytes is small. The exact format specification is provided in the following section.

Note that implementing the above design decisions in a profiling system enables the discussed requirements for effective profiling R_1–R_7. Therefore, the proposed profiling system CP-Profiler is based on these design decisions.

3.3 Development and Evaluation of a Prototype

A working prototype of the system, CP-Profiler, has been implemented for the purpose of evaluating the system itself, as well as the profiling techniques discussed in the subsequent chapters. The search tree visualisation component of CP-Profiler is based on the visualisation module *Gist* from the *Gecode* solver [Schulte et al., 2016] (mentioned in the previous chapter), extended with additional functionality (see Chapter 4). All other components of the system have been developed from scratch.

CP-Profiler is currently undergoing an integration process with the MiniZinc IDE[1], an environment that enhances the experience of modelling and executing constraint programs. However, a standalone version is also being maintained, and its full implementation is available under an open-source license at `https://github.com/cp-profiler`.

3.3.1 The Protocol Specification

In the protocol used by CP-Profiler each message starts with a four-byte integer encoding the size of the remainder of the message in bytes. This information is followed by a single byte, which encodes the type of the message. The corresponding values are: `Node: 0`, `Done: 1`, `Start: 2`, `Restart: 3`. Then, if the message is of a `Node` type, the following fields are added in order: `id`, `pid`, `alt`, `children`, and `status`. These fields correspond to the

[1] http://www.minizinc.org/ide/

required information about each node, as discussed previously: the node's Id, its parent's Id, which alternative the node is, the number of children it has, and its status, respectively.

All multi-byte integer values are encoded using the two's complement notation in the *big-endian order*, that is, with the most significant byte written first. The node identifiers `id` and `pid` are represented as a pair of four byte integers each: one integer is used for identifying a node within the thread it is created by (each thread represents one of the parallel workers in a parallel search), and one for identifying the thread itself. The `alt` and `children` fields are represented by a single four byte integer each. The `status` field is represented by a single byte. Its possible values are: `SOLVED: 0`, `FAILED: 1`, `BRANCH: 2`, `SKIPPED: 3`. All five fields are required, it can thus be assumed that they are present and have static positions within a message. All other fields are optional and can come in any order.

The binary representation of each optional field starts with a byte identifying the field. The optional fields currently used are: `label`, `no-good`, and `info`, which correspond to a branching decision, a no-good representation (relevant for a learning solver), and arbitrary information, respectively). Other fields could be added in the future without breaking backward compatibility. Since the number of bytes that each field takes up is predetermined by the protocol, the number of bytes remaining for that field is also known once the field is identified. The exceptions are the fields of textual format (for example, label, solution etc.), which can significantly vary in size. For such textual fields the size is encoded in the four leading bytes as a four-byte integer. Field identifiers and their sizes in bytes are shown in Table 3.2.

field name	field id	size (in bytes)	field name	field id	size (in bytes)
id	n/a	8	label	0	unk.
pid	n/a	8	no-good	1	unk.
alt	n/a	4	info	2	unk.
children	n/a	4	...	...	...
status	n/a	1			

Table 3.2: Field types in the protocol.

Example 3.3.1. Consider an execution represented by a tree with a single node labelled "Root". A possible correspondence between a solver and the profiler in that execution is shown in Table 3.3, where the order in which different fields arrive is shown from top to bottom, and the rows are numbered for convenience.

The table shows that three messages are communicated in total. The first message is represented by rows 1 to 5 and contains the command to initiate profiling, and additionally provides the execution's descriptor. The first row shows the first four bytes representing the number 33, which is the size of the remainder of the message in bytes. The next byte (row 2) encodes the message type, which is *START* in this case. As the message represents a command, rather than a node, it only contains optional fields rather than those required to create a node. The next byte (row 3) encodes the type of the optional field provided. In this case it is the *info* field, which is used for communicating arbitrary information in the JSON[2] format. Being a string-based field, its representation starts with a four-byte integer (row 4) encoding the size of the following string (27 bytes). The following 27 bytes (row 5) encode the string “`{"name": "minimal example"}`”, which communicates the execution's descriptor as the value associated with the key “`name`”. By this point, all 33 bytes have been interpreted, thus the subsequent bytes correspond to a new message.

[2]https://www.json.org/

Row Number	**Bytes**	**Interpretation**
1	`00 00 00 21`	message size (33)
2	`02`	message type (*START*)
3	`02`	field (*info*)
4	`00 00 00 1B`	string size (27)

5	`7b 22 6e 61 6d 65 22 3a 20`
	`22 6d 69 6e 69 6d 61 6c 20`
	`65 78 61 6d 70 6c 65 22 7d`
		“`{"name": `”
		“`"minimal `”
		“`example"}`”

6	`00 00 00 23`	message size (35)
7	`00`	message type (*NODE*)
8	`00 00 00 00`	node id (0)
9	`FF FF FF FF`	node thread id (-1)
10	`FF FF FF FF`	parent id (-1)
11	`FF FF FF FF`	parent thread id (-1)
12	`FF FF FF FF`	alternative (-1)
13	`00 00 00 02`	children (2)
14	`02`	node status (*BRANCH*)
15	`00`	field (*label*)
16	`00 00 00 04`	string size (4)
17	`52 6f 6f 74`	“`Root`”
18	`00 00 00 01`	string size (1)
19	`01`	message type (*DONE*)

Table 3.3: Example of a correspondence between a solver and the profiler.

The second message is represented by rows 6 to 17 and contains information relevant for creation of the single (root) node of the search tree in the execution. Similar to the first message, the bytes in rows 6 and 7 represent the message size (35) and its type (*NODE*). Fields shown in rows 8 through 14 represent the required node related information in the following order: node id and the id of its thread, parent id and the id of its thread, alternative, number of children, and the node's status. The bytes shown in rows 15 through 17 represent the optional “label” field (row 15), which is a string of size 4 (row 16) and content “Root” (row 17). Finally, the last message is represented by rows 18 to 19 and contains the *DONE* command, which finalizes the execution.

Note in making decisions regarding the protocol for CP-Profiler, I have considered taking advantage of protocol generators, namely, the Protocol Buffers serialisation library [Google, 2015], which is the industry standard for constructing binary messages of dynamic size and content. However, the dedicated protocol was selected in favour of the one generated by Protocol Buffers, as the latter would either require a run-time dependency (associated with its dynamic library) or a large code base to be included along with integration library.

3.3.2 Efficiency Evaluation of the Profiling System

Visualising the search explored during a solver execution inevitably introduces overheads resulting from the need to (a) store the relevant information in memory, (b) draw the tree, and (c) update the tree as the search progresses in case the visualisation is a real-time one. Additionally, if the visualisation is decoupled from the solver, which is the case in the solver-independent CP-Profiler, the communication between the solver and the

visualiser (d) can slow down the process even further. How much these extra operations will affect the execution time depends on how much time the solver spends in processing the nodes. The overhead for communication is roughly constant per node (assuming that the messages are of similar size); therefore, the smaller the solving time per node, the bigger the communication overhead. To measure the overhead in a worst-case scenario, I built a simple model (see Listing 3.1) for which virtually no propagation can be done, and thus, the solver spends very little time in processing each node. In that model, line 1 declares parameter n, which determines the size of the array of decision variables y (declared in line 2), and thus allows one to control the size of the search space. An additional array of variables is declared in line 3, whose values are restricted by the unsatisfiable constraint declared in line 4. The search item in line 5 ensures that the values for the y array are assigned first, resulting in a large search tree.

```
1 int: n;
2 array[1..n] of var 1..2: y;
3 array[1..3] of var 1..2: x;
4 constraint forall (i,j in 1..3 where i<j) (x[i]!=x[j]);
5 solve ::int_search(y++x, input_order, indomain, complete) satisfy;
```

Listing 3.1: MiniZinc model with virtually no propagation.

Due to the small processing time required by this model for each node, it can be used to obtain an upper bound to the overhead introduced by the profiler. Note that, while one could use a model with no constraints and require a solver to produce all solutions to the problem instead, such an execution would not be representative of a regular solver execution: in a typical execution, there are much fewer solution nodes than other types of nodes. In fact, processing a solution node often requires more time (for example, in the Gecode solver) than processing a branch or a failure node, even when a small amount of propagation is necessary for those nodes.

Table 3.4 shows the execution times (wall-clock time in seconds) needed to solve different instances of this model with no visualisation, visualising using Gist (which incurs overheads from tasks a, b, and c as discussed above), and visualising using CP-Profiler (which incurs overheads from all a, b, c, and d). As expected, both Gist and CP-Profiler introduce significant overhead for this case, with CP-Profiler being approximately 7 times slower than the original execution. Nonetheless, the results also show that a solver-independent visualisation of multi-million node search trees can be obtained at a reasonable cost. Of course, on more realistic constraint models most of the time is spent on propagation. For example, Table 3.5 and Table 3.6 show the results on the Golomb Ruler problem and the well-known N-Queens problem, respectively, with profiling taking approximately 70–80% longer than the original execution.

n	Size (nodes)	No visual.	Gecode Gist (solver-specific visual.)	CP-Profiler (solver-independent visual.)	Overhead (CP-Profiler over "No visual.")
19	2.1M	0.64s	2.84s	4.68s	625.8%
20	4.2M	1.32s	5.75s	9.07s	585.2%
21	8.4M	2.68s	11.07s	18.13s	575.9%
22	16.8M	5.39s	22.78s	38.36s	611.4%
23	33.6M	11.09s	46.71s	73.24s	560.7%

Table 3.4: Time taken in a worst-case scenario (no propagation).

Instance (size)	No visual.	Gecode Gist (solver-specific visual.)	CP-Profiler (solver-independent visual.)	Overhead (CP-Profiler over "No visual.")
10 marks (0.1M)	0.91s	1.10s	1.25s	38.0%
11 marks (1.7M)	16.23s	20.44s	28.61s	76.2%
12 marks (13.2M)	156.61s	192.97s	284.09s	81.4%

Table 3.5: Time taken for the Golomb Ruler (GR) problem.

Instance (size)	No visual.	Gecode Gist (solver-specific visual.)	CP-Profiler (solver-independent visual.)	Overhead (CP-Profiler over "No visual.")
35 queens (0.3M)	2.46s	2.87s	3.81s	55.3%
36 queens (0.8M)	6.86s	8.01s	11.27s	64.3%
37 queens (3.9M)	34.72s	38.87s	57.64s	66.0%

Table 3.6: Time taken for the N-Queens problem.

Note that the current implementation of CP-Profiler is a prototype, and thus there are many opportunities to further improve its performance and reduce the overhead.

3.4 Solver Integrations

Similar to other profiling systems, my approach to profiling requires some solver instrumentation in order to produce the information necessary for profiling. In principle, any solver implementing the protocol defined above can be used with CP-Profiler. In addition, I have developed libraries for C++ and Java that encapsulate the protocol and network-related routines, and thus lessen the work required to integrate a solver. An example of a simple use of the C++ library is presented in Listing 3.2, where the program creates a tree as depicted in Figure 3.4.

Communication and node creation are encapsulated within an instance of the Connector class, which is created in line 4 and receives as argument the port number the profiler is listening on (its network address defaults to *localhost*). Between the calls to `connect` and `disconnect` (lines 5 and 19, respectively) that handle the network connection, the program first initiates a new execution on the profiler side by calling `start` in line 7 with the name of the execution as an optional parameter. It then creates three nodes and sends them to the profiler in lines 10 through 16. In this example the thread ids are omitted for all nodes, as the execution is single-threaded, and thus these fields can take default values (-1). Note that the required fields are always present by construction. At the same time, optional fields (such as *labels*, which normally indicate branching decisions in search) can be set via a builder pattern, which allows the user to chain methods and thus construct and send a node in a single line.

Thanks to the integration library, adding support for new solvers should not be an onerous task. For example, adding the profiling capabilities to Gecode and CPX (the two have similar architecture) required only about ten lines of code for each of their search engines. The integration of Choco (a solver whose source code I was unfamiliar with) for

```
1  int main() {
2    using Profiling::NodeStatus;
3
4    Profiling::Connector c(6565);
5    c.connect();
6
7    c.start("minimal example");
8
9    /// SEND ROOT NODE
10   c.createNode(0,-1,-1,2,NodeStatus::BRANCH).set_label("Root").send();
11
12   // SEND LEFT CHILD
13   c.createNode(1,0,0,0,NodeStatus::FAILED).set_label("Failure").send();
14
15   // SEND RIGHT CHILD
16   c.createNode(2,0,1,0,NodeStatus::SOLVED).set_label("Solution").send();
17
18   c.done();
19   c.disconnect();
20 }
```

Listing 3.2: Simple use of the integration library in C++.

most non-advanced profiling capabilities required adding about 30 lines of code, and took about 5 hours of work. Note, however, that most of that time I spent on familiarising myself with the source code. The exact code modifications applied to Choco for its integration are shown in Listing 3.3 and Listing 3.4, where the added lines are shown in green colour.

Apart from the two import statements (lines 1 and 2), all the modifications were done inside the SearchLoop class, which is responsible for the search tree exploration in Choco. Lines 6 through 9 extend the class with additional members. Line 6 initialises an instance of the Connector. Line 7 initialises a node counter (`nc`), which is used for generating node identifiers as the search progresses. Lines 8 and 9 initialise two stacks, which are necessary for keeping track of the node identifiers during the search tree traversal: parent id (`pid_stack`) and the id of the currently explored branch (`alt_stack`).

Lines 11 through 47 instrument the *loop* method. Line 13 notifies the profiler of a new solver execution. Lines 19 and 20 correspond to the start of the search, and initialise the aforementioned stacks with values for the root node. Lines 26 and 27 correspond to advancing to a new decision level, and extend the parent id stack with the identifier of the previous node. Note that the two "push" calls correspond to the two new children nodes (left-hand-side and right-hand-side) created in that case (binary branching is assumed). Line 28 increments the node counter for the left-hand-side node, and line 29 extends `alt_stack` with 0, indicating that the left-hand-side node is being explored. Similarly, lines 33 and 34 correspond to the exploration of the right-hand-side node. This time, however, `alt_stack` is extended with 1, indicating that the current node is the second alternative, or the right-hand side node in this case. Lines 38 through 40 correspond to search backtracking and, accordingly, remove the latest values from the stacks. Finally, lines 44 and 45 tell the profiler that the search is done and close the connection.

In Choco, certain procedures are taken out from the main search loop into separate methods. One is `openNode`, which is called for every new node in the search tree. My modifications to this method are as follows. Lines 52 and 53 are called when a new decision node is created, and send the required information about this node. Lines 56 through 64 are called when a solution is found: first, a textual representation of the solution is created in lines 56 through 60. Then, in lines 61 through 64, the solution node

```
1  import cpprofiler.Connector;
2  import cpprofiler.NodeStatus;
3
4  public class SearchLoop implements ISearchLoop {
5    ...
6    Connector c = new Connector(6565);
7    int nc = 0;
8    Stack pid_stack = new Stack();
9    Stack alt_stack = new Stack();
10
11   private void loop() {
12     ...
13     c.start();
14
15     while(alive) {
16       switch (nextState) {
17
18         case INIT:
19           alt_stack.push(-1);
20           pid_stack.push(-1);
21           ...
22         case OPEN_NODE:
23           openNode();
24           ...
25         case DOWN_LEFT_BRANCH:
26           pid_stack.push(nc);
27           pid_stack.push(nc);
28           nc++;
29           alt_stack.push(0);
30           downBranch();
31           ...
32         case DOWN_RIGHT_BRANCH:
33           nc++;
34           alt_stack.push(1);
35           downBranch();
36           ...
37         case UP_BRANCH:
38           pid_stack.pop();
39           alt_stack.pop();
40           alt_stack.pop();
41           ...
42       }
43     }
44     c.done();
45     c.disconnect();
46     ...
47   }
```

Listing 3.3: The use of the integration library in the Choco solver (part 1 of 2).

is sent to the profiler. Note that the optional info field is used in this case to store the solution representation.

The last modified method is `downBranch`, which is responsible for applying a new decision in the search. The added lines are 74 through 77, which send a node to the profiler that represents a failure. The optional info field in this case is used to communicate the reason of the failure, as provided by Choco.

```
48   private void openNode() {
49     ...
50     if (decision != null) {
51       ...
52       c.createNode(nc, pid_stack.lastElement(), alt_stack.pop(), 2,
53           NodeStatus.BRANCH).setLabel(decision.toString()).send();
54     } else {
55       ...
56       Variable[] vars = solver.getStrategy().getVariables();
57       StringBuilder solution = new StringBuilder(32);
58       for (Variable v : vars) {
59           solution.append(v).append(' ');
60       }
61       c.createNode(nc, pid_stack.lastElement(), alt_stack.pop(), 0,
62           NodeStatus.SOLVED)
63        .setLabel(decision.toString())
64        .setInfo("solution:" + solution.toString() + "").send();
65     }
66   }
67
68   private void downBranch() {
69     try {
70       decision.buildNext();
71       ...
72     } catch (ContradictionException e) {
73       ...
74       c.createNode(nc, pid_stack.lastElement(), alt_stack.pop(), 0,
75           NodeStatus.FAILED)
76         .setLabel(decision.toString())
77         .setInfo("reason:" + e.toString() + "").send();
78     }
79   }
80 }
```

Listing 3.4: The use of the integration library in the Choco solver (part 2 of 2).

3.5 Related Work

As already mentioned in the previous chapter, a few existing profiling systems were decoupled from a solver, and thus, could be used for solver-independent profiling. One such system is OPL Studio [Bracchi et al., 2001], which uses a client-server architecture, where the profiler (server) and ILOG Solver (client) communicate via an XML-based protocol over TCP sockets. Although the decoupling of the profiler from the solver allows other solvers to be profiled in principle, an integration library (encapsulating the protocol and the communication layer) is only available for ILOG-based C++ applications. In particular, instrumentation of such applications requires instantiating ILOG-specific classes `IlcSolverDebugger` and `IloSolver`. The former provides a means of controlling the profiler through a set of commands, while the latter is responsible for generating the search and propagation events as defined by ILOG Solver.

Among the commands that the OPL Studio supports are, for example the `displayResult` command for visualising solutions, the `sendConsole` command for sending arbitrary information to be displayed in the profiler's console, and the `newSearchTreeView` command for starting a new tree visualisation. Note that OPL Studio additionally allows users to specify the constraints to be communicated to the profiler through commands such as `registerVariable` and `registerConstraint`, which specify which variables and constraints should be tracked, respectively. In addition, conditional breakpoints can be

specified through, for example, `breakIfBound` command, which will activate when a specified variable is assigned a specified value. Despite its profiling capabilities, OPL Studio has been solely used by ILOG-based applications due to the lack of an integration library for them.

CLPGUI [Fages et al., 2004] was another solver-independent profiling system that, similarly to OPL Studio, used a client-server architecture. For communication between a solver and CLPGUI the protocol GenTra4CP was used, which was specifically designed for profiling of constraint executions. This protocol is XML-based, although its binary variant WBXML could alternatively be used in order to reduce the size of each message.

Solvers communicated to the system through a sequence of pre-defined events: nine for describing search, and six for describing propagation. Among the propagation events are those notifying of a domain reduction, awakening of a constraint, and unsatisfiability of a constraint. The search events relevant to this thesis are those indicating solution, failure and branch nodes, as well as the "back-to" event, which indicated the event of revisiting some previously created node during, for example, backtracking or backjumping. Additionally, CLPGUI allowed users to control solvers during execution. For example, one could dynamically add constraints or recompute the solver state of a given node to obtain extra information about the state.

Compared to CP-Profiler, CLPGUI supported more functionality in the form of profiling constraints and controlling the solver. However, despite its use of the "back-to" event to enable an arbitrary exploration order for nodes, CLPGUI did not support parallel search as the system relied on the notion of a "current node" and could not distinguish between different parallel threads. Additionally, the integration of solvers for profiling with CLPGUI required significant amount of work, and as a consequence, few solvers were integrated with the system.

The architecture of CP-Viz [Simonis et al., 2010] is different to the previously discussed solver-independent systems: due to its post-mortem nature, CP-Viz does not require a direct communication between a solver and the system, and thus uses file-based communication instead. In particular, solvers that support CP-Viz are required to create two XML-based log files per execution: one for the structure of the search tree, and another for detailed information about constraints and variables.

The structure of the search tree is represented by a list of node items. Each such item contains the node id, the id of its parent, the name of the variable in the search decision represented by the node, the size of that variable's domain, and the decision in a string form. In order for solvers to generate the log files required by CP-Viz, a communication interface has been proposed. The following are some of the methods used for recording the structure of the search tree in that interface:

```
1 addRootNode(int id)
2 addSuccessNode(int id, int parentId, String varName, int size, String choice)
3 addFailureNode(int id, int parentId, String varName, int size, String choice)
```

Listing 3.5: A few methods from the communication interface by CP-Viz.

The root node is created using the method in line 1, and success (solution) and failure nodes are created using the methods in lines 2 and 3, respectively. Note that the information recorded per node for CP-Viz is similar to the protocol used in CP-Profiler. One notable difference is the absence of the total number of children and the field to identify the position of the node relative to its parent (alt) in CP-Viz. Although the protocol in CP-Viz allows nodes to come in arbitrary order, the absence of these fields suggests that unexplored nodes cannot be shown in that system (and neither can skipped nodes).

Another difference is the absence of thread identifiers, suggesting that parallel search is not supported: although it is possible to maintain unique node identifiers across different threads, that would be inefficient, and the user would not be able to see the distribution of work among threads.

3.6 Summary

In this chapter I have proposed an architecture for profiling of CP executions that meets the requirements for effective profiling as defined in the beginning of this chapter. In particular, this architecture supports real-time search tree visualisation of large executions (R_1, R_2, R_5), which was demonstrated in Section 3.3.2. The efficiency of the system was achieved thanks to the use of a lightweight binary protocol, direct interprocess communication, and a modular, parallel architecture that takes advantage of modern multiprocessor computers.

While solver instrumentation will always be necessary to enable profiling, the proposed system is nonetheless solver-independent (R_3), as integrating new solvers is a relatively easy task due to the choice of a flexible protocol and the availability of integration libraries for popular languages (C++ and Java). The ease of integrating a new solver has been demonstrated (using the Choco solver as an example) in Section 3.4.

Additionally, the protocol is designed to support a range of modern search exploration strategies (R_4) and to be extensible (R_6), and thus it can be used with modern (or indeed, future) solvers, allowing them to communicate new information, potentially specific to them. This property of the system will be demonstrated in the following chapters using the examples of restart-based search (Chapters 4 and 6) and backjump search associated with modern learning solvers (Chapter 7).

Chapter 4

Visualisation of Search Trees

Whenever the execution of a program takes longer than expected, the programmer would like to analyse it and determine what undesired behaviour, if any, is the reason for the slow execution. Most solvers provide some form of coarse statistical information, or aggregate measures, about their executions. Such information often includes the number of different types of nodes in the search tree (e.g. solution nodes, failure nodes, branch nodes), the maximal or average depth in the tree, and the average propagation time. One could, for instance, see if propagation is taking too long, and it should be weakened in favour of exploring more of the search space. Alternatively, one could use the depth of the search tree and the number of unexplored nodes to roughly estimate which portion of the search space has been explored.

Coarse statistical information, while valuable, rarely allows a modeller to make a concrete decision on how to improve the execution of a model. A more common and, arguably, more effective approach is to use visual abstractions, since they are believed to be cognitively easier for a human brain to process than text, making the presentation of more detailed information viable. Common visual abstractions used for combinatorial problems include node-link diagrams for visualising search trees, constraint graphs for visualising connections between constraints, colored matrices for visualising the domains of variables, and line graphs for visualising how scalar values, such as the objective, change over time.

Perhaps the most successful use of visual abstractions in the literature is the interactive visualisation of search trees. Such visualisations allow users to examine the search tree of an execution in order to determine which parts of the search could correspond to some undesired behaviour and, hopefully, determine which changes to the program might remove that behaviour, thus speeding up the execution.

Traditionally, search trees are drawn as node-link diagrams, where each node represents a propagation step (a fixpoint) and each link represents a search decision. Such diagrams are relatively easy for humans to comprehend for a small number of nodes, for example, by following individual search decisions and examining individual nodes. Node-link diagrams, however, tend to become less effective fairly quickly as the number of nodes increases, and large, multi-million node trees are very difficult to comprehend and navigate. Such diagrams have poor *cognitive* scalability: while there exist fast algorithms for laying out and drawing large trees with many thousands or even millions of nodes, such trees are nearly impossible to examine by a human eye. As a consequence, a tree may *contain* the information we seek, but it is often so large that we struggle to *find* the interesting parts of the tree.

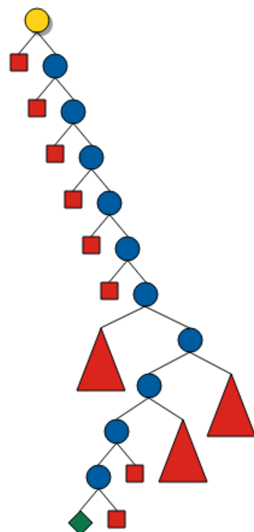

Figure 4.1: Search Tree with Collapsed Subtrees.

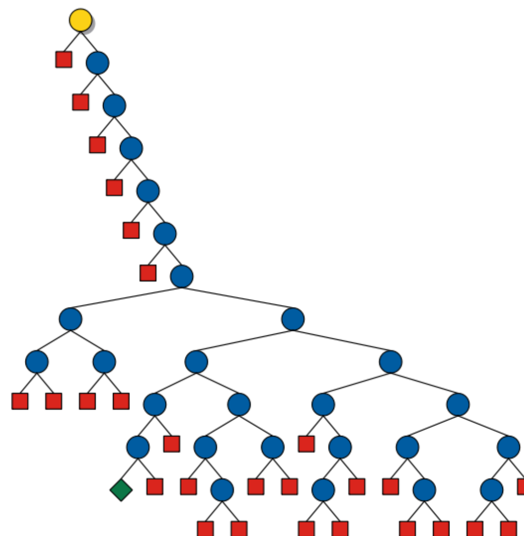

Figure 4.2: Search Tree (No Collapsing).

Various techniques for tackling the problem of scalability of tree visualisations have been considered in the literature. These techniques span from enabling interactivity in the traditional search-tree visualisation to designing whole new, unconventional ways of displaying the search (e.g., [Schulte, 1996], [Simonis et al., 2000], [Fages et al., 2004], [Bracchi et al., 2001], [Simonis et al., 2010]). While many of these techniques improve the user's experience of exploring the tree, dealing with large trees still remains a challenging task. This chapter first investigates how the traditional search tree visualisation can be further improved, making large search trees easier to comprehend. This is followed by a study of two unconventional ways of visualising search trees, which enhance the user's understanding of the search.

4.1 Traditional Search Tree Visualisation

4.1.1 The State of the Art in Search Tree Visualisation

One of the earliest works in drawing search trees in the context of Constraint Programming was available in Oz Explorer [Schulte, 1996]. To make the best use of the screen space, the visualisation utilised a space efficient layout algorithm that was first proposed in [Kennedy, 1996] and is suitable for rooted trees of equal size nodes.

Further, Oz Explorer reduced the visual complexity of a tree by allowing the user to collapse entire subtrees and focus on more interesting regions of search. Subtrees with no solutions (i.e., failed subtrees) could be collapsed automatically during search, resulting in a view similar to the one displayed in Figure 4.1, where the original, uncollapsed tree is displayed next to it, in Figure 4.2. As mentioned in Chapter 2, search tree visualisations often distinguish between three types of nodes: branch nodes, failure nodes, and solution nodes. In Figures 4.2 and 4.1 these nodes are depicted as blue circles, red squares, and green diamonds, respectively.

The space efficiency, clarity and collapsing capabilities of Oz Explorer have made it the most popular method to display search trees, with modern CP systems like CP-Viz profiler and the modern CP solver Gecode still using its approach. In fact, Gecode directly inherited its search tree visualisation (now called *Gist*) from Schulte's system, and it represents the state of the art in search tree visualisation.

Other visualisations have been proposed in the literature, both of the node-link kind ([Bracchi et al., 2001], [Fages et al., 2004], [Simonis et al., 2000]), as well as more unconventional visualisations ([Simonis et al., 2000]). Notably, both the APT Tool and OPL Studio made attempts at incorporating statistical information into the way node-link trees were drawn (see Section 2.6.2). It is not clear, however, whether the existing visualisations were superior to that in Oz Explorer, and, to the best of my knowledge, they have not received much traction.

For the reasons above, I have selected Gist as the baseline visualisation in my work. Indeed, the prototype of CP-Profiler reuses a substantial part of Gecode's source code related to tree visualisation by decoupling it from the solver proper.

4.1.2 Visually Determining the Size of Subtrees

Unfortunately, the way in which existing in CP techniques automatically collapse failed subtrees makes the collapsed subtrees indistinguishable from one another. This is the case, for example, in Gecode and CP-Viz, where collapsed subtrees are displayed as red triangles of equal size. Therefore, the collapsed nodes do not readily convey any information regarding their contents and, in particular, regarding the amount of search effort they represent. One subtree with, say, three nodes would look no different from another subtree with three thousand.

While detailed information about subtrees, such as their size, might be available in existing systems, one is required to manually query this information for every node individually. In Gecode, for example, the user can select a node of interest to show a dialog window with statistical information about the node and the underlying subtree.

A better approach is to provide the user with the ability to estimate the amount of nodes at a glance. In fact, the visualisation in the APT Tool (see Chapter 2) could draw collapsed subtrees as triangles using different shades of grey in order to convey their size. However, this way of visualising collapsed subtrees has not been adopted in modern systems, as colours are difficult for a human eye to interpret and compare. Further, the APT Tool did not provide a way to collapse subtrees with a given granularity, and thus required users to expand collapsed nodes manually, which is impractical for large trees.

In this section I propose an alternative way for collapsing subtrees that is aimed to attain the property of conveying the amount of search performed in each subtree in a more effective way.

Lantern Tree Visualisation

One challenge when collapsing subtrees is that the size of the resulting subtrees is difficult to control with high precision: very few subtrees will have exactly the desired number of nodes. It is, however, possible to guarantee that the number of nodes of collapsed subtrees does not exceed a certain amount. The new technique for collapsing subtrees is based on that idea: it only collapses subtrees that contain as many nodes as possible, but fewer than N, where N is a user-defined parameter. Note that this approach allows the user to control the amount of detail shown in the visualisation by means of adjusting the

parameter N. This is an improvement over the existing approaches for subtree collapsing used in Gecode and CP-Viz, as the amount of detail shown in them is determined by the presence and distribution of solutions nodes in the tree. As a consequence, they cannot effectively visualise a tree with either no solutions or a large number of solutions, as in those cases the user would be required to manually collapse/uncollapse a large number of nodes to achieve the desired level of detail.

The collapsed nodes can be visualised as a special "lantern" node, which can be seen as a triangle extended with a rectangular section at the bottom. The triangle part resembles failed triangle nodes commonly used for failed subtrees. The rectangular section, unlike the triangle, is dynamic in size: its height indicates the number of underlying nodes in the corresponding subtree.

If the user-defined N (the maximal number of nodes in a subtree) is small, it can directly determine the subtree size, for example, as the rectangle's height in pixels. Larger subtrees, containing thousands or millions of nodes require a different approach to be practical. A solution is to rescale the height, such that the largest collapsed subtrees – subtrees of size N – do not exceed some practical maximum M for such nodes.

Formally, given M as the maximum node height, the height of a node representing subtrees with n nodes can be easily calculated as follows:

$$height(n) = \begin{cases} \mathrm{n}, & N \leq M \\ \mathrm{n} \cdot \frac{M}{N}, & N > M. \end{cases}$$

Note that there is a reason not to scale the height for smaller values of N: doing so will result in a misleading visualisation, where small subtrees in a small tree appear large.

In the implementation within CP-Profiler the maximal height for a lantern node M is set to 127, which I found to be a reasonable compromise between the high "resolution" of a lantern node (that is, the number of different lantern nodes one could distinguish between in principle) and the compactness of the resulting visualisation. Additionally, subtrees that would be collapsed into the smallest lantern node (with $height = 1$), are chosen to be presented in their original uncollapsed form. This is important, for example, for single nodes whose siblings represent large subtrees, such as those created as a result of a "deep dive" in the search – a behaviour that can be useful to identify visually.

Example 4.1.1. Consider the search tree in Figure 4.3 visualised in two different ways. The visualisation on the left is the traditional node-link visualisation, where failed trees are collapsed entirely and displayed, in this case, as large red triangles labelled as "A", "B", "C", and "D". Note that it is unclear what portion of the search each of such triangles represents, that is, how many nodes they contain.

On the right, the same tree is visualised using the lantern tree visualisation, with the node limit N set to 1600. Labelled nodes correspond to those on the left view. The subtrees are highlighted using grey background to make it easier to identify which nodes belong to them. The lantern tree makes it easy to visually determine that subtree "A" is substantially larger than "B", while subtree "C" contains, perhaps, six times fewer nodes than "B". Further, subtree "D" is trivial, containing only two failure nodes. The visualisation on the left does not have this information readily available: subtrees "A", "B", "C", and "D" are visually indistinguishable from each other.

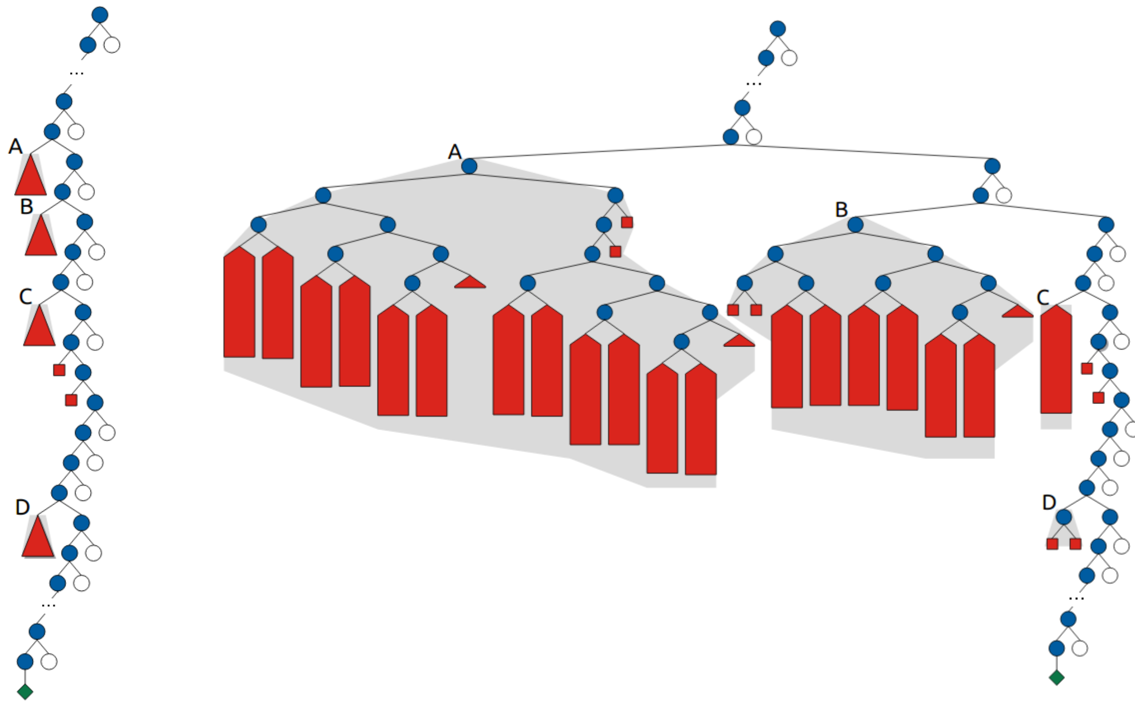

Figure 4.3: Collapsing of failed subtrees: traditional (left) and proposed (right).

Example 4.1.2. Consider a restart-based execution of the Golomb Ruler problem (introduced in Chapter 2) with 9 marks. Figure 4.4 shows its search tree with failed subtrees collapsed in the traditional way. Each restart branch, associated with one of the direct children of the root node, finds one solution. The exception is the last branch, where the proof of optimality is obtained.

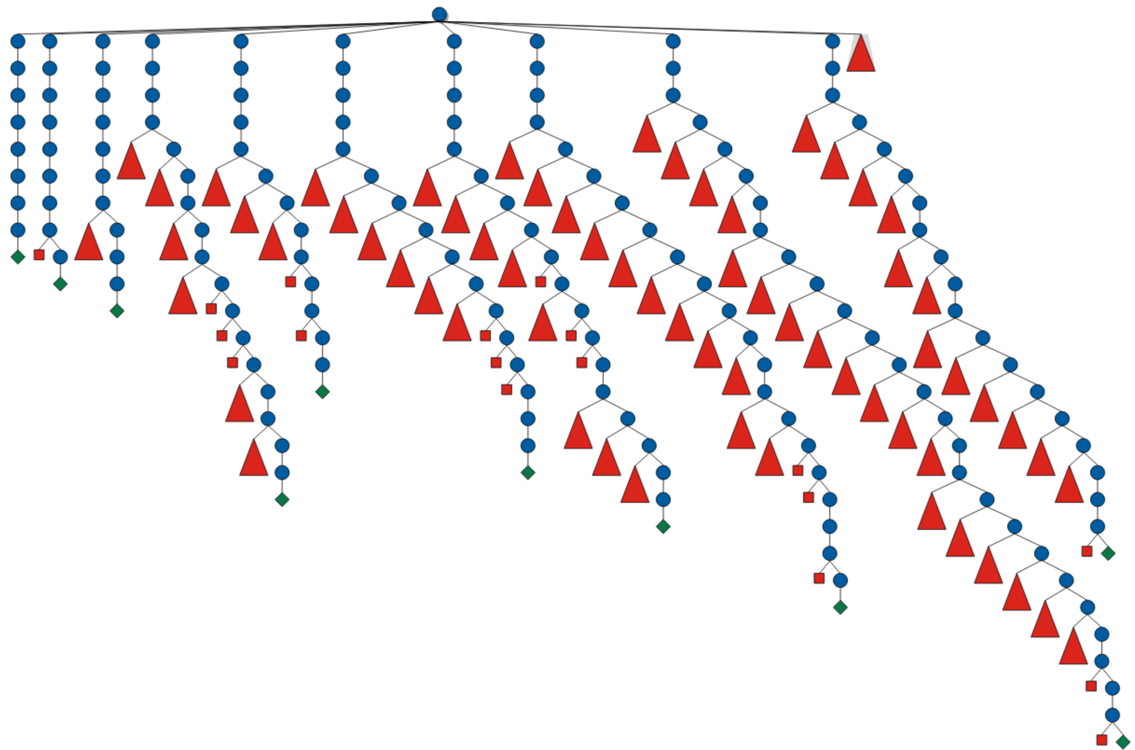

Figure 4.4: Traditional collapsing of failed subtrees (execution with restarts).

It is easy for a user to believe that most of the search is spent finding those solutions. The tree visualised in Figure 4.5 shows this is not the case. This figure shows the same tree visualised using the lantern tree technique, where the node limit is set to 5000 nodes. The visualisation clearly indicates that finding the first few solutions is trivial. Proving optimality required by far the most search effort: the last restart branch (highlighted) occupies the most space.

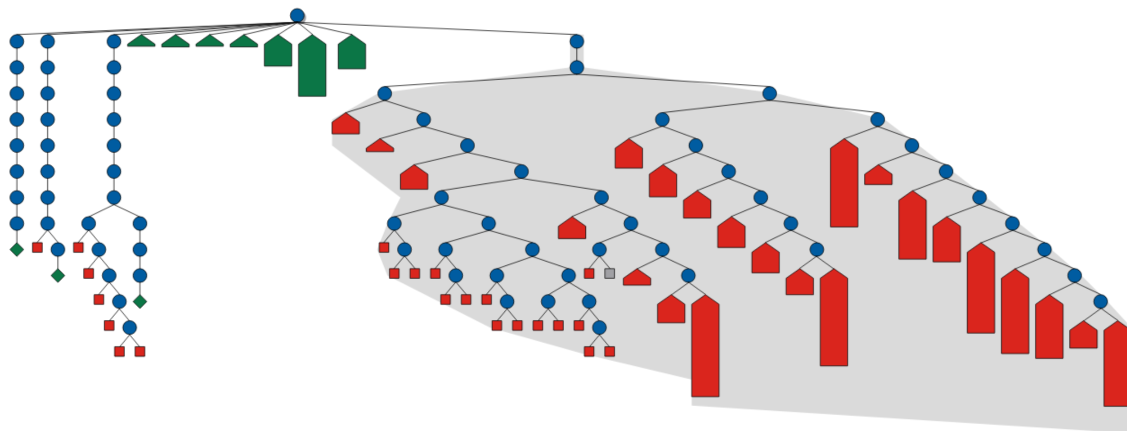

Figure 4.5: Proposed collapsing of failed subtrees (execution with restarts).

4.2 Overview Visualisations

As argued in [Schulte, 1996], two of the important insights that can be gained from a search tree visualisation are the amount of work performed by a solver before the first solution is found, and how the solutions are distributed in the tree. A potential way to facilitate these insights for large search trees is to provide users with an alternative search-tree visualisation with a good overview property, which would not necessarily replace the traditional node-link visualisation, but rather complement its functionality.

While there have been very few visualisations of this kind, a notable one was the phase-line display (seen in Chapter 2.6) implemented in CHIP, wherein the meaning of edges, normally representing node-parent relationships, was altered: instead, they connected nodes assigning the same variables. Such a view is not suitable for inspecting, for example, sequences of search decisions, since the order in which nodes are explored is not as clear as in the traditional visualisation. It could, however, give users a general idea about how dynamic the search of a given execution is in terms of the variable assignment order.

This section explores other ways in which alternative, or unconventional, visualisations can enhance the user's experience in viewing search trees. In particular, it investigates whether two tree visualisation techniques – *Indented Pixel Tree Plots* [Burch et al., 2010] and *Icicle Plots* [Kruskal et al., 1983] – can help provide a good overview of large search trees (and thus help users gain insights regarding the distribution of solutions) and allow users to navigate trees more effectively.

4.2.1 Indented Pixel Tree Plots

An Indented Pixel Tree Plot [Burch et al., 2010], or a *Pixel Tree*, is a visualisation technique for hierarchical structures, designed to clearly represent their major components, sizes and depths, while scaling up to millions of elements. As demonstrated in [Burch et al.,

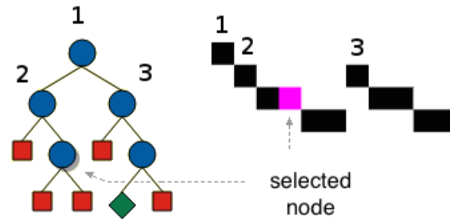

Figure 4.6: Basic Search Tree (left) and Corresponding Pixel Tree (right).

2010], the resulting plots can be easily scaled vertically and horizontally, can incorporate additional information using different colours, and they are easy to interact with (expand, collapse, filter etc). Importantly, even when trees are large, they allow users to perform tasks such as visually detecting similar subtrees and estimating which of two subtrees is larger.

Pixel trees represent nodes as single square objects, or pixels, while edges between nodes are represented only implicitly through the indentation between the nodes: parent nodes are placed directly above and immediately to the left of their left-most children nodes, and sibling nodes can be identified as nodes on the same depth level sharing the same parent. Figure 4.6 provides an example of a simple search tree visualised both in the traditional way (left) and as a pixel tree (right). For each node from the tree on the left, there is a corresponding pixel, or a square, in the pixel-tree on the right. For illustration, I have scaled the pixels up, and nodes and pixel with the same label represent the same entity.

Note that pixel trees preserve the vertical position, or depth, of a node. For example, node "2" lies on the second level in the search tree, and so does the corresponding node on the pixel tree. However, the horizontal positions of nodes on the pixel-tree represent depth-first-search exploration order: node "2" is positioned to the right of its parent "1". The same is not the case in the traditional tree visualisation on the left: in the tree on the left, for example, node "1" is explored before node "2", but it is closer to the right. Nevertheless, the hierarchical structure is still preserved in the pixel tree. For example, the parent of node "3" can always be found as the closest node to the left of node "3" that lies one level above (node "1" in this case).

While pixel trees are not as familiar and easy to interpret as traditional trees, they have two particularly important properties:

(a) they show very clearly the relationship between node exploration and time (i.e. a node is guaranteed to have been explored prior to all the nodes to its right);

(b) they scale well horizontally without losing much information regarding time progression, which is crucial for showing large trees.

The scalability of a pixel tree is achieved by compressing horizontally adjacent nodes to form vertical lines of pixels. Note that this might lead to some nodes being collapsed into a single pixel if they also incidentally belong to the same depth level. As a result of this node overlapping, compressed pixel trees can convey less information than the original tree. The necessary information loss due to collapsed pixels, however, can be partly compensated by colouring pixels, so that the intensity of the colour represents the node "density", or how many nodes it contains.

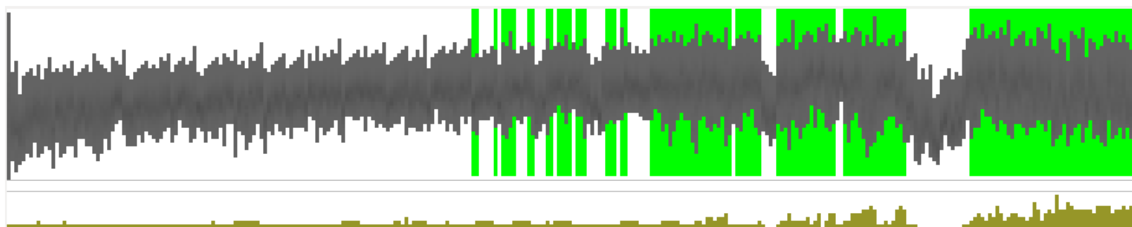

Figure 4.7: An example of a pixel tree view applied for search trees.

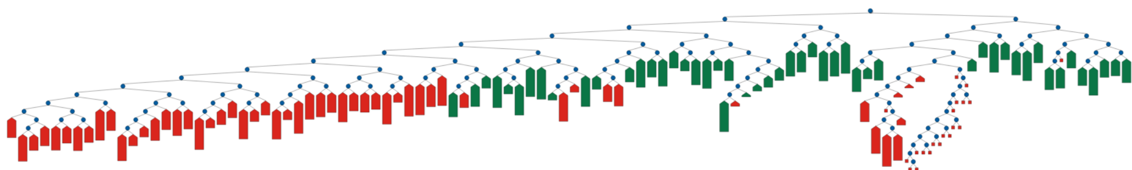

Figure 4.8: Lantern tree for a problem with many solutions.

Pixel Tree Implementation in CP-Profiler

The implementation of the pixel tree within CP-Profiler indeed takes advantage of the above properties: scalability and connection to time progression of the pixel tree. The benefit of the better scalability is clear, as it can help address one of the main shortcomings of the traditional tree visualisation: node-link diagrams take up a lot of space and commonly used techniques for collapsing traditional search trees lose all the structure behind collapsed nodes, leading to visualisations that do not provide a good overview of the whole tree, and can be misleading. In contrast, when the pixel tree visualisation is applied to a large search tree produced by a CP solver, it can convey some information about the structure of the search, even when it is compressed to fit in a single computer screen. I have additionally extended this visualisation to highlight the location of each solution with a vertical green line. This allows for a clear representation of the amount of effort between solutions, and can be used to immediately identify different patterns of behaviour.

Further, the clear correspondence between node and time allows pixel trees to display various statistical measures, and their change over time, in a way that their association with the part of the tree explored is clear. In CP-Profiler, for example, this is done through histogram-like views, where the height of a histogram bar associated with some point in time, represents the value of some measure at that point in time in the execution. Note that traditional tree visualisations are not suited for this task, as they sacrifice the node-time relationship in favour of the node's ancestry relationships. The examples below illustrate the benefits of using a pixel tree for visualising search.

Example 4.2.1. Consider the pixel tree visualisation in Figure 4.7, which represents the search tree, obtained as the result of running a model to find all solutions in a satisfaction problem. (This model is available as `wolf_goat_cabbage.mzn` from the official MiniZinc distribution[1].) To get an overview of the execution, the presented tree is compressed, such that the whole execution fits in one screen (the compression factor is 1,000 in this case). Note that due to the solution highlighting it is easy to see the contrast between regions of search rich in solutions and regions with no solutions at all. It also appears that regions with no solutions tend to require more search: the pixel tree is deeper in those regions, indicating a large amount of search decisions before backtracking. This observation is most prominent on the right-most region with no solutions, which is also the largest, indicating

[1] https://github.com/MiniZinc/libminizinc/blob/master/tests/examples/wolf_goat_cabbage.mzn

it took the search a very long time to find new solutions – something the modeller might want to take into account to modify the model.

Note that traditional search tree visualisations do not consider collapsing solution nodes, which makes them impractical for problems with a large numbers of solutions, like the one discussed here. On the other hand, the lantern tree visualisation introduced earlier in this chapter can easily handle such trees. The result of applying it to the same search is presented in Figure 4.8. Note its resemblance to the pixel tree: large areas with no solutions closely match large failed subtrees of the lantern tree. The perception of time, however, is not as clear in the lantern tree as it is in the pixel tree, where the number of nodes per vertical slice is user-defined and equals to the compression factor. Figure 4.7 illustrates how that property can be exploited: in addition to the pixel tree itself, the figure depicts a histogram underneath, indicating the average solver time per node. Looking at the histogram, it can be noted, for example, that propagation in the nodes appearing in the last region with no solutions is relatively inexpensive, compared to the rest of the tree.

Example 4.2.2. Consider the three visualisations displayed in Figures 4.9, 4.10, and 4.11, which correspond to three different views of the search tree obtained by solving the Golomb Ruler (introduced in Chapter 2) problem with 11 marks using the Gecode solver: the pixel tree, the traditional visualisation, and the lantern tree visualisation, respectively. Clearly the pixel tree view in Figure 4.9 depicts a very different search behaviour to that, for example, in Figure 4.7: the pixel tree of the Golomb Ruler resembles a fractal-like pattern – a behaviour that I believe can be potentially attributed to a generate-and-test-style search due to lack of propagation.

The same behaviour is apparent in the lantern tree (Figure 4.11), but not in the traditional search tree visualisation (Figure 4.10) due to the way failure trees are collapsed without regard for their size. In fact, the traditional visualisation is misleading, as it seems to suggest that the solutions are spread out almost uniformly in time, which is, as the pixel tree shows, clearly not the case.

Example 4.2.3. Figure 4.12 depicts a pixel tree for the Radiation problem (described in detail in Section 6.4.2) solved with the CP solver Chuffed. This time, a different kind of information is displayed under the pixel tree: the variable profile. In this view, pixels indicate both through their color and through their vertical positions, which variables correspond to the decisions made in a given part of the tree.

The way the color and the position is assigned to a variable is as follows. First, a list of all decision variables is obtained by, for example, traversing the entire tree and extracting variables from corresponding search decisions. The list is then sorted alphabetically. This way of sorting often leads to positioning related variables (for example, variables from the same array) with similar vertical coordinates and similar colour, thus forming groups (however, the boundaries between groups are not always clear). Note that other assignments could be implemented: for example, it could be left up to the user to select which variables to highlight in which colour.

The pixel tree in Figure 4.12 clearly displays different regions of search. This is especially prominent with the second half of the search, where the tree is substantially deeper. The variable profile reinforces this observation by indicating that the different regions also differ in the range of variables that get assigned. At the beginning of the search (on the left in the pixel tree), a less prominent "stairway" pattern on the variable profile also suggests different regions of search associated with different groups of variables.

My implementation of the pixel tree within CP-Profiler is interactive to allow the user to investigate patterns identified visually in more detail. For example, one can hover the mouse over the pixel elements in order to see the actual variable names. In this particular

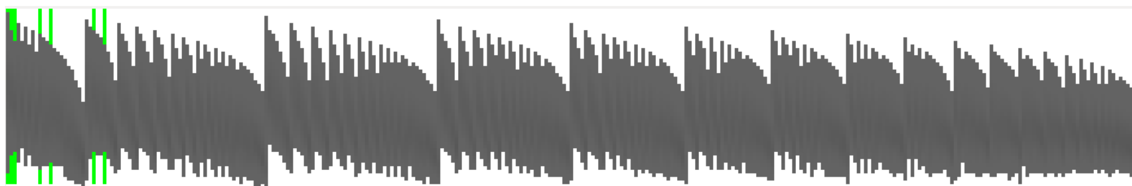

Figure 4.9: Golomb Ruler with 11 marks, Gecode.

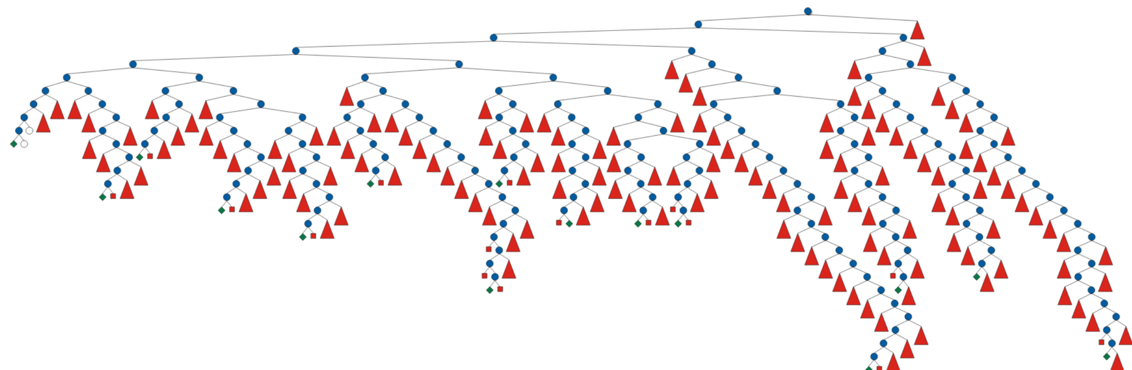

Figure 4.10: Golomb Ruler with 11 marks, Gecode.

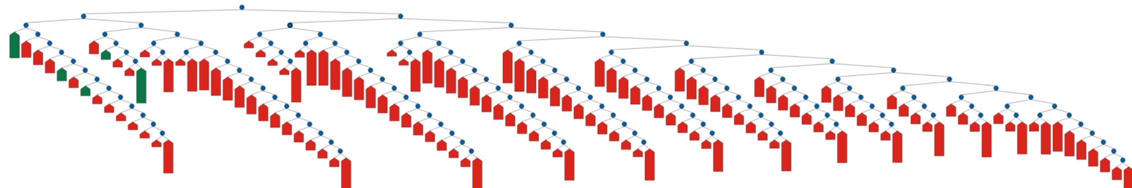

Figure 4.11: Golomb Ruler with 11 marks, Gecode.

case, it could be confirmed that the different regions can be associated with the search progressing by making decisions for different arrays (or rows in a matrix, to be precise) of decision variables, which were not involved in the search until that point in time. The interactivity of the pixel tree additionally allows users to navigate from the pixel tree to the traditional search tree visualisation, where the search can be studied in more detail. This property is explored in detail in Section 4.3.

4.2.2 Icicle Tree

Similarly to pixel trees, icicle trees [Kruskal et al., 1983] do not draw edges. However, they differ from pixel trees in the way child-parent relations are realised: in icicle trees parent nodes are positioned strictly above their children, that is, they span horizontally over the same space as their children nodes. Intermediate nodes, therefore, take a rectangular shape where the width is equal to the sum of all of its underlying leaves. This often results in very wide intermediate nodes, particularly for those nodes that are closer to the root. Despite this, icicle trees are twice as compact as pixel trees when uncompressed, since leaf nodes have the same vertical coordinates as their ancestors. More importantly, this property allows users to quickly find a node's ancestry and determine the size of any subtree at a glance.

A reasonable approach for compressing an icicle tree is to cut off its leaves, so that the corresponding parent nodes become the new leaf nodes. Doing so in a binary tree, for example, reduces the display area required for the visualisation by a factor of two. Note

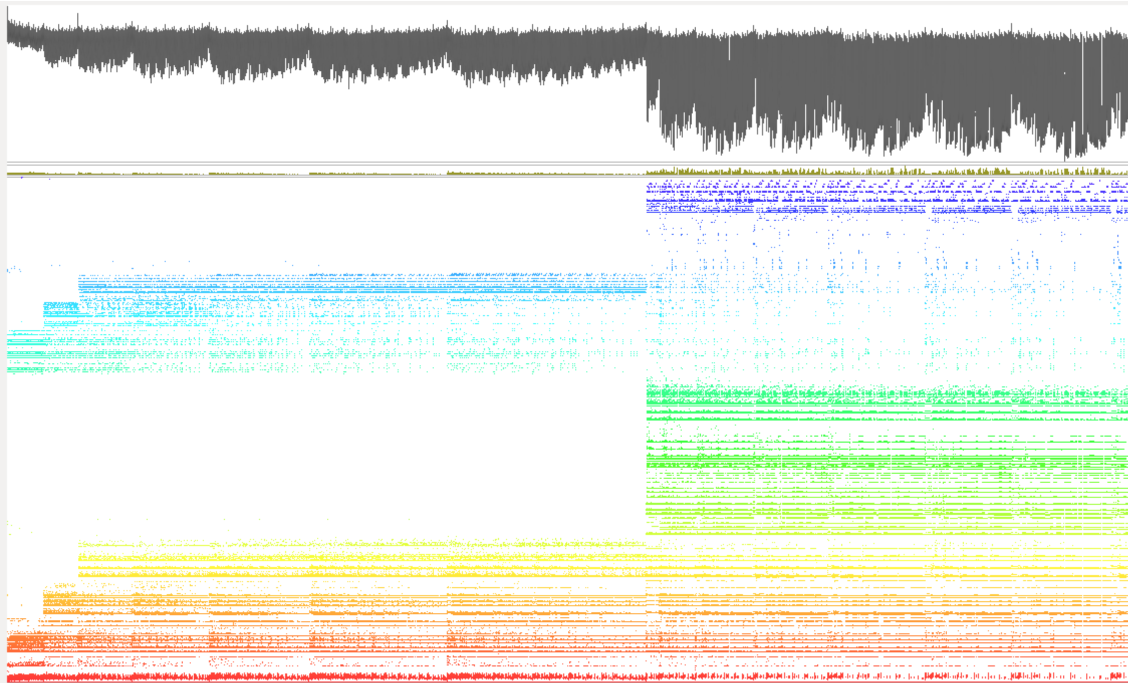

Figure 4.12: The Pixel Tree with a variable profile (Radiation problem instance solved with Chuffed).

that no matter how many of such compression steps are performed, one invariant is always maintained: for a given node its underlying subtree is displayed directly below and takes up exactly the width of that node. This fact makes intermediate nodes good candidates for representing subtree structure even in a compressed form.

This is in contrast to a pixel tree, where the focus is mostly on the temporal information, and the depiction of the search tree structure is not as clear. This is especially the case when a pixel tree is compressed and its visual elements, or pixels, cease to convey most of the structural meaning. For example, a pixel (square) in a compressed pixel tree can represent multiple nodes that happen to be located at the same level in the search and explored roughly in the same time window, but otherwise unrelated. Likewise, while a vertical slice of a compressed pixel tree corresponds to nodes explored in the same time window, these nodes can rarely form a single subtree.

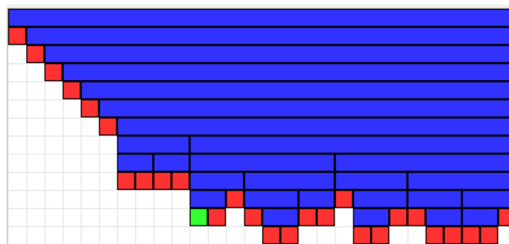

Figure 4.13: Icicle Tree Visualisation.

For consistency, we can colour the elements of the icicle tree visualisation – rectangles and squares – in a similar way to that of the traditional node-link visualisation. For example, in the Icicle Tree view shown in Figure 4.13, the elements colored in blue, red and green represent the internal nodes, failure nodes and solution nodes, respectively, of the traditional visualisation in Figure 4.2, which depicts the same search tree. However, as Figure 4.13 clearly shows, internal nodes, while comprising only half of the tree in terms

of number of nodes (in a binary search), dominate the amount of space they require in the Icicle Tree visualisation. As a result, most of the space would be coloured in blue, and thus can be used to represent other kinds of interesting information without a significant loss of information. This, combined with the fact that the icicle tree is easy to compress, makes the icicle tree a very useful tool for conveying statistical information about the search. For example, internal nodes can be colour-coded based on the time it took the solver to explore each subtree, and that information will be valid even for a compressed tree.

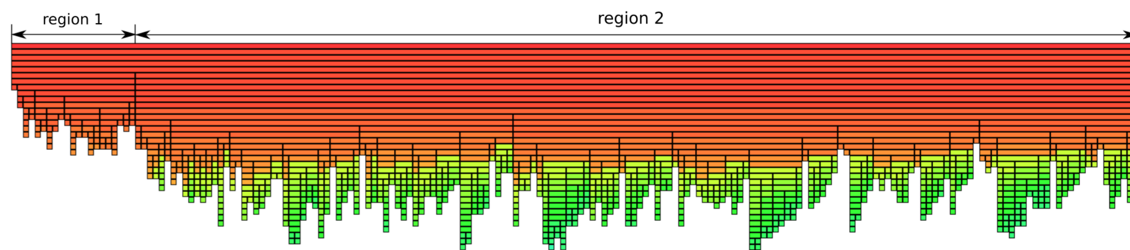

Figure 4.14: Icicle tree color mapped using decision variables.

Figure 4.14 shows an example of an icicle tree of the Radiation problem, colour-mapped based on decision variables in a way similar to that used in the pixel tree of Figure 4.12. As in the pixel tree, one can recognise distinct regions in search in this view: one on the left, at the beginning of the search – and one comprising the rest of the tree, with deep branches involving new decision variables (colored in a spectrum of green). Note, however, that the region on the left in this view appears small, which is misleading. This is due to the way the icicle tree is compressed: it cuts off fewer nodes from deep branches, making them appear larger in the resulting compressed view.

Another example of a potential use of iclcle-trees with colour-mapping is to display the node exploration order whenever it differs from depth-first search order: in this case the nodes could be coloured, for instance, based on the time at which they have been received from the solver.

4.3 Navigation

Importantly, the alternative visualisations discussed above are not intended to replace the traditional visualisation, but rather to complement it by providing an alternative view that might provide better insight on issues related to the structure of large trees. In addition, my study of pixel trees and icicle trees in the context of Constraint Programming led to a realisation that these overview visualisations can be very useful in driving the navigation of the corresponding traditional tree view. In CP-Profiler, this is achieved in pixel trees by selecting a time windows on a pixel tree, which will then highlight the corresponding part of the traditional tree and place it in the center.

Figure 4.15: Pixel tree with a selected slice.

For example, consider the pixel tree depicted in Figure 4.15, which shows a certain area of the tree highlighted in pink. The figure was obtained as the result of selecting a region of the tree of interest by performing the common task of moving the mouse cursor while holding the left mouse button. This action caused the traditional visualisation to navigate to the corresponding region in the search and highlight the nodes from that region, which is demonstrated in Figure 4.16. Highlighting in the node-link diagram is performed by both drawing the corresponding nodes with a thicker outline and changing the background of the two subtrees they belong to. Note that some of the nodes in the highlighted subtree on the right do not have the thick outline. This is due to the fact that a given slice in a compressed pixel tree rarely correspond to nodes that can form a subtree.

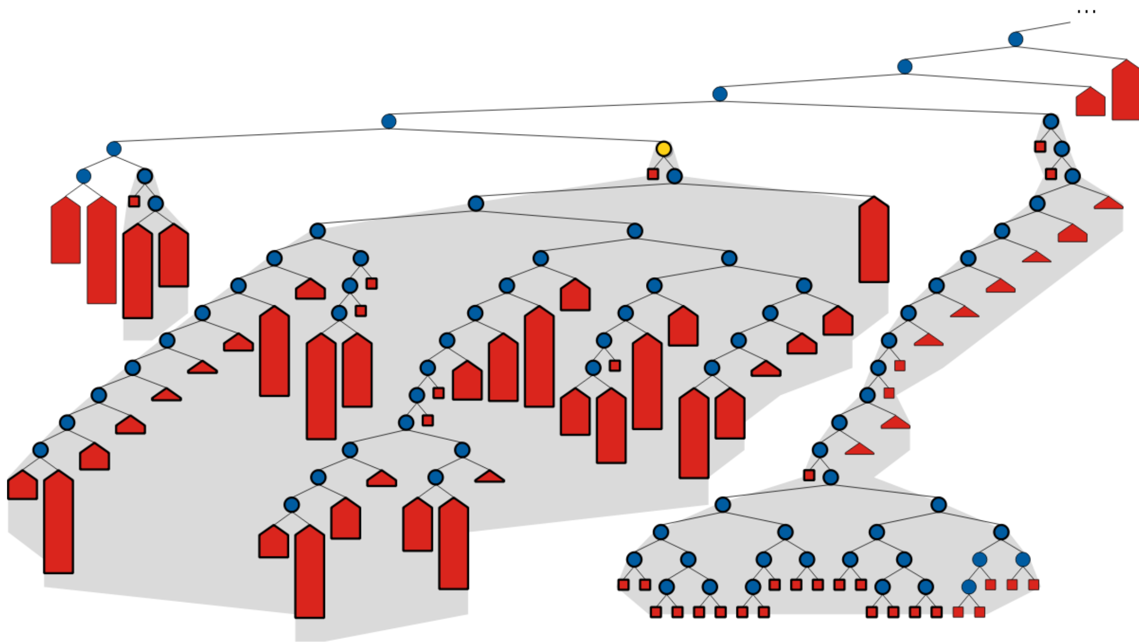

Figure 4.16: Distinct parts of the tree highlighted using the pixel tree.

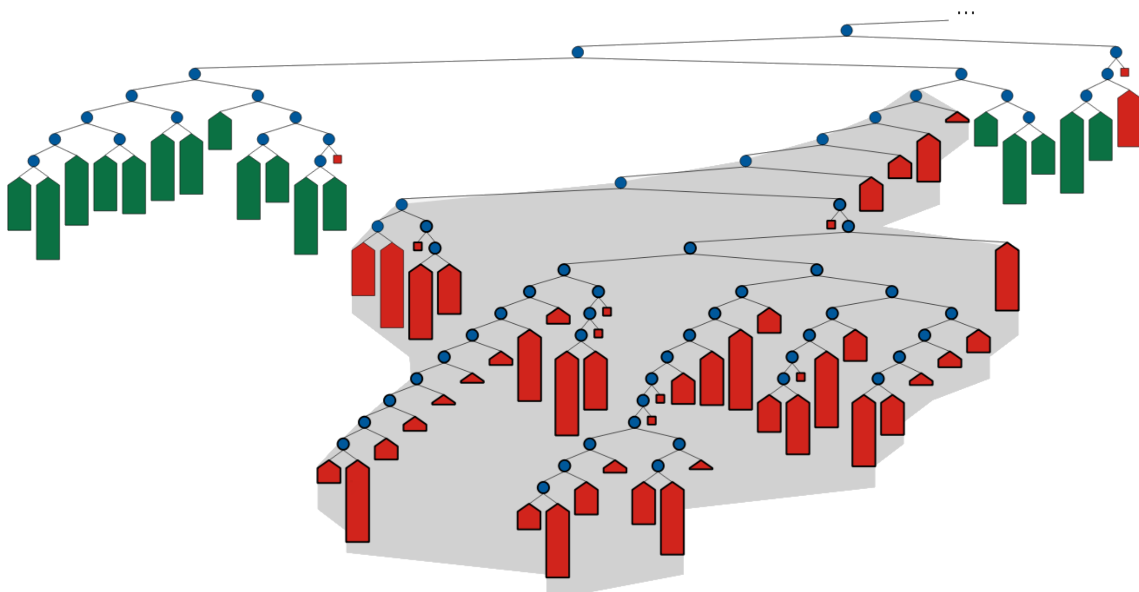

Figure 4.17: A single subtree highlighted using the icicle tree.

In contrast, the visual elements of the icicle tree always correspond to subtrees, and thus can be more meaningful than the slices in the pixel tree. This enables a more intuitive two-way mapping between both views: the position of the currently selected node on the traditional node-link visualisation can be highlighted on the icicle tree and vice versa. For example, one could select the root of the largest highlighted subtree depicted in the middle in Figure 4.16 (the selected node is colored in yellow) and see the corresponding element on the icicle tree highlighted (shown in Figure 4.18). Using the icicle tree, it becomes apparent that the selected node is only part of a larger entirely failed subtree: there is a node five levels above in the selected node's ancestry representing a subtree with no solutions. One could select the corresponding visual element on the icicle tree as in Figure 4.19 to highlight the corresponding subtree (navigate to it, if necessary) in the traditional visualisation.

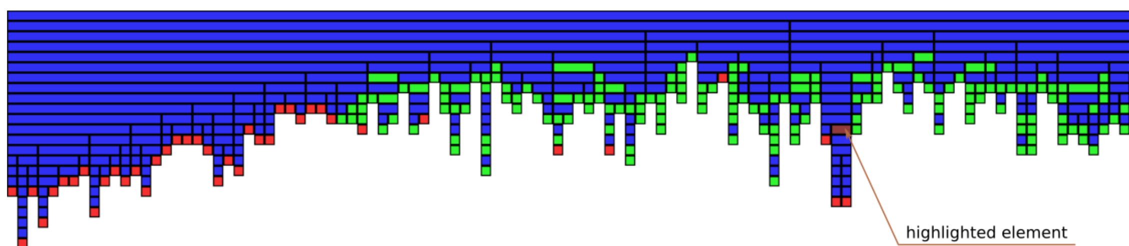

Figure 4.18: Currently selected node is indicated on the icicle tree.

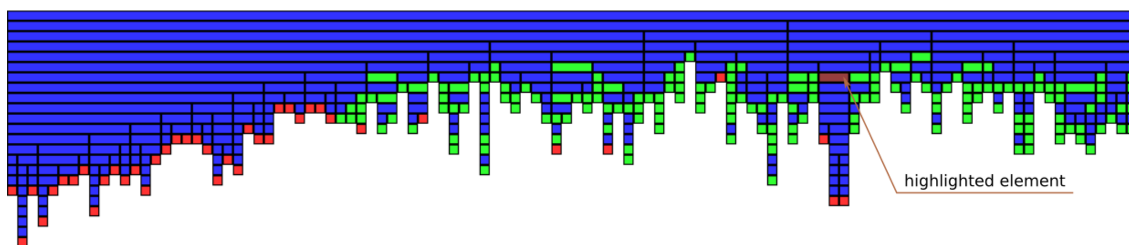

Figure 4.19: Using the icicle tree to navigate to a particular node in the tree.

Note that the above examples were intentionally made small to fit in one page. This was partly done by selecting a relatively small area of the search, and finding the right granularity for the lantern tree visualisation. Unfortunately, in the large trees that are often obtained in practice, the areas of search that interest the user might not be as easy to observe at a glance: for example, the layout algorithm in the traditional node-link visualisation might position two sibling nodes too far away from each other to fit in one screen if the underlying subtrees are large. While collapsing the failed subtrees addresses this problem, the resulting view might not show the desired level of detail. Alternative overview visualisations, such as the pixel tree and the icicle tree, offer a more practical way for navigating the tree, as they allow the user to quickly identify particular regions of the tree by using the two alternative tree visualisations as a form of "map" for the search.

The work in [Goodwin et al., 2017] presented further studies on how icicle tree visualisations can give insights into constraint solver executions.

4.4 Summary

This chapter first brought attention to one of the drawbacks in traditional node-link visualisations which represent the state of the art in search tree visualisation: their cognitive scalability. In particular, to the way collapsing of subtrees is currently performed: once collapsed, the subtrees lose all information regarding their underlying subtrees, and thus the entire search tree becomes difficult to reason about.

The chapter then introduced an alternative technique for visualising large trees: collapsing subtrees into special "lantern" nodes that convey information about their underlying subtrees via their height. Additionally, the collapsing is performed selectively: the user can specify the maximum number of nodes that the largest lantern node can contain, thus controlling the degree of detail in the resulting visualisation. Side by side comparisons of the state of the art and the proposed techniques are provided, which clearly demonstrate the advantages of the new technique.

The chapter then demonstrated how two unconventional techniques for visualising generic trees – pixel tree visualisation and icicle tree visualisation – can be applied in the context of search trees to provide a complementary view of the search. In particular, it demonstrates that even in a compressed form, which is necessary for large trees, pixel trees allow users to distinguish between different regions in the search, and, unlike traditional visualisation, pixel trees clearly convey time progression. Further, displaying statistical information under a pixel tree allows users to simultaneously see how that information changes in time and its relation to the different regions in the search.

Regarding icicle trees, the chapter shows that, while they are not suited for depicting time progression, they are also easy to compress and, in contrast to pixel trees, the meaning of the visual elements in icicle tree is mostly clear even in a compressed form: they represent individual nodes, or underlying subtrees. This fact allows users to colour-code individual nodes or subtrees based on different statistical information. The notion of time, however, is not as clear as in the pixel tree.

Finally, the chapter demonstrated how both alternative visualisations make it easier to navigate the traditional node-link visualisation: one can use them to obtain a good overview of the search and select regions of interest, and CP-Profiler will highlight the same regions in the traditional visualisation, where the search can be examined in more detail.

Chapter 5

Finding Recurring Patterns in Executions

Search trees for large real-world problems often contain many thousands or even millions of nodes. Partial understanding of such large executions can be obtained through good "overview" visualisations like the ones suggested in Chapter 4. However, as I show in this chapter, some helpful insights about an execution are often buried in detail, making it impractical to uncover them without the aid of automated analysis.

One of the ways to facilitate the examination of a program's execution is to provide users with guidance regarding the parts of the search tree on which to focus their attention. This can be achieved in a variety of ways, each useful for detecting different problems. For example, CP-Viz [Simonis et al., 2010] provides an invariant checker that can be used to detect nodes where the constraint propagation performed by the solver is not as strong as it could be, and might lead to a very costly search. In this way, the user can focus on the search nodes that show poor propagation. Another example is the "Christmas Tree" visualisation of search trees provided by the APT tool (see Chapter 2), which allowed users to draw nodes in different sizes/colours, depending on some statistical measure (e.g. propagation time).

In this chapter I propose a new approach for improving the user's focus – one that is based on automatically analysing a search tree in order to find repeated patterns in the form of similar subtrees. (Hereafter I refer to this kind of analysis as the *similar subtree analysis*.) The subtrees identified as similar are of interest to the modeller, as they often indicate repeated, and thus inefficient behaviour, such as that caused by the presence of symmetries or dominance in the model. Once the analysis is performed, the user can focus on examining the similar subtrees only (rather than the entire tree) to determine whether they indicate inefficient behaviour, and if so, investigate the reasons and potentially improve the model by, for example, adding symmetry/dominance breaking constraints.

The structure of this chapter is as follows. First, Section 5.1 discusses different similarity criteria that can be used for the analysis. Then, Section 5.2 presents algorithms that underlie the similar subtree analysis. This is followed by the discussion of the implementation of the analysis within CP-Profiler in Section 5.3. Section 5.4 presents a case study, which demonstrates that the analysis can be used for model improvement. Finally, Section 5.5 summarises the chapter.

5.1 Measuring the Similarity among Subtrees

To perform the similar subtree analysis, one first needs to determine when two subtrees should be considered similar. This section discusses different similarity criteria for subtrees appropriate for the analysis.

Two or more subtrees are said to form a *pattern* if they conform to one of the similarity criteria discussed here. Two subtrees are said to be *identical* if they have the exact same node structure and the exact same decisions on corresponding branches. This definition leads to the first way to define similar subtrees: two or more subtrees are similar if they are *identical*. Note that such identical subtrees can still contain nodes representing different states, e.g., the corresponding variable domains could differ.

Since the above definition of similar subtrees is quite restrictive, the user can easily reason about the similarities found by the analysis. The disadvantage, however, is that the analysis can be too conservative and disregard otherwise interesting patterns in a tree. Therefore, it might be interesting to relax this criteria by, for example, disregarding branching decisions, or by allowing a few internal nodes to be different. One such possibility that I explored is using a notion of similarity by *contour*.

The contour of a subtree describes how far the subtree extends horizontally at each decision level. That is, for each decision level of a subtree, the contour records horizontal position (for example, in pixels) of the leftmost and rightmost nodes at that level, relative to the root of the subtree. As an example, consider the subtree depicted in Figure 5.1. Its contour can be represented by the following sequence of pairs of numbers: $[(0, 0), (-15, 15), (0, 30)]$ (with 30 as the minimal distance between nodes in this case). The first pair corresponds to the level of the root node, the second corresponds to the level of its children etc. In total, this subtree contains three decision levels, and therefore, its *height* is 3.

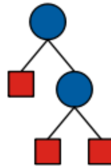

Figure 5.1: Visualisation of a basic search tree.

Note that the definition of a contour disregards the exact decisions taken by the solver in each branch, and only captures the outside contour of the tree, without distinguishing between the inner nodes. For example, note how the two trees depicted in Figure 5.2 have the same contour (outlined by the grey shapes around both subtrees), and thus would be identified as similar. However, the trees are clearly not identical, as the tree on the right lacks two nodes in the middle. A significant advantage of this pattern definition is that its computation is already required by the layout algorithm for drawing the tree and, therefore, it can be obtained without any overhead.

Algorithm 1 outlines a possible method for calculating contours (only binary trees are considered for simplicity, but the extension to n-ary trees is straightforward). It is based on the tree layout algorithm that was first described in [Kennedy, 1996] and then implemented in, among others, Oz Explorer, Gist and now CP-Profiler.

The algorithm takes the root of the tree of interest (*Root*) as a parameter, and populates the *extents* data structure for every node in the tree. Note that *Root* represents the

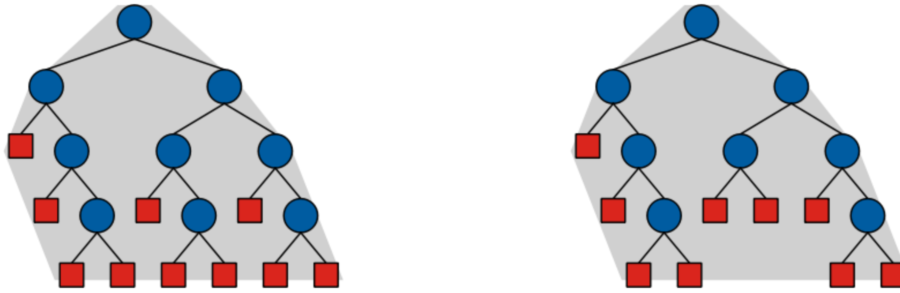

Figure 5.2: Different trees of the same contour.

entire tree in this case. Similarly, individual subtrees can be represented by their corresponding roots. The following algorithm as well as Algorithms 4 and 5 take advantage of this fact.

The *extents* for a node N is a vector of tuples (L_i, R_i), representing the contour of the subtree rooted at node N: every element of the vector indicates the relative horizontal positions of the left-most (L_i) and the right-most (R_i) nodes at height level i.

The contour for a subtree rooted at node N can be calculated as a result of conceptually merging the contours of the underlying subtrees, prepended with the contour of the root node N: $[(0,0)]$, which represents the extents of a single node (not accounting for its dimensions). Consequently, the contour of any leaf node is $[(0,0)]$ (assigned in line 2). If a node has children, the layout procedure is recursively applied to them first (line 5), and then its extents are computed based on the extents of the children. In the case of a node with a single child C, its extents are calculated as: $\mathit{contour}(N) = [(0,0)] +\!+ \mathit{contour}(C)$, since the child is positioned directly under N.

If two children, C_1 and C_2, are present, the algorithm first computes the minimal (horizontal) distance between the two root nodes required for their contours not to overlap. This is done by calculating the maximum of the "overlaps" on each level i $\in [0, MinHeight)$, where $MinHeight$ is the height of the shallower of the subtrees rooted at C_1 and C_2 (calculated in line 13). The overlap on level `i` is the difference between the right extent of the right-most node in C_1 and the left extent of the left-most node in C_2, both on level `i`. Additionally, the overlap is increased by a constant $MinDist$, representing the desired minimal (horizontal) distance between any two nodes (line 14).

For aesthetic reasons, the algorithm positions the subtrees in such a way that the root node N is the same distance apart from its children C_1 and C_2. To achieve this, the left and the right subtrees are conceptually moved by adding $Shift_L$ and $Shift_R$ (computed in line 15 as half of the maximum overlap in the absolute values, but opposite in signs) to their the horizontal coordinates, respectively. This is done by adding the corresponding values to the extents of the children subtrees after copying them over. For the range of levels where both subtrees have nodes (i.e., for i $\in 0..MinHeight$), the node's extents are appended by a tuple of two values for each level: the left extent of the left-most node of the left subtree and the right extent of the right-most node of the right subtree (lines 16 to 18). Further, if one of the subtrees is deeper than the other, its extents for the deeper levels (those farther from the root) are copied as the parent's extents for the corresponding levels (lines 19 through 27).

A similar algorithm can be used for n-ary trees as well. In that case, a contour for node N with m children $C_i, i \in 1..m$ can be calculated as above, with the exception that the "merging" procedure would have to be performed $m - 1$ times: first merging the subtrees rooted at C_1 and C_2, then merging the result of the previous merge with C_3 etc.

Algorithm 1: Algorithm for calculating contours of subtrees [Kennedy, 1996].

```
   Global Data: MinDist: minimal horizontal distance between nodes
 1 Procedure Layout(Root)
 2     Root.extents ← [(0, 0)]
 3     if |children(Root)| = 0 then return
 4
 5     for Child ∈ children(Root) do Layout(Child)
 6
 7     if |children(Root)| = 1 then
 8         Root.extents ← Root.extents ++ child(Root).extents
 9     end
10
11     if |children(Root)| = 2 then
12         (ext_L, ext_R) ← children(Root).extents
13         MinHeight ← min(height(ext_L), height(ext_R))
14         Overlap ← MinDist +
             max[ext_L[i].right − ext_R[i].left | ∀i ∈ [0..MinHeight)]
15         (Shift_L, Shift_R) ← (−Overlap/2, Overlap/2)
16         for i ∈ [0..MinHeight) do
17             Root.extents ← Root.extents ++ (ext_L[i].left+Shift_L,
                 ext_R[i].right+Shift_R)
18         end
19         if height(ext_L) > height(ext_R) then
20             for i ∈ [MinHeight..height(ext_L)) do
21                 Root.extents ← Root.extents ++ (ext_L[i].left+Shift_L,
                     ext_L[i].right+Shift_L)
22             end
23         else
24             for i ∈ [MinHeight..height(ext_R)) do
25                 Root.extents ← Root.extents ++ (ext_R[i].left+Shift_R,
                     ext_R[i].right+Shift_R)
26             end
27         end
28     end
```

5.2 Algorithms for Similar Subtree Analysis

The criteria for similarity between subtrees often affects the choice of the appropriate algorithm for performing similar subtree analysis. In the following two sections I describe the algorithms designed for the two similarity definitions introduced above (one for contours, and one for identical subtrees).

5.2.1 Similarity by Contour

The algorithm that finds similar subtrees based on their contour is outlined in Algorithm 2. At the core of the algorithm is the comparison function $Less(a, b)$ that provides a strict weak ordering for subtrees. In particular, its definition (see Algorithm 3 below) requires that for any pair of subtrees (S_1, S_2): S_1 and S_2 belong to the same group iff $Less(S_1, S_2) \Leftrightarrow Less(S_2, S_1)$.

First, an empty set of sets is initialised in line 2, where each internal set contains subtrees that together form a pattern. Then, for every *Subtree* from *Subtrees*, the algorithm searches for a pattern G in *Patterns*, such that *Subtree* belongs to G, that is, for every element $G_i \in G$: $\text{Less}(Subtree, G_i) \Leftrightarrow \text{Less}(G_i, Subtree)$. It is sufficient, however, to check this condition for any one element from G, as it is done in line 4, showing the element as G_0. If such a pattern already exists, *Subtree* is added to the corresponding set (line 5). Otherwise, a new pattern is created with *Subtree* as its first element and added to *Patterns* (line 7). Once all *Subtrees* have been processed this way, *Patterns* will contain the final result of the algorithm, which is returned in line 10.

Note that if *Patterns* is implemented as a sorted list (with sorting performed using the "Less" function), the search in line 4 could be performed in $O(logN)$ time using binary search, for example. Additionally, most of the algorithm could be abstracted away using a standard "multiset" data structure for *Patterns* (available as `std::multiset` in C++, for example), which will arrange elements in groups as they get inserted (based on the comparison function "Less").

Algorithm 2: Clustering subtrees using contours.

Input : *Subtrees*: a set of all subtrees in a tree, *Less*: comparison function
Output: *Patterns*: a set of groups of subtrees (each group representing a contour)

```
1  Function AnalyseContours(Subtrees, Less)
2      Patterns ← ∅
3      foreach Subtree ∈ Subtrees do
4          if ∃ G ∈ Patterns : G_0 ∈ G, Less(Subtree, G_0) ⇔ Less(G_0, Subtree) then
5              G ← G ∪ {Subtree}
6          else
7              Patterns ← Patterns ∪ {{Subtree}}
8          end
9      end
10     return Patterns
```

Algorithm 3: "Less" operator for contours.

Input : $Subtree_1$, $Subtree_2$
Output: bool

```
1  Function Less(Subtree_1, Subtree_2)
2      (Contour_1, Contour_2) ← (contour(Subtree_1), contour(Subtree_2))
3      (Height_1, Height_2) ← (height(Contour_1), height(Contour_2))
4      if Height_1 < Height_2 then return true
5      if Height_1 > Height_2 then return false
6      for i ← 0 to Height_1 do
7          (L_1, R_1) ← Contour_1[i]
8          (L_2, R_2) ← Contour_2[i]
9          if (L_1 > L_2 or R_1 < R_2) then return true
10         if (L_1 < L_2 or R_1 > R_2) then return false
11     end
12     return false
```

I propose the following function for comparing for two subtrees based on their contours (see Algorithm 3). The function takes $Subtree_1$ and $Subtree_2$ as arguments, and provides a means for strict weak ordering of subtrees: it returns `true` if $Subtree_1$ precedes $Subtree_2$

and `false` if $Subtree_2$ precedes $Subtree_1$. The algorithm first extracts contours of the subtrees (line 2) and queries their heights (line 3). The lines 4 and 5 state that a shallower subtree must precede a deeper one. For subtrees of equal height the ordering is determined based on the extents on each level: left and right extents of the two contours are compared, starting from the height of 0 to $Height_1$ until a discrepancy is found (lines 6 to 11). (The way extents determine the ordering is quite arbitrary, but the intuition is that the "narrower" subtree should appear first.) If extents of both subtrees are equal on each level, the algorithm will return `false` (line 12) for both $Less(Subtree_1, Subtree_2)$ and $Less(Subtree_2, Subtree_1)$, indicating that the corresponding subtrees are similar. Defined this way, the comparison function guarantees that in the resulting clustering, any two subtrees belong to the same cluster (or pattern) iff they have the same contours.

It is easy to see that the comparison function has time complexity proportional to the height of the tree, which in the worst case is O(N), where N represents the number of nodes in the tree. In turn, processing of a single subtree requires at most $logN$ comparisons, necessary to find its group (line 4 of Algorithm 2). The overall worst-case complexity of the similar subtree algorithm for contours is, therefore, $O(N^2 \times logN)$. The best-case scenario for the algorithm is when the tree is fully-balanced, and thus the height of its subtrees is bounded by $O(logN)$, turning the complexity of the algorithm into $\Omega(N \times logN \times logN)$.

5.2.2 Finding Identical Subtrees

When the definition of similar subtrees considers grouping only *identical* subtrees, an approach similar to the one above will not work as well, since the comparison function would be more expensive to compute: in the worst-case scenario, it would require accessing all nodes in both subtrees in order to perform pairwise comparison. The number of nodes in a subtree in the worst-case scenario is $O(N)$ for a tree with N nodes. Analogously to the above reasoning for contours, the worst-case complexity for identical subtrees would be $O(N^2 \times logN)$. Contrary to the contours, however, there is no apparent best-case scenario for the case of identical subtrees: the number of nodes in a subtree of a fully-balanced tree is still $O(N)$.

Instead of taking the same approach as for contours, a *partition refinement* algorithm can be more appropriate for finding identical subtrees. One such algorithm [Horbach et al., 2002] is outlined in Algorithm 4 and can be described as follows. Let G be the initial set of groups of subtrees that we want to partition. Let G_p and G_r be disjoint sets of groups, such that $G_p \cup G_r = G$, corresponding to, respectively, processed and remaining groups. Initially, G_r contains all subtrees rooted at an internal node (n) grouped by the number of direct children of n. For example, if a binary branching is used to build the tree, every node of that tree will have either 0 or 2 children, so G_r will contain a single element/group. In turn, G_p contains all subtrees rooted at a leaf node grouped by their status (which indicates whether a node represent a solution or a failure).

For some set S, let $S[i]$ denote the subset of S containing all (and only) subtrees of height i. The algorithm starts with $G_p[1]$ (i.e., the set of all groups of subtrees of height 1) processing one group g_p at a time. For every group g_p, all subtrees associated with a root n, where n is the first child ($j = 1$) of its parent node, are considered first. This condition is checked in line 7. For every such node n, its parent node (its associated subtree) is marked as a candidate for separation, adding it to the $Marked$ set (line 8). Groups containing at least one subtree in $Marked$ will be bisected based on whether a subtree is in $Marked$, with the two resulting groups replacing the original group in G_r (lines 11–12).

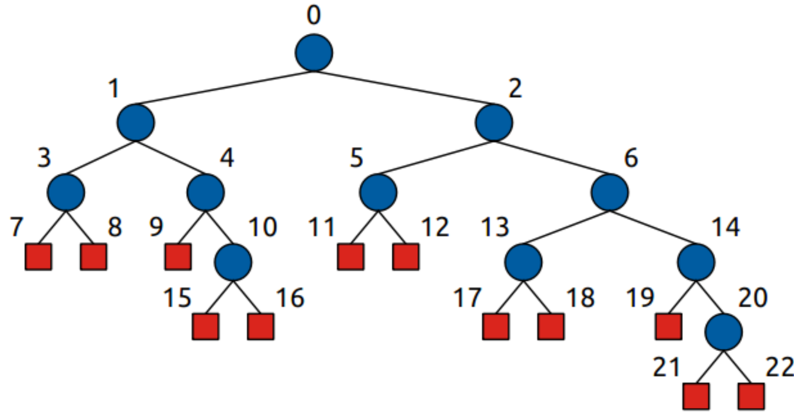

Figure 5.3: A simple tree used for demonstrating Algorithm 4.

The above repeats for all j, until $Children$ (the maximum number of direct children for an individual node within G_r, computed in line 4) is reached. By that point, all subtrees of height $i+1$ are fully partitioned, so $G_r[i+1]$ is moved into G_p as processed (lines 16–17). The algorithm then proceeds to the next height $i+1$, repeating the procedure performed for i. Once the maximum height of the tree ($MaxHeight$, available globally) is reached, the algorithm finishes with all subtrees fully partitioned in G_p.

Algorithm 4: Partition refinement algorithm for subtrees.

Global Data: $MaxHeight$: height of the tree
Input : G_p, G_r: initial partition of subtrees
Result : G_p: fully partitioned subtrees

1 **Procedure** Partition(G_p, G_r)
2   **foreach** $i \in [1..MaxHeight-1]$ **do**
3   **foreach** $g_p \in G_p[i]$ **do**
4       $Children \leftarrow$ maxChildren(G_r)
5   **for** $j \in [1..Children]$ **do**
6         $Marked \leftarrow \emptyset$
7   **foreach** $n \in g_p$, n is a jth child **do**
8           $Marked \leftarrow Marked \cup$ parent(n)
9   **end**
10   **foreach** $g_r \in G_r, g_r \cap Marked \neq \emptyset$ **do**
11           $G_r \leftarrow G_r - g_r$
12           $G_r \leftarrow G_r \cup \{g_r - Marked\} \cup \{g_r \cap Marked\}$
13   **end**
14   **end**
15   **end**
16     $G_p \leftarrow G_p \cup G_r[i+1]$
17     $G_r \leftarrow G_r - G_r[i+1]$
18   **end**

The key insight that the algorithm is built upon is the fact that for any two trees to be identical, their corresponding subtrees must be identical as well. The algorithm thus requires some initial partition (in this case, G_p) that it can use as the "ground truth". At the beginning, G_p only contains patterns of height 1 specified by the initial partition.

When all such patterns are processed, the subtrees of height 2 become fully partitioned and can no longer change, and thus groups of height 2 can also serve as the "ground truth" extending G_p. More precisely, when processing a single group from g_p for some j, the algorithm splits groups from G_r based on whether or not their subtrees have an element from g_p as their jth child. Since all child subtrees for patterns of height $i+1$ must be in $G_p[k], k \leq i$, processing them guarantees that subtrees in $G_r[i+1]$ have been fully partitioned. The set of processed groups G_p can thus be extended by $G_r[i+1]$.

Note that this algorithm is relatively easy to modify to incorporate other information on nodes. For example, the variables assigned in the search decision, or the full decisions (labels) can be used for this purpose. In that case, the initial partition has to be modified in such a way that only nodes with the same decisions can be within the same group. Additionally, the procedure in lines 6– 13 would need to be run separately for nodes in g_p that have different labels (requiring an additional loop over labels).

Example 5.2.1. Consider the tree depicted in Figure 5.3. For demonstration, all nodes are labelled with their unique identifiers: for example, the identifier of the root node is 0, and its children are labelled as 1 and 2. However the algorithm in this example will not take labels into account, that is, otherwise identical subtrees with different labels on nodes will be treated as identical.

The algorithm starts with an initial partition, consisting of two parts: G_p and G_r, representing processed and remaining groups, respectively. In this example, all failed nodes {7 8 9 11 12 15 16 17 18 19 21 22} comprise the only group within G_p. Any other types of leaf nodes that should be distinguished from failed nodes (such as solution nodes) would be placed in their respective groups within G_p alongside the group for failed nodes. All branch nodes {0 1 2 3 4 5 6 10 13 14 20} comprise the only group within G_r.

The algorithm first goes through subtrees of height 1, that is, leaf nodes. In the initial partition all such nodes are in the set of processed groups, that is, within G_p. In this example, there is only one group of nodes of height 1 – the group of failed nodes. The loop starting in line 5 initiates with $j = 0$, indicating that only jth children nodes (of their corresponding parents) will be considered first. For the group of failure nodes, only nodes {7 9 11 15 17 19 21} satisfy that condition. According to line 8, parents of the selected nodes are obtained as $Marked$: {3 4 5 10 13 14 20}. Then, according to lines 10–13, any nodes that are in $Marked$ are separated from their respective groups. In our case, the only group within G_r is separated into two: {3 4 5 10 13 14 20} and {0 1 2 6}.

The algorithm proceeds to second (right) children by incrementing j. The nodes in question are {8, 12, 16, 18, 22}. Similarly to the above, parents of the selected nodes {3 5 10 13 20} are obtained and used to separate unprocessed groups further, resulting in the following partition: {3 5 10 13 20} {4 14} {0 1 2 6}. Since the tree in this example is binary ($Children = 2$), the loop started in line 5 exits. The outer loop also exist since at this point there is only one group in G_p, which we have just examined. By this point, all groups of height 2 have been processed (since we have examined all their children), so all such groups are moved from G_r to G_p.

The algorithm then proceeds to subtrees of height 2 and repeats the above steps. All steps required to complete the algorithm are outlined Table 5.1, which shows how the partition changes over time (the changes shown in red with bold font). The algorithm finishes when G_r becomes empty, and the resulting partition can be found in G_p. Note that, for the purpose of similar subtree analysis, only groups with multiple subtrees are of interest. Excluding trivial patterns (comprised of a single element or single nodes), the resulting patterns are: {1 6}, {4 14}, and {3 5 10 13 20}.

Step	Remaining groups (G_r)	Processed groups (G_p)
Initial	{0 1 2 3 4 5 6 10 13 14 20}	{7 8 9 11 12 15 16 17 18 19 21 22}
i = 1; j = 0	**{0 1 2 6} {3 4 5 10 13 14 20}**	{7 8 9 11 12 15 16 17 18 19 21 22}
i = 1; j = 1	{0 1 2 6} **{4 14} {3 5 10 13 20}**	{7 8 9 11 12 15 16 17 18 19 21 22}
move h = 2	{0 1 2 6} {4 14}	**{3 5 10 13 20}** {7 8 9 11 12 15 16 17 18 19 21 22}
i = 2; j = 0	**{0} {1 2 6}** {4 14}	{3 5 10 13 20} {7 8 9 11 12 15 16 17 18 19 21 22}
i = 2; j = 1	(no change: {4 14} are marked for splitting, but they already comprise their own set)	
move h = 3	{0} {1 2 6}	**{4 14}** {3 5 10 13 20} {7 8 9 11 12 15 16 17 18 19 21 22}
i = 3; j = 0	(no change: no nodes in $G_p[3]$ for $j = 0$)	
i = 3; j = 1	{0} **{1 6} {2}**	{4 14} {3 5 10 13 20} {7 8 9 11 12 15 16 17 18 19 21 22}
move h = 4	{0} {2}	**{1 6}** {4 14} {3 5 10 13 20} {7 8 9 11 12 15 16 17 18 19 21 22}
i = 4; j = 0	(no change: {0} is marked for splitting, but it already comprises its own set)	
i = 4; j = 1	(no change: {2} is marked for splitting, but it already comprises its own set)	
move h = 5	{0}	**{2}** {1 6} {4 14} {3 5 10 13 20} {7 8 9 11 12 15 16 17 18 19 21 22}
i = 5; j = 0	(no change: no nodes in $G_p[5]$ for $j = 0$)	
i = 5; j = 1	(no change: {0} is marked for splitting, but it already comprises its own set)	
move h = 6		**{0}** {2} {1 6} {4 14} {3 5 10 13 20} {7 8 9 11 12 15 16 17 18 19 21 22}

Table 5.1: Steps performed for finding identical subtrees in Example 5.2.1.

5.2.3 Filtering the Results

When applied to large search trees, the similar subtree algorithm can find many patterns, potentially making the results overwhelming. It is thus essential to help the user find the patterns that are most likely to lead to a fruitful insight regarding the execution by automatically filtering out the rest. This can be done by only showing those patterns that exceed a certain height and/or occur at least as often as some user defined number. For example, if a subtree is unique (i.e., no similar subtree can be found in the search tree), the algorithms above will associate it with a pattern of cardinality 1. Such unique subtrees are of no interest, at least during the similar subtree analysis, and should be discarded. Likewise, patterns of shallow subtrees, such as leaf nodes or subtrees of two to four levels of height, convey little information and should be discarded: it is most likely that the similarities found there are purely due to chance.

In addition to such trivial patterns, there is another class of subtrees that one should avoid: tall subtrees with relatively few nodes corresponding to a solver “deep dive”. Often solvers make a deep dive, backtracking at most one level up, resulting in tall subtrees with relatively few nodes. Such subtrees rarely convey interesting information even when they

appear frequently (especially, if the similarity criteria is loose, e.g., no labels are taken into account) and should be discarded like other trivial patterns.

In addition to simply filtering patterns by their height and frequency, one can greatly simplify the results of the analysis, without loss of generality, by eliminating *subsumed* patterns.

5.2.4 Eliminating Subsumed Patterns

By definition, if two subtrees are identical, their corresponding underlying subtrees must be identical. Similarly and more generally, if two subtrees can be associated with some pattern (for example, they have the same contour), their corresponding underlying subtrees might form patterns of their own.

Consider again the two contours depicted in Figure 5.2. The subtrees have been identified as having the same contour – a fact that can also be observed by comparing the areas around the subtrees highlighted in grey. The contours of their underlying subtrees are depicted in Figure 5.4. As indicated by the highlighted areas, the two left subtrees have the same contour, while the right subtrees have different ones.

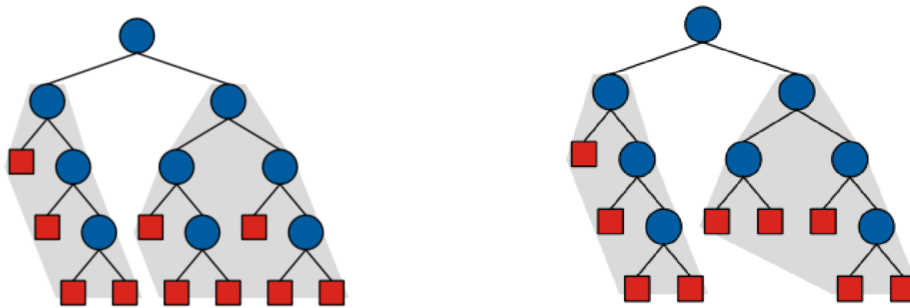

Figure 5.4: Contours of underlying subtrees can differ.

In practice, we may wish to focus on the larger, non-subsumed patterns only, since the smaller, *subsumed*, ones convey little additional information. Subsumed subtrees could be eliminated using the algorithm outlined in Algorithm 5. The key idea behind the algorithm is to use the input set of patterns to construct a set of subtrees that they would subsume had they been part of a pattern. Once this set is obtained, it can is used to determine if a pattern is subsumed, that is, all its subtrees are subsumed.

The detailed algorithm description is as follows. Let a *Pattern* represent a group of subtrees that satisfy a given similarity criteria. The algorithm above takes a set of patterns as input and returns all patterns that cannot be subsumed by others from the same set. For every subtree associated with a *Root* node found within a pattern from *Patterns*, the algorithm obtains a list of the *Root*'s children and adds them to $Marked$ – a set of nodes representing subtree candidates for subsumption, initialised in line 2 to an empty set. After that, for every pattern from *Patterns* the algorithm checks if all of its subtrees (their roots) are within $Marked$ (line 11). If that is the case, the pattern is known to be subsumed and thus omitted. Otherwise, the pattern is added to the $FilteredPatterns$ set, which is returned as the result of the algorithm (line 15).

Algorithm 5: Removing subsumed patterns

```
   Input  : Patterns – original patterns
   Output: FilteredPatterns – patterns less subsumed
1  Procedure RemoveSubsumed(Patterns)
2      Marked ← {}
3      foreach Pattern ∈ Patterns do
4          foreach Root ∈ Pattern do
5              Marked ← Marked ∪ children(Root)
6          end
7      end
8      FilteredPatterns ← {}
9      foreach Pattern ∈ Patterns do
10         Roots ← getRootsOf(Pattern)
11         if Roots ⊈ Marked then
12             FilteredPatterns ← FilteredPatterns ∪ {Pattern}
13         end
14     end
15     return FilteredPatterns
```

Example 5.2.2. Consider again the tree from Figure 5.3 used in Example 5.2.1. In that example, Algorithm 4 found the following patterns of subtrees: {1 6}, {4 14}, and {3 5 10 13 20}. Let us apply Algorithm 5 in order to eliminate subsumed patterns (if any) from this set.

The algorithm starts by walking through the original set of patterns. The first pattern is {1 6}, which represents subtrees rooted at nodes 1 and 6. The algorithm obtains children of nodes 1 and 6: {3 4} and {13 14}, respectively, and adds them to the $Marked$ set. Thus, $Marked$ currently contains: {3 4 13 14}. The algorithm repeats the above procedure for remaining patterns {4 14} and {3 5 10 13 20}, extending the $Marked$ set with {9 10 19 20} and {7 8 15 16 17 18 21 22}, respectively. The resulting $Marked$ set consists of the following nodes: {3 4 7 8 9 10 13 14 15 16 17 18 19 20}.

Note that any pattern whose roots are entirely in $Marked$ is considered subsumed by some other pattern (or multiple patterns) from the original set. The algorithm thus walks through the original set of patterns again in order to check for this condition. Pattern {1 6}, for example, is not subsumed since neither 1 nor 6 is in $Marked$. On the other hand, both 4 and 14 of pattern {4 14} can be found in $Marked$, and thus the pattern is subsumed. Finally, pattern {3 5 10 13 20} is not subsumed, even though four out of five of its nodes/subtrees ({3 10 13 20}) are in $Marked$. The result of the algorithm is thus comprised of the following patterns: {4 14} and {3 5 10 13 20}, which are now ready to be presented to the user.

5.3 Implementation within CP-Profiler

I have extended CP-Profiler to be able to perform the similar subtree analysis using both criteria proposed in this work: by contour and by identity. Figure 5.5 shows the result of applying the analysis to a search tree, with the similarity criteria by contour selected (the model used in the execution is discussed in Section 5.4). The window displaying the result is vertically divided into two parts. A histogram of the patterns found is shown on

the left, where each horizontal bar represents a single pattern found in the tree. For each pattern the user can also see:

1. *Height*: the number of decision levels in the subtrees represented by the pattern
2. *Size*: the number representing the pattern's *size*
3. *Count*: the number of times the pattern occurs in the tree

A pattern's *size* is designed to resemble the area that the pattern occupies with the motivation that it should work for both kinds of patterns: subtrees and contours. More precisely, the size of a pattern p of height H is defined as the sum of the widths of the pattern's extents on levels from 1 to H, where the width of extent (L_i, R_i) is equal to $R_i - L_i$. Note that other definitions for *size* can be viable as well. For example, if a pattern represents subtrees with identical structure, the number of nodes in the subtree could be more appropriate.

CP-Profiler supports sorting and filtering patterns by these three properties, and choosing which parameter will determine the lengths of the horizontal bars (shown in light red colour). This enables users to focus on two parameters at once, thus helping them find patterns of interest. For example, in Figure 5.5 the patterns are sorted by height, while the lengths of the bars indicate their sizes as defined above. The user could change the bars to indicate the patterns' *Count* instead, and thus clearly separate large patterns that are frequent from those that are less frequent.

Once the user selects a pattern (as is the case for the ninth pattern from the top in this example, indicated by a dashed outline), a subtree representing the pattern is shown in the preview part of the window on the right. All subtrees in the original tree that belong to the selected pattern are also highlighted on the traditional node-link tree visualisation window. This is achieved by changing the color of the subtrees' contours and collapsing everything else, as demonstrated in Figure 5.6.

From there, the user can study the similarities more closely by focusing on the highlighted areas only. For example, the user might be interested in the decisions made by the solver that led to the same subtrees or, more precisely, in the difference between the decisions made on the two search paths, that nonetheless led to similar searches.

Note how in Figure 5.6 the two paths that lead to the two highlighted subtrees share a large number of nodes (at the beginning of the tree). To aid the user in identifying the differences, the path differences can be automatically calculated and shown upon pattern selection. This is achieved in CP-Profiler by showing the different decisions textually below the histogram, as displayed in Figure 5.5. This feature will be demonstrated in more detail in the following section.

To give an idea about the performance of the algorithms used in the similar subtree analysis, Tables 5.2 and 5.3 show the time required to run their CP-Profiler implementation on a consumer-grade CPU (Intel Core i7-4790 used in this case). In particular, Table 5.2 shows from left to right: the tree's size, the time taken by the similar subtree analysis with the similarity by contour (Section 5.2.1), the number of patterns found initially, the time taken by the subsumed subtree elimination algorithm (Algorithm 5), and the number of patterns after the elimination. Table 5.3 shows the same information as in Table 5.2, but for the analysis with similarity by identity (Algorithm 4).

Note that the running time of the similar subtree analysis by contour has better scalability than that of the algorithm by identity. Despite that, both algorithms find patterns in a reasonable amount of time for large trees with several millions of nodes. Importantly,

Figure 5.5: Viewing similar subtrees.

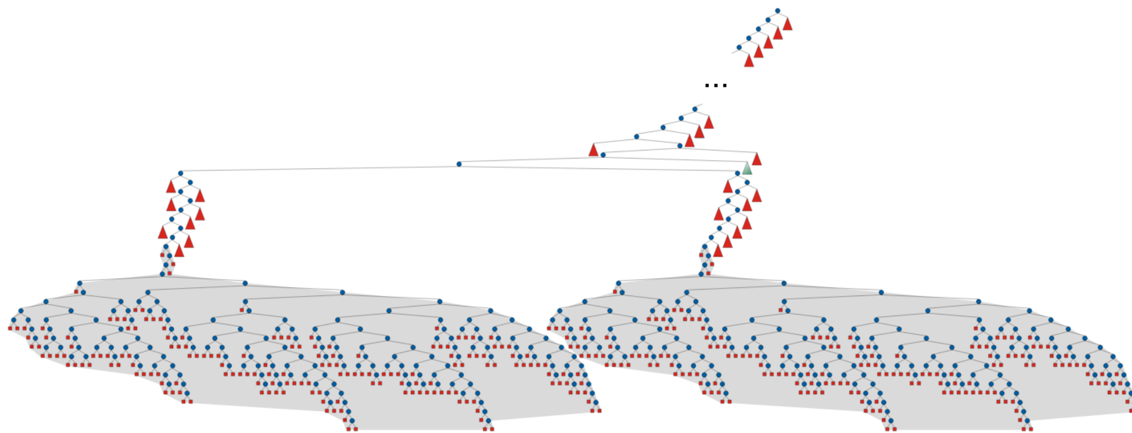

Figure 5.6: Subtrees of interest highlighted on the tree visualisation.

the similar subtree analysis does not require the execution to finish before it could be applied to it and still be meaningful: patterns in partial executions tend to resemble those in the corresponding complete ones.

Tree Size (nodes)	Analysis	Patterns (init)	Elimination	Patterns (final)
1,677.1K	2.39s	9,684	1.06s	9,197
5,895.0K	9.42s	27,449	4.41s	26,156
9,367.0K	16.64s	45,351	8.26s	39,644

Table 5.2: Similar subtree analysis performance (by contour).

Tree Size (nodes)	Analysis	Patterns (init)	Elimination	Patterns (final)
1,677.1K	6.22s	8,635	1.12s	6,841
5,895.0K	43.63s	21,202	4.97s	16,137
9,367.0K	61.87s	28,672	8.61s	21,823

Table 5.3: Similar subtree analysis performance (by identity).

5.4 Case Study

I present a case study for using the similar subtree analysis on an execution of the Social Golfers problem. In this problem, a cohort of N golfers play R rounds of golf, wherein each round, the golfers are divided into G groups containing S players each. The goal of the problem is to find R partitions of golfers into G groups, such that no two golfers play together in the same group more than once.

The following model (Listing 5.1), which can be found as `golfers1` in the MiniZinc benchmarks library[1], was used in the study:

```
1  include "globals.mzn";
2
3  int: G;    % The number of groups.
4  int: S;    % The size of each group.
5  int: R;    % The number of rounds.
6
7  int: N = G * S;
8
9  set of int: ROUNDS = 1..R;
10 set of int: GOLFERS = 1..N;
11 set of int: PLACES = 1..N;
12
13 array [ROUNDS, PLACES] of var GOLFERS: rpg;
14 array [GOLFERS, GOLFERS] of var 0..R: ggr;
15
16     % Each member of each group must be distinct.
17 constraint
18     forall (r in ROUNDS) (
19         alldifferent (p in PLACES) (rpg[r, p])
20     );
21
22     % Each pair can play together at most once.
23 constraint
24     forall (r in ROUNDS, g in 0..(G - 1), i, j in 1..S where i < j) (
25         ggr[rpg[r, S * g + i], rpg[r, S * g + j]] = r
26     );
27
28     % Break some symmetry by strictly ordering players
29     % within each group in each round.
30 constraint
31     forall (r in ROUNDS, p in PLACES) (
32         if p mod S != 0
33         then rpg[r, p] < rpg[r, p + 1]
34         else true
35         endif
36     );
37
38 solve
39     :: int_search([rpg[r, p] | r in ROUNDS, p in PLACES],
40         first_fail, indomain_min, complete)
41 satisfy;
```

Listing 5.1: MiniZinc Model for the Social Golfers Problem.

An assignment of a golfer to a group in each round is modeled through the golfers' position within the *rpg* array (defined in line 13), such that two golfers in positions P_1 and P_2 during some round belong to the same group iff: $\lfloor \frac{P_1}{S} \rfloor = \lfloor \frac{P_2}{S} \rfloor$. The constraint in lines 17 through 20 guarantees that each player appears only in one group each round, and there is no duplicate entry for the same player. For any two golfers, the array of variables in *ggr* (defined in line 14) represents the round which golfers g_1 and g_2 play together in a group. For example, $ggr[g_1, g_2] = 1$ would indicate that g_1 and g_2 play in the same group in the first round. The second constraint (lines 23 through 26) makes *ggr* consistent with *rpg*, i.e., it ensures that any pair of golfers play together in at most one round, the one

[1]See `https://github.com/MiniZinc/minizinc-benchmarks`

stated explicitly within ggr. Finally, the third constraint (lines 30 through 36) significantly reduces the search space by enforcing that players within each group are strictly ordered by their index, thus avoiding symmetrical solutions that assign golfers within a group to different positions within that group.

Figure 5.5 shows the result of identifying similar subtrees in the search performed by the Gecode solver for the instance of the problem with $R = 6$, $G = 5$, and $S = 5$ (i.e. for each of the 6 rounds, golfers need to split into 5 groups of 5 players).

Let us consider the top-most pattern – one of the largest patterns found both in terms of subtree height and size. In this study, I assume that the information about the actual domains of variables is not available to the user, as it can be rather costly to communicate this information for every node. Therefore, in the following analysis I focus on the decisions made on the path to the subtrees of interest, thus ignoring any variable assignments made implicitly by propagation.

The different decisions are shown at the bottom of the figure as:

left subtree	right subtree
`rpg[3,11]=3 rpg[3,12]≠6`	`rpg[3,11]≠3 rpg[3,11]≠4`
`rpg[3,12]=9 rpg[3,13]=15`	`rpg[3,11]=5 rpg[3,12]=6`
`rpg[3,14]=16 rpg[3,16]≠4`	`rpg[3,13]=14 rpg[3,14]=17`
`rpg[3,16]=5 rpg[3,17]≠6`	`rpg[3,16]=3 rpg[3,17]=9`

Since decisions of the form $X = a$ subsume $X \neq b$ $(a \neq b)$, the above decisions can be simplified as:

left subtree	right subtree
`rpg[3,11]=3`	`rpg[3,11]=5`
`rpg[3,12]=9`	`rpg[3,12]=6`
`rpg[3,13]=15`	`rpg[3,13]=14`
`rpg[3,14]=16`	`rpg[3,13]=17`
`rpg[3,16]=5`	`rpg[3,14]=3`
`rpg[3,17]=6`	`rpg[3,14]=9`

Recall that an expression of the form $rpg[R, P] = G$ indicates that golfer G played in position P during round R. In all of the above assignments $R = 3$, indicating that the difference in decisions are only in the third round. The table below illustrates which golfers are assigned to which positions in the third round, omitting the first two groups (associated with positions from 1..5 and 6..10), as they are not present in the different decisions.

position P	11	12	13	14	15	16	17	18	19	20
golfer G (left subtree)	3	9	15	16		5	6			
golfer G (right subtree)	5	6	14	17		3	9			

Empty cells indicate variables not yet assigned at the associated nodes, or assigned implicitly through propagation. Even with this limited information, one can notice a symmetrical relationship between subtrees. For instance, one can notice that, while players 3 and 9 occupy the first two places within group 3 (associated with positions 11..15) in the left subtree, the same players occupy the same places within group 4 (associated with positions 16..20) in the right subtree. Similarly, in the left subtree, players 5 and 6 are

placed first within group 3, and in the right subtree they are placed first within group 4. This observation suggests the groups 3 and 4 might be simply swapped around. (Note that this fact can be confirmed by inspecting the exact variable domains at the corresponding nodes, if they are provided by the solver.)

The fact that swapping groups can result in the same search, as indicated by similar (identical in this case) subtrees of large size, suggests that the two corresponding subproblems are also equivalent (that is, symmetrical). Indeed, the order of groups is irrelevant in the definition of the problem (which could be confirmed by the modeller), and leads to the exploration of symmetrical areas of the search space.

Having identified the presence and cause of some redundant search, the user could eliminate it by, for example, adding a symmetry breaking constraint on the rounds – the original `golfers1` model has constraints that break the symmetry *within* each group in each round, but not between *different* groups of the same round.

Specifically, I added a constraint that forces the groups within each round to be ordered; that is, the "smallest" golfer in the first group must be less than the smallest golfer in the second group, and so on for all adjacent groups:

```
1        constraint
2            forall (r in ROUNDS, g in [1 + i * S | i in 0..G-2])
3              (rpg[r, g] < rpg[r, g+S]);
```

The extended model requires exploring a smaller search space and, unsurprisingly, shows better performance than the original model. For example, when tested on the instances provided in the MiniZinc benchmarks library, the extended model either required the same amount of time (for the smallest instance), or 1-2 orders of magnitude less time (for larger instances). The extended model is now part of the MiniZinc benchmark suite as `golfers1b`.

5.5 Summary

This chapter presented a profiling technique that can discover inefficiencies in large executions: *the similar subtree analysis*. The technique works by automatically finding repeated patterns in the execution – something that often indicates the presence of inefficient search. The chapter presented a case study that demonstrated how the similar subtree analysis can be used to detect a symmetry present in the problem, but not addressed in the model. Eliminating the found symmetry has led to an improvement of the model.

Chapter 6

Techniques for Comparing Executions

6.1 Introduction

In order obtain a fast execution of a given problem, the modeller often considers multiple *encodings* of the problem, tries different search strategies or, indeed, different solvers until a satisfactory result is reached. Trying all possible combinations of models, search strategies, and solvers is impractical. It is therefore extremely important for the modeller to know which changes in the execution are favorable. This chapter aims to achieve the goal of designing profiling techniques that provide the modeller with that knowledge.

It particular, this chapter discusses two techniques for comparing the executions of different versions of the same model, and shows how utilising them can be beneficial for model improvement. First, Section 6.2 introduces *search merging* as a technique for comparing two executions that is most useful when their searches are relatively similar. Then, Section 6.3 proposes a technique for making two executions follow the same search, which broadens the circumstances under which the *search merging* technique can be effective. I then continue by demonstrating a case study, where these two techniques are used together to gain previously unknown information about a modelling case from the literature. Finally, I conclude by summarising the contributions of this chapter and discussing some potential limitations of the presented techniques.

6.2 Comparison of Executions

In order to know how a given model modification affected the model's execution (and, thus, determine which modifications lead to the biggest impact), it is essential to be able to compare the executions before and after the modification. Currently, this comparison is only possible by means of relatively crude aggregate measures, such as execution runtimes or the size of the corresponding search trees. A more detailed analysis of search trees is impractical to perform by hand, as search trees often contain many millions of nodes. This section explores the possibility of comparing execution through comparing their search trees automatically the resulting comparison tool is available within CP-Profiler as a proof of concept.

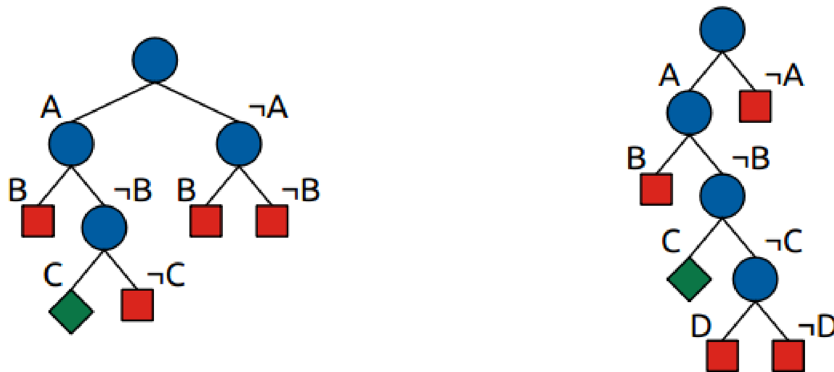

Figure 6.1: Two simple search trees used for demonstrating search merging.

6.2.1 Comparison Algorithm

Consider a situation where the difference between two executions of a problem is due to a relatively small modification in the model, and the search strategy is unchanged. Then, the search trees corresponding to the executions before and after the modification will often make similar decisions, especially at the higher decision levels in the tree (i.e. closer to the root). This is due to the fact that when there is a difference in propagation, the effect it will have on the search is likely to be more profound in the deeper levels in the search tree, as a consequence of the compounding effect of propagation: values pruned earlier lead to further pruning as the search progresses deeper. For that reason, such executions will tend to have similar structure near the higher decision levels.

The approach for comparison that I propose takes advantage of this fact, and compares the two executions by performing an inexpensive merging procedure of their search trees. A high-level description of this procedure is as follows. First, an empty tree is created for the purpose of displaying the result. Then, the process of merging proceeds by traversing both trees in lockstep in a depth-first-search manner, starting at the root nodes and comparing corresponding nodes at each step. A node is created on the resulting merged tree for every pair of corresponding nodes that are identical. Once a difference is found in corresponding nodes N_1 and N_2, a special *pentagon* node is added where the trees start to diverge. The underlying subtrees from the two original trees are copied into the resulting tree as the children of the *pentagon* node. The traversal then backtracks to find the next point of divergence, until the two trees are exhausted.

Example 6.2.1. Consider the two trees (intentionally small for demonstration purposes) depicted in Figure 6.1. Both trees find the same solution (identified by path $\{A, \neg B, C\}$). However, the execution on the left was able to identify a dead end earlier on the path $\{A, \neg B, \neg C\}$, while it also required more search on the path $\{\neg A\}$ than the execution on the right. The result of *merging* the two trees is shown in Figure 6.2, where the pentagons mark the parts in the search trees that are different. Note that the tree on the left has its pentagons collapsed (that is, their contents are hidden), and the one on the right is expanded. As a visual aid, my implementation within CP-Profiler highlights the different subtrees in two different colours depending on which execution they belong to, as it is shown in the figure.

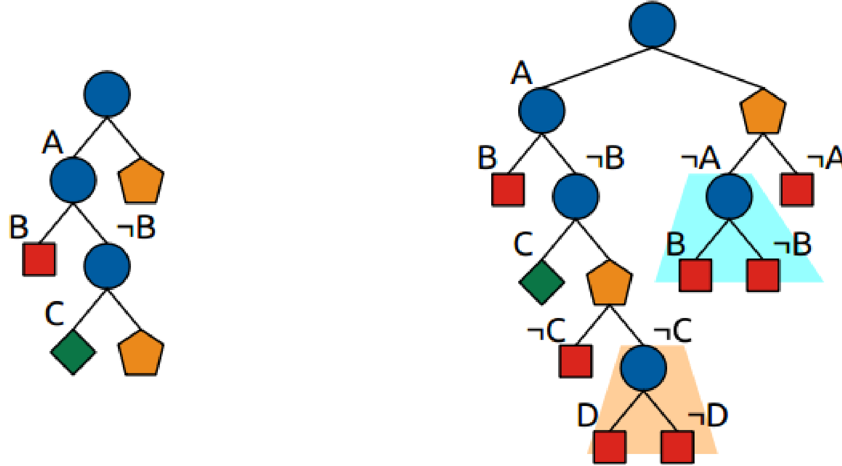

Figure 6.2: Comparison result collapsed (left) and with pentagons expanded (right).

Pseudo-code for a possible implementation of the merging algorithm is presented in Algorithm 6. Three stacks are initialised in line 2 and maintained throughout the code to perform the lockstep depth-first-search traversal of three search trees: two ($Stack_1$ and $Stack_2$) are for the executions being compared and one (*Stack*) is for the resulting tree. Every time a pair of nodes is extracted from their corresponding stacks (in lines 7 and 8), a reference *Node* to the top-of-the-stack node of the resulting tree is also acquired (line 9).

If $Node_1$ and $Node_2$ are not equal (checked in line 10), *Node* is turned into a *pentagon* with children $Node_1$ and $Node_2$ (and their underlying subtrees) copied into the resulting tree as children of *Node* (lines 20 through 23). Otherwise, *Node* is used to represent both $Node_1$ and $Node_2$ (which is achieved in line 11 by substituting *Node* with a copy of $Node_1$ – $Node_2$ could have been used instead). Additionally, all three stacks are extended with the children of the corresponding nodes (shown in lines 14 through 18).

In this context, two nodes are equal *iff* they both have the same *status* (that is, if the nodes both represent failures, solutions etc.), the same number of children, and the same *labels* on the paths to them. The equality checking is taken out into a separate procedure and outlined in Algorithm 7.

Note that other definitions for node equality can be considered. For example, the definition outlined in Algorithm 7 can be relaxed by taking only the variable names in labels into account (as opposed to requiring labels to match exactly). This can identify where the searches in two executions start to diverge in terms of variable selection.

The algorithm for merging trees is fast and scales well to large trees. It is easy to show that its time complexity, given a pair of trees T_1 and T_2, is linear in the number of nodes ($O(N_1+N_2)$), where N_1 and N_2 are the number of nodes in T_1 and T_2, respectively. The algorithm performs a depth-first-search traversal of T_1 and T_2 visiting at most $min(N_1, N_2)$ nodes. Every visited pair of nodes ($Node_1$ and $Node_2$) is either copied into the resulting tree T_3 (when the two nodes are equal), or causes the corresponding subtrees (for which $Node_1$ and $Node_2$ are roots) to be copied into T_3 under a newly created pentagon node. In either case, a node from T_1 or T_2 is only copied once per merging. Assuming that copying a single node is a $O(1)$ operation, the algorithm runs in $O(N_1+N_2)$, since only constant-time operations are performed for at most $N_1 + N_2$ nodes.

Algorithm 6: The algorithm for merging search trees.

```
   Input : Root1, Root2 : roots of trees to compare
   Result: merged tree is constructed rooted at Root
 1 Procedure Compare(Root1, Root2, Root)
 2     Stack1, Stack2, Stack ← empty stacks
 3     Stack1.push(Root1)
 4     Stack2.push(Root2)
 5     Stack.push(Root)
 6     while Stack1.size() > 0 do
 7         Node1 ← Stack1.pop()
 8         Node2 ← Stack2.pop()
 9         Node ← Stack.pop()
10         if Equal(Node1, Node2) then
11             copyInto(Node1, Node)
12             numberOfChildren ← size(Node1.children())
13             extend Node with numberOfChildren child nodes
14             for i ∈ [numberOfChildren, 0) do
15                 Stack1.push(Node1.getChild(i))
16                 Stack2.push(Node2.getChild(i))
17                 Stack.push(Node.getChild(i))
18             end
19         else
20             Node ← create pentagon
21             ChildL, ChildR ← get left and right children of Node
22             copyTree(Node1, ChildL)
23             copyTree(Node2, ChildR)
24         end
25     end
```

Algorithm 7: The procedure for comparing nodes.

```
1 Function Equal(Node1, Node2)
2     return size(Node1.children()) = size(Node2.children()) AND
3            statusOf(Node1) = statusOf(Node2) AND
4            labelOf(Node1) = labelOf(Node2)
```

6.2.2 Performance experiments

Consider three different models of the Golomb Ruler problem introduced in Section 2.5. In the first model, a global constraint *alldifferent* is utilised to ensure that the differences between any pair of marks are unique (the model is the same as was presented in Listing 2.1). The second model uses instead the decomposition of the alldifferent global constraint (a set of disequality constraints). The third model is similar to the first one, and differs only in that it lacks the symmetry breaking constraint (declared in line 17 of Listing 2.1). All three models are solved with the Gecode solver using a static search strategy that selects variables from the `mark` array in order from first to last, and selects values from lowest to highest from the corresponding domains.

Table 6.1 presents statistical results of a pairwise comparison between the discussed model variations for different lengths of the ruler: 9, 10, and 11. In particular, the first column shows the instance used (determined by the problem's parameter m) and the

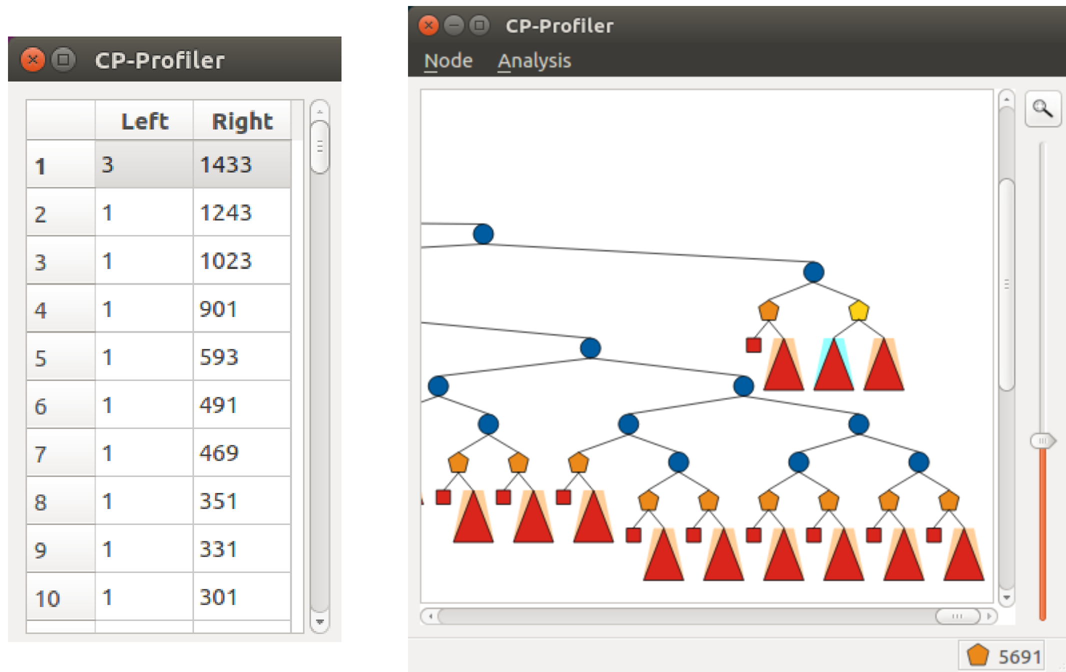

Figure 6.3: Sorted list of pentagons (left), which allows to locate them on the tree (right).

number of nodes in the tree obtained using the first model. Other columns show for the two alternative models the number of nodes in their search trees, and the result of merging their trees with that from the first model: the number of *pentagons* and the time (averaged over 3 runs) required to merge the trees. The results demonstrate that multi-million node trees can be merged in a relatively short time. Additionally, these results suggest that the search trees of executions that use fixed search share a lot in common near the roots, as indicated by the large number of pentagons.

Note that the number of pentagons is presented for demonstration purpose only and is not intended to be used by the modeller to extract insights. Instead, the modeller is presented with a table of pentagons sorted based on the difference in the number of nodes between the left and the right subtrees. An example of such table is presented in Figure 6.3 on the left. The modeller can select a row representing a pentagon (first row is selected in the example), and the pentagon from the traditional search tree visualisation will be highlighted (shown in Figure 6.3 on the right, where the selected pentagon is colored in yellow). The modeller can thus learn that the largest difference in two executions is near the end of the search. Studying that area closer, for example, by examining the search decisions near the pentagon might reveal further insights.

Instance (nodes)	Golomb (decomposed)			Golomb (no symmetry breaking)		
	nodes	pentagons	time	nodes	pentagons	time
m = 9 (19’635)	83’517	5’691	0.07	34’909	2’433	0.04
m = 10 (115’931)	633’103	34’355	1.35	191’045	12’530	0.99
m = 11 (1’677’105)	9’367’521	478’095	7.37	3’140’787	220’175	3.24

Table 6.1: Experimental results of merging search trees.

The algorithm for merging (in the form as presented here) does not work well for *n-ary* branching, where the number of children for every node depends on the domain of the corresponding variable. This fact can lead to some undesirable behaviour: when a pair of nodes from different executions have the same path but differ in the number of children, they will be treated as different by the algorithm. However, some of their children (and, indeed, entire subtrees) can match, and, thus, it would be logical to merge them. To accommodate cases like that, the algorithm could be modified by: (1) removing the requirement that equal nodes have the same number of children, and then (2) identifying the subset of children in the first execution that match those in the second execution. These modifications would lead to more accurate results, but would also raise the level of time complexity above linear, since a sorting operation appears to be necessary for modification (2).

A further limitation of the proposed method is that it requires the two executions to have a similar search near the roots of the search trees. As shown earlier, this situation is not uncommon, and it arises, for example, when the two executions search over the same variables using fixed search. However, the methodology can be useful even when a dynamic search strategy is employed by one of the executions. In particular, if the search in one execution can be made to match the dynamic search of another execution, the search merging can be used to determine the effect of different propagation. Aiming to achieve that, I propose a methodology for *search replaying* in the following section.

6.3 Search Replaying

Given a solver, the two main factors that affect execution are propagation and search. However, the two are not independent, that is, one can significantly affect the other. For example, when the *first-fail* dynamic search strategy is employed, which branches on variables with the smallest domains, domain reduction in decision variables caused by propagation can significantly alter the search. Usually the modeller specifies the search and the propagation separately: search – through choosing a search strategy; and propagation – implicitly through the choice of the variables and constraints used to describe the problem in the model, and explicitly by enforcing the consistency level for each constraint propagator used in the model (such as bounds or domain consistency). Importantly, when comparing two executions (before and after some modification), it can be useful to consider propagation and search in isolation, decouple them, so that user modifications to one or the other can be evaluated.

One way to achieve this decoupling is by forcing the same decisions on both executions, and, thus, making sure the executions have the same (or quite similar) search. To achieve this, I propose the *search replaying* methodology, which involves recording all search decisions made in one execution (E_1) to a file (called *search log*), and then using this *search log* to drive the search in another execution (E_2). I refer to E_1 as the *recording* execution, and to E_2 as the *replaying* execution. Clearly, applying this technique is different from simply using the same search strategy in both E_1 and E_2, since, for example, if a dynamic search strategy is used, even small differences in variable domains (caused, for example, by a difference in propagation level) may lead to different search decisions.

When a search is replayed this way, two problems need to be addressed: (1) some decisions specified in the search log might no longer be necessary in E_2 due to differences in propagation; (2) contrary to the first problem, decisions made in E_1 may not be sufficient for an exhaustive search in E_2.

To better understand how problem (1) can arise, let us consider the case where at some node N in the search, the execution E_1 makes decision $V \leftarrow k$, where V is the branching variable and $k \in dom(V)$. It is possible that in E_2, due to a difference in propagation, $k \notin dom(V)$ in node N' (corresponding node to N from E_1), indicating that $V \leftarrow k$ will lead to a dead-end in E_2. In this case, I call $V \leftarrow k$ a *failing* decision. It is also possible that in E_2, $k \in dom(V)$ in node N' but $|dom(V)| = 1$ (i.e. the variable V is already assigned to k in execution E_2). In this case, I call the decision $V \leftarrow k$ an *implied* decision, as it is unnecessary (it does not prune any variable domains).

One way to address the problem of both *implied* and *failing* decisions being present in the *search log* is to simply skip them. Unfortunately, this might harm the effectiveness of the *merging* algorithm discussed earlier in this chapter, since this might lead to small (and, perhaps, unimportant) discrepancies in the two trees, potentially focusing the user on the uninteresting parts of the comparison. Instead, I suggest processing these decisions normally, thus causing the creation of unnecessary nodes in the search tree. This rule does not apply, however, in the case where $|dom(V) = 0|$ in N', that is, where the node is a dead-end, and the decision corresponding to its descendants can be safely skipped (without hurting the *merging*). Note that posting *implied* or *failing* decisions this way should not trigger any propagation, and, therefore, it should not significantly affect the execution time of E_2. Additionally, if the unnecessary nodes are correspondingly marked as *implied* or *failing*, the *merging* can take the provided information into account when, for example, the node difference within a *pentagon* is calculated.

Regarding problem (2), the problem arises when a node N of execution E_1 is a leaf node (either a dead-end, or a solution), but the corresponding node N' in E_2 has some unassigned variables and, therefore, needs to search further with no branching decisions available in the search log. In this case, I propose for the solver of E_2 to continue the search in N' with some form of default branching until it backtracks out of N'. Note that if N is a solution node, the search of E_2 is also guaranteed to discover a solution at some point under N'. This follows from the fact that nodes N and N' represent the same subproblem and the invariant that if a problem has solutions, any two solvers will find a solution given enough time. Following the same reasoning we can conclude that if N is a failed node, N' will represent a failed subtree in execution E_2.

The methodology for search replaying as described here fits well within the profiling system proposed in Chapter 3. This is because the branching information is readily available on the search tree, and thus no solver modifications are required in order to record the search. Instead, the search can be recovered by traversing the search tree in a depth-first search manner.

However, it is still necessary to modify the solver used for *replaying* execution E_2, as solvers do not usually provide that functionality. In order to evaluate the methodology for search replaying, I modified the Gecode solver to be able to read and follow any search recorded accordingly to the protocol as described below.

6.3.1 The Protocol for Replaying

The replaying scheme that I propose requires the following information: a unique identifier per node, the number of children of every node, and the decision made for reaching every child node. More specifically, my implementation records a search in a *search log* – a human-readable text file, where each line, or *entry*, corresponds to a single node in the search of E_1. Each entry, in turn, is divided into *tokens* separated by a whitespace character. The first two tokens in each entry are the identifier of the corresponding node

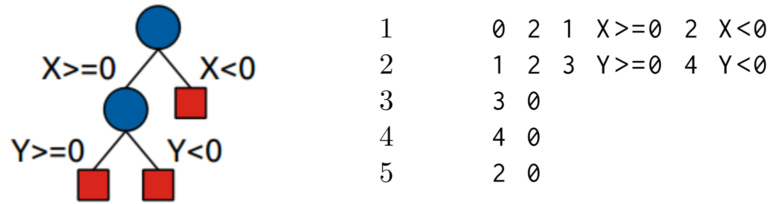

Figure 6.4: Basic search tree to replay (left) and corresponding search log (right).

and the number of direct children nodes it has. For a leaf node, the number of children is 0, and the corresponding entry does not contain any further tokens. For a non-leaf node, the entry is expanded by a pair of tokens per child, containing the child's identifier and the branching decision it represents.

Figure 6.4 provides an example of a basic search tree (on the left) and the search log that would be produced by following the protocol discussed above. The root node, for example, is represented by the first line, where its `id` is specified as 0, the number of children nodes is 2; the first child's id is 1 representing decision $X \geq 0$, while the second node's id is 2 representing decision $X < 0$.

6.3.2 The Algorithm for Recording Search

The algorithm for recording search where *Root* represents the root of the search tree E_1 is outlined in Algorithm 8. I again use a *Stack* (initialising it in line 2, and updating it in lines 4, 6, and 15) to follow a depth-first-search traversal. Additionally, an empty string *Log* is initialised in line 3, and will be used to store the resulting search log. (The search log will normally be saved to a file upon the completion of the algorithm.) While performing the traversal, *Log* is extended by the current node's identifier (line 7) and by the number of children nodes it has (line 8). (The operator `<<` is assumed to add separators between each token.) If the current node has children, each child node further extends *Log* with its identifier (line 10) and its corresponding branching decision (line 11). Each node entry is separated by a new line character in line 13. Finally, the constructed the search log is returned once all nodes in the search tree have been visited.

6.3.3 The Algorithm for Replaying

A basic search engine for replaying a previously recorded search is outlined in Algorithm 9. A stack (*Stack*) of branching choices is created in line 2 and maintained throughout the execution to perform a depth-first-search traversal of the search tree. The choices are represented as a list of tuples (Id, $Constraint$), each representing a branching decision (in the form of a constraint), and the identifier of the corresponding child node. The stack is first populated in line 3 with an entry representing the root node (identifier 0) and a dummy decision/constraint *true* (it does not modify the problem). The following steps are then repeated until the stack is exhausted (which will indicate exhausting the search space). First, a choice is retrieved from the stack in the form of a tuple (Id, $Constraint$). The $Constraint$ is applied at the current node in line 6. This will run propagation algorithms (omitted here) and return a *Status* taking one of the following values: *fail*, *solution*, or *branch*. If *Status* holds the value of *failure* or *solution* (indicating that the current node represents a dead-end or a solution, respectively), the search backtracks by undoing the last decision (line 8). Otherwise, the search continues branching in line 10.

Algorithm 8: The algorithm for recording search.

```
   Input : Root – root of the search tree to record
   Result: Log – string representing the search log
 1 Function RecordSearch(Root)
 2   Stack ← <empty stack>
 3   Log ← <empty string>
 4   Stack.push(Root)
 5   while Stack.size() > 0 do
 6     Node ← Stack.pop()
 7     Log << Node.id()
 8     Log << size(Node.children())
 9     for Child ∈ Node.children() do
10       Log << Child.id()
11       Log << Child.decision()
12     end
13     Log << <new line>
14     for Child ∈ reverse(Node.children()) do
15       Stack.push(Child)
16     end
17   end
```

The *getBranching* procedure retrieves the branching choices for the node identified as Id. If the list of choices returned by *getBranching* in not empty, the choices are pushed onto the $Choices$ stack in reversed order, ensuring that the left-most choice is applied first (lines 12 to 14). If, on the other hand, the list of choices is empty (indicating that, contrary to the current execution, the original search did not require any additional branching and arrived at either a *solution* or a *failure* in the corresponding node), the replaying solver is expected to solve the problem in the current node without consulting the search log, but utilising some default branching strategy instead (line 16). This "unguided" search continues until the search space is exhausted at the node Id, and the replaying resumes with the new iteration of the while loop thereafter.

The algorithm for extracting branching decisions from the recorded search log is outlined in Algorithm 10. A readable stream of recorded search is assumed to be available through the *LogStream* handle (defined globally elsewhere). A node identifier $Current$, which is supplied by the search engine, represents the node that should be extended next. The algorithm obtains the next line of the log file from the stream (line 4) and extracts the first two tokens (separated by a *whitespace* character): $Node$ and $NumberOfChildren$, which represent, respectively, the node's identifier and the number of alternatives of the next node in the recorded execution. All such entries are skipped until the one relevant is found, that is, until the log entry corresponds to the node we look for – $Current$ (this condition is checked in line 6). Since both recording and replaying are done following the depth-first-search order traversal, it is guaranteed that all entries can be safely skipped this way. (This situation only arises when the replaying execution E_2 doesn't require branching information regarding a part of the tree, for example, because the replaying execution is able to identify a failure earlier than the original execution, and the associated search in the original search tree E_1 needs to be skipped.)

Once a match in the search log is found, the corresponding identifiers (Id) and branching constraints ($Choice$) for each alternative are extracted from the remainder of the entry and added to the list of choices (lines 10 and 11). The latter is then returned as the result

Algorithm 9: Outline of the search engine used for replaying.

```
1  Function Replay()
2      Stack ← <empty stack>
3      Stack.push((0, true))
4      while Stack.size() > 0 do
5          (Id, Constraint) ← Stack.pop()
6          Status ← apply(Constraint)
7          if (Status = failure) OR (Status = solution) then
8              undo(Constraint)
9          end
10         Choices ← getBranching(Id)
11         if Choices.size() > 0 then
12             for Choice in reverse(Choices) do
13                 Stack.push(Choice)
14             end
15         else
16             solveUnguided(Id)
17         end
18     end
```

to the search engine (line 13). The next time the same procedure is called, reading from the *LogStream* will continue from where it last stopped.

Algorithm 10: Procedure for extracting next branching from a search log.

```
   Data  : LogStream – readable stream for the search log
1  Function getBranching(Current)
2      Choices ← <empty list>
3      while true do
4          Entry ← getLine(LogStream)
5          Tokens ← Entry.split(' ')
6          (Node, NumberOfChildren) ← (Tokens[0], Tokens[1])
7          if Node ≠ Current then  continue
8
9          for Child ∈ 1..NumberOfChildren do
10             (Id, Choice) ← (Tokens[Child+2], Tokens[Child+2+1])
11             add (Id, Choice) to Choices
12         end
13         return Choices
14     end
```

Example 6.3.1. Consider the model of the problem of Golomb Ruler presented in Chapter 2. The model was first instantiated with a parameter `m=10`, signifying that the shortest ruler with 10 marks is to be found. Then, Gecode was used to solve the problem using a restart search exploration strategy, generating the search tree displayed in Figure 6.5. The search required over a million backtracks (1'100'954) and restarted 28 times to find an optimal solution and prove optimality.

After the search had finished, it was recorded to a file (54Mb in size) and replayed by the same solver. This time, however, it was not necessary to instruct Gecode to perform restarts. The resulting execution (obtained as the result of such replaying) was

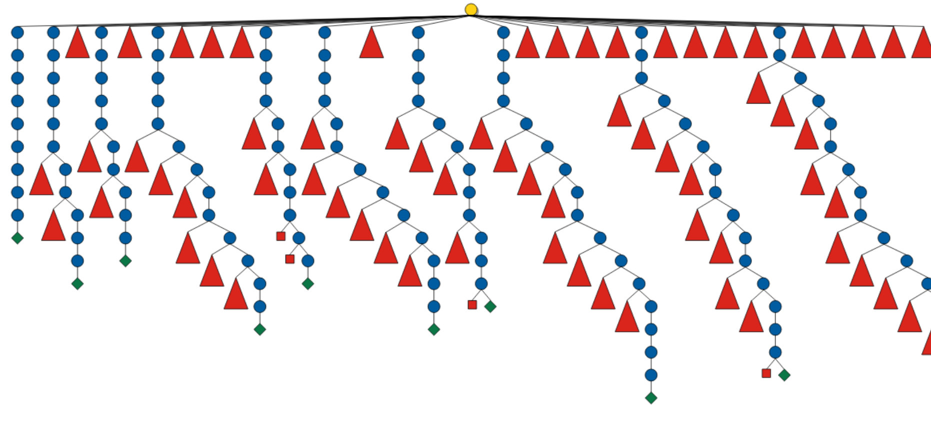

Figure 6.5: Search tree as a result of solving Golomb 10 with restarts (Gecode).

identical to the original, recorded execution. This fact was confirmed by merging the two corresponding search trees, which resulted in two trees merged entirely, without creating any *pentagons*. This experiment was important, as it demonstrated that replaying works even for advanced exploration strategies, such as restarting search.

6.4 Case Studies

In this section I demonstrate how search replaying can be an effective technique for determining the effect of a model change, by means of case study on two problems.

6.4.1 Case Study: The All Interval Series Problem

Consider the All Interval Series problem, which requires finding a permutation of numbers 1 through n, such that the differences between all adjacent pairs of numbers form a permutation of the numbers 1 to $n-1$ (all different). Consider, for example, an instance of the problem where $n = 5$. A possible solution to this instance can be a sequence of numbers 1, 5, 2, 4, 3. This sequence can be encoded through the X variables, where each X_i variable represents the number located in position i ($i \in 1..5$): $X_1 = 1$, $X_2 = 5$, $X_3 = 2$, $X_4 = 4$ and $X_5 = 3$. Alternatively, the same sequence can be encoded through the Y variables, where each Y_i variable represents the position of number i ($i \in 1..5$): $Y_1 = 1$, $Y_2 = 3$, $Y_3 = 5$, $Y_4 = 4$ and $Y_5 = 2$. These two different encodings can be used to model the same problem in two different (dual) ways.

In [Choi et al., 2007] the authors study two models (or *viewpoints*) for each encoding of the problem (M_X and M_Y, respectively). While using either of the models is sufficient to solve the problem, Choi et al. also considered simultaneously using both *viewpoints* linked through *channeling* constraints (which show how to convert assignments between the variables of each viewpoint), thus creating a combined model (M_{XY}). The technique of combining different viewpoints in a single model – *redundant modelling* [Cheng et al., 1999] – is akin to adding *redundant constraints* and, thus, aims to achieve stronger propagation at the cost of performing more computations.

The experimental results of Choi et al., where a solver is instructed to find all possible solutions, show that the M_{XY} model performs significantly better than M_X or M_Y, both in terms of search space reduction and time reduction. Indeed, it was expected that the

```
1  int: n;
2  array[1..n] of var 1..n: X;
3  array[1..n-1] of var 1..n-1: U;
4
5  include "alldifferent.mzn";
6  constraint alldifferent(X);
7  constraint alldifferent(U);
8
9  constraint forall(i in 1..n-1)(U[i] = abs(X[i] - X[i+1]));
10
11 solve :: int_search(X, first_fail, indomain_min, complete) satisfy;
```

Listing 6.1: M_X variant of the *all-intervals* model in *MiniZinc*.

execution of M_{XY} would lead to a smaller search tree, as a result of stronger propagation in the combined model. However, the authors used a dynamic search strategy (branching on variables with smallest domains first), where the individual branching decisions depend on how much the constraint propagation engine has been able to narrow down the variable domains at each node. As a consequence, based only on the number of nodes in the search tree and the execution time, the authors did not verify whether the improvement was caused by the stronger propagation obtained from the extra constraints, or by the difference in the search decisions due to the differences in propagation. This information could, for example, help determine whether further improvements are possible. To answer this question I applied the *replaying* methodology discussed above.

In order to reproduce the results of Choi et al., I used corresponding MiniZinc models M_X, M_Y and M_{XY}, presented in Listings 6.1, 6.2, and 6.3, respectively. The variables and constraints are equivalent to those used in [Choi et al., 2007].

In the M_X model (Listing 6.1), the order of numbers is declared using the array of decision variables `X` in line 2, such that X_i denotes the number at position `i`. The array of decision variables `U` (line 3) denotes the differences between adjacent numbers, as computed by the constraint in line 9. The values of the variables in `X` are declared to be unique in line 6 through the use of the *alldifferent* global constraint. (*MiniZinc* requires explicitly including global constraints used in the model, hence the *include* statement in line 5.) Similarly, the values of the variables in `U` are declared to be unique in line 7.

```
1  int: n;
2  array[1..n] of var 1..n: Y;
3  array[1..n-1] of var 1..n-1: V;
4
5  include "alldifferent.mzn";
6  constraint alldifferent(Y);
7  constraint alldifferent(V);
8
9  constraint forall(i,j in 1..n where i < j)
10                   (Y[i] - Y[j] = 1 -> (V[j-i] = Y[j]));
11 constraint forall(i,j in 1..n where i < j)
12                   (Y[j] - Y[i] = 1 -> (V[j-i] = Y[i]));
13
14 constraint abs(Y[1] - Y[n]) = 1;
15 constraint V[n-1] = min(Y[1],Y[n]);
16
17 solve :: int_search(Y, first_fail, indomain_min, complete) satisfy;
```

Listing 6.2: M_Y variant of the *all-intervals* model in *MiniZinc*.

In the M_Y model (Listing 6.2), the order of numbers is declared using the array of decision variables `Y` in line 2, such that Y_i denotes the position of number i. Auxiliary variables V_i are declared in line 3 to denote the differences between adjacent numbers,

n	M_X	M_Y	M_{XY}	M_{XY} (M_Y)	M_{XY} (M_X)
10	10,876	48,876	1,446	1,353	9,644
11	48,305	264,637	4,605	4,220	42,505
12	225,310	1,562,749	15,648	13,950	195,987
13	1,119,821	9,799,788	57,037	49,565	974,213

Table 6.2: All Interval Series Problem, number of failures to find all solutions.

as computed by the constraints in lines 9 to 12. Again, the constraints in lines 6 and 7 ensure that the numbers are unique, and the differences between adjacent numbers are also unique. Redundant constraints in lines 14 and 15 indicate that the only way to get the difference $n - 1$ is to make 1 and n adjacent.

The combined model M_{XY} (Listing 6.3) combines the two alternative encodings through the arrays of variables X and U declared in lines 5 and 6, and the constraints declared in lines 11 and 12. Finally, the two viewpoints are connected through an "inverse" global constraints in lines 24 and 25. The global constraint `inverse(X, Y)` is semantically equivalent to the logical constraint $X_i = j \iff Y_j = i$, for all $i, j \in 1..|X|$. The models is executed using a first-fail search strategy on the Y variables, that is, branching on the Y variables and choosing at each step the Y variable with the smallest domain.

```
1  int: n;
2
3  array[1..n] of var 1..n: Y;
4  array[1..n-1] of var 1..n-1: V;
5  array[1..n] of var 1..n: X;
6  array[1..n-1] of var 1..n-1: U;
7
8  include "alldifferent.mzn";
9  constraint alldifferent(Y);
10 constraint alldifferent(V);
11 constraint alldifferent(X);
12 constraint alldifferent(U);
13
14
15 constraint forall(i in 1..n-1)(U[i] = abs(X[i] - X[i+1]));
16
17 constraint forall(i,j in 1..n where i < j)
18                 (Y[i] - Y[j] = 1 -> (V[j-i] = Y[j]));
19
20 constraint forall(i,j in 1..n where i < j)
21                 (Y[j] - Y[i] = 1 -> (V[j-i] = Y[i]));
22
23 include "inverse.mzn";
24 constraint inverse(Y,X);
25 constraint inverse(U,V);
26
27 constraint abs(Y[1] - Y[n]) = 1;
28 constraint V[n-1] = min(Y[1],Y[n]);
29
30 solve :: int_search(Y, first_fail, indomain_min, complete) satisfy;
```

Listing 6.3: M_{XY} variant of the *all-intervals* model in *MiniZinc* (combined).

The experimental results in Table 6.2 show the number of failures as a measure of the size of the search trees for the several executions on four instances ($n = 10, n = 11, n = 12$, and $n = 13$). In particular, columns "M_X", "M_Y", and "M_{XY}" represent the execution of the associated models with the *first-fail* strategy, as used in [Choi et al., 2007]. The results match those obtained by Choi et al. in that M_{XY} significantly outperforms both M_X and (to a larger degree) M_Y.

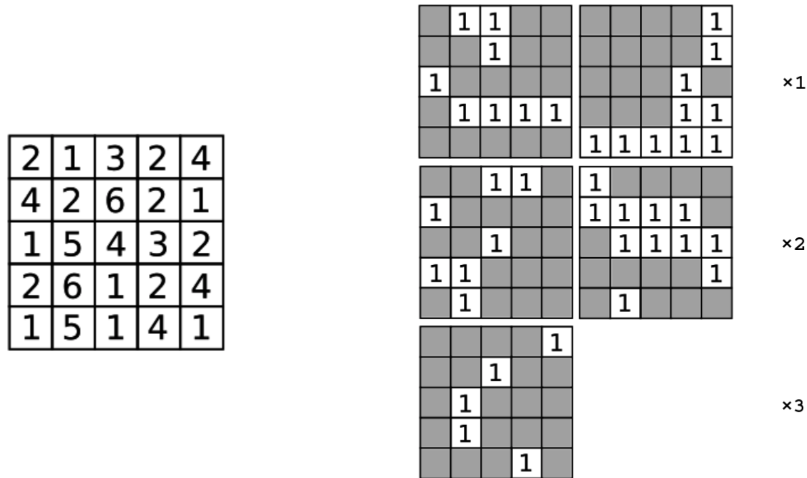

Figure 6.6: Intensity matrix (left) and its decomposition (right) in the Radiation problem.

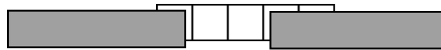

Figure 6.7: Two out of five cells are exposed to radiation using two horizontal rods.

Following the methodology for replaying, I recorded the search decisions made when solving the M_X and the M_Y models and replayed these two searches using the combined model M_{XY}, which resulted in executions shown in Table 6.2 as M_{XY} (M_X) and M_{XY} (M_Y), respectively. These replayed executions additionally show that solving M_{XY} by replaying the search of M_Y search (shown as M_{XY} (M_Y)) also outperforms the original execution of M_Y. Since every decision taken during the execution of M_{XY} in the replay search is the same as for M_Y, this shows that the considerable reduction in search space is entirely due to propagation. Interestingly, the execution M_{XY} (M_Y), which uses the model M_{XY} and replays the search of M_Y (the worst model based on the size of the search tree in the original experiment) gives even better results than those obtained using the same model with the first-fail search strategy. Investigating why the decisions guided by the *weaker* propagation consistently yield *better* behaviour than those guided by the stronger propagation is extremely interesting, but falls outside the scope of this thesis. However, it could be a promising avenue for future work.

6.4.2 Case Study: The Intensity-Modulated Radiation Therapy Problem

Consider the Intensity-Modulated Radiation Therapy problem [Baatar et al., 2011], where radiation is given through repeated exposures of a device that delivers radiation of uniform intensity through a rectangular field. This rectangular field is shaped using two horizontal lead rods per row of the rectangle. Each two rods are positioned at the left and right of the row, respectively, and can slide horizontally to block the radiation. In each exposure, the rods are moved into a given position, the radiation source is switched on for a specified length of time and then switched off, to move to a new position. The left part of Figure 6.6 shows an example of an intensity matrix with five rows and columns as a possible input data for this problem. The numbers in cells indicate the amount of radiation individual cells require to receive. Figure 6.7 demonstrates how the amount of radiation in this problem is controlled by using two lead rods per row; in this case, the rods expose the second and the third cells of the row.

The model I studied is the one used in the MiniZinc Challenge 2015. The input data is an $m \times n$ `Intensity` matrix of non-negative integers, where `Intensity[i,j]` represents the total amount of radiation that the cell in row i, column j should expose. The goal is to find a decomposition of the matrix into a positive linear combination of the smallest number of binary matrices (representing the different positions), each with the consecutive-ones property (i.e., all `1`s in any row are consecutive), where the `0`s represent the part of the row occluded by the rods and the `1`s the non-occluded part which exposes radiation. The reason to minimise the number of binary matrices is to reduce the number of positions and, thus, the cost and time required for the radiation procedure.

An example of such decomposition for the example on the left in Figure 6.6 is shown on the right in Figure 6.6. Note that these decomposition matrices are associated with the durations for which they are applied (indicated by the numbers on the right), which in this figure range from one to three time units.

The model is as follows:

```
1  int: m;  % Rows
2  int: n;  % Columns
3  set of int: Rows    = 1..m;
4  set of int: Columns = 1..n;
5  array[Rows, Columns] of int: Intensity; % Intensity matrix
6  set of int: BTimes = 1..Bt_max;
7  int: Bt_max   = max(i in Rows, j in Columns) (Intensity[i,j]);
8  int: Ints_sum = sum(i in Rows, j in Columns) (Intensity[i,j]);
9
10 var 0..Ints_sum: Beamtime; % Total beam-on time
11 var 0..m*n: K; % Number of shape matrices
12   % N[b] is the number of shape matrices with associated beam-on time b
13 array[BTimes] of var 0..m*n: N;
14   % Q[i,j,b] is the number of shape matrices with associated beam-on time
15   % b that expose cell (i,j)
16 array[Rows, Columns, BTimes] of var 0..m*n: Q;
17
18 constraint Beamtime = sum(b in BTimes) (b * N[b]);
19 constraint K = sum(b in BTimes) (N[b]);
20 constraint forall(i in Rows, j in Columns) (
21   Intensity[i,j] = sum([b * Q[i,j,b] | b in BTimes])
22 );
23 constraint forall(i in Rows, b in BTimes) (
24   upper_bound_on_increments(N[b], [Q[i,j,b] | j in Columns])
25 );
26
27 predicate upper_bound_on_increments(var int: N_b, array[int] of var int: L) =
28   N_b >= L[1] + sum([ max(L[j] - L[j-1], 0) | j in 2..n ]);
29
30 int: obj_min = lb((m*n + 1) * Beamtime + K);
31 int: obj_max = ub((m*n + 1) * Beamtime + K);
32 var obj_min..obj_max: objective = (m*n + 1) * Beamtime + K;
33
34 solve :: int_search([Beamtime] ++ N ++
35         [Q[i,j,b] | i in Rows, j in Columns, b in BTimes],
36         input_order, indomain_split, complete) minimize objective;
```

Listing 6.4: The model used for the radiation problem.

The first 8 lines introduce the parameters of the problem: lines 1 and 2 introduce m and n, respectively, line 5 introduces the intensity matrix, line 7 computes in `Bt_max` the maximum intensity value in the matrix, and in `Ints_sum` the sum of all intensity values in the matrix. The latter serves as an upper bound to the total amount of time the radiation beam will have to be on. The next lines introduce the variables of the problem: line 10 defines the total exposure time `Beamtime` for the solution, line 11 defines the total number `K` of binary matrices in the solution (with `m` $\times$ `n` as the upper bound), line 13 defines array `N`, where variable `N[b]` is the number of matrices with the same exposure time `b`, and

line 16 defines array Q, where variable Q[i,j,b] is the number of binary matrices exposing cell (i,j) for the duration of time b.

The constraints start on line 18, which computes the total exposure time as the result of adding the exposure time used for every binary matrix. The constraint in line 19 computes the total number of matrices as the result of adding those associated with every exposure duration. The constraint starting in line 20 states that the intensity required by each cell (i,j) in the intensity matrix must be achieved by the solution, that is, it must be equal to the sum of exposure times for each of the binary matrices that expose that cell. The constraint starting in line 23 ensures the consecutive-ones property of the binary matrices by requiring that, for every exposure duration b and every row i of Q[i,j,b], the number of times the intensity increases from a column j-1 to the next j, is equal to or less than the number N[b] of binary matrices with that exposure duration. Finally, the solve item in line 34 states that the goal is to minimise the value of the objective by using a search strategy that considers variables in the following order: Beamtime, variables from N, and variables from Q. The labeling is performed by making search decisions of the form $X < v$ and $X \geq v$, where X is a decision variable, and v is a value from its domain. Effectively, this search strategy tries values in order from the smallest to the largest values from the domains of the variable.

The results of the MiniZinc Challenge 2015 show that learning solvers, such as Chuffed and Opturion CPX, significantly outperform non-learning solvers, such as Gecode, on this problem. To investigate the reasons for this, I first executed one of the instances of this problem from the challenge (namely, the i7-9 instance) using Chuffed. A lantern tree visualisation for this execution is shown in Figure 6.8, which contains around 25 thousand nodes, with "lantern" nodes representing subtrees of up to 800 nodes. An interesting property of this search is that after reaching the first solution (represented by the green "lantern" node), the solver restarts and performs a relatively small search to prove the optimality of this solution (the corresponding node is shown with grey background). In fact, executing all other instances shows that their executions also find only one solution, which is proved to be optimal in the second restart. By examining the search strategy stated in the model, it can be easily confirmed that the search is designed to find the optimal solution first: the smallest values for the Beamtime and the variables from N are assigned first, and the objective is a monotonically increasing function of Beamtime and $\mathtt{N}_i$, which is to be minimised. It can thus be concluded that the proof of optimality is unnecessary.

Executing the same instance using Gecode, I visually observed that its search tree did not reveal any interesting patterns. Therefore, I executed this instance using Gecode again, but this time replaying the search decisions from the Chuffed's execution. Note that, although both Gecode and Chuffed use the same search annotation (line 34 of the model), the actual search decisions made by the two solvers are likely to deviate: regardless of the search annotation, Chuffed would sometimes perform a certain procedure during search called a "backjump", which can significantly affect the search (see Chapter 7 dedicated to learning solvers).

The search tree of the resulting replaying execution is shown in Figure 6.9, and contains around one million nodes in total, with "lantern" nodes representing subtrees of up to 200'000 nodes. Note that, as in the tree for Chuffed, the subtree with the grey background corresponds to the proof of optimality. This time, however, the proof of optimality required by far the most search effort – a very different behaviour from that of Chuffed.

Interestingly, while Gecode's tree is significantly larger than that of Chuffed, the amount of search required to find the optimal solution by both solvers seems to be much

closer in magnitude than that of the proof of optimality. In fact, we can consult the tree to find that Gecode required exploring 3.93 times more nodes to find the solution (17,780 nodes in Chuffed against 69,931 nodes in Gecode), while it required 6,755.36 times more nodes to prove optimality (150 nodes in Chuffed against 1,013,304 nodes in Gecode). This is an interesting observation, because, while the original results from the MiniZinc competition show that Gecode often fails to find solutions early, this is not the case when Gecode replays Chuffed's search.

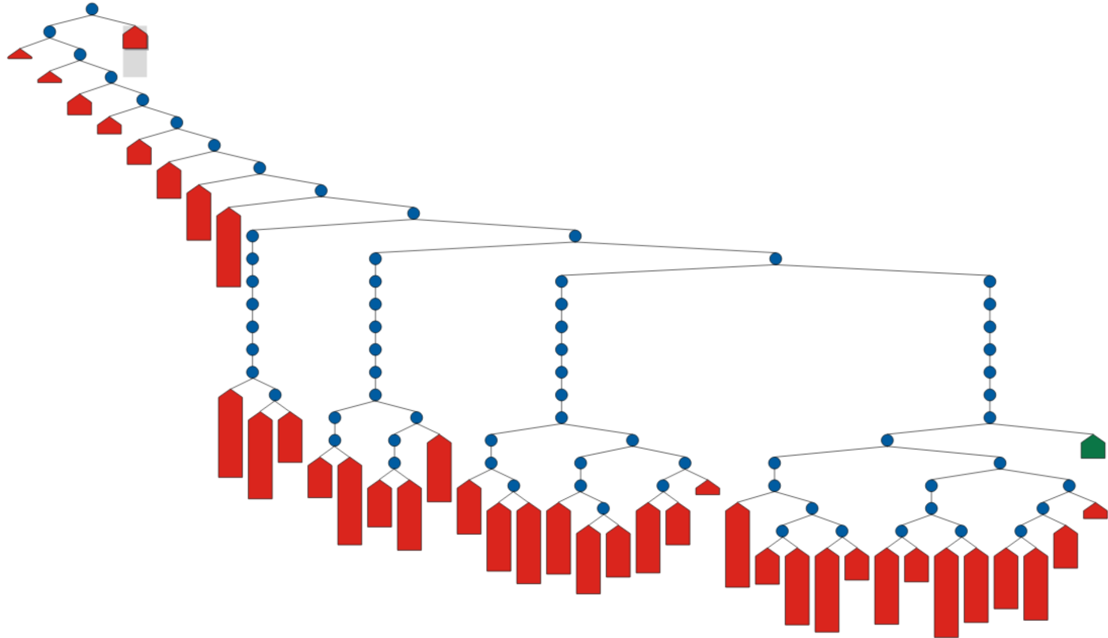

Figure 6.8: The radiation problem solved using Chuffed.

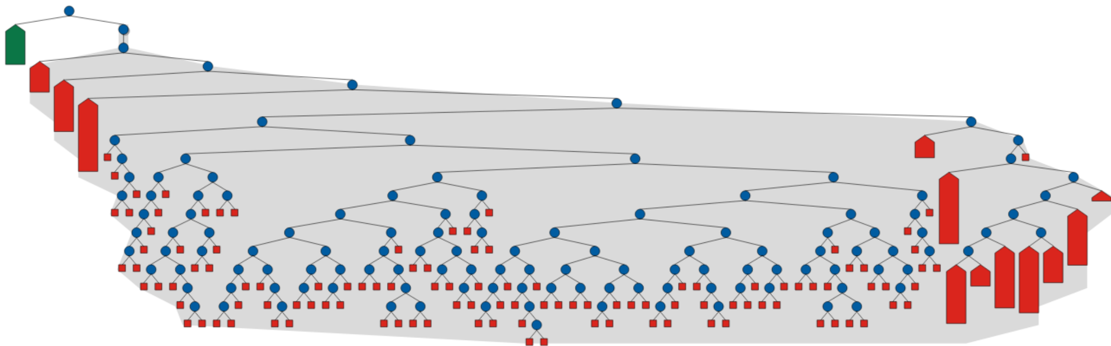

Figure 6.9: The radiation problem solved using Gecode with search replaying.

To investigate further, I decided to focus on the part of the execution responsible for finding the optimal solution. To achieve this, I replaced the text "`minimize objective`" in line 36 of the model with "`satisfy`". This change results in the solvers producing the same search as before for finding the first solution, but terminating immediately after that. I then considered some of the smaller instances used in the challenge (namely, *i*6-11, *i*7-9, and *i*9-11), and solved each instance with Chuffed, with Gecode, and with Gecode replaying the search performed by Chuffed.

The obtained results are shown in Table 6.3, where the first two columns correspond to the executions using Chuffed, the next two columns correspond to the executions using

Instance	Chuffed		Gecode		Gecode (rep)	
	time (s)	nodes	time (s)	nodes	time(s)	nodes
$i6$-11	6.03	106'187	408.65	28'751'319	18.54	1'252'774
$i7$-9	2.29	24'899	16.57	1'013'254	1.73	69'932
$i9$-11	68.29	348'044	1'980.19	60'458'164	50.23	1'294'069

Table 6.3: Solving the Radiation problem using Gecode, Chuffed, and Gecode with search replaying.

Gecode, and the last two columns correspond to Gecode replaying Chuffed's search. The columns show the solving time in seconds and the total number of nodes in the search tree.

Indeed, the results are consistent with those from the MiniZinc Challenge: they show that Chuffed is significantly faster on this problem than Gecode, even when no optimality proof is necessary. However, when Gecode is executed by replaying Chuffed's search, its executions are at least one order of magnitude faster than the original ones, where Gecode performed its own search. This fact demonstrates that once the difference in searches is eliminated, the performance of Gecode is much closer to that of Chuffed on this problem. We can thus conclude that the reason for Chuffed outperforming Gecode on this problem is mainly due to the differences in their searches, rather than to the advantages given by no-good learning propagation.

This case study will be discussed further in the next chapter, which is dedicated to the analysis of learning solvers.

6.5 Related Work

The idea of replaying search was developed simultaneously with (and independently of) that of [Van Cauwelaert et al., 2015], who use a similar method to compare propagators. Similar to my work, the authors recognise the importance of replaying for making comparison of executions fair when dynamic search is employed. Their approach for replaying is, however, different to what I propose in this thesis. Since the work in [Van Cauwelaert et al., 2015] was done in isolation to any profiling framework, the authors had to instrument a solver for the purpose of recording, rather than recover it from the search tree as proposed in my work. Further, the authors propose a different protocol for replaying. In particular, the protocol represents each node as a triple $\langle b, c, d \rangle$ where b, c and d denote the branching decision, the number of children, and the number of descendants, respectively. When the replaying execution arrives at a failure, but more search is expected by the search log, d node entries can be skipped, and the next node entry in the log will correctly correspond to the next node in replaying execution.

This is different from my approach, where I use node identifiers for the same purpose. Note that Van Cauwelaert et al. do not explore the opposite case wherein the replaying execution is weaker, and, thus, requires more search than is specified in the search log (and, in fact, the authors state it as a limitation of their methodology). My approach, on the other hand, supports replaying any execution regardless of the propagation used. In fact, the ability to replay an execution with stronger propagation is central for the methodology of analysing learning solvers, which I present in the following chapter.

6.6 Summary

In this chapter I presented two techniques that help modellers compare the effects of their model modifications: *search merging* and *search replaying*. Search merging refers to the merging of the search trees, and it can be used to pinpoint where two executions differ in terms of their search. It is mostly effective when a fixed search strategy is employed by both executions, or when the dynamic search strategy does not make the searches diverge too soon in the search tree. In the former case, search merging will pinpoint where one execution required less search compared to the other (for example, due to the difference in propagation strength). In the latter case, search merging will pinpoint the nodes in the search tree where the two executions started to diverge in terms of search. This technique scales well to large executions, but requires the executions to be relatively similar, particularly near the roots.

The second technique – search replaying – allows users to record search decisions in one execution and use them to drive the search in another execution, thus making the searches in both executions match. Since the effect of the search strategy is eliminated, this can be useful to evaluate the effect of propagation in the two executions. The two techniques, search merging and search replaying, work hand in hand, as the latter makes the merging work, even when a dynamic search strategy is employed in one of the executions.

Chapter 7

Analysis of Learning Solvers

A new generation of efficient constraint solvers has emerged in recent years that can learn from past failures to arrive at solutions faster. These solvers are based on the Lazy Clause Generation (LCG) technique [Ohrimenko et al., 2009], which combines the strengths of Constraint Programming and SAT solving, and often results in dramatic reductions of the search space, allowing the solver to solve hard constraint problems faster. Such solvers, referred to as *learning* solvers, are indeed the state of the art for solving a number of combinatorial problems, including Resource Constrained Project Scheduling Problems [Schutt et al., 2009] and the Carpet Cutting Problem [Schutt et al., 2011]. Further, they consistently exhibit better performance than traditional Constraint Programming solvers for a large number of problems in the annual MiniZinc Challenge [Stuckey et al., 2014].

However, for some problems, learning solvers seem to be unable to benefit from the learning and perform poorly compared to non-learning solvers. The reasons for these differences in behaviour are not well understood in practice, as learning solvers are even more complex than traditional CP solvers. Thus, it is not yet clear to the research community under what circumstances learning solvers are better, or even how to identify when or why a learning solver is performing poorly or not.

To be able to better understand the behaviour of learning solvers, I designed a methodology for studying the execution of such solvers. This chapter presents this methodology and demonstrates through case studies that analysing the execution of a learning solver in this way can be useful, not only to better understand its behaviour — opening what is typically a black box — but also to infer modifications to the original constraint model that can improve the performance for both learning and non-learning solvers.

The chapter first (Section 7.1) briefly describes the technology that underlies learning solvers. Section 7.2 presents the methodology for analysing learning solvers. Section 7.3 briefly discusses the implementation of this methodology in CP-Profiler. This is followed by two case studies that demonstrate the effectiveness of this methodology for improving models. Finally, Section 7.5 draws conclusions for this chapter and discusses potential future work.

7.1 Background on No-Good Learning

Learning solvers extend CP solvers by instrumenting their propagators to explain the domain changes caused by propagation. In the context of Lazy Clause Generation, the domain changes are represented by literals, that is, Boolean expressions in the form of *equalities* ($x = d$ for $d \in D(x)$), *disequalities* ($x \neq d$) or *inequalities* ($x \geq d$ or $x \leq d$). An

explanation for literal ℓ is $S \rightarrow \ell$, where S is a set of literals. For example, consider the constraint $x \neq y$. If variable x is set to 5, the propagator associated to $x \neq y$ will infer $y \neq 5$. Thus, a learning solver will instrument the propagation to produce the explanation $\{x = 5\} \rightarrow y \neq 5$. Explanations make the reasons behind constraint propagation explicit and can be used later when a failure occurs.

In learning solvers, each new literal inferred by a propagator is recorded in a stack in the order it was generated and attached to its explanation. Decision literals, that is, the literals associated with branching decisions, are also added to the stack and marked as such. The *decision level* for any literal is the number of decision literals pushed in the stack before it. The literals on the stack form the vertices of the implication graph, which is described in an example below.

When encountering a failure, learning solvers generate a *no-good* that explains the failure, i.e., a conjunction of literals that cannot be extended to a solution. Given an implication graph, a no-good is computed by starting with the direct cause of the failure, and then eliminating literals by replacing them with their explanations until only one literal at the current decision level remains. A no-good obtained this way is a 1UIP (First Unique Implication Point [Marques Silva et al., 1996]) no-good. While other ways of generating no-goods are possible, the resulting no-goods generally do not perform as well as the 1UIP no-goods, and they are not used in practice.

Example 7.1.1. Consider the *free pizza* problem from the MiniZinc Challenge 2015. In this problem a customer is to obtain a number of pizzas by either paying for them or by using vouchers to get them for free. Each voucher is represented by a pair of numbers a/b, indicating that the voucher allows customers to get b number of pizzas for free, as long as they pay for a number of pizzas, and none of the b pizzas are more expensive than any of the a pizzas. Given a customer who has m such vouchers and wants n pizzas (the prices of the pizzas are specified separately), the aim is to minimise the total price paid for the n pizzas. The model that was used in the annual MiniZinc Challenge (denoted as `freepizza`) is shown in Listing 7.1.

The first 9 lines introduce the problem parameters. Lines 1 and 5 introduce n and m, respectively, while line 3 introduces an array to store the price of each pizza. Lines 7 and 8 introduce two arrays for vouchers, `buy` and `free`, where `buy[i]` and `free[i]` represent the parameters a and b for the ith voucher, respectively. Finally, line 9 introduces the `total` parameter, which is computed as the collective price of all pizzas, and acts as the upper bound on the amount of money that could be spent.

Lines 13 through 15 introduce the decision variables. Line 13 defines the array of variables `how`, indicating for every pizza how it is obtained. More precisely, `how[p]` holds value:

- `v`, if pizza `p` was obtained for free thanks to voucher `v`;
- `0`, if `p` was paid for and not used in any voucher;
- `-v`, if `p` was paid for and used to get free pizzas with voucher `v`.

Line 14 introduces the array of variables `used`, indicating for every voucher whether the voucher is used. Line 15 introduces the objective, and defines it as the total price paid for the pizzas.

The lines 17 through 28 introduce the constraints in the problem. The constraint in lines 17 through 19 states that if voucher `v` is used (`used[v]` holds), then the total number of pizzas bought and assigned to `v` must be greater than or equal to `buy[v]`, the

```
1  int: n;                                  % number of pizzas wanted
2  set of int: PIZZA = 1..n;
3  array[PIZZA]   of int: price;            % price of each pizza
4
5  int: m;                                  % number of vouchers
6  set of int: VOUCHER = 1..m;
7  array[VOUCHER] of int: buy;              % buy this many to use voucher
8  array[VOUCHER] of int: free;             % get this many free
9  int: total = sum(price);
10
11 set of int: ASSIGN = -m .. m;
12
13 array[PIZZA] of var ASSIGN: how;
14 array[VOUCHER] of var bool: used;
15 var 0..total: objective = sum(p in PIZZA)((how[p] <= 0)*price[p]);
16
17 constraint forall(v in VOUCHER) (
18   used[v]<->sum(p in PIZZA)(how[p]=-v) >= buy[v]
19 );
20 constraint forall(v in VOUCHER) (
21   sum(p in PIZZA)(how[p]=-v) <= used[v]*buy[v]
22 );
23 constraint forall(v in VOUCHER) (
24   sum(p in PIZZA)(how[p]=v) <= used[v]*free[v]
25 );
26 constraint forall(p1, p2 in PIZZA)(
27   (how[p1] < how[p2] /\ how[p1]= -how[p2]) -> price[p2] <= price[p1]
28 );
29
30 solve :: int_search(how, input_order, indomain_min, complete) minimize objective;
```

Listing 7.1: Free pizza original model.

number of pizzas required by the voucher. The constraint in lines 20 through 22 states similar information but in the opposite direction: the total number of pizzas bought and assigned to voucher `v` must be less than or equal to `used[v]*buy[v]`. Together, the two constraints make the total number of pizzas bought for voucher `v` to be equal to `buy[v]` if the voucher is used. The constraint in lines 23 through 25 states that the total number of free pizzas obtained thanks to voucher `v` must be smaller than or equal to the number of free pizzas allowed by `v` if the voucher is used, and `0` otherwise. The last constraint in lines 26 through 28 states that if two pizzas `p1` and `p2` are assigned to the same voucher, and `p2` is obtained for free while `p1` is purchased (given `how[p1] < how[p2]` and `how[p1] = -how[p2]`), then the price of `p2` must be lower than or equal to that of `p1`. Finally, the solve item in line 30 states that the goal is to minimise the objective, and it specifies the search strategy that should be employed by a solver: variables from the `how` array are to be assigned in order from first to last, trying smallest values from their domains first.

Figure 7.1 shows a search tree for the execution of the model using Chuffed [Chu, 2011] with the following input data:

```
n = 5;
price = [17, 98, 76, 36, 69];
m = 8;
buy = [4, 4, 1, 4, 2, 1, 1, 3];
free = [2, 4, 1, 1, 4, 2, 3, 3];
```

The third branch of the tree shows a restart after bound 259 has been established for the `objective`. Note that during the MiniZinc compilation process, variable names are modified and, thus, variables `how_i` and `used_j` in the tree correspond to variables `how[i]` and `used[j]` in the model, respectively.

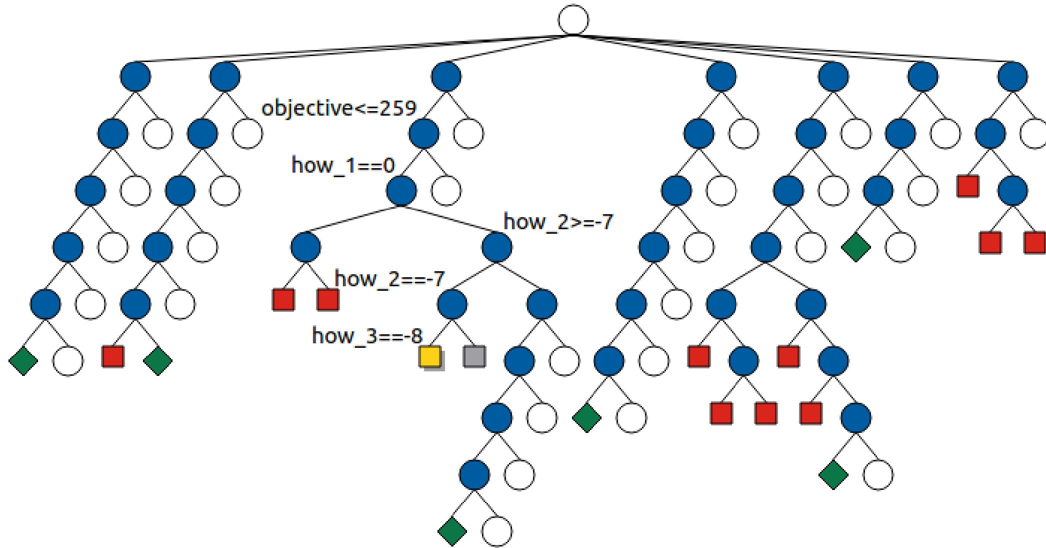

Figure 7.1: Search tree for `freepizza` using Chuffed. The path to the highlighted node is labelled.

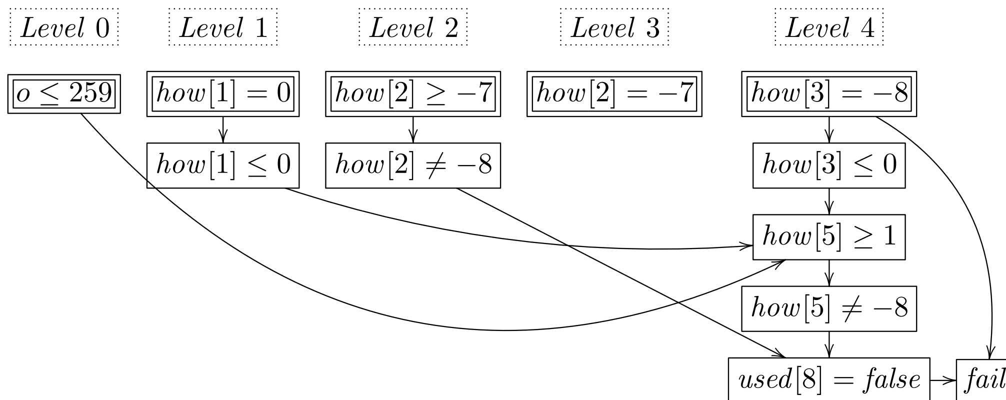

Figure 7.2: Part of the implication graph for `freepizza`. Decision literals are double boxed.

Figure 7.2 shows part of the implication graph used to derive a no-good after the failure caused by search decision `how[3]=-8` (marked as yellow in Figure 7.1). In this figure, literals are outlined with rectangular borders and arranged in columns, which correspond to different decision levels. There is one decision literal per decision level, and it is depicted with double borders. Arrows between literals show implication relationships.

The initial no-good associated with this failure is {`how[3]=-8`, `used[8]=false`}. Since the no-good has two literals belonging to the current decision level (level 4), the last literal is reduced obtaining {`how[3]=-8`, `how[2]`$\neq$`-8`, `how[5]`$\neq$`-8`}. This set still has two literals belonging to the current decision level, so literal `how[5]`$\neq$`-8` is reduced to obtain {`how[3]=-8`, `how[2]`$\neq$`-8`, `how[5]`$\geq$`1`}. This reduction process continues until only one literal remains at the current decision level, yielding the 1UIP no-good {`objective`$\leq$`259`, `how[1]`$\leq$`0`, `how[2]`$\neq$`-8`, `how[3]=-8`}.

Once a no-good N is obtained, its negation ($\neg N$, also called a *learnt clause*) is added as a clausal propagator. In other words, the learnt clause is treated as a regular model constraint, participating in propagation in every node, and thus preventing the same failure

from recurring. In this case, the learnt clause is {`objective>259`, `how[1]>0`, `how[2]=-8`, `how[3]≠-8`}, which is interpreted as the disjunction of its literals.

The search then backtracks to the decision level of the second deepest literal in the no-good, where it applies the newly learnt clause. Importantly, if the deepest latest literal is not from the immediately preceding decision level, the search performs a *backjump*, skipping decisions that were unrelated to the failure. In this case, the search backjumps to level 2, as the no-good has no literal at level 3. This jump indicates that the decision at level 3 (`how[2]=-7`) is unrelated to the failure.

Thus, the impact that a no-good makes on the process of solving is twofold:

- It strengthens the propagation due to the additional clause, potentially allowing subsequent failures to be identified earlier. If so, we say that the no-good and its corresponding learnt clause *contribute* to the failure.
- It creates the possibility of altering the search order, potentially yielding a more favourable one due to backjumping.

Note, however, that the benefit of the additional propagation due to the learnt clause can be outweighed by the cost of performing the extra propagation call. Therefore, modern learning solvers have to make decisions as to which learnt clauses should be kept and which of them should be discarded.

While identifying learnt clauses with high impact is important for solvers, it can also be valuable information for the modeller. In the remainder of this chapter I present a methodology for determining which learnt clauses have the most impact on the execution, and demonstrate how this information can be helpful for model improvement.

7.2 The Methodology

In order to analyse why learning is beneficial for a given model, let us compare two executions of the same instance of that model: one execution with learning and another execution without learning. Since it is important to ensure that the only difference in the two executions is entirely due to the benefits of learning, the two solvers used for this comparison (learning and non-learning) should employ similar propagation algorithms. Ideally, the two executions would be obtained by the same learning solver that additionally supports a traditional backtracking search with no learning.

Recall that learning solvers receive benefits from the clauses they learn, which might help to (a) determine a failure earlier in the search, and/or (b) backjump over some unexplored nodes, altering the search order. This methodology is focused on studying the effect that learnt clauses have on the propagation (a), and uses the search replaying technique (presented in the previous chapter) to separate the effect of different searches, including the effect of backjumping (b).

Following the methodology for search replaying, the profiler first records the search decisions taken by the learning solver in solving a given model instance, and saves the resulting search log. The search log is then used for replaying the same instance, this time without learning. Figure 7.3 shows search trees of two executions, obtained following the steps above.

As discussed, the main differences between such two executions come from the clauses learnt by the learning solver. To pinpoint the exact areas in the trees where the difference

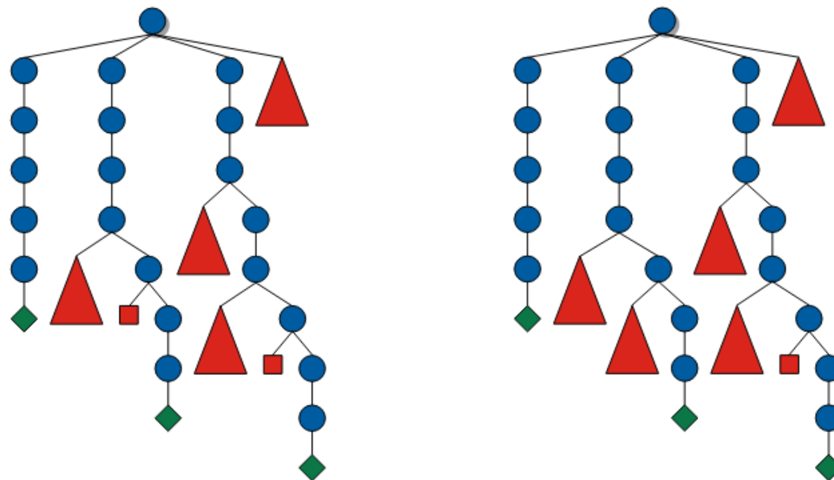

Figure 7.3: Model executed with learning (left) and the resulting search is replayed without learning (right).

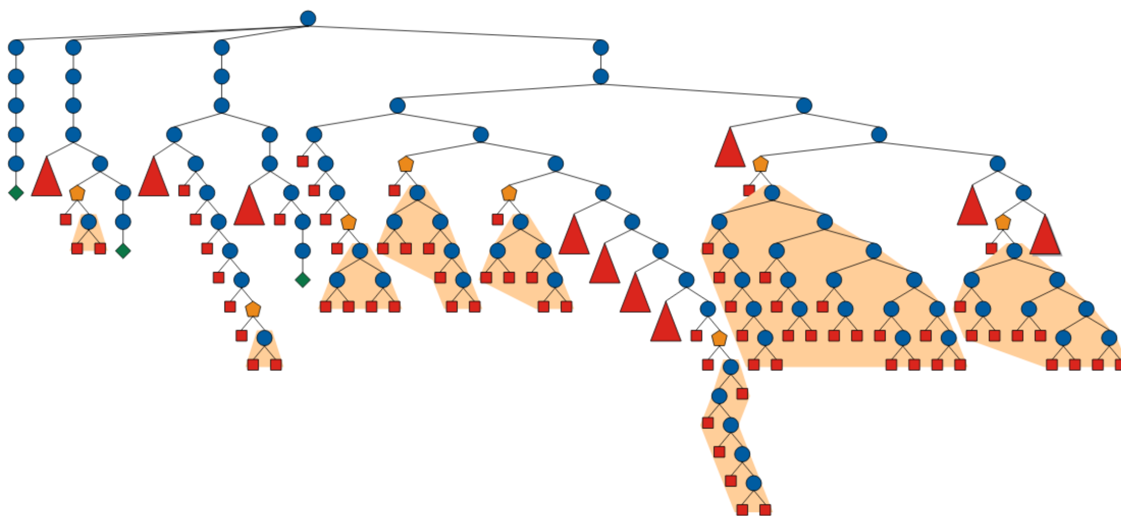

Figure 7.4: Two trees are merged to pinpoint the effect of no-goods.

in propagation due to no-goods made an impact, the two trees are merged using the search merging technique presented in the previous chapter. Recall that search merging shows the differences by means of pentagons, which display two subtrees as their children – one per search tree at the corresponding point of divergence. Note that in this methodology the subtree corresponding to the learning execution will always[1] contain a single failure node, while the corresponding non-learning subtree will contain more than one node. Figure 7.4 shows the result of merging the two trees from Figure 7.3 with eight such pentagons. Note that the trees used in this demonstration are quite small (containing less than 200 nodes each), and the number of pentagons is much larger for real-world models.

The difference in the number of nodes between the two subtrees under a pentagon, which equals the number of nodes in the non-learning subtree minus one, is referred to as the *search reduction*. Since the difference in the two executions is only due to the additional propagation caused by the learnt clauses, it is reasonable to attribute the search reduction in a given pentagon to the clauses that "contributed" to the early failure in the learning

[1] Provided that the difference in propagation is entirely due to no-goods.

execution. Recall that each no-good discovered during search is recorded as its negation – a learnt clause. This clause participates in propagation, and thus can be part of the explanation for a future failure. In that case, the no-good is said to *contribute* to that failure.

To obtain the information regarding contributing no-goods, the learning solver requires additional instrumentation. In particular, in addition to the structure of the search tree normally required by CP-Profiler, the solver needs be modified to send the following information for every failure node:

- the no-good generated at that node;
- a list of the previously recorded no-goods that contributed to that failure.

More often than not, the list of contributing no-goods will contain multiple items. When multiple no-goods collectively contribute to a failure, each of them are thought of as *partially* contributing to the failure. For simplicity, in this case the amount of search reduction is split equally between each of the contributing no-goods, allowing us to estimate individual contributions of the no-goods.

In addition, it is not unusual for a single no-good to (partially) contribute to multiple pentagons, that is, to different search reductions. In this methodology, the individual contributions of a given no-good are aggregated across the entire execution to obtain the total search reduction for that no-good. The no-goods can then be sorted by the total search reduction producing a ranking that reveals the "top" no-goods, that is, the no-goods that contribute the most to search reduction.

7.3 Implementation

Once we have this information, we can extend CP-Profiler to measure the impact of the no-goods by comparing the learning execution with a similar execution without learning, and examining the points at which their behaviours diverge.

For my studies I have chosen Chuffed as the learning solver, as it represents the state of the art in LCG, and its source code is readily available on-line, making it easy to instrument. Although it is possible to run Chuffed in a non-learning mode, I have chosen Gecode [Schulte et al., 2016] – an efficient backtracking CP solver – as Chuffed's non-learning counterpart, since its instrumentation already supported replaying. This choice should not, however, lead to major inaccuracies in the methodology (in associating no-goods with search reduction) since, for most constraints, Gecode implements the same or stronger level of propagation as Chuffed.

On the profiler side, I modified the search merging procedure to additionally identify all cases (pentagons) where a single failure node in the learning execution corresponds to a larger subtree in the non-learning execution. Further, the profiler aggregates all pentagons as discussed in Section 7.2, and displays the results in a table form (see Table 7.1 for an example).

7.4 Case Studies

With all the necessary implementation completed, I used the new methodology to compare Chuffed's and Gecode's behaviour on two different problems. The following section

Rank	Activity	Reduced Search	Clause
1	159	3425	how[1] = -1 how[2] = -1 how[3] = -1 how[4] = -1 how[5] = -1 how[1] = -2 how[2] = -2 how[3] = -2 how[4] = -2 how[5] = -2 how[6] $\leq$ 0 how[6] $\geq$ 3
2	176	2068	how[7] $\leq$ 2 how[7] $\geq$ 4 how[1] $\neq$ -3 how[1] $\geq$ -2
3	34	1712	how[4] $\neq$ 3 how[1] = -3 how[2] = -3 how[3] = -3 how[4] = -3
4	8	1636	how[5] $\neq$ -3 how[3] $\neq$ 3
5	8	1636	how[8] $\neq$ -3 how[3] $\neq$ 3
6	8	1636	how[9] $\neq$ -3 how[3] $\neq$ 3
7	8	1636	how[10] $\neq$ -3 how[3] $\neq$ 3
8	143	1489	how[6] $\leq$ 2 how[6] $\geq$ 4 how[1] $\neq$ -3 how[1] $\geq$ -2
9	25	1404	how[5] $\neq$ -3 how[4] $\neq$ 3 how[4] $\leq$ 2
10	24	1403	how[10] $\neq$ -3 how[4] $\neq$ 3

Table 7.1: Most effective learnt clauses in the Free Pizza problem.

presents these case studies and demonstrates how applying this methodology can lead to insights that result in effective model transformations.

7.4.1 First Case Study: the Free Pizza Problem

Consider an instance of the free pizza problem and its model introduced in Section 7.1 with the following input data:

```
n = 10;
price = [70, 10, 60, 65, 30, 100, 75, 40, 45, 20];
m = 4;
buy = [1, 2, 3, 3];
free = [1, 1, 2, 1];
```

Following the methodology introduced in the previous section, I executed the model for the data shown above using Chuffed, replayed its search using Gecode, merged the resulting execution trees, and ranked the learnt clauses in terms of their associated reduced search. Table 7.1 shows the top 10 clauses together with information regarding their *activity* (that is, the number of times they contributed to a failure) and number of search nodes they reduced. Note that the total number of nodes in the execution with Gecode is $77,100$, therefore the nodes reduced by these top clauses represent a significant portion of the search.

I first concentrated on some of the shorter clauses, like `how[5]` $\neq$ `-3, how[3]` $\neq$ `3`, which is ranked number four. It states that pizza 3 cannot be obtained for free with voucher 3 by buying pizza 5 and assigning it to this voucher. This is a direct consequence of the constraint starting in line 26 of the model and the fact that pizza 3 (cost 60) is more expensive than pizza 5 (cost 30). This observation helped me understand the longer clauses and realise that some of them were weaker (and more complex) than I expected.

Consider, for example, the top clause, which captures information about the relationship between obtaining pizza 6 with vouchers 1 or 2 (as `how[6]` $\leq$ `0, how[6]` $\geq$ `3` indicates that `how[6]` cannot be 1 or 2), and buying pizzas 1, 2, 3, 4, and 5, assigning them to these vouchers. Consulting the input data, I realised that pizza 6 is more expensive than any other pizza and, thus, it cannot be obtained for free with any voucher (not just 1 and 2) and must be purchased. Therefore, the clause could be strengthened by simply expressing it as `how[6]` $\leq$ `0`. It was to a certain degree surprising to realise that this simple fact (and its cousin: the cheapest pizza cannot be used to obtain any other pizza for free) was not being learnt by the solver. This interesting insight further motivates the need for

studying the learnt clauses to better understand the information learnt (or not learnt) by the solver.

While the learnings (how[6]≤0 and how[2]≥0) were instance specific, the same ideas can be stated in a generic way and used as redundant constraints in the model. Here is one way to do it (note that != represents disequality):

```
% the most expensive pizza can never be bought with a voucher
constraint forall(p in PIZZA) (
  if forall(o in PIZZA where o != p) (price[p] > price[o])
       then how[p] <= 0 else true endif
);
% the cheapest pizza can never be used with a voucher
constraint forall(p in PIZZA) (
  if forall(o in PIZZA where o != p) (price[p] < price[o])
       then how[p] >= 0 else true endif
);
```

The first constraint requires that if pizza p is the (single) most expensive one, it cannot be obtained for free, and thus the value $how[p]$ cannot be positive. The second constraint requires that if pizza p is the cheapest, it cannot (should not) be used with a voucher, and thus the value $how[p]$ cannot be negative. It should be noted, however, that these redundant constraints will be vacuous if there is no single most expensive/cheapest pizza.

I was also surprised to realise that many of the top clauses (4 to 7) allowed Chuffed to avoid exploring significant amounts of nodes when compared to Gecode, despite the fact that these clauses are direct consequences of a single constraint (the one starting in line 26). One would expect Gecode to also avoid exploring those nodes by directly propagating the constraint. This observation indicates that the constraint was not propagating as well as expected. Focusing on this constraint, I realised that the how[p1] < how[p2] part of the constraint could be replaced by how[p2] > 0, since it only needs to ensure that pizza p2 is free. This is clearly stronger information and connects with the way the objective function is expressed, thus allowing stronger propagation when the objective is bounded. The modified constraint can be expressed as follows (with the modified part shown in green):

```
constraint forall(p1,p2 in PIZZA) (
  (how[p2] > 0 /\ how[p1] = -how[p2]) -> price[p2] <= price[p1]);
```

To assess the modifications made to the model, I randomly generated 100 input data files and measured the solving time using Gecode and Chuffed with fixed search (as specified in the original model) and with free search. Aiming to solve all instances to completion within a reasonable amount of time for both solvers (not too easy, not too difficult), I generated the input data using between 2 to 10 vouchers for Gecode and 6 to 10 vouchers for Chuffed, each voucher requiring 1 to 4 pizzas to be purchased and allowing customers to get 1 to 4 pizzas for free. The number of pizzas to obtain was between 9 to 10 for Gecode with fixed search, 12 to 13 for Gecode with free search, and 15 to 16 for Chuffed. Any data file that for a given solver and search was solved in under one second using all models was excluded from the final results. As a result, 74 and 80 data files were used for Gecode with fixed and free search, respectively, and 98 and 95 data files were used for Chuffed, respectively.

Aggregated results for these data files are shown in Table 7.2, which compares the performance of the two solvers in terms of execution time and number of failures using three models: the original one, the one obtained by adding the two redundant constraints, and the final one obtained by also modifying the constraint starting in line 26 as indicated above. For a given combination of search and solver, the table presents the results of

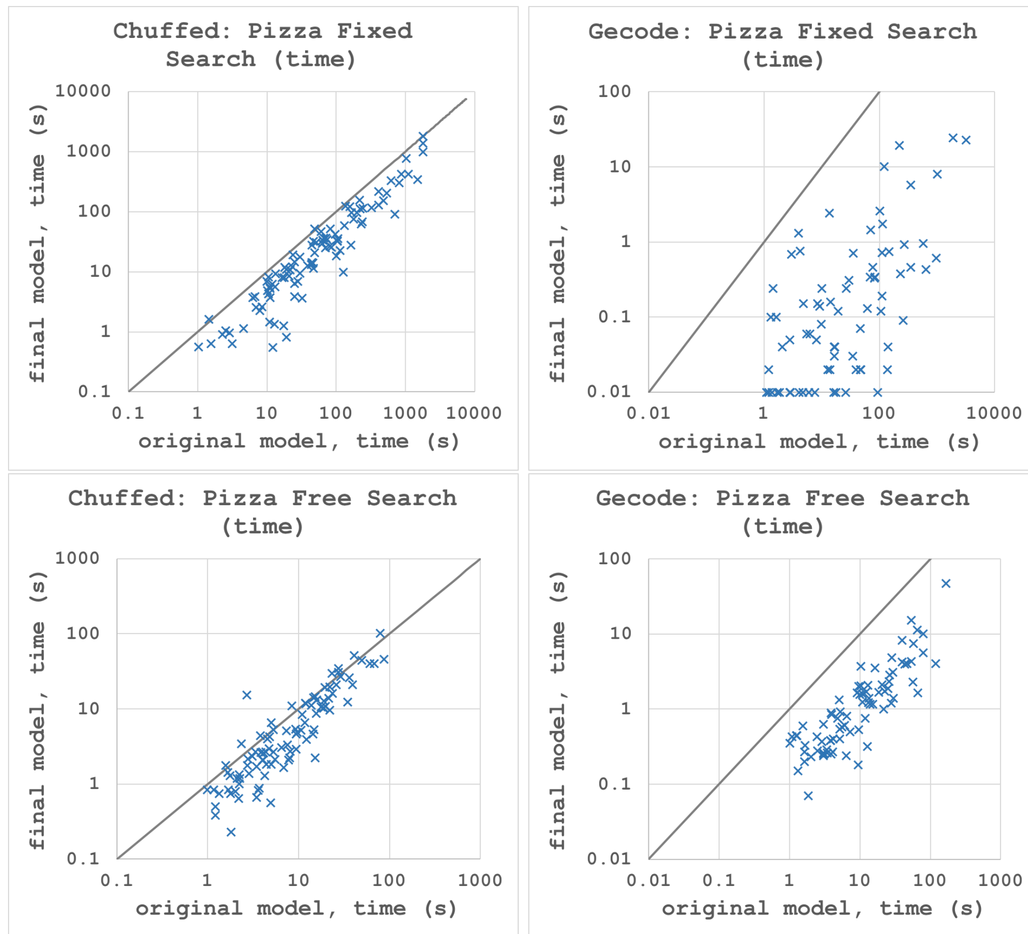

Figure 7.5: Execution time of original and improved pizza models (logarithmic scale).

pairwise comparison of the three models in terms of execution time and the number of nodes in their search trees in terms of geometric mean and median of the corresponding ratios. For example, the first row compares the original model to the model with the redundant constraint, both executed using Chuffed with fixed search. Clearly, the modifications improved the performance of both solvers (as all numbers are below 1), with the results being particularly significant for Gecode with fixed search, where the difference reaches two orders of magnitude.

Detailed results are presented in Figure 7.5, where each dot shows the solving time for a given data file using the original and the final models in the horizontal and vertical axes, respectively. The scatter plots show that the vast majority of the instances lie below the identity line, and thus, that the final model consistently performs better than the original one.

7.4.2 Second Case Study: the Golomb Ruler Problem

Recall the Golomb ruler problem introduced in Chapter 2, which involves finding a minimal ruler with n integer marks, such that no two pairs of marks are the same distance apart. I use the model from the MiniZinc benchmarks set[2] for this problem, which was introduced also in Chapter 2.

[2]https://github.com/MiniZinc/minizinc-benchmarks

		Models Ratio	GeoMean(time)	Median(time)	GeoMean(fails)	Median(fails)
Fixed Search	Chuffed	redundant/original	0.4885	0.5497	0.5186	0.5737
		final/redundant	0.7746	0.7905	0.9159	0.9368
		final/original	0.3784	0.4152	0.9368	0.5199
	Gecode	redundant/original	0.0904	0.1250	0.0925	0.1234
		final/redundant	0.0569	0.0786	0.0435	0.0461
		final/original	0.0051	0.0056	0.0040	0.0042
Free Search	Chuffed	redundant/original	0.7039	0.7162	0.7625	0.7426
		final/redundant	0.8228	0.8070	0.8872	0.8944
		final/original	0.5791	0.5830	0.6765	0.6876
	Gecode	redundant/original	0.1526	0.1468	0.1493	0.1459
		final/redundant	0.7205	0.7330	0.7991	0.8104
		final/original	0.1100	0.1050	0.1193	0.1187

Table 7.2: Aggregate Results for Free Pizza over a set of random instances (relative).

	Rank	Activity	Reduced Search	Clause
Modified	1	2	55	mark[6] $\geq$ 19, mark[4] $\leq$ 14, mark[3] $\leq$ 13
	2	3	55	mark[6] $\geq$ 18, mark[4] $\leq$ 13, mark[3] $\leq$ 12
	3	3	41	(mark[8] - mark[6] $\leq$ 5), (mark[8] - mark[6] $\geq$ 7), (mark[10] $\geq$ 55), (mark[6] $\leq$ 41), (mark[10] - mark[8] $\leq$ 5)

Table 7.3: Most effective learnt clauses for Golomb Ruler (after the first modification).

This model is known to be difficult for learning solvers. Indeed, it can be shown that Gecode is consistently faster than Chuffed on this model (see "Original Model" in Table 7.5). Nonetheless, Chuffed requires fewer failures to solve the problem and, thus, I decided to examine Chuffed's behaviour to see if I could determine how to improve the model.

Using the methodology introduced earlier in this chapter, I executed the instance of this model with parameter $m = 10$ using Chuffed, replayed its search using Gecode, merged the resulting executions trees, and ranked the learnt clauses in terms of their associated reduced search. Table 7.4 shows the 5 most effective clauses learnt by Chuffed. All these clauses are of the form

```
(mark[i] >= n) /\ (mark[i+1] >= n+1) -> (mark[i+2] >= n+3),
```

for some `i` and `n`. (Note that a clause of the form $\{A, B, C\}$ can be interpreted as $\neg B \wedge \neg C \rightarrow A$.) This represents an implied constraint, which the solver is effectively rediscovering and explicitly adding to the execution. By manually analysing the problem, I confirmed it was correct to add the following redundant constraint:

```
mark[i] + 3 <= mark[i+2],
```

for all `i`. Clearly `mark[i+2]` is at least one more than `mark[i+1]`, which is at least one more than `mark[i]`. Thus, `mark[i+2]` and `mark[i]` are at least two apart and, if so, both intermediate differences must be one, which is forbidden.

	Rank	Activity	Reduced Search	Clause
Original	1	49	193	mark[6] $\geq$ 38, mark[5] $\leq$ 35, mark[4] $\leq$ 34
	2	6	170	mark[5] $\geq$ 18, mark[4] $\leq$ 15, mark[3] $\leq$ 14
	3	6	170	mark[5] $\geq$ 15, mark[4] $\leq$ 12, mark[3] $\leq$ 11
	4	5	168	mark[5] $\geq$ 22, mark[4] $\leq$ 19, mark[3] $\leq$ 18
	5	50	163	mark[6] $\geq$ 36, mark[5] $\leq$ 33, mark[4] $\leq$ 32

Table 7.4: Most effective learnt clauses for Golomb Ruler (before the first modification).

Once this redundant constraint is added to the model, I again applied the methodology for studying no-goods. The new top 3 learnt clauses are shown in Table 7.3. The first two follow the pattern:

```
(mark[i] >= n) /\ (mark[i+1] >= n+1) -> (mark[i+3] >= n+5).
```

The third clause illustrates a property of connected differences: if the difference between `mark[i]` and `mark[j]` is, say, 6, and the difference between `mark[j]` and `mark[k]` is at least 6, then the difference between `mark[i]` and `mark[k]` is at least 6+6+1=13. The 1 in the sum appears for the same reason as above.

Note that all these clauses refer to the idea that the distance between two marks that are k positions apart is equal to the sum of the inner distances, which are all different. As a result, this distance is at least as large as the sum of the arithmetic sequence of natural numbers, i.e., $\frac{k(k+1)}{2}$. In fact, by following this methodology I had rediscovered a redundant constraint for this problem, unknown to me, but discussed earlier in [Barták, 2005].

The inclusion of this redundant constraint in the model reduces the search effort required to solve it, both in number of nodes and in time (see “Improved Model” in Table 7.5). Interestingly, even the non-learning solver Gecode benefited from the constraint. This demonstrates how, even when learning solvers are not the strongest for a given problem, we can gain useful insights from examining the clauses they learn.

		Original Model		Improved Model	
	Size n	Time (s)	Number of Failures	Time (s)	Number of Failures
Chuffed	$n = 10$	1.99	20,912	1.49	19,343
	$n = 11$	72.36	307,957	54.25	288,071
	$n = 12$	616.2	2,329,959	512.63	2,254,206
Gecode	$n = 10$	0.74	23,463	0.57	19,928
	$n = 11$	15.81	374,886	12.09	321,419
	$n = 12$	147.00	3,002,474	117.83	2,656,663

Table 7.5: Golomb Ruler Results.

7.4.3 Third Case Study: the Radiation Problem

In Section 6.4.2 I introduced the Intensity-Modulated Radiation Therapy problem and showed, using the search replaying technique, that the reason for Chuffed’s good performance while finding the first (and optimal) solution to this problem is mainly due to its search.

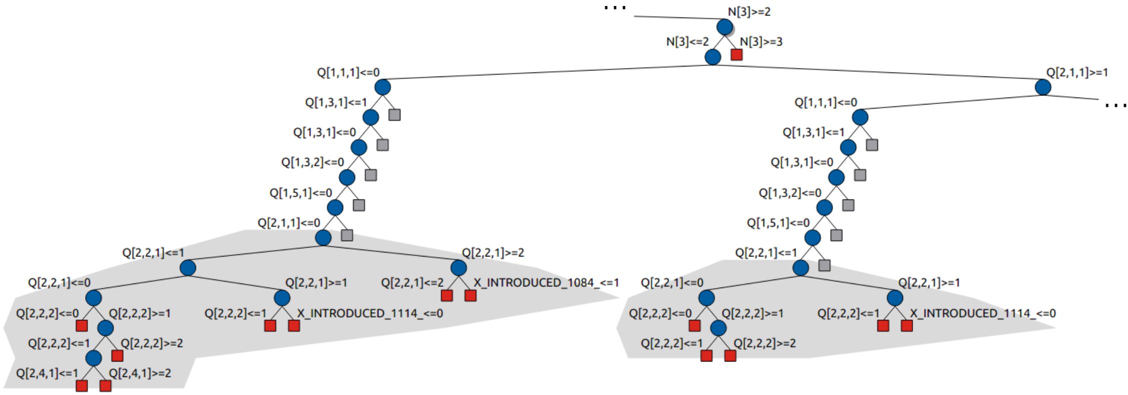

Figure 7.6: Chuffed backjumps in the Radiation problem.

Closer examination of the Chuffed's search tree (shown in Figure 7.6 for instance *i*7-9) reveals that the search frequently backjumps (grey square nodes indicate the parts of the search not explored due to backjumping). Note that this tree shows three distinct regions of search. The first region, represented by the top nodes, corresponds to searching on variables N_i. Below this point on the tree, all variables N_i have been assigned, as required by the search strategy. The search then proceeds to the second region, which corresponds to searching on variables $Q_{1,j,k}$ associated with the first row in the matrix. These variables seem underconstrained, and the search quickly assigns all of them without the need to backtrack, arriving at the third region (represented by two subtrees highlighted with grey background). This region corresponds to searching on variables $Q_{2,j,k}$ associated with the second row in the matrix. Note that this region requires more search as indicated by backtracks.

Unable to find satisfying assignments to the variables of the second row, the search does not backtrack to the second region, but rather backjumps directly to the first region. This observation suggests that variables of the first row do not participate in the failure caused by assigning variables from the second row. Analysing the model more closely reveals that variables that correspond to different rows (such as $Q_{1,j,k}$ and $Q_{2,j,k}$) become independent once all variables N_i have been assigned. The fact that there is no satisfying assignment to variables $Q_{2,j,k}$ for a given assignment to N_i, holds for any other assignments to $Q_{1,j,k}$. Fortunately, Chuffed is able to identify this and thus leaves regions two and three rather quickly to change the value of N_i that is incompatible with the second row assignments.

Gecode, on the other hand, is unable to backjump, and thus it would repeat the same fruitless search for all satisfying assignments in stage two. This fact can be confirmed using the similar subtree analysis introduced in Section 5. Indeed, this analysis reveals a great deal of identical subtrees. For example, the search represented by the subtree shown in Figure 7.7, which corresponds to searching on variables $Q_{2,j,k}$ (second row in the matrix), is repeated 240 times throughout the execution. The lantern tree visualisation in Figure 7.8 depicts a fraction of Gecode's search tree, where the subtrees representing the search from Figure 7.7 are shown as lantern nodes. The analysis additionally reports that the difference in search decisions between the first two of these identical subtrees is only in the value of a single variable: $Q_{1,3,1}$ (first row in the matrix).

The main reason for Chuffed's better performance on this problem compared to Gecode is, indeed, its search due to backjumping, which reinforces the conclusion made in the previous chapter. However, the benefit of extra propagation caused by the no-goods in

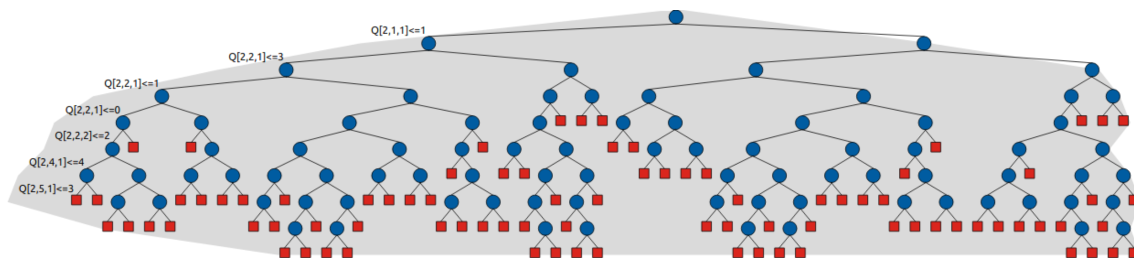

Figure 7.7: Gecode's subtree repeated 240 times in the Radiation problem.

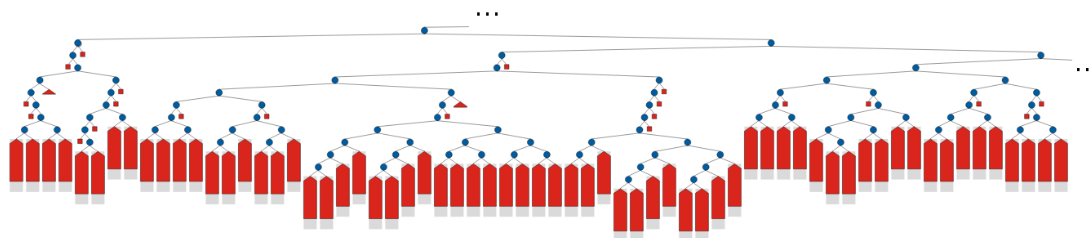

Figure 7.8: Gecode's tree for the Radiation problem with similar subtrees highlighted.

Chuffed's execution is still significant, as it explores less nodes than Gecode (see Section 6.4.2), even when Gecode follows the same search decisions and performs the same backjumps.

Although the performance of Gecode is likely to benefit more from a different search than from a better propagation by, for example, redundant constraints identified by the no-good analysis, I decided to apply this analysis to the model. The following data was used for the analysis (corresponds to the $i7$-9 instance from the MiniZinc Challenge 2015):

```
m = 9;  n = 9;  % rows and columns
Intensity = [| 4,  8, 11,  2,  5,  7,  1, 10,  4 |
              11,  4,  4,  5,  1,  8,  9,  3,  9 |
               2,  9,  6,  2,  4,  1,  5,  2,  6 |
              11,  9,  8,  9,  3,  2, 11,  6,  7 |
               2,  8, 11,  2, 10,  5,  5,  4,  5 |
               5,  9,  8,  1,  6,  3,  5, 11,  5 |
               ...
               7,  1,  6, 10,  0,  8,  1,  0,  0 |];
```

Following the methodology introduced before, I executed the model for the Radiation problem for the data shown above using Chuffed, replayed it using Gecode, merged the resulting executions and ranked the learnt clauses in terms of their associated reduced search. Table 7.6 shows the 5 top learnt clauses obtained as the result of this analysis.

As in the previous case studies, I focused on the shortest clauses first. For example, the clause in the second row $(N_2 \geq 2) \vee (Q_{2,2,2} \leq 1) \vee (Q_{2,2,2} - Q_{2,1,2} \geq 2)$ is logically equivalent to the following implication: $(Q_{2,2,2} - Q_{2,1,2} < 2) \wedge (Q_{2,2,2} > 1) \Rightarrow (N_2 \geq 2)$. Recall that variable N_2 represents the number of shape matrices associated with beam-on-time 2, and variables $Q_{2,1,2}$ and $Q_{2,2,2}$ represent the number of shape matrices associated with beam-on-time 2 that expose cells $[2,1]$ and $[2,2]$, respectively. Clearly, N_2 must be at least as large as the largest $Q_{i,j,2}$ $(\forall i \in 1..m, j \in 1..n)$. (In fact, it appears that the literal $(Q_{2,2,2} - Q_{2,1,2} \geq 2)$ in the clause is redundant.) This observation can be expressed in the model as the following constraint:

```
constraint forall(b in BTimes, i in Rows, j in Columns) (N[b] >= Q[i,j,b]);
```

Rank	Activity	Reduced Search	Clause
1	50	53072	$(Q_{2,2,1} \neq 4)$ $(Q_{2,2,2} \geq 2)$ $(Q_{2,2,3} \geq 2)$ $(Q_{2,2,4} \geq 1)$ $(Q_{2,2,5} \geq 1)$ $(Q_{2,2,6} \geq 1)$ $(Q_{2,2,7} \geq 1)$ $(Q_{2,2,8} \geq 1)$
2	25	45079	$(N_2 \geq 2)$ $(Q_{2,2,2} \leq 1)$ $(Q_{2,2,2} - Q_{2,1,2} \geq 2)$
3	17	44796	$(N_2 \geq 3)$ $(Q_{2,2,2} \leq 2)$ $(Q_{2,2,2} - Q_{2,1,2} \geq 3)$
4	16	44471	$(N_2 \geq 2)$ $(Q_{1,3,2} \geq 2)$ $(Q_{1,4,2} \leq 1)(Q_{1,4,2} - Q_{1,3,2} \geq 2)$
5	33	43149	$(N_3 \geq 2)$ $(Q_{1,4,3} \leq 1)$ $(Q_{1,4,3} - Q_{1,3,3} \geq 2)$

Table 7.6: Most effective learnt clauses in `radiation`.

This constraint demands that for every matrix cell $[i, j]$ ($\forall i \in 1..m, j \in 1..n$) the total number N_b of shape matrices associated with beam-on-time b ($b \in Btimes$) is at least as large as the number of those shape matrices among the N_b that expose a given cell. I added this constraint to the model, and evaluated the augmented model by comparing its performance to that of the original model. The three instances used for the study are the same as those used in the MiniZinc Challenge. The results of the comparison in terms of the execution time and the number of nodes are shown in Table 7.7.

		Original Model		Augmented Model	
	Instance	Time (s)	Number of Failures	Time (s)	Number of Failures
Gecode	*i*6-11	408.65	28,751,319	352.06	25,896,142
	*i*7-9	17.28	1,013,473	14.83	896.536
	*i*9-11	1969.25	60,463,317	1827.96	55,837,438
Chuffed	*i*6-11	5.46	108,372	5.46	107,038
	*i*7-9	2.09	25,056	2.13	24,407
	*i*9-11	53.93	351,494	55.35	341,717

Table 7.7: Adding a redundant constraint to the model for the Radiation problem.

For Gecode, the results show around 10% reduction in time and in the number of nodes in the search trees for the augmented model. For Chuffed, on the other hand, the reduction in the number of nodes is insignificant ($1-2\%$), and it is sometimes outweighed by the cost of propagation the extra constraint, resulting in a small slow down.

Note, however, that not all top clauses identified by this analysis describe the same "insight" as the clause ranked number two, that was the focus of the above study. For example, close examination of the model reveals that the clause ranked number one relates to the constraint defined in line 20 of Listing 6.4, while the clause ranked number two relates to the constraint defined in line 23. This fact identifies an opportunity for future work: automatically evaluate the potential effectiveness of a redundant constraint based on, perhaps, the amount of search reduction achieved by its associated learnt clauses. In fact, running other instances of this model or performing different searches often reveals different "insights". For instance, the clause $(Q_{2,5,1} \geq 1) \vee (Q_{2,5,3} \geq 1) \vee (Q_{2,5,5} \geq 1)$ was identified during the analysis of the same instance executed only until the first solution was found. This clause states that there should be a matrix that exposes cell [2,4] for the duration of 1, 3 or 5. The rational for this restriction becomes clear upon the examination of the input data, which requires the amount of radiation received by cell [2,4] to add up to exactly 5 units – an odd number. Therefore, there needs to be at least one matrix with an odd duration time. In particular, for 5 this requires a matrix with the duration of 1, 3, or 5, and anything longer than 5 would result in the overexposure of the cell, which is exactly what the clause conveys. This observation can be expressed in the model as the following redundant constraint:

```
constraint
  forall(b in BTimes where b mod 2 = 1)
    (forall(i in Rows, j in Columns where Intensity[i, j] = b)
      (sum([Q[i,j,k] | k in 1..b where k mod 2 = 1]) > 0));
```

This constraint states that for every odd beam-on-time b and for every cell $[i, j]$ in the matrix there should be at least one shape matrix $Q_{i,j,k}$ with an odd beam-on-time k. Adding this constraint, however, results in only $2 - 3\%$ search reduction on average, and is again not worthwhile, especially since it results in longer propagation time in each node. Part of the reason this constraint is ineffective might be its instance-dependency, as it targets cells with a specific intensity requirement (although odd numbers are not uncommon, the constraint seems to be mainly effective for small numbers like 3 or 5). This fact identifies an opportunity for future work: to rank learnt clauses across multiple instances, so that the insights they represent are instance-independent.

7.5 Summary

This chapter presented a new methodology for analysing the execution of learning solvers. The methodology was applied to three case studies, involving three different constraint problems. For two problems, the methodology discovered redundant constraints that significantly reduced both the solving time and the amount of nodes explored. In the free pizza problem, the methodology additionally helped identify a constraint with poor performance, and thus showed what part of the model requires more attention. Applying a small modification to this constraint further improved the free pizza model.

Importantly, while the benefit of these modifications was greater for the non-learning solver, the learning solver also benefited. Note that, although the added redundant constraints are reminiscent of the no-goods (learnt clauses) in the learning solver, they are more general: they are stronger and are often effective across instances. Additionally, the Golomb ruler study demonstrated that this methodology can be useful even when the learning solver is outperformed by the traditional backtracking solver.

The case study of the Radiation problem also resulted in finding a redundant constraint. However, it only marginally improved the executions for Gecode and was not worthwhile for Chuffed, as its search reduction was outweighed the longer propagation time it caused.

Note that the current implementation of this methodology is at its early stage and can be improved further. For example, it currently requires the modeller to process the list of top no-goods manually to infer model improvements. One opportunity for future work is to automate this process by, for example, automatically generating redundant constraints. Another way to improve the analysis is to automatically link the top no-goods to the constraints in the model, thus making it easier for the modeller to determine which constraints require more attention.

Chapter 8

Conclusions

The profiling of Constraint Programming executions can help modellers improve their models, and thus allows them to solve hard combinatorial problems faster, obtaining better quality solutions. In this thesis I have explored the area of profiling in Constraint Programming and presented significant improvements to the profiling process. In particular, the thesis introduces several effective profiling techniques along with a new profiling system architecture that integrates and enables these techniques. This chapter discusses the conclusions to the thesis by recapitulating its contributions and identifying the potential future work that arises from them.

The CP-Profiler System

Many profiling techniques are most effective when they are used together, as a part of a profiling system. At the same time, the fundamental architectural decisions taken to design such systems determine the kind of profiling techniques that can be integrated. This thesis provides the following contributions regarding the profiling system design:

- This thesis identifies the requirements a profiling system for Constraint Programming must fulfil in order to be effective. In particular, it must be:
  - Capable of visualising search-trees.
  - Efficient, that is, it scales for large real world executions.
  - Solver-independent, that is, easy to integrate with new solvers by requiring only reasonably small changes to the solver's implementation.
  - Search-independent, that is, capable of supporting both simple search exploration paradigms and complex ones such as restart-based search, parallel search, and that of learning solvers.
  - Real-time, that is, it allows users to start analysing executions before they are finished.
  - Extensible, that is, it can adapt to new profiling techniques.

  Although many existing profiling systems have been successful to an extent, no single one of them is able to fulfill all the above requirements (with CP-Viz [Simonis et al., 2010] being the most effective existing system, only failing to fulfill the requirement of being real-time).

- This thesis presents an architecture and associated design decisions, including a protocol specification for solver-profiler communication, that fulfill the identified requirements for an effective profiling system.
- A prototype profiling system called CP-Profiler based on this architecture has been designed and made available open-source[1]. Despite its prototype status, CP-Profiler has been used both by my collaborators and externally. A few solvers (namely, Gecode, Chuffed, and Choco) have already been integrated into the system and, importantly, these solver integrations have been adopted (and extended) by their solver developers.

One possibility for future work is the integration of CP-Profiler into the MiniZinc IDE in order to maximise its utilisation and impact in the community. Additionally, it would be useful to integrate other popular solvers (e.g., Mistral-2.0 [Hebrard, 2008], Picat [Zhou et al., 2015], JaCoP [Kuchcinski et al., 2013]) into CP-Profiler.

Enhancing Search Tree Visualisation and Navigation

Studying an execution by examining its search tree is one of the most popular tasks performed when profiling an execution, and thus search tree visualisations are at the heart of most CP profiling systems. However, despite the many advancements in search visualisation techniques, the existing profiling systems are still not very effective at dealing with the common case of large search trees. This thesis provides the following contributions to the way large search trees are visualised:

- It presents a new search tree visualisation: *the lantern tree*, to reduce the scale of large trees without losing their general structure. The technique builds on the state-of-the-art search tree visualisation implemented by the Gecode solver, but improves the way subtrees are collapsed. In particular, this technique collapses subtrees into special "lantern" nodes that convey information about their underlying subtrees via their height. This collapsing is performed selectively: the user can specify the maximum number of nodes that the largest lantern node can contain, thus controlling the degree of detail in the resulting visualisation. Side by side comparison of the state of the art and the proposed techniques are provided, which demonstrate the advantages of the new technique.
- It studies the use of two unconventional techniques for visualising generic trees – pixel tree and icicle tree – in the context of search trees. In particular, this thesis demonstrates that even in a compressed form, which is necessary for large trees, pixel trees allow users to easily distinguish between different regions in the search, such as regions with many solutions or those corresponding to the proof of optimality. Additionally, unlike the traditional node-link visualisation, pixel trees clearly convey time progression, and thus they can be used to display statistical information that changes in time, clearly linking the information with the different regions in the search.

  Regarding icicle trees, the thesis demonstrates that they are easy to compress and, while they are not suited for depicting time progression, the relation of the visual elements in an icicle tree (horizontal rectangles) to the tree structure (nodes and subtrees) is clear even in a compressed form. This property allows users to colour-code individual nodes or subtrees based on different statistical information. For

[1] https://github.com/cp-profiler

example, colour-coding the nodes based on the branching decisions can show how variable assignment order changes during the execution.

- It demonstrates how the two unconventional techniques mentioned above can make it easier for users to navigate the traditional node-link visualisation: when selecting regions of interest, CP-Profiler will highlight the same regions in the traditional visualisation, where the search tree can be examined in more detail.

- The lantern tree visualisation and the two novel visualisations (pixel tree and icicle tree) have been implemented and integrated into CP-Profiler, and thus made available to the users of the Constraint Programming technology.

  In regard to the lantern tree visualisation, one opportunity for future work is to develop an algorithm to automatically identify the right level of detail to present to the user. In regard to the use of unconventional search tree visualisations, one opportunity for future work is to identify other statistical information that can be displayed using the pixel tree and the icicle tree to gain insights about the execution.

Finding Recurring Patterns in Executions

The search trees for large real-world problems often contain many thousands or even millions of nodes. While partial understanding of such large executions can be obtained through good "overview" visualisations like the ones contributed by this thesis, some helpful insights regarding the execution are often buried in detail, making it impractical to uncover them without the aid from automated analysis. Although the need for automated analysis is clear, there seems to be no such analysis available in the existing profiling systems. The contributions of this thesis in this regard are as follows:

- It presents *the similar subtree analysis* – a technique capable of finding repeated patterns in large executions, which often indicate the presence of inefficient search behaviours.

- It implements the similar subtree analysis and integrates it into CP-Profiler, additionally demonstrating that it scales to large executions.

- It presents a case study that demonstrates how the similar subtree analysis can be used to detect a symmetry present in the problem model. It also shows that eliminating the found symmetry leads to a more efficient model.

Although the similar subtree analysis has been demonstrated to be effective in its current form, further improvements are indeed possible. For example, this thesis only explores two criteria for similarity: by shape and by identity. While the former is less restrictive than the latter, they tend to find the same patterns. Developing a less restrictive criteria might be beneficial, as it would be able to find more patterns, and thus provide more insights to the user. Further, the automatic analysis of search tree structure seems to be a promising area in general. One opportunity for future work is the identification of common patterns that can be linked to inefficient behaviour, such as thrashing (where the solver repeatedly searches fruitless areas of the search space) or blind inference (where a great deal of computational effort is spent by the solver in producing information of little use).

Comparison of executions

Modifying the model, search strategy, or the selection of solver is often an inevitable part of modelling. Therefore, being able to compare two executions to evaluate the impact of a particular change is crucial for model improvement. The existing approaches for comparing two executions are limited to a manual side-by-side comparison of the respective search trees. The contributions of this thesis to support the comparison of executions are as follows:

- It presents *search merging* – a profiling technique that allows the modeller to visually compare the search trees of two different executions, before and after performing some change to the model, solver, or search strategy, and assess the effect of that change in detail. It is most effective when a fixed search strategy is employed, or when a dynamic search strategy is used in both executions that does not make the searches diverge too soon in the search tree. In the former case, it will pinpoint where one execution required less search compared to the other (for example, due to a difference in propagation strength). In the latter case, the technique will pinpoint the nodes in the search tree where the two executions start to diverge. This technique scales well for large executions, but requires their early search decisions (near the roots) to be similar in order to be really useful.
- It presents *search replaying* – a profiling technique that allows the modeller to isolate the effect of a change in terms of constraint propagation. In particular, it ensures that the two executions follow exactly the same search decisions up to a failure. Since the difference in the two searches is minimised, this allows the modeller to evaluate the difference in terms of propagation between the two executions.
- It implements both search merging and search replaying and integrates them into CP-Profiler, additionally demonstrating their good scalability for large executions.
- It presents a case study that demonstrates that the two comparison techniques can be used hand-in-hand to gain better insights regarding model modifications, that in turn could suggest further model improvements.

Although these techniques can already be effectively used to support the comparison of two executions, further improvements are possible. For example, the search merging is limited to executions with similar search decisions near the root. One opportunity for future work is to extend this technique to additionally compare searches after they have diverged.

Learning from Learning Solvers

A generation of no-good learning solvers has emerged in recent years, and they often solve hard constraint problems faster than traditional, non-learning solvers. However, for some problems, learning solvers seem to be unable to benefit from their learning and perform poorly compared to non-learning solvers. The reasons for these differences in behaviour are not well understood in practice, as learning solvers are even more complex than traditional CP solvers. This thesis explores the profiling of learning solvers, and in doing so it presents a surprising discovery: the profiling information can be used to aid modellers to improve their models. In particular, the contributions of this thesis in this regard are as follows:

- It presents *no-good analysis* – a technique for studying the execution of learning solvers. This technique is built on top of the search replaying and the search merging

techniques, and adds extra functionality specific for learning solvers that allows the modeller to identify effective no-goods – those that have the most impact on the execution (i.e., reduce the highest amount of search).

- It demonstrates how the most effective no-goods can be used to identify constraints in the model that do not propagate well, and thus should be replaced by more effective ones to speed up the execution.
- It demonstrates how the most effective no-goods can also be used to discover redundant constraints that can be added to the model. Importantly, the redundant constraints obtained this way are effective (that is, they improve the resulting model when they are added), as they are based on the no-goods that significantly reduce search space.
- It demonstrates that the model modifications obtained using the no-good analysis can lead to significantly faster executions for both learning and non-learning solvers, often with a few orders of magnitude improvement.

Although the no-good analysis is already effective in its current form, it can be improved further. For example, its current implementation requires the modeller to process the list of effective no-goods manually in order to infer model improvements. One interesting and challenging opportunity for future work is to automate this process by, for example, automatically generating redundant constraints. Another way to improve the analysis is to automatically link the effective no-goods to the constraints in the model – a task that is performed manually in the current implementation – thus showing the modeller which constraints require more attention.

Conclusion

Profiling in Constraint Programming, despite the great deal of previous research dedicated to it, is still not as commonly employed in day-to-day modelling as it is in imperative programming. This thesis has a potential to change that by demonstrating that profiling can be effective for model improvement, and thus, worthwhile. In fact, the optimisation team at Monash University has recognised its value, and provided me with the opportunity to work on CP-Profiler further to make it more robust, document its functionality, and help in integrating it into the MiniZinc IDE, which I gladly accepted.

Bibliography

Armando, A., C. Castellini, and E. Giunchiglia (1999). "SAT-Based Procedures for Temporal Reasoning". In: *Recent Advances in AI Planning, 5th European Conference on Planning, Proceedings*. Ed. by S. Biundo and M. Fox. Vol. 1809. Lecture Notes in Computer Science. Springer, pp. 97–108.

Baatar, D., N. Boland, S. Brand, and P. J. Stuckey (2011). "CP and IP Approaches to Cancer Radiotherapy Delivery Optimization". In: *Constraints*. Vol. 16. 2. Springer, pp. 173–194.

Barták, R. (2005). "Effective Modeling With Constraints". In: *Applications of Declarative Programming and Knowledge Management*. Springer, pp. 149–165.

Biere, A., M. Heule, H. van Maaren, and T. Walsh, eds. (2009). *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press.

Bouvier, P. (2000). "Visual Tools to Debug Prolog IV Programs". In: *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*. Ed. by P. Deransart, M. V. Hermenegildo, and J. Maluszynski. Vol. 1870. Lecture Notes in Computer Science. Springer, pp. 177–190.

Bracchi, C., C. Gefflot, and F. Paulin (2001). "Combining Propagation Information and Search Tree Visualization Using ILOG OPL Studio". In: *arXiv preprint cs/0111040*.

Burch, M., M. Raschke, and D. Weiskopf (2010). "Indented Pixel Tree Plots". In: *Advances in Visual Computing – 6th International Symposium, Proceedings, Part I*. Ed. by G. Bebis, R. D. Boyle, B. Parvin, D. Koracin, R. Chung, R. I. Hammoud, M. Hussain, K. Tan, R. Crawfis, D. Thalmann, D. Kao, and L. Avila. Vol. 6453. Lecture Notes in Computer Science. Springer, pp. 338–349.

Carro, M. and M. V. Hermenegildo (2000a). "Tools for Constraint Visualisation: The VIFID/TRIFID Tool". In: *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*. Ed. by P. Deransart, M. V. Hermenegildo, and J. Maluszynski. Vol. 1870. Lecture Notes in Computer Science. Springer, pp. 253–272.

— (2000b). "Tools for Search-Tree Visualisation: The APT Tool". In: *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*. Ed. by P. Deransart, M. V. Hermenegildo, and J. Maluszynski. Vol. 1870. Lecture Notes in Computer Science. Springer, pp. 237–252.

Cheng, B., K. M. Choi, J. H.-M. Lee, and J. Wu (1999). "Increasing Constraint Propagation by Redundant Modeling: an Experience Report". In: *Constraints*. Vol. 4. 2. Springer, pp. 167–192.

Choi, C. W., J. H. Lee, and P. J. Stuckey (2007). "Removing Propagation Redundant Constraints in Redundant Modeling". In: *ACM Transactions on Computational Logic*. Vol. 8. 4. ACM, p. 23.

Chu, G. G. (2011). "Improving Combinatorial Optimization". PhD thesis. The University of Melbourne.

Dakin, R. J. (1965). "A Tree-Search Algorithm for Mixed Integer Programming Problems". In: *The Computer Journal*. Vol. 8. 3. Oxford University Press, pp. 250–255.

Dantzig, G. B. (1963). *Linear Programming and Extensions. Princeton landmarks in mathematics and physics.*

Davis, M. and H. Putnam (1960). "A Computing Procedure for Quantification Theory". In: *Journal of the ACM*. Vol. 7. 3. ACM, pp. 201–215.

Deransart, P. (2004). "Main Results of the OADymPPaC Project". In: *Logic Programming, 20th International Conference, Proceedings*. Ed. by B. Demoen and V. Lifschitz. Vol. 3132. Lecture Notes in Computer Science. Springer, pp. 456–457.

— (2011). "Generic Traces and Constraints, GenTra4CP revisited". In: *arXiv preprint arXiv:1105.6210*.

Deransart, P., M. V. Hermenegildo, and J. Maluszynski, eds. (2000). *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*. Vol. 1870. Lecture Notes in Computer Science. Springer.

Dooms, G., P. V. Hentenryck, and L. Michel (2009). "Model-Driven Visualizations of Constraint-Based Local Search". In: *Constraints*. Vol. 14. 3. Springer, pp. 294–324.

Fages, F., S. Soliman, and R. Coolen (2004). "CLPGUI: A Generic Graphical User Interface for Constraint Logic Programming". In: *Constraints*. Vol. 9. 4. Springer, pp. 241–262.

Frisch, A. M., M. Grum, C. Jefferson, B. M. Hernández, and I. Miguel (2007). "The Design of ESSENCE: A Constraint Language for Specifying Combinatorial Problems". In: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*. Ed. by M. M. Veloso. AAAI Press, pp. 80–87.

Goodwin, S., C. Mears, T. Dwyer, M. Garcia de la Banda, G. Tack, and M. Wallace (2017). "What do Constraint Programming Users Want to See? Exploring the Role of Visualisation in Profiling of Models and Search". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1, pp. 281–290.

Google (2015). *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers/.

Goualard, F. and F. Benhamou (2000). "Debugging Constraint Programs by Store Inspection". In: *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*. Ed. by P. Deransart, M. V. Hermenegildo, and J. Maluszynski. Vol. 1870. Lecture Notes in Computer Science. Springer, pp. 273–297.

Graham, S. L., P. B. Kessler, and M. K. McKusick (1982). "Gprof: A Call Graph Execution Profiler". In: *ACM Sigplan Notices*. Vol. 17. 6. ACM, pp. 120–126.

Hart, P. E., N. J. Nilsson, and B. Raphael (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Trans. Systems Science and Cybernetics*. Vol. 4. 2. IEEE, pp. 100–107.

Hartert, R. (2017). "Kiwi-A Minimalist CP Solver". In: *arXiv preprint arXiv:1705.00047*.

Hebrard, E. (2008). "Mistral, a Constraint Satisfaction Library". In: *Proceedings of the Third International CSP Solver Competition* 3, p. 3.

Held, M. and R. M. Karp (1971). "The Traveling-Salesman Problem and Minimum Spanning Trees: Part II". In: *Mathematical Programming*. Vol. 1. 1. Springer, pp. 6–25.

Horbach, M. and S. Woop (2002). "Incremental Algorithms and a Minimal Graph Representation for Regular Trees". Bachelor Thesis. Saarland University.

Johnson, B. (2012). *Professional Visual Studio 2012*. John Wiley & Sons.

Kennedy, A. J. (1996). "Functional Pearls". In: vol. 6. 3. Cambridge University Press, pp. 527–534.

Kruskal, J. B. and J. M. Landwehr (1983). "Icicle Plots: Better Displays for Hierarchical Clustering". In: vol. 37. 2. Taylor & Francis Group, pp. 162–168.

Kuchcinski, K. and R. Szymanek (2013). "Jacop-Java Constraint Programming Solver". In: *CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming.*

Land, A. H. and A. G. Doig (1960). "An Automatic Method of Solving Discrete Programming Problems". In: vol. 28. 3. The Econometric Society, pp. 497–520.

Langevine, L. (2005). "Gentra4cp: A Generic Trace Format for Constraint Programming". In: *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings*. Ed. by M. Gabbrielli and G. Gupta. Vol. 3668. Lecture Notes in Computer Science. Springer, pp. 433–434.

Mackworth, A. K. (1977). "Consistency in Networks of Relations". In: vol. 8. 1. Elsevier, pp. 99–118.

Marques Silva, J. P. and K. A. Sakallah (1996). "GRASP – a New Search Algorithm for Satisfiability". In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*. IEEE Computer Society, pp. 220–227.

Meier, M. (1995). “Debugging Constraint Programs”. In: *Principles and Practice of Constraint Programming, First International Conference, Proceedings.* Ed. by U. Montanari and F. Rossi. Vol. 976. Lecture Notes in Computer Science. Springer, pp. 204–221.

Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik (2001). “Chaff: Engineering an Efficient SAT Solver”. In: *Proceedings of the 38th Design Automation Conference.* ACM, pp. 530–535.

Nelson, G. and D. C. Oppen (1979). “Simplification by Cooperating Decision Procedures”. In: *ACM Transactions on Programming Languages and Systems.* Vol. 1. 2. ACM, pp. 245–257.

Nethercote, N., P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack (2007). “MiniZinc: Towards a Standard CP Modelling Language”. In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, Proceedings.* Ed. by C. Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, pp. 529–543.

Newsham, Z., W. Lindsay, V. Ganesh, J. H. Liang, S. Fischmeister, and K. Czarnecki (2015). “SATGraf: Visualizing the Evolution of SAT Formula Structure in Solvers”. In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Proceedings.* Ed. by M. Heule and S. Weaver. Vol. 9340. Lecture Notes in Computer Science. Springer, pp. 62–70.

Ohrimenko, O., P. J. Stuckey, and M. Codish (2009). “Propagation via Lazy Clause Generation”. In: vol. 14. 3. Springer, pp. 357–391.

OscaR Team (2012). *OscaR: Scala in OR.* URL: https://bitbucket.org/oscarlib/oscar.

Prud’homme, C., J.-G. Fages, and X. Lorca (2017). *Choco Documentation.* TASC, LS2N, CNRS UMR 6241 and COSLING S.A.S. URL: http://www.choco-solver.org.

Rossi, F., P. van Beek, and T. Walsh (2008). “Constraint Programming”. In: *Handbook of Knowledge Representation.* Ed. by F. van Harmelen, V. Lifschitz, and B. W. Porter. Vol. 3. Foundations of Artificial Intelligence. Elsevier, pp. 181–211.

Ryder, B. G. (1979). “Constructing the Call Graph of a Program”. In: vol. 5. 3. IEEE, pp. 216–226.

Schimpf, J. and K. Shen (2012). “ECL^iPS^e – from LP to CLP”. In: *Theory and Practice of Logic Programming.* Vol. 12. 1-2. Cambridge University Press, pp. 127–156.

Schulte, C. (1996). “Oz Explorer: A Visual Constraint Programming Tool”. In: *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium.* Ed. by H. Kuchen and S. D. Swierstra. Vol. 1140. Lecture Notes in Computer Science. Springer, pp. 477–478.

Schulte, C., G. Tack, and M. Z. Lagerkvist (2016). *Modeling and Programming with Gecode.* URL: http://www.gecode.org.

Schutt, A., T. Feydy, P. J. Stuckey, and M. Wallace (2009). "Why Cumulative Decomposition Is Not as Bad as It Sounds". In: *CP 2009*. Ed. by I. P. Gent. Vol. 5732. Lecture Notes in Computer Science. Springer, pp. 746–761.

Schutt, A., P. J. Stuckey, and A. R. Verden (2011). "Optimal Carpet Cutting". In: *CP 2011*. Ed. by J. H. Lee. Vol. 6876. Lecture Notes in Computer Science. Springer, pp. 69–84.

Shirazi, J. (2002). "Tool Report: JProfiler". In: *Java Performance Tuning*.

Shishmarev, M., C. Mears, G. Tack, and M. Garcia de la Banda (2016a). "Learning from Learning Solvers". In: *Principles and Practice of Constraint Programming – 22nd International Conference, Proceedings*. Vol. 9892. Lecture Notes in Computer Science. Springer, pp. 455–472.

— (2016b). "Visual Search Tree Profiling". In: *Constraints*. Vol. 21. 1. Springer, pp. 77–94.

Simonis, H., P. Davern, J. Feldman, D. Mehta, L. Quesada, and M. Carlsson (2010). "A Generic Visualization Platform for CP". In: *Principles and Practice of Constraint Programming – 16th International Conference, Proceedings*. Vol. 6308. Lecture Notes in Computer Science. Springer, pp. 460–474.

Simonis, H. and A. Aggoun (2000). "Search-Tree Visualisation". In: *Analysis and Visualization Tools for Constraint Programming*. Ed. by P. Deransart, M. V. Hermenegildo, and J. Maluszynski. Vol. 1870. Lecture Notes in Computer Science. Springer, pp. 191–208.

Sinz, C. and E. Dieringer (2005). "DPvis - A Tool to Visualize the Structure of SAT Instances". In: *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*. Ed. by F. Bacchus and T. Walsh. Vol. 3569. Lecture Notes in Computer Science. Springer, pp. 257–268.

Stuckey, P. J., T. Feydy, A. Schutt, G. Tack, and J. Fischer (2014). "The MiniZinc Challenge 2008–2013". In: *AI Magazine*. Vol. 35. 2. AAAI, pp. 55–60.

Van Cauwelaert, S., M. Lombardi, and P. Schaus (2015). "Understanding the Potential of Propagators". In: *Integration of AI and OR Techniques in Constraint Programming – 12th International Conference, Proceedings*. Springer, pp. 427–436.

Van Hentenryck, P., L. Michel, L. Perron, and J.-C. Régin (1999). "Constraint Programming in OPL". In: *Principles and Practice of Declarative Programming, International Conference, Proceedings*. Ed. by G. Nadathur. Vol. 1702. Lecture Notes in Computer Science. Springer, pp. 98–116.

Zhou, N.-F., H. Kjellerstrand, and J. Fruhman (2015). *Constraint Solving and Planning with Picat*. Springer.