Discrete Optimization over Graph Problems

Diego De Uña

Submitted in total fulfilment of the requirements of the degree of
Doctor of Philosophy

Department of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

June 2018
Abstract

Discrete optimization problems are ubiquitous both in industry and theoretical computer science. They appear in a wide range of fields, such as manufacturing, planning, packing, design and scheduling. It is often the case that the problem at hand is in the form of a graph, or can be represented as such. It is therefore natural to have an interest in tackling this specific kind of problems.

In this thesis we mainly use Constraint Programming as a technology to solve discrete optimization problems over graphs. We identify substructures that occur repeatedly in different problems that are usually tackled using Constraint Programming. We then study these combinatorial substructures to develop advanced algorithms within Constraint Programming solvers that attempt to drastically reduce the time spent by solvers on the problem. These are called propagators, and this thesis explores a series of them that deal with graph structures. All these algorithms have been implemented and different set of experiments show the value of each one of them.

We also investigate the use of graphical structures not as a tool to represent the model, but rather as a tool within a Constraint Programming solver to increase its performance when tackling highly constrained problems. We do so through the use of Multi-valued Decision Diagrams (MDD) and Deterministic Decomposable Negation Normal Form Formulae (d-DNNF) for which propagators already exist. Our work on this focuses on improving the model provided by a user of the solver, by pre-processing parts of it in the hope of decreasing the total runtime.

Additionally, we identified a graph problem in the field of computational sustainability for which Constraint Programming solvers were not adequate. In order to solve this problem, we developed a completely new Local Search algorithm that was specifically
designed for it.

The common thread between all components of this thesis are graphs and discrete optimization. We have looked at how to improve Constraint Programming technology to solve discrete problems containing graphs, how to use graphs within Constraint Programming to solve any discrete problem, and lastly, how to tackle discrete graph problems when Constraint Programming is not an adequate tool.
Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,

2. due acknowledgement has been made in the text to all other material used,

3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Diego De Uña, June 2018
Acknowledgements

First and foremost I would like to thank my three supervisors: Peter Stuckey, Graeme Gange and Peter Schachte. I cannot emphasize enough how much the success of this thesis is, in big part, thanks to them. I don’t think I could have possibly ever had a better main Ph.D. supervisor than Peter Stuckey. He was to me what I hoped a supervisor would be: involved, efficient, knowledgeable and did not waste my energy making me do things I did not want to. The support and ideas of all my supervisors (and Graeme’s understanding of the intricacies of CHUFFED) are what have driven this thesis to success.

I would like to thank Harald Søndergaard and Tim Baldwin for allowing me to teach their Master’s and Bachelor’s classes. It was definitely a big help to distract my mind when I have been stuck during my Ph.D.. To my only collaborator outside of my supervisors, Nataliia Rümelé, also thank you. I take this opportunity to also thank the University of Melbourne, the Fundación LaCaixa, Data 61 and Google Australia for providing me with financial support during my thesis in the form of scholarships.

I would also like to thank the friends I have made during my stay in Melbourne, specially to Doris for showing me that, if you buy a $10 inflatable mattress, camping is actually quite nice; Aili, Felix, Mohammad and everyone in room 8.19; and to Julian for all those discussions over lunch.

Lastly, I would like to thank my parents and my brother for their support during these three and a half years. I never took the due time to properly explain to them what I was doing in my thesis, but they still supported me all this time and followed my progress eagerly.
Preface

- The content of Chapter 3 has been published (in its majority) as:

- The content of Chapter 4 has been published (in its majority) as:
  Diego De Uña, Nataliia Rümelle, Graeme Gange, Peter Schachte, and Peter J. Stuckey. Machine Learning and Constraint Programming for Relational-To-Ontology Schema Mapping. In *International Joint Conference on Artificial Intelligence (IJCAI)*, (To appear at the moment of submission of this thesis), 2018

- The content of Chapter 5 has been published (in its majority) as:

- The content of Chapter 6 has been published (in its majority) as:

- The content of Chapter 7 has been published (in its majority) as:
  Diego De Uña, Graeme Gange, Peter Schachte, and Peter J. Stuckey. Minimizing

- The content of Chapter 8 has been submitted for publication (in its majority) at the *Constraints* journal and is, at the time of submitting this thesis, under review.
To my parents and my brother,
Contents

1 Introduction 1

2 Background 5
2.1 Constraint Programming 5
  2.1.1 SAT Solvers 8
  2.1.2 Constraint Programming Solvers 10
  2.1.3 Clause Learning 14
2.2 Graphs in Constraint Programming 20
  2.2.1 Definitions and Notation 20
  2.2.2 Graph Variables and Constraints 22
2.3 Other Discrete Optimization Techniques 25
  2.3.1 Mixed Integer Programming 25
  2.3.2 Local Search 30
2.4 Decision Diagrams 32
  2.4.1 Multivalued Decision Diagrams 32
  2.4.2 Edge-valued MDDs 34
  2.4.3 Deterministic Decomposable Negation Normal Form Formulae (d-DNNFs) 34

3 Steiner Trees in Constraint Programming 37
3.1 The Steiner Tree Problem 37
3.2 State of the Art 38
3.3 Applying Constraint Programming 40
  3.3.1 Basic Graph Propagation with Explanations 40
  3.3.2 A Connectedness Propagator with Explanations 41
  3.3.3 An Acyclicity Propagator with Explanations 49
  3.3.4 A Tree Propagator with Explanations 53
  3.3.5 Steiner Tree Global Constraint 53
  3.3.6 Lower-bounding for Steiner Trees 55
3.4 Experimental Evaluation and Results 62
  3.4.1 The Grade of Service STP (GoSST) 63
  3.4.2 The Terminal Steiner Tree Problem (TSTP) 65
3.5 Concluding Remarks 66
# 4 Steiner Trees Applied to the Relational-To-Ontology Schema Mapping Problem

4.1 The Rel2Onto Problem ........................................ 69
  4.1.1 Problem Statement and Vocabulary ....................... 71
4.2 Machine Learning for Training on Source Descriptions ........ 74
  4.2.1 Overview on Machine Learning .......................... 74
  4.2.2 Semantic Labeling ....................................... 76
  4.2.3 Alignment Graph .......................................... 79
  4.2.4 Frequent Graph Pattern Mining .......................... 80
4.3 Steiner Tree Formulation .................................... 81
  4.3.1 Using Patterns ........................................... 83
  4.3.2 Unmatched Attributes ................................... 83
4.4 Modeling in Constraint Programming .......................... 84
4.5 Experimental Evaluation and Results ........................ 86
  4.5.1 Experimental Setup ....................................... 86
  4.5.2 Experimental Results .................................... 88
4.6 Related Work .................................................. 91
4.7 Concluding Remarks .......................................... 93

# 5 Spanning Trees in Constraint Programming

5.1 The Minimum Spanning Tree Problem ........................ 95
5.2 State of the Art .............................................. 97
5.3 Global Constraint for the Weighted Spanning Tree ......... 98
  5.3.1 Lower Bound with Explanations ......................... 98
  5.3.2 Propagation Rule with Explanations .................... 102
5.4 Experimental Evaluation and Results ......................... 103
  5.4.1 The Diameter Constrained Minimum Spanning Tree .... 103
  5.4.2 Non-crossing Edges MST .................................. 106
5.5 Concluding Remarks .......................................... 111

# 6 Graph Paths in Constraint Programming

6.1 Path Problems in Constraint Programming .................. 113
6.2 State of the Art .............................................. 116
6.3 From a Directed Reachability Propagator to a Path Constraint 117
  6.3.1 A Directed Reachability Propagator with Explanations 118
  6.3.2 A Directed Tree Propagator with Explanations ........ 120
  6.3.3 DAG Propagator with Explanations ...................... 120
  6.3.4 A Path Global Constraint ............................... 122
6.4 A Distance-Bounded Path Propagator with Explanations ...... 122
  6.4.1 Propagating Simple Distances ........................... 123
  6.4.2 Propagating Combined Distances ......................... 125
  6.4.3 Stronger Bounding Using Dynamic Programming .......... 126
  6.4.4 Improving the Explanations of the Bounded Path Propagator 129
6.5 Experimental Evaluation and Results ........................ 133
  6.5.1 The Maximum Directed Acyclic Subgraph Problem (MDSP) 134
  6.5.2 Node Constrained Shortest Paths ....................... 135
6.5.3 Metabolic Networks ........................................... 136
6.5.4 Task Constrained Shortest Path .............................. 137
6.5.5 Non-Overlapping Paths ..................................... 138
6.5.6 Profitable Tourist Path .................................... 139
6.6 Concluding Remarks ........................................... 140

7 Discrete Optimization for Habitat Conservation ................. 143
  7.1 Habitat Conservation By Resistance Model ..................... 143
     7.1.1 Corridors Model ......................................... 144
     7.1.2 Multi-species Area-driven Model ......................... 144
     7.1.3 Resistance-based Model .................................. 145
  7.2 Problem Formulation and Required Notions .................... 147
     7.2.1 Preliminaries ............................................. 147
     7.2.2 Problem Formulation ..................................... 149
     7.2.3 Complexity ................................................. 151
  7.3 Solving the Minimum Effective Resistance Problem ............ 152
     7.3.1 Greedy Algorithm Approach .............................. 152
     7.3.2 Mixed Integer Programming Approach .................... 153
     7.3.3 Local Search Approach ................................... 155
  7.4 Experimental Evaluation and Results .......................... 157
     7.4.1 Instances .................................................. 157
     7.4.2 Improvement Over the Initial Landscape ................. 159
     7.4.3 Optimality Gap ............................................. 160
     7.4.4 Number of Iterations ..................................... 161
     7.4.5 Destruction Rate .......................................... 162
  7.5 Concluding Remarks ........................................... 163

8 Graphs for Constraint Programming: MDDs and d-DNNFs .......... 165
  8.1 Compiling Constraint Programming Subproblems into Tractable Structures 165
  8.2 State of the Art ................................................. 167
     8.2.1 MDD Compilations .......................................... 167
     8.2.2 NNF Compilations ......................................... 170
  8.3 Subproblem Equivalence ........................................ 172
     8.3.1 Key Construction ........................................... 173
  8.4 CP-to-MDD and CP-to-Cost-MDD Compilations .................... 176
     8.4.1 Compiling a Cost-MDD ...................................... 180
  8.5 CP-to-d-DNNF Compilation ...................................... 180
     8.5.1 Splitting Subproblems for d-DNNFs ....................... 182
  8.6 Experimental Evaluation and Results .......................... 186
     8.6.1 Problem Descriptions ...................................... 187
     8.6.2 Compilation Effort ........................................ 192
     8.6.3 Splitting Globals for d-DNNFs ............................. 194
     8.6.4 Using the Compiled Structures ............................ 196
  8.7 Concluding Remarks ............................................ 201

9 Conclusion ......................................................... 203
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of search tree and clause learning</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of the derivation of a 1UIP</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of drawing convention for graph variables</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>Geometric view of a linear program in 2D</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>Objective function progressing through Simplex algorithm</td>
<td>28</td>
</tr>
<tr>
<td>2.6</td>
<td>Example of objective function and Local Search solutions</td>
<td>32</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of an MDD</td>
<td>33</td>
</tr>
<tr>
<td>2.8</td>
<td>Example of a cost-MDD</td>
<td>34</td>
</tr>
<tr>
<td>2.9</td>
<td>Example of a d-DNNF</td>
<td>35</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of two Steiner trees for the same graph</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of unreachable nodes</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of oposing DFS to find explanations in a graph</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of bridges and articulations detection</td>
<td>45</td>
</tr>
<tr>
<td>3.5</td>
<td>Example of the DFS to build explanations for articulations and bridges</td>
<td>46</td>
</tr>
<tr>
<td>3.6</td>
<td>Example of potential cycles</td>
<td>50</td>
</tr>
<tr>
<td>3.7</td>
<td>Visual example of Rerooted-Union-Find</td>
<td>52</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of the application of contraction function ( \mu )</td>
<td>56</td>
</tr>
<tr>
<td>3.9</td>
<td>Example of non-satisfied GoSST</td>
<td>64</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of an Ontology</td>
<td>71</td>
</tr>
<tr>
<td>4.2</td>
<td>Examples of two semantic models</td>
<td>72</td>
</tr>
<tr>
<td>4.3</td>
<td>Expected semantic model</td>
<td>73</td>
</tr>
<tr>
<td>4.4</td>
<td>Example of regression problem</td>
<td>75</td>
</tr>
<tr>
<td>4.5</td>
<td>Example of alignment graph</td>
<td>80</td>
</tr>
<tr>
<td>4.6</td>
<td>Example of integration graph</td>
<td>82</td>
</tr>
<tr>
<td>4.7</td>
<td>Performance for the museum and soccer domains</td>
<td>89</td>
</tr>
<tr>
<td>4.8</td>
<td>Performance of SERENE for leave-one-out strategy on the museum domain</td>
<td>91</td>
</tr>
<tr>
<td>5.1</td>
<td>Example of the computation of the lower bound for a WST</td>
<td>102</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of a graph where a greedy algorithm would fail</td>
<td>107</td>
</tr>
<tr>
<td>5.3</td>
<td>Gain in nodes provided by explanations</td>
<td>109</td>
</tr>
<tr>
<td>5.4</td>
<td>Gain in time provided by explanations</td>
<td>110</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison between our propagator against CHOCO</td>
<td>110</td>
</tr>
<tr>
<td>6.1</td>
<td>Example of basic explanations for bounded paths</td>
<td>124</td>
</tr>
</tbody>
</table>
List of Tables

3.1 List of algorithms applied in our tree propagator .......................... 54
3.2 Results for the pure Steiner Tree problem .................................. 63
3.3 Results for the GoSST .......................................................... 65
3.4 Results for the TSTP ............................................................. 66

4.1 Example of tables to be integrated in the ontology ......................... 73
4.2 Feature extractors that produce a scalar value ............................... 77
4.3 Feature extractors that produce a vector value ................................ 78
4.4 Description of data sources .................................................... 86
4.5 Average performance of semantic labeling models for leave-one-out strategy .......................................................... 88

5.1 Comparison in time (seconds) and nodes for the DCMST models ........ 105

6.1 Comparison of propagator vs. decomposition for dag constraint .......... 134
6.2 Comparison of bounded path propagators with artificial benchmarks .... 135
6.3 Solving metabolic pathways in real-world networks ......................... 137
6.4 Comparing BPATH and DPBPATH using PATH and CIRCUIT constraints . 138
6.5 Cooperative shortest path finding ............................................. 139
6.6 Profitable tourist path problem results ...................................... 140

7.1 Averages of the ratios of updated resistance over original resistance .... 159
7.2 Average of ratio between LS and proven optimum ............................ 161
7.3 Improvement of solution over destruction rate ............................... 163

8.1 Time spent compiling Tables, MDDs or d-DNNFs ........................... 192
8.2 Comparison of compilation time and size of d-DNNFs with or without splitting .......................................................... 194
8.3 Comparison of compilation time and size of d-DNNFs with or without splitting (tree constraint) .................................................. 195
8.4 Comparison of conflicts, nodes and time for the shift scheduling problem 200
List of Algorithms

2.1 DPLL Algorithm ................................................. 9
3.1 Auxiliary DFS functions ....................................... 44
3.2 Algorithm to ensure connectivity in a graph .................. 44
3.3 Auxiliary DFS functions ....................................... 47
3.4 Algorithm to propagate articulations and bridges for connectivity .... 48
3.5 Algorithm to explain propagation on articulations and bridge ........ 48
3.6 Algorithms for cycle detection and prevention .................. 50
3.7 Algorithms for R-Uf ........................................... 52
5.1 Computing the lower bound with explanation .................... 101
6.1 Enforcing dominators to be in the graph ....................... 120
6.2 Algorithms for ITC ............................................ 121
6.3 Shortest path from s to d containing all mandatory nodes M ........ 127
6.4 Explaining why n is at distance more than w from the source ........ 130
6.5 Better explanations ............................................ 133
6.6 Avoiding state explosion ....................................... 133
7.1 Greedy algorithm .............................................. 152
8.1 Constructing MDD via propagation ................................ 177
8.2 Constructing d-DNNF via propagation .......................... 181
8.3 Constructing d-DNNF via propagation while splitting subproblems ..... 186
Chapter 1
Introduction

Discrete optimization problems are problems where one has to choose the best solution for a problem, based on some measure or objective, amongst a discrete set of solutions. These problems appear in all kinds of situations, such as scheduling the tasks performed by a machine in a factory, packing objects into a cargo container, laying out a computer or electric network, or choosing the areas in a landscape where reforestation is most likely to help native species survive.

Unlike most problems students in computer science would see in their syllabus, there is no algorithms to solve these problems. Even more interesting is the fact that many of the algorithms one may learn during their studies are only applicable to a very specific, often simplistic, version of some problem. For example, finding the shortest path in a graph can be efficiently solved using Dijkstra’s algorithm. But finding the shortest path for an actual car cannot be solved by that algorithm, as the vehicle will have limited fuel and need to refuel.

The goal of this thesis is precisely to allow humans to solve some of these problems more easily. The world of discrete problems is too vast for the scope of a doctorate, and would require multiple lifetimes to study. Therefore, in this thesis, we focus on a small subset of problems: combinatorial problems over graphs. These are problems where the solution depends on the topology or weight of the edges in a graph.

The main technology we use in this thesis is Constraint Programming. This is a programming paradigm that allows a user to model a problem without specifying the way the model needs to be solved. A Constraint Programming solver will then take the model and try to find solutions for it. This is a brilliant technique because, although the prob-
lem at hand may not have an algorithm to solve it, there is some reasoning that can be
done about it to lead the search to a solution. This reasoning is the core of Constraint
Programming solvers, implemented as propagators.

The main focus of this thesis is precisely the design and implementation of these propa-
gagators. We implement several of them, all related to graphs, as we often see some combi-
natorial problem containing a graph as a core element to the problem, or as a sub-part.
For example, in a problem where we need to find a path in a city, a graph is at the core of
the problem. Similarly if we are designing a computer network, we also need to represent
the network as a graph. The propagators will do reasoning over the graph for specific
constraints. We will see propagators that ensure that a graph is connected, or a directed
graph is acyclic, or that a path is of a certain length. One particularity of our implement-
ation of propagators is that all of them generate explanations for no-good learning. This is
a powerful technique that allows the solver to remember the work done by propagators,
so that if a similar subproblem needs to be solved later, the inferences done earlier can be
reused immediately. This effectively reduces the total solving time.

Developing these propagators is not the same as developing an algorithm to solve
the entire problem where this graph appears. The solver will make the different propa-
gagators interact with each other until a solutions is found, but none of the propagators is
actually solving the entire problem. This is a key element of Constraint Programming,
and probably its greatest asset: flexibility. Constraint Programming is modular and flex-
ible by design. The propagators deal with separate parts of the problem in a hermetic
manner, and the solver conjoins them to find a solution. In addition to that, Constraint
Programming is extremely efficient and the fastest way of solving many problems, as
we will see in this thesis. Furthermore, our experiments show that no-good learning is
key to achieve this great performance, and it is well worth the implementation effort.
Interestingly enough, no-good learning “breaks” the hermeticity of propagators by creat-
ing clauses that combine the reasoning done by multiple propagators at once. Yet, from
a software architecture perspective, the design of each propagator is still independent,
thus keeping the flexibility of Constraint Programming intact.

The last two chapters of this thesis are slightly different. In Chapter 7, we look at
a discrete optimization problem with a graph at its core, but we do not use Constraint Programming to solve it. This is because Constraint Programming is not always the most suitable technology to tackle discrete optimization problems. As we will see, some problems are very unconstrained and propagators would have very little reasoning to do, thus making the solver “blind” in the problem, and trying solutions with no true reasoning. Chapter 8 is different in that we do not tackle discrete optimization problems involving graphs, but rather we tackle discrete optimization problems through graphs. Multi-valued Decision Diagrams (MDDs) and Deterministic Decomposable Negation Normal Form formulae (d-DNNFs) are data-structures that can be easily represented as graphs. Their goal is to hold an enormous amount of information whilst using a small amount of memory. We use these graph-based structures to improve the speed of Constraint Programming solvers for generic problems (not just graph-related).

This thesis is divided into 9 chapters.

Chapter 2 provides the reader with all the definitions and knowledge required to understand the rest of this thesis. It is intended, ideally, to be readable by someone that knows little or nothing about this field and can hopefully be accessible to anyone about to start their research on the topic.

Chapter 3 presents propagators to ensure that a graph is a tree. The goal of the chapter was to solve variants of the well known Steiner Tree Problem. Therefore, in addition to algorithms involving the topology of the graph, it also presents algorithms for a propagator dedicated to Steiner trees, taking into account weights on the edges of the graph.

Chapter 4 is a direct application of the previous chapter. This was a problem that was brought to us by Natalia Rümelle, researcher at Data61 (at that time), after reading our paper of the previous chapter. This is probably one of the most interesting chapters, as it showcases the value of Constraint Programming when brought to researchers that were not necessarily aware of this technology.

Chapter 5 deals with a special case of Steiner Trees, where all the nodes need to be connected. It is closely related to the Minimum Spanning Tree problem. This is a perfect example of a well known algorithm only working in a specific situation: Kruskal’s algorithm does solve the Minimum Spanning Tree problem. But in real life, this prob-
lem contains side constraints that make such algorithms inapplicable and the problems combinatorial. We create specialized propagators to deal with weighted spanning trees in Constraint Programming.

Chapter 6 is yet another example of classic algorithms not being applicable in general. This chapter tackles bounded paths as a propagator. This means any problem that involves finding a path on a (weighted) graph can be modeled in Constraint Programming using the work developed here to enhance the reasoning of the solver.

Chapter 7 leaves Constraint Programming to the side and tackles a problem in computational sustainability using an ad-hoc Local Search approach. The problem consists in finding the best parts of some landscape to improve (in an ecological sense, e.g. reforest) in order to ease the survival of some species on that land. Surprisingly, the core of the problem is a graph.

Chapter 8 deals with the precompilation of parts of a Constraint Programming model using Multi-valued Decision Diagrams (MDDs) and Deterministic Decomposable Negation Normal Form formulae (d-DNNFs). The goal is to then enhance the overall performance of the solver. We show how, for very hard problems, this can mean a substantial improvement.

Chapter 9 concludes this thesis, given an overview of the contributions and some take-away messages for the reader.
Chapter 2
Background

This chapter presents all the knowledge that the reader needs to understand the technical details of this thesis. We provide definitions and notations that will be used throughout this thesis. The first part of this chapter presents Constraint Programming (CP) as a programming paradigm, building from SAT solvers to state of the art CP solvers. The second part provides an overview of past work where graphs have been used in CP. This is not intended to be the entire overview of the related work to this thesis, as more specific details are provided in each chapter. The third part discusses other approaches to Discrete Optimization Problems. As we will see in Chapter 7, CP is not necessarily the best approach to all problems. Therefore, we provide an overview on other technologies. Finally, the last section discusses decision diagrams. These are graph-based data structures that can be used to improve CP solvers.

2.1 Constraint Programming

Constraint Programming is one form of declarative programming. Roughly speaking, in Constraint Programming (CP), a user provides an input and describes the expected output of their program rather than the algorithms to achieve this output from the input. Given the input and the description, some black-box gives an output that matches the description.

Example 2.1. In the Sudoku game, a player is given a 9×9 grid of numbers, subdivided into 9 grids of 3×3 cells.

Description: The aim is to fill this grid with numbers from 1 to 9 in such a way that the same number does not appear twice in the same row, column or 3×3 grid.

Input: The player is provided a few cells where numbers are already selected.
If we give the description and some input as in Example 2.1 to the “CP black-box”, we expect that it provides us with a solution to the Sudoku problem.

The idea of Constraint Programming is that the user does not need to provide an algorithm to solve the problem at hand. Instead, a generic CP solver (the black-box) solves the problem given the description and input. This description is formally known as the model, and the user is known as the modeler (rather than a programmer).

Following are some definitions to formalize the concepts we will use throughout the entire thesis.

**Definition 2.1.** A variable $v$ is a mathematical object that can take some value. We will mostly talk about Boolean variables and integer variables.

**Definition 2.2.** The domain of a variable $D(v)$ is a function mapping $v$ to the set of values it can take.

We call the lower bound (denoted $lb(v)$) the smallest value a variable can take, and the upper bound (denoted $ub(v)$) the biggest value it can take. A variable $v$ is said to be fixed when $lb(v) = ub(v)$, that is, there is only one value in $D(v)$.

**Definition 2.3.** A valuation, or assignment, $\theta : v \mapsto \{x \in D(v) \mid v \in v\}$ is a mapping over a set of variables $v$ that associates a value $x \in D(v)$ to each variable $v$ in $v$. We say that a variable $v$ is assigned a value $x$. We denote assignments as $\theta = \{v_1 \mapsto x_1, v_2 \mapsto x_2, \ldots, v_n \mapsto x_n\}$.

**Definition 2.4.** A constraint $c(v)$ is a logical function over a set of variables $v$ and their domains. It behaves as a mapping from valuations of the set of variables $v$ to true or false.

Note how the domain of a variable is just a specific case of a constraint that enforces a variable to take only some values (the ones in the domain). We will sometimes refer to them as “domain constraints”.

We say that a valuation satisfies a constraint $c$ if by substituting the variables in $c$ by their value in $\theta$, the logical expression $c$ holds. Otherwise, we say that the constraint is not satisfied. A constraint is unsatisfiable if no valuation exists such that the constraint is satisfied.

**Definition 2.5.** A Constraint Satisfaction Problem (CSP) is a pair $(v, C)$ where $v$ is a set of variables and $C$ is a set of constraints (including domain constraints).
We denote \( \text{vars}(O) \) the set of variables of object \( O \) (\( O \) can be, for example, a CSP or a constraint).

**Definition 2.6.** A **solution** to a CSP \((v, C)\) is a valuation \( \theta \) over \( v \) that satisfies all the constraints in \( C \).

**Example 2.2.** Formalizing the Example 2.1, we would have 81 integer variables with domains \([1..9]\). The constraints would be that, for each pair of variables \( a \) and \( b \) in the same row, column and subgrid, \( a \neq b \). Additionally, some variables will have a fixed value, given by the input.

But constraint programming is not limited to satisfaction problems. We can also solve optimization problems.

**Definition 2.7.** A **Constraint Optimization Problem (COP)** is a triple \((v, C, o)\) where \( v \) is a set of variables, \( C \) is a set of constraints (including domain constraints) and \( o : v \mapsto \mathbb{R} \) is a function to be minimized (or maximized). This function is called the **objective function**.

By analogy to domains, we call **lower bound** (denoted \( \text{lb}(o) \)) the smallest value the objective function \( o \) can take, and **upper bound** (denoted \( \text{ub}(o) \)) the biggest value it can take. Some examples of objective functions are minimizing the cost of a construction, maximizing the sales at a bakery, minimizing the length of cable used in an electric circuit or the number of vehicles to transport some goods. We also extend the notation of objective functions to take a valuation as a parameter. For example, \( o(\theta(v)) \) will evaluate function \( o \) over the set of variables \( v \) that are mapped into some value in their domain.

In the interest of being succinct, several definitions in this thesis will only talk about minimization. This does not lose generality of any definition, as a maximization problem with objective function \( f \) is equivalent to a minimization problem with the negated objective function \(-f\).

**Definition 2.8.** A solution \( \theta^* \) of a COP \((v, C, o)\) is **optimal** if no solution \( \theta \) such that \( o(\theta(v)) < o(\theta^*(v)) \) exists, for minimization problems.

We denote by \( \text{SOLS}(P) \) the set of all solutions to the CSP or COP \( P \).

The next subsection will provide information about the internal workings of CP solvers. First, SAT solvers will be discussed, as they are the core of some CP solvers, and in par-
ticular the one we use in this thesis. We will then explain how this evolves to become a CP solver. We will end this section with Clause Learning and Lazy Clause Generation.

### 2.1.1 SAT Solvers

The SAT problem is a well known NP-complete problem that consists in finding a satisfying valuation for a Boolean formula. Usually SAT problems are written in Conjunctive Normal Form (CNF), that is a conjunction of disjunctions of literals.

**Definition 2.9.** A literal is a variable or the negation of the variable. That is, $v$ and $\neg v$ are the literals for variable $v$.

**Definition 2.10.** A clause is a disjunction of literals. We can think of them as a specific type of constraint. Therefore the SAT problem is finding a valuation for all the variables of the problem such that all the clauses are satisfied.

**Example 2.3.** The following formula is a SAT problem in CNF form: $F = (a \lor b) \land (c \lor \neg a) \land \neg b$. This formula $F$ has 3 clauses and 3 Boolean variables.

A SAT solver solves this specific kind of problems. Although it is not always obvious how to model a problem as a SAT problem, many practical problems can be solved with this technology.

The DPLL algorithm [56, 57] is the core of SAT solvers. It combines search and propagation. Search is the task of choosing a value for a variable (e.g. $v = true$). This is often referred as “making a decision” or “branching on a variable”. A brute-force complete search algorithm would assign a value to each variable in turn and check whether the problem is satisfied by the current assignment or not. This obviously has a prohibitive cost, with a complexity of $\Theta(2^n)$ for $n$ variables, so propagation is interleaved with search. Propagation is the task of making inferences from the current partial assignment of the variables. For example, in Example 2.3, after assigning $a = true$, we can automatically determine that $c = true$, otherwise the clause $(c \lor \neg a)$ cannot be satisfied. After each decision, we say that we start a new decision level, and all the propagations that happen between this decision and the next one we make are said to happen at this decision level. This has complexity $O(2^n)$ for $n$ variables, but in reality often behaves substantially better.
SAT solvers perform only one type of propagation, called unit-propagation. In short, when all the literals of a clause are false except one, it forces the last one to be true, so that the clause is satisfied. If all the literals of a clause become false, then the current assignment does not satisfy that constraint. In this case, the solver will backtrack to the last decision and make the opposite decision (i.e. if the last decision was to branch on $b = true$, it will branch on $b = false$ and vice-versa), until all options are exhausted. After trying both options for a same variable, the solver needs to backtrack.

**Definition 2.11.** A conflict is a partial assignment that cannot be extended to satisfy the problem. In other words, it refers to the fact of having made a “wrong” set of decisions, leading to unsatisfiability.

<table>
<thead>
<tr>
<th>Algorithm 2.1 DPLL Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: function DPLL($v, C$)</td>
</tr>
<tr>
<td>2: if $v = \emptyset$ then</td>
</tr>
<tr>
<td>3: return true</td>
</tr>
<tr>
<td>4: if $\neg$ UNITPROPAGATION($v, C$) then</td>
</tr>
<tr>
<td>5: return false</td>
</tr>
<tr>
<td>6: $b \leftarrow$ CHOOSE($v$)</td>
</tr>
<tr>
<td>7: $v \leftarrow v \setminus {b}$</td>
</tr>
<tr>
<td>8: return DPLL($v, C \land (b = false)) \lor DPLL(v, C \land (b = true))$</td>
</tr>
<tr>
<td>9: end function</td>
</tr>
</tbody>
</table>

Algorithm 2.1 provides pseudo-code for DPLL. It is a recursive algorithm that terminates successfully when there are no more variables to assign, or unsuccessfully when the unit propagation fails (because the current assignment makes at least one clause become unsatisfiable).

Extensive research has been done to improve unit propagation, as it is the bottleneck of this algorithm [136, 148, 223]. We will not go into much detail on how this is done, since this is outside of the scope of the current thesis. In short, all literals reference the clauses where they appear, and each clause watches two literals in the clause. When one of the watched literals becomes false, a new one is looked for and watched. Eventually, either one of the two watches will watch a true literal, and the clause is then satisfied, or one of them will watch a false literal. In this second case, if the other watch points to an unassigned literal, unit-propagation will fix this literal to true. If both watches point to
false literals, it means all literals in the clause are false, and this is a conflict. The reason to have two watches is to convey the information of how many unassigned literals remain. If that number is one, then the unit propagation will assign that literal to true, in order to satisfy the constraint.

Another key element of the algorithm is the order of choosing the variables (the call to \texttt{CHOOSE}(v)). Different orders yield different running times for the algorithm, depending on how much a decision propagates to the clauses. Similarly, if we are only collecting one solution to the SAT problem, the order in which the values true and false are tried for each variable can also greatly impact the runtime. Choosing the order of variables and the order of the values for each variable to branch one is known as the search strategy.

A conference is run annually about the topic of SAT solvers, called the \textit{International Conferences on Theory and Applications of Satisfiability Testing (SAT)}. A SAT competition is also run annually to compare the state of the art solvers in industrial, handmade and random instances of SAT problems.\footnote{Available at http://www.satcompetition.org/}

### 2.1.2 Constraint Programming Solvers

#### 2.1.2.1 SAT Generalization

Constraint Programming is a generalization of SAT solving. It provides a more expressive language with more variable and constraint types and a generalization of unit-propagation to these new variables and constraints. Namely, CP introduces integer variables and operations to create relations between them to constrain their values (comparison and arithmetic operations). For example, it is possible to write constraints such as \(a + b + c = d\) or \(a < b\) and not just clausal constraints as in SAT. Note that because the “=” operator exists in CP, we will use \(\equiv\) to denote semantic equality to avoid ambiguity.

Because domains in CP are larger than in SAT, we have a dominance relation between domains. We say that a domain \(D\) of variable \(v\) is \textit{stronger} than a domain \(D'\) (denoted \(D \subseteq D'\)) if \(D(v) \subseteq D'(v)\).

Analogously to unit-propagation over Booleans in SAT solvers, CP solvers do prop-
aggregation on the integer variables. For example, given two variables $a$ and $b$ both with domain $D(a) = D(b) = [1..9]$, and a constraint $a < b$, after making the decision that $a = 3$, the CP solver will automatically propagate that the domain of $b$ is $[4..9]$. This particular example is often called finite domain propagation.

Each constraint in CP is implemented using a propagator. These are specialized algorithms within the solver that can do strong reasoning over the current assignment of the variables to prune the search space sooner (just like unit-propagation in SAT).

**Example 2.4.** Consider the constraint $C \equiv a + b + c = d$ where $D(a) = D(b) = D(c) = [1..9]$ and $D(d) = [7..14]$. After making the decisions that $a = 4$ and $b = 5$, a clever algorithm would figure that $D(c)$ is now $[1..5]$ and $D(d) = [10..14]$. This is because all other pairs of values for $(c, d)$ in their original domains would not satisfy the constraint at hand.

This leads to the following definition (equivalent to the definition by [192]):

**Definition 2.12.** A propagator is a function $f$ over domains $D$ such that $f(D) \subseteq D$ (i.e. $f$ is contracting), $D \subseteq D' \Rightarrow f(D) \subseteq f(D')$ (i.e. monotonic) and, for any problem $P = (v, C)$ with domain constraint $D$, $\text{SOLS}((v, C \land f(D))) = \text{SOLS}((v, C))$ (i.e. no solution is lost).

In short, propagators should remove values from the domains of variables (if possible) whilst maintaining all solutions to the problem. They are filtering algorithms, intended to reduce the search space. As in the Example 2.4, the search will never need to try the values $[6..9]$ of $c$, because there is no solution within those values. It is worth noting that propagators do not need to completely filter the domains of the variables. Some propagators may choose to not do some propagation if it is too costly in time. Note that if this is the case, the solver will need to then run checks on the solutions to ensure that the solution satisfies all the constraints, as these may not be enforced by the propagators.

Apart from integer variables, CP has other types of variables, depending on the expressiveness of the modeling language used. All CP systems support Boolean variables as they are equivalent to integer with domain $\{0, 1\}$, and they support logic operators over Booleans. In addition to this, some may support set variables [153], string variables [9] or graph variables [72] as we will see later.
2.1.2.2 Predicates and Global Constraints

**Definition 2.13.** Predicates are a tool to capture complex constraints in an abstract manner.

Predicates are the equivalent to a procedure in imperative programming. They allow the modeler to capture a constraint that is, conceptually, only one constraint but due to its complexity needs to be written as a composition of multiple constraints.

For example, if we want a set of variables $A = \{a_1, a_2, \ldots, a_n\}$ to all take different values, we need to write a conjunction of disequalities. We can capture this in a predicate so that conceptually we have an alldifferent constraint, but it is in fact just a conjunction of simpler constraints.

**Definition 2.14.** Global constraints (or simply “globals”) are a specific type of constraints that, because they are prominent in Discrete Optimization problems, are built-in/standard across most CP systems, and use specialized algorithms to perform propagation.

Global constraints [34] can be modeled with basic constraints (actually, alldifferent is an example of a global constraint). We call this approach decomposition of the global. But, typically, CP solvers will have specialized algorithms for them. This is because these constraints appear often, and having fast propagation associated with them can be extremely beneficial in accelerating solvers.

**Example 2.5.** Let us consider a simple alldifferent example with 4 variables $a$, $b$, $c$ and $d$ all with domain $[1..3]$. Clearly, because each variable can only take 3 values and there are 4 variables they cannot be all different. Nonetheless, a decomposition approach would not be able to detect this quickly. Indeed $\text{alldifferent}([a, b, c, d]) \equiv a \neq b \land a \neq c \land a \neq d \land b \neq c \land b \neq d \land c \neq d$.

If the search decides that $a = 1$, then automatically $D(b) = D(c) = D(d) = [2..3]$. Then if $b = 2$, we have $D(c) = D(d) = \{3\}$. At this point failure is detected because it is not possible for the constraint $c \neq d$ to hold anymore. Therefore there is no solution for this assignment. But this will have to be done for all sequences of numbers $\{1, 2, 3\}$ before the CP solvers detects that the constraint is unsatisfiable.

For this reason, an important topic of research in CP is the development of new or improved propagators for global constraints. And indeed, the alldifferent propagator...
by Régis [179] would have detected unsatisfiability immediately in Example 2.5. In this thesis, we present propagators for some global constraints to improve the pruning in the solver.

There is a catalog of global constraints [25] that provides a thorough list and analysis of these global constraints. A survey was also presented by Régis [182]. It is worth noting that not all solvers implement all global constraints, and even if they do, they might have different propagators. For this reason, solver-agnostic languages like MiniZinc [153] provide a decomposition to all the globals, in case the selected solver does not provide an implementation.

### 2.1.2.3 Optimization Problems

The most used technique to solve COPs in CP is Branch-and-Bound [128]. This technique keeps a record of the “best” objective value (either the smallest for minimization or the biggest for maximization) found so far.

The value of the objective function is stored in a variable $\text{obj}$, and during search the lower and upper bounds of this variable are changed until eventually the variable becomes fixed. The goal of Branch-and-Bound is to update the upper bound of the variable $\text{obj}$ (or the lower bound if maximized) to match the best solution found so far. During the search, after making some decisions, it may be the case that the objective function $o$ changes bounds in such manner that $\text{lb}(o) > \text{ub}(\text{obj})$, in which case we know no better solution can be found in the current search space. Therefore, the solver can backtrack and we avoid searching part of the search space.

**Example 2.6.** Consider a problem with objective function $o(a, b, c) = a + 2b - 4c + d$ to be minimized, and only one constraint that is $C \equiv (b \neq 3) \lor (c = 0)$. The domains are $D(a) = D(b) = D(c) = D(d) = [1..5]$. Assume we had found the solution $\theta = \{a \mapsto 3, b \mapsto 2, c \mapsto 3, d \mapsto 1\}$, with objective value 1. We now restart the search and choose $a = 4$, followed by $b = 3$. The constraint $C$ propagates and we infer that $c = 0$. Now the lower bound of $o$ is $4 + 3 - 0 + \text{lb}(d) = 8$, which is higher than the best solution found so far. Therefore we know there is no point in exploring further this region, as we are not going to find better solutions than $\theta$. 
2.1.3 Clause Learning

Clause Learning, also known as “no-good learning” or simply “learning” is a powerful technique that, conceptually, allows solvers to learn from their mistakes. This technology comes from SAT solvers. The idea is that during search, the solver may take the wrong decisions and arrive to a conflict. But it can be the case that only a subset of those decisions (or what those decisions implied) caused the conflict, and the same situation may repeat itself in another branch of the search, thus leading to the same conflict.

**Example 2.7.** Consider the SAT problem \( P = (\{a,b,c,d,e,f,g\}, C_1 \land C_2 \land C_3 \land C_4) \) where \( C_1 = (\neg b \lor \neg c) \), \( C_2 = (\neg d \lor e) \), \( C_3 = (\neg d \lor c \lor \neg e) \) and \( C_4 = (f \lor g) \) are the only four clauses in the problem. Figure 2.1a shows an example of search path for this problem.

Making the decision that \( a = \text{false} \) or \( a = \text{true} \) changes nothing, so whatever we encounter on the first case, we will also encounter it in the second case. Let us choose \( a = \text{false} \) first. Then \( b = \text{true} \). This, due to clause \( C_2 \), implies that \( c = \text{false} \). The next decision we make is \( f = \text{false} \) which, by \( C_4 \), implies that \( g = \text{true} \). Now, we choose \( d = \text{true} \). Due to \( C_2 \) we now have \( e = \text{true} \). But due to \( C_3 \) we would have that \( e = \text{false} \). So we get to a conflict.

We can see that the conflict is due to the fact that \( c = \text{false} \) and \( d = \text{true} \). Therefore, we learn a clause \( C_L = (c \lor \neg d) \) that we add to \( P \). We will explain later how to derive this learnt clause.

Later, we backtrack to the first level, and choose \( a = \text{true} \). Figure 2.1b. We then choose \( b = \text{true} \), which implies \( c = \text{false} \) by \( C_1 \). Note how we are in the exact same situation as before, and we could repeat the same mistake, and waste a lot of time getting the same conflicts over and over. However, thanks to \( C_L \), we also get the propagation that \( d = \text{false} \) at the second decision level, thus avoiding the conflict.

2.1.3.1 Deriving Learnt Clauses

In order to derive the learnt clause from a conflict, we use the implication graph. This is actually represented in Figure 2.1a as the green, blue and red rectangular nodes.

A cut \( K = (A, B) \) on the implication graph separates 2 sets of nodes, \( A \) and \( B \). Any cut such that all decision nodes are on the same side, and the conflict node is on the other side, is a valid cut to induce a learnt clause. Assume the decisions are on the set \( A \) and
the conflicts are on set $B$. In Figure 2.1a, two such cuts (dashed orange and blue lines) are given. Let $K_E$ be the set of edges of the form $e = (a, b)$ such that $a \in A$ and $b \in B$. These edges “traverse” the cut $K$. Let $A' = \{a \mid (a, b) \in K_E\}$. Then, the conjunction $\land_{a_i \in A'} a_i$ of the $a$ nodes is enough to cause the failure.

This means that if all the nodes in $A'$ are true at the same time, the failure will happen again, regardless of what decisions led to this situation. Looking at the example in Figure 2.1a, for the blue cut, we would have $(\neg c \land d) \Rightarrow fail$ and for the orange cut we obtain $(b \land d) \Rightarrow fail$. Both are enough conditions to cause the failure.

Nonetheless, this does not produce a clause. Note how the derived terms are a conjunction, and a clause is a disjunction. This disjunction is called a no-good. In order to obtain an equivalent clause, we build an equivalent conjunction. In the example we would obtain $(c \lor \neg d)$ and $(\neg b \lor \neg d)$. These are called learnt clauses. When adding either of these clauses to the problem $P$ in Example 2.7 we obtain the behavior seen in Figure 2.1b, that is, the conflict is avoided. In the example we chose to keep $C_L = (c \lor \neg d)$, but both are valid. Note how adding this clause to the problem does not remove any solution: it is a redundant constraint.

**Definition 2.15.** For a CSP $P = (v, C)$, a redundant constraint $c$ over a set of variables $v' \subseteq v$ is a constraint that was already implied by the constraints in $C$ implicitly (and thus does not remove any solution from $P$).

Although there are multiple choices for learnt clauses, usually SAT solvers will only
add one to the clause store. The reason is that these clauses need to be propagated, and therefore can slow down the unit-propagation process. Indeed, there is no guarantee that these learnt clauses will ever be used. For that reason, periodically, the store of learnt clauses needs to be cleaned to only keep the most used ones (that is, the ones that propagate often). We will not discuss cleaning techniques as this is out of the scope of this thesis.

2.1.3.2 Unique Implication Point

As seen earlier, there may be multiple choices for choosing the clause to be added to the problem. Usually, the Unique Implication Point (1UIP) is chosen as it provides a good tradeoff between quick computation and reusability [65, 200, 224].

The algorithm to derive 1UIPs keeps track of a conjunction of literals that caused the failure and repeatedly substitutes each literal \( l \) with its cause, that is the literals that implied \( l \) in the first place. The algorithm starts with the conflict as the only literal (i.e., the false literal), and only stops when exactly one literal is at the same decision level as the conflict, this is the reason why it is called 1UIP. The order of the substitutions is the order of the propagations reversed. This means that a difference in the propagation engine can cause a difference in the resulting UIP. Figure 2.2 shows the same implication graph as Figure 2.1a, laid out in a more appropriate form to show the 1UIP derivation.

**Example 2.8.** Continuing with Example 2.7, we show the process of building the no-good in Figure 2.2. First, the no-good is \( n_g = \text{false} \). The false literal gets then substituted with the literals that implied it, that is \( n_g = e \land \neg e \). Because \( \neg e \) was the last propagation, we then substitute \( \neg e \) with the literals that implied it, so \( n_g = e \land \neg c \land d \). As there are still two literals in the current decision level (\( e \) and \( d \)), we continue the substitution. The last propagation was the one implying \( e \), and therefore it is the next literal to be substituted. We obtain \( n_g = d \land \neg c \land d = \neg c \land d \). Only one literal is at the current decision level (\( d \)), therefore we stop here, and the learnt clause is \( C_L = \neg n_g = c \lor \neg d \).

Note that, in practice, solvers won’t have \( e \) and \( \neg e \) as true literals at once. Therefore the starting no-good to perform this derivation is in fact the second step shown in Figure 2.2, \( n_g = e \land \neg c \land d \).
2.1 Constraint Programming

2.1.3.3 Backjumping vs. Backtracking

Usually, the DPLL algorithm (shown in Algorithm 2.1) would backtrack after each decision, to make a different decision at the previous decision level. But reaching a conflict, it is not necessarily only the last decision that caused it. Clearly, in Example 2.7, it was not just the decision $d = \text{true}$ that caused the conflict.

Backjumping [61] is a technique used after discovering a learnt clause, by which, instead of going back to the previous level, we go back to the earliest level $l_j$ where the learnt clause could have propagated. This idea helps performance, as otherwise the learnt clause would need to be propagated repeatedly at every level between the level where the conflict happened and level $l_j$. As the number of learnt clauses grows this adds more overhead. In addition to this, having this backjump acts as a restart where new information is taken into account (in this case the propagation of the clause). This in turn can allow for better decisions to be taken later as the new propagation may affect the search strategy.

Example 2.9. With the problem $P$ in Example 2.7, the decision done at level 3 ($g = \text{false}$) has nothing to do with the conflict. This can be clearly seen in Figure 2.2, as the level 3 is never touched by the algorithm. For this reason, with backjumping we would go back to level 2, make the exact same decision (that is, $b = \text{true}$) and continue the DPLL algorithm. The learnt clause will propagate at level 2 and the rest of the search can take place as usual.
2.1.3.4 Extending Clause Learning to Constraint Programming

The principle of deriving clauses (1UIP) is exactly the same in CP as in SAT. Nonetheless, clause learning requires learning from clauses, and in CP clauses are not necessarily obvious in the problem. Indeed, as we saw in Section 2.1.2, CP generalizes SAT in such a manner that literals do not necessarily appear in the model, specially when we look at constraints involving integer variables.

Because clause learning relies on a representation of the model as literals, Constraint Programming solvers will create literals corresponding to different possible assignments of a variable. Namely, for an integer variable $v$, CP solvers that use Lazy Clause Generation will (usually) create $2 \times |D(v)|$ literals. We note literals $[v \oplus x]$ where $x$ is a value, and $\oplus$ an arithmetic operation. The literals for a variable $v$ in a CP solver are, typically, $L(v) = \{[v = x] \mid x \in D(v)\} \cup \{[v \leq x] \mid x \in D(v)\}$.

When a decision is made over an integer variable $a$, for example $a = 5$, a literal $[a = 5]$ is added to the implication graph, just like for the SAT examples we saw earlier. We say that such literal is set to true. Of course, the literals are always consistent between them. For example if $[a = 5]$ is true, the literals $[a \neq 5]$ or $[a \leq 4]$ are false, whereas $[a \leq 6]$ and $[a \geq 3]$ are true.

The more delicate part is the propagations. By looking at Figure 2.2 we see that we need to know what triggered a propagation to then replace it in the no-good to derive the clause (c.f. Example 2.8). But in CP these propagations are “opaque” to each propagator. This prompts the introduction of Lazy Clause Generation (LCG) [155]. In this technique, propagators need to be extended to explain their propagation as a clause. Every time a propagator changes the domain of a variable, the propagator must provide an explanation for the propagation. This is often a considerable extra effort for the implementation of the propagator, but is key to allow clause learning in CP.

**Definition 2.16.** An explanation is a conjunction of literals that is sufficient to trigger an assignment over some other literal that has been propagated within a propagator.

Propagators might also detect conflict, when the domain of some variable is inconsistent (i.e. the propagator needs to set the variable $v$ to some value $d \notin D(v)$ to maintain
the soundness of the constraint). This is equivalent to unit-propagation detecting a conflict. In such cases, the propagator must also provide an explanation for failure. This is typically the explanation as to why it intended to set \( v \) to the forbidden value \( d \), conjoined with the literal \( [v \neq d] \).

Just as we saw in Example 2.8, the explanations provided by the propagator are added to the implication graph. The explanations are what is used to perform the successive substitutions during the conflict resolution process as illustrated in Figure 2.2. Example 2.10 provides an example, based on Example 2.7, that illustrates the explanations provided by the propagators.

**Example 2.10.** Consider the CSP \( P' \) defined as follows. The set of variables of \( P' \) is \( v = \{a, \ldots, g\} \) with domains \( D(a) = [1..9], D(b) = [1..8], D(c) = [1..3], D(d) = [1..7], D(e) = [1..9] \) and \( D(f) = D(g) = [0..1] \). The constraints in \( P' \) are \( C'_1 \equiv b + c \geq 5 \), \( C'_2 \equiv d = e - 1 \), \( C'_3 \equiv d + c + e = 8 \) and \( C'_4 \equiv f + g = 1 \).

Assume the search starts by trying \( a = 1 \). The literal \( [a = 1] \) is added to the implication graph. This produces no propagation as \( a \) is not involved in any constraint. At level 2, the decision is \( b = 4 \), which implies that \( c > 1 \) (or in other words \( \neg[c \leq 1] \)) as otherwise \( C'_1 \) would fail. So the implication \( [b = 4] \Rightarrow [c \geq 2] \) is added to the implication graph by the propagator of \( C'_1 \).

The next decision is \( f = 0 \), thus \( [f = 0] \Rightarrow [g = 1] \) is added by \( C'_4 \). We now decide that \( d = 5 \).

Because of \( C'_2 \), this means that \( [e = 4] \) is now true, and the propagator adds \( [d = 5] \Rightarrow [e = 4] \).

But also, because of \( C'_3 \), \( [e \leq 1] \) is true, since \( d \) is now 5, \( c \) is either 2 or 3 and \( d + c + e \) must add to 8. But of course, \( [e = 4] \) and \( [e \leq 1] \) cannot be true at the same time. Therefore there is a conflict and the explanation for the conflict is \( [d = 5] \wedge \neg[c \leq 1] \wedge \neg[e \leq 1] \Rightarrow \text{fail} \).

The implication graph for this example is actually the same as for Example 2.7, and therefore the implication graph and conflict resolution for the present one is also shown in Figure 2.2. The learnt clause in this example is \( C'_L \equiv [c \leq 1] \lor [d \neq 5] \).

Because there may be too many literals to comfortably fit in memory and this may affect the performance of the solver, the literals may be created lazily. That is, they may be created only when needed. In practice, this means that solvers will use some criteria to decide for which variables they should create all the literal before-hand and for which variables they should create literals lazily. The lazy literals are only created when they
are needed in an explanation and they are be mapped to the integer variable they relate to (for example using a hash-table). Solvers like CHUFFED [48] will create them lazily for variables with big domains (“big” being a parameter) and eagerly for the rest.

2.2 Graphs in Constraint Programming

In the previous section we saw that Booleans and integers are the main variables types in Constraint Programming. Nonetheless, often we wish to reason on graphs.

2.2.1 Definitions and Notation

Definition 2.17. A graph \( G = (V, E) \) is a pair composed of i) a set of nodes (or vertices) and ii) a set of edges \( E \subseteq V^2 \). A graph is directed if the edges have a direction (that is, a non-symmetrical relation between its endpoints) or undirected otherwise (in which case \( (a, b) = (b, a) \)).

If an edge \( (a, b) \) exists in a graph, we say that \( a \) and \( b \) are neighbors and qualify them as adjacent. A directed edge \( e = (t, h) \), sometimes called “arc”, goes from \( t = \text{tail}(e) \) to \( h = \text{head}(e) \), and is drawn ‘\( \longrightarrow \)’. The inverse of a directed edge \( e = (t, h) \) is \( e^{-1} = (h, t) \). We say that an edge is outgoing from its tail and incoming to its head. We call the incoming degree (resp. outgoing degree) of a node \( n \), denoted \( \text{ideg}(n) \) (resp. \( \text{odeg}(n) \)), the number of incoming (resp. outgoing) edges to \( n \). For undirected graphs, we use either word incoming or outgoing equivalently to refer to the edges containing a certain node. Accordingly, we define the degree, denoted \( \text{deg}(n) \) as the number of outgoing edges of \( n \).

Definition 2.18. The inverse (noted \( G^{-1} \)) of a directed graph \( G = (V, E) \) is defined as \( G^{-1} = (V, \{e^{-1} \mid e \in E\}) \).

Definition 2.19. A weighted graph \( G = (V, E, w) \) is a graph to which a function \( w : E \mapsto \mathbb{R} \) is associated. For any edge \( e \in E \), \( w(e) \) is called the weight of the edge \( e \). Unless otherwise specified, the weight of a graph is defined as \( w((V, E)) = \sum_{e \in E} w(e) \).

Definition 2.20. Let \( G_1 = (V_1, E_1) \) and \( G_2 = (V_2, E_2) \) be two graphs. If \( V_1 \subseteq V_2 \) and \( E_1 \subseteq E_2 \), then we say that \( G_1 \) is a subgraph of \( G_2 \), and we denote it \( G_1 \subseteq G_2 \).
Definition 2.21. A path $p$ from $n_1$ to $n_l$ is an ordered sequence of edges of the form $p = \langle (n_1,n_2), (n_2,n_3), \ldots, (n_{l-1},n_l) \rangle$. The nodes of a path are the endpoints of all the edges in the path.

Definition 2.22. A simple path $p$ is a path where no nodes are repeated.

Definition 2.23. The length of a path in an unweighted graph is the number of edges in the path. For a path in a weighted graph, when we talk about “length of a path” we are referring to the weight of the path (i.e. the sum of the weight of the edges in the path), by analogy to real world distances, and we will explicitly say “number of edges” otherwise.

Note that, by definition, in an undirected graph a path is simply a sequence of edges where consecutive edges share a node. For example, in an undirected graph $G_u = (\{a,b,c\},\{(a,b),(c,b)\})$, $p = \langle (a,b),(c,b) \rangle$ is a valid path, since $(c,b) = (b,c)$. This is not the case for a directed graph: the direction of the edges must be consistent, i.e., for two consecutive edges, the head of the first must be the tail of the second, that is, $p = \langle (a,b),(c,b) \rangle$ is not a path, since $(c,b) \neq (b,c)$.

Definition 2.24. A graph is connected if between all pairs of nodes in the graph there exists at least one path.

Definition 2.25. For any connected undirected graph $G$, an articulation is a node $n$ such that, if $n$ is removed from $G$, $G$ becomes disconnected.

Definition 2.26. For any connected directed graph $G$, a dominator of node $t$ from node $s$ is a node $d$ such that all paths from $s$ to $t$ go through $d$. We say that $t$ is dominated by $d$ from $s$.

Definition 2.27. For any connected directed graph $G$, the immediate dominator of node $t$ from node $s$ is the dominator $d$ of $t$ from $s$ that is the closest to $t$ in all the paths from $s$ to $t$. Here “closest” refers to the smallest number of nodes between a dominator of $t$ and $t$ itself.

Definition 2.28. For any connected graph $G$, a bridge is an edge $e$ such that, if $e$ is removed from $G$, $G$ becomes disconnected.

Definition 2.29. A circuit, or cycle, in a graph is a path $p$ such that no nodes are repeated except for the first node, which is also the last (i.e., the path “loops” on that node).
Definition 2.30. A graph is \textit{acyclic} if it contains no circuits.

Definition 2.31. A \textit{tree} is a connected acyclic graph.

In a directed graph, a tree must also have a \textit{root} from which there is exactly one path to all the other nodes. We will often call these \textit{directed trees}.

Definition 2.32. A \textit{directed acyclic graph (DAG)} is a directed graph with no directed cycles.

The difference between trees and DAGs is that DAGs allow undirected cycles but not directed cycles. Neither of them are allowed in trees.

2.2.2 Graph Variables and Constraints

As these data structures appear often in many combinatorial problems, it is natural to consider using them as variables as well. We could imagine constraints over graphs that force them to be connected, or form a tree (connected acyclic graph), or contain some nodes only if they don’t contain some others. Any arbitrary constraint could be imagined.

For this reason Dooms et al. [72] created a graph computation domain in Constraint Programming, called CP(GRAPH). The idea was to create graph variables and a series of graph-related global constraints to augment Constraint Programming. This would provide more expressiveness in CP: the modeler would have access to a self-contained graph variable to pass as a parameter to constraints rather than having to chose an encoding for the graph.

Following the definitions of Dooms et al. [72], if $G_1$ and $G_2$ are two graphs, such that $G_1 \subseteq G_2$, then $[G_1, G_2]$ is a graph \textit{interval}, by analogy to integer intervals. It contains the set of graphs $G$ such that $G_1 \subseteq G \subseteq G_2$. This defines the domain of a graph variable $G$ (note that a graph variable takes a graph as a value, and we will often refer to the nodes or edges of a graph variable as a shorthand for the nodes or the edges of the value of a graph variable). In addition, there are a few basic constraints that were introduced for CP(GRAPH). Namely, the user can constrain the set of nodes (or edges) of a graph to be some set that may have its own constraints (e.g. cardinality, etc.). More advanced constraints include, but are not limited to, enforcing a graph to be a subgraph of another,
enforcing a node to have a specific set of neighbors, enforcing the graph to be connected, to be a path, to be a circuit, to be a tree, or to have some weight.

Earlier, Beldiceanu [20] produced a list of global constraints over graphs, but without giving much detail about how these should be implemented. The definitions and vocabulary to be used when referring to graph was further formalized [24], but no propagators were suggested for these constraints. Furthermore, the doctoral thesis by Dooms [69] provides a well structured survey over the most essential constraints over graphs, and it was shown how it is possible to elegantly combine them to achieve more complex ones. For example, the tree constraint, as presented by Dooms et al. [72], is simply a disjunction of the connected and acyclic constraints. The implementation and evaluations of the latter were done in GECODE [193]. Although this decomposition approach is elegant, we will see that there is a need for propagators, as CP(GRAPH) performs substantially worse than approaches where globals are implemented through dedicated propagators. It is worth noting that CHOCO 3.3.0 [172] provides the most up to date implementation of the CP(GRAPH) framework, with graph variables as such. We will see in the technical chapters some comparisons against these systems.

The more recent thesis by Fages [84] investigated propagators over graphs. Their work substantially improved previous work by updating propagators that had previously been introduced, for example, for the tree constraint [22, 85]. They also discuss the arbo and antiarbo constraints (directed tree and reversed directed tree) as well as path and circuit constraints. We will see these in detail in Chapters 3 to 6. Their implementation [84] is actually the aforementioned one in CHOCO.

There are other relevant research outcomes in constraints over graphs. As such, there is a number of global constraints that exists over graphs, although they were not necessarily presented using the graph variables from CP(GRAPH). We will talk about the previous work on some globals in the sections of this thesis where these constraints are relevant.
2.2.2.1 Our Modeling Choice

For most of this thesis, we will use the MiniZinc language as a modeling language and Chuffed as our CP solver. Neither of them has built-in support for graph variables as in CP(Graph) or Choco. Thus we choose an equivalent Boolean-based encoding.

For a graph variable $G = (V, E)$, we need to define its domain. We will have a graph $G = (V, E)$ that defines the upper bound of $G$. By definition of the graph variables, $G$ must be a subgraph of $G$. We encode $G$ using two sets of Booleans. The first set $v_V$ contains a Boolean for each node of $G$, and there is a bijective mapping between the set $V$ and the set $v_V$. The second set $v_E$ of Booleans is defined similarly with a mapping from $v_E$ to $E$. Each of these Booleans will be subscripted with the node or edge they map to. The semantics of this encoding is the following:

\[
\forall b_v \in v_V, \ b_v \iff v \in V \\
\forall b_e \in v_E, \ b_e \iff e \in E
\]

That is, if a Boolean is true, the corresponding node or edge is in $G$, and we say that such a node or edge is mandatory (drawn ‘‘’ or ‘___’ respectively). If the Boolean is false, the corresponding node or edge is not in $G$ and we say that such a node or edge is forbidden (drawn ‘..’ or ‘……’ respectively). Until the Boolean is not assigned a value, the membership of the edge or node to the graph is unknown. We say that such an edge or node is unfixed (drawn ‘O’ or ‘_.._’ respectively). During the search, nodes and edges that are not forbidden are named available. An example is given in 2.3.

Figure 2.3: Example of drawing convention for graph variables.

Note that the Boolean encoding is of size $|V| + |E|$ as it encodes only the variables seen by the solver. The graph itself is encoded by $G$, for which any classic graph structure (e.g.
adjacency list) can be used.

As for the lower bound of the graph variable $G$, it suffices to fix the Boolean variables corresponding to the lower bounding graph, to ensure that at least all the nodes/edges in the lower bounding graph are in $G$. For example, if we want to find a subgraph of $\mathcal{G}$ containing a node $n$, we just need to define the corresponding graph variable $G$ and set $b_n = \text{true}$.

### 2.3 Other Discrete Optimization Techniques

Like Constraint Programming, there are multiple other techniques to solve discrete optimization problems that arise in the real world. In this section we will discuss two specific techniques that we will use later in this thesis.

#### 2.3.1 Mixed Integer Programming

In this section we are going to introduce the basics of Mixed Integer Programming (MIP) and Linear Programming (LP). We will not go into much detail about the internal workings of a MIP/LP solver, as this is outside of the scope of this thesis, and providing thorough details about this technology is a task worthy of a book, such as the ones by Schrijver [191], Sierksma [198], Pochet and Wolsey [170], or Gass [96].

#### 2.3.1.1 Linear Programming

**Definition 2.33.** A linear program (LP) is a mathematical model defined by a set of linear functions over a set of real variables (i.e. in $\mathbb{R}$). The canonical form of a linear program is:

$$
\text{maximize } c^T \cdot x \\
\text{such that } A \cdot x \leq b \\
\text{and } x \in \mathbb{R}^{\mid x \mid}_{\geq 0}
$$
where all the bold symbols are vectors, \( x \) contains a set of non-negative real variables, \( c \) and \( b \) are two vectors of real numbers, and \( A \) is a matrix of real numbers.

Similarly to CP, the first line of an LP is the objective function (to be maximized in this canonical form, but can be minimized by negating all the coefficients in \( c \)). Equivalently to Constraint Programming, the goal is to find a valuation for vector \( x \) such that all the linear constraints given by \( A \cdot x \leq b \) are satisfied, and all the elements in \( x \) are non-negative. Although this canonical form may seem restrictive, there are techniques to overcome some of the limitations. For example, it is perfectly possible to use variables with negative domains, by encoding them as a difference of two positive variables. Note how each inequality in the system defines a half-space (e.g. in 2D, each inequality splits the Cartesian space in 2 regions, one where the inequality is “satisfied”, and one where it is not). Figure 2.4 shows a geometrical view of a linear program. A key element of linear programming is that the equations in the linear program must define a convex set. Intuitively, for this to be the case, the intersection of the half-spaces defined by the inequations must not be empty, and this create a finite polytope.

![Figure 2.4: Geometric view of a linear program in 2D](image)

Unlike all problems that we will be tackling in this thesis, this is not a discrete optimization problem. Indeed, the fact that the values in \( x \) are real numbers makes the problem continuous. As such, Khachiyan [120] proved that the Ellipsoid Method to solve LPs is a polynomial time algorithm. Karmarkar also proposed a polynomial time algorithm...
to solve LPs [3]. Nonetheless, the most used method is the Simplex algorithm by Dantzig [53]. Although its worst case complexity is exponential [122], in practice the Simplex algorithm performs better than the previously mentioned ones and thus it is widely used to solve most LPs.

**The Simplex Algorithm** In order to use the Simplex algorithm by Dantzig, a small transformation needs to be done to the LP. Namely, the Simplex algorithm does not support explicit inequalities like $A \cdot x \leq b$. Instead, these have to be converted into equivalent equalities $A \cdot x = b$ by using slack variables. For example, an inequation such as $x_1 + 3x_2 \leq 9$ becomes $x_1 + 3x_2 + s_1 = 9$, where $s_1$ is a non-negative slack variable. Both are equivalent.

Once the problem is written in this form, the Simplex algorithm can be applied. The principle of the algorithm is to find a solution and iteratively improve it (in terms of the objective function) until no improvement can be done. The key to this algorithm is that the objective function is maximal at, at least, one of the extreme points (i.e. the corners) of the polytope [151]. The Simplex algorithm finds which extreme point of the polytope conveys an optimal solution.

To start, the Simplex algorithm needs a first solution. An easy way to find it is to simply express a subset of the variables (called basic variables) in terms of the other variables (called non-basic), as equalities. Then it suffices to set all the non-basic variables to 0, and let the basic variables take a value that satisfies the equations (their value will only depend on $b$ and $A$, which are known). This produces a basic feasible solution. Note that it is important to choose the basic variables in such a way that their assignment is non-negative.

Once we have a basic feasible solution, the Simplex algorithm will make a move to go to an adjacent solution. To do so, it swaps a non-basic variable with a basic one, and re-expresses all the equations accordingly. Setting the non-basic variables to 0 will produce a new basic feasible solution. This move is done in an intelligent manner to achieve two goals. First, the new solution must have non-negative values, as before (one must take care of inspecting the values of $b$, which are known, to guarantee this). More importantly,
this move needs to be made in a manner such that the objective function is not worsened (i.e. either it improves, or remains with the same value). Indeed, it has been shown [151] that any vertex of the polytope will have a neighboring vertex that will improve the solution (or have a solution of same objective value), unless it is optimal. The terminating condition for the Simplex algorithm is when no better solution can be found.

Figure 2.5 shows a small example of the objective function moving between vertices until reaching the optimal solution.

Figure 2.5: Objective function progressing through Simplex algorithm for objective function maximize $x_1 + 0.22x_2$

2.3.1.2 From Linear Programs to Mixed-Integer Programs

Mixed-Integer Programming is similar to Linear Programming, except for one detail: some variables are required to take an integer value. This problem was shown to be NP-complete by Karp [115].

**Definition 2.34.** A **Mixed Integer program (MIP)** is a mathematical model defined by a set of linear functions over a set of variables in $\mathbb{R}$ where at least one variable is in $\mathbb{N}$. The canonical form of a MIP is:

$$\begin{align*}
\text{maximize} \quad & c^T \cdot x \\
\text{such that} \quad & A \cdot x \leq b \\
& x \in \mathbb{R}^{|x|} \\
& \exists x^I \subseteq x, x^I \in \mathbb{N}^{|x^I|}
\end{align*}$$

where all the bold symbols are vectors, $x$ contains a set of variables (a mixture of real and integer
variables, the later noted $x^1$, $c$ and $b$ are two vectors of real numbers, and $A$ is a matrix of real numbers. The last equation is called the integrality constraint. Note that $\mathbb{N}$ includes 0.

MIP problems are ubiquitous in the Operations Research and as such, a good number of solvers exist to solve them. Amongst them, CPLEX12.4 [50] and Gurobi [156] show the best performance, and SCIP [99] claims to be the best amongst the non-commercial solvers. We say that, for a given MIP $P$, the same problem $P$ without the integrality constraint is the linear relaxation (or “LP relaxation” or simply “relaxation”) of $P$.

There exist several techniques to solve MIP problems, the most popular ones being Gomory cuts [16], Branch-and-Bound methods [130] (we already saw them in CP, for optimization) or Branch-And-Cut methods [157] (a combination of the two previous ones). The idea of all these methods is the same. First, find a solution $S$ for the relaxation of the MIP problem at hand. Then, if all the variables that were supposed to be integer are integers in $S$, then the current solution is a feasible solution to the MIP problem. If not, ensure that the current solution $S$ is removed from the solution space (that is, the polytope), and the process is repeated until all required variables are integral.

Briefly, Gomory cuts are a technique by which a new linear inequality is added to the model that ensures the current solution is cut off the polytope (because we do not want this solution anymore), but no integral solution is removed. In Branch-and-Bound, a variable $x^{(N)}$ that must be integer but has a fractional value $v \in \mathbb{R}_{\geq 0} \setminus \mathbb{N}$ is selected and two new MIP problems are created: one with the decision that $x^{(N)} \leq \lfloor v \rfloor$ and a second one with $x^{(N)} \geq \lceil v \rceil$. This is called branching just as we described for SAT in Section 2.1.1. This is the most similar approach to what SAT and CP solvers do with search, and indeed we also call this search in MIP. Graphically, branching corresponds to splitting the polytope into two disconnected regions, and searching in each part for the next solution.

Just as for CP (see Section 2.1.2), when we are optimizing we would like to be able to detect that no solution better than our current optimal one can be found in some branch. This is the bounding part of Branch-and-Bound. It can be seen that the best solution of the relaxation of a MIP is a valid bound for the MIP itself. Intuitively, the optimal integer solution is inside the polytope, so an optimal solution on the border of the polytope (i.e. the solution to the relaxation) can only be better. For maximization (respectively minimiza-
tion) problems, the optimal solution of the relaxation provides an upper bound (resp. lower bound) to the MIP. Thus, after each branching, MIP solvers will use the objective value of the solution of the relaxed problem to stop the search if no better solution can be found in the current branch of the problem. For this reason, a key element to optimize MIP problems efficiently is to find a formulation with a “good relaxation”. That is, we want to formulate MIP problems in such a way that the objective value of the relaxation of the MIP is very close to the objective value of the MIP itself. The closer the solution to the relaxation is, the less search we will have to do.

2.3.2 Local Search

Local Search (LS) is drastically different to the other techniques discussed until now. One important difference is that Local Search does not provide a proof of optimality, it is an approximate method.

The idea is quite simple: try assignments until we find some feasible solution or until we run out of time. For CSPs, each variable gets assigned a value from its domain, so we obtain an assignment $\theta$. If $\theta$ does not violate any constraint, we found a valid assignment. Otherwise, we modify something in $\theta$ to create another assignment $\theta'$ and repeat this process until we find an assignment that satisfies all constraints. For pure optimization problems (no constraints), we repeatedly try assignments while keeping track of the best solution until some time limit is reached. At the end the best solution is returned. It may appear like a very naive approach, but it has shown good results in several applications. The book by Aarts and Lenstra [1] goes through great detail of state of the art techniques in Local Search and applications such as the Traveling Salesman Problem, Vehicle Routing, Machine Scheduling and VLSI Design. A chapter in the Handbook of Metaheuristics is also dedicated to Local Search [139]. It shows several examples and applications such as, for example, Flow Shop Scheduling, Job Shop Scheduling, Graph Partitioning and Prize-Collecting Steiner Tree problems.

**Definition 2.35.** A local move is a function $f$ over an assignment $\theta$ that converts it into an other assignment $\theta'$ such that $\theta \neq \theta'$. 
Definition 2.36. The neighborhood $N$ of an assignment $\theta$ is the set of assignments $\Theta'$ that can be obtained by applying a local move to $\theta$.

Example 2.11. Given a graph $G = (V, E)$ a valid coloring is an assignment of a color to each node in $V$ such that any two adjacent nodes have different colors. This is known as the Graph Coloring Problem. Given some coloring $\theta$ that does not satisfy the constraints, a local move could be to swap the colors of two nodes.

Example 2.12. Given a graph $G = (V, E)$, we try to find the shortest cycle that visits all the nodes in $V$. This is known as the Traveling Salesman Problem. Given a solution path $\theta$, a possible move could be swapping the position of two nodes in the cycle to make it shorter.

The major difficulty in LS is finding the best moves that lead to solutions of good quality fast. These tend to be problem specific. We will use this technique in Chapter 7, and we will specify the moves used then.

2.3.2.1 Local vs. Global Optimal

One of the major difficulties of Local Search is avoiding local mimima/maxima. Assume we want to minimize a function over one variable $x$ as in Figure 2.6. If we only make moves that lower the value of the objective function (improving moves) we might get stuck in a local minimum, as the figure shows.

To avoid this, we must allow the possibility of worsening the solution.

2.3.2.2 Using Simulated Annealing

One of the techniques to avoid getting stuck in a local optimal is simulated annealing, introduced by Ropke and Pisinger [188] for the Pickup and Delivery Problem with Time Windows. There exist other such as Tabu Search [100], and Quantum Annealing [88].

Assume we are optimizing a function $f$. The idea of simulated annealing is to always have 3 solutions stored: $\theta^*$, $\theta^a$, and $\theta^f$. The first is the solution found so far with best objective value. The second is the accepted solution that we use to generate new solutions through local moves. The last, is the new solution which we obtained by modifying $\theta^a$.
using local moves. If we accept a solution $\theta^i$ only when it is better than the accepted solution we might get trapped in a local minimum. With simulated annealing, we accept a solution $\theta^i$ with probability $e^{(f(\theta^i(v)) - f(\theta^{i-1}(v)))}/T$, where $T$ is the temperature. The temperature is a parameter that decreases at each iteration by a factor called the cooling rate: $T_{i+1} = c_r \times T_i$, $0 < c_r < 1$. This allows us to accept solutions that are worse than our currently accepted solution, thus allowing us to get out of local minima.

Initially we will have a higher tendency to accept worsening solutions, and as the temperature drops slowly over the iterations, we will only accept improving solutions.

2.4 Decision Diagrams

2.4.1 Multivalued Decision Diagrams

Multivalued Decision Diagrams (MDDs) [32, 202] can be thought of as “compressed” decision trees. MDDs represent a set of tuples (in our case a set of valuations) that need to be stored in memory. One way of storing these sets would be to simply enumerate them in a table or array, but when the set is too big enumerating all of the tuples becomes impractical. MDDs offer the opportunity of “merging” redundant parts of the tuples thus
2.4 Decision Diagrams

Reducing the memory required to represent them. An MDD $m$ is a connected directed acyclic graph where nodes are layered by their depth from the root. An MDD node $n$ is either one of the terminals $\text{TRUE}$ or $\text{FALSE}$, or is of the form $(x, [(v_1, n_1), \ldots, (v_k, n_k)])$ where $x$ is a variable, $v_1, \ldots, v_k$ are integer values and $n_1, \ldots, n_k$ are MDD nodes. This represents $k$ labeled edges $n \xrightarrow{x=v_1} n_1, \ldots, n \xrightarrow{x=v_k} n_k$. We let $\phi$ define the semantics of an MDD node as $\phi((x, [(v_1, n_1), \ldots, (v_k, n_k)])) \equiv (x = v_1 \land \phi(n_1)) \lor \cdots \lor (x = v_k \land \phi(n_k))$ and $\phi(\text{TRUE}) \equiv \text{true}$ and $\phi(\text{FALSE}) \equiv \text{false}$.

We will restrict our attention to layered MDDs where each node $n$ is in layer $i$ (the shortest path from the root to $n$) and each edge in a given layer is labeled by the same variable. Each path from the root $r$ of an MDD to $\text{TRUE}$ represents a solution of $\phi(r)$. To simplify the presentation, we elide the FALSE node and all edges to it.

Example 2.13. Figure 2.7 shows an example of an MDD that captures the constraints $a + 2b \leq 5 \land a + b + 2c \leq 8$ for $\{a, b, c\} \subseteq [1..3]^3$.

![Figure 2.7: Example of an MDD](image)

The valuation $\{a \mapsto 1, b \mapsto 1, c \mapsto 2\}$ is a solution but $\{a \mapsto 1, b \mapsto 2, c \mapsto 3\}$ is not a solution, for example.

Global propagation algorithms for constraints represented by MDDs are well understood (see e.g. [42, 160, 162, 164]) including versions with explanations [93]. Note that MDDs are a generalization of Binary decision Diagrams (BDDs), where the variables are Boolean.
2.4.2 Edge-valued MDDs

Cost-MDDs (or edge-valued MDDs, or weighted-MDDs) are a variation of MDDs where each edge has a weight associated with it. Similarly to MDDs, the goal is to store a large set of tuples, but in this case each tuple is associated a weight or cost. Cost-MDDs take these weights into account to ensure than when retrieving a tuple we know the weight that was associated to it. The cost of the solution to an MDD is the sum of the weights on the edges representing the path from the root to \textsc{True}. Cost-MDDs allow better Branch-and-Bound search when the cost is part of the objective since paths that cannot lead to better solutions can be pruned.

An example of cost-MDD is given in Figure 2.8. Similarly to regular MDDs, propagation algorithms for cost-MDDs already exist (see e.g [64, 94]).

![Cost-MDD Example Diagram]

Figure 2.8: Example of a cost-MDD (costs are indicated in parenthesis)

2.4.3 Deterministic Decomposable Negation Normal Form Formulae (d-DNNFs)

d-DNNFs are a less common data structure in the CP community, but widely used in \#\textsc{Sat} works [149].

A formula is considered to be in \textit{Negational Normal Form} (NNF) if the only logic operators used are $\wedge$, $\lor$ and $\neg$ and the latter (logical negation) is only applied to elementary variables (rather than sub-formulas).

An NNF is decomposable (DNNF) if, for all $\wedge$ operators, no variable appears in both operands. That is, a formula $F_{\text{left}} \wedge F_{\text{right}}$ is DNNF if, and only if, both formulas $F_{\text{left}}$ and
$F_{\text{right}}$ are DNNF and $\text{vars}(F_{\text{left}}) \cap \text{vars}(F_{\text{right}}) = \emptyset$.

A DNNF is deterministic (d-DNNF) if, for all $\lor$ operators, the conjunction of its operands is unsatisfiable. That is, a formula $F_{\text{left}} \lor F_{\text{right}}$ is d-DNNF if, and only if, both formulas $F_{\text{left}}$ and $F_{\text{right}}$ are d-DNNF and $F_{\text{left}} \land F_{\text{right}}$ is unsatisfiable.

For clarity, d-DNNFs are often represented as diagrams. Figure 2.9 gives an example of one.

Figure 2.9: Example of a d-DNNF (formula $(a \land b) \lor (\neg a \land b) \land (c \lor (\neg c \land d))$)

Propagators for this type of formulae exist in Constraint Programming. We use the one presented by Gange and Stuckey [91], as it is present in our solver of choice, CHUFFED.
Chapter 3
Steiner Trees in Constraint Programming

The Steiner Tree Problem is a well known NP-complete problem that is well studied and for which fast algorithms are already available. Nonetheless, in the real world, the Steiner Tree Problem is almost always accompanied by side constraints which means these approaches cannot be applied. For many problems with side constraints, only approximation algorithms are known. We introduce here a propagator for the connected, acyclic and tree constraints with explanations, as well as lower bounding techniques and a novel constraint programming approach for the Steiner Tree Problem and two of its variants. The experiments showed that these propagators with explanations were highly advantageous when it comes to solving variants of the Steiner Tree Problem.

3.1 The Steiner Tree Problem

The Steiner Tree Problem (STP) is a combinatorial problem on graphs. Given a non-empty graph $G = (V, E)$ and a subset of its nodes $\tau \subseteq V$ called terminals, a Steiner Tree $T = (V, E)$ is a tree such that $\tau \subseteq V \subseteq V$ and $E \subseteq E$. That is to say, $T$ is a subgraph of $G$ that spans all the nodes in $\tau$. We call the non-terminal nodes Steiner nodes.

Following the definition of Dreyfus and Wagner [77], the Steiner Tree Problem (STP) is an NP-complete problem (proved by Karp [115]) stated as follows.

Definition 3.1. Given a weighted graph $G = (V, E, w_f)$, find the Steiner Tree $T = (V, E) \subseteq G$, of minimal weight where the total weight is the sum of the weights of the edges in $E$ given by $w_f$.

To illustrate the problem, consider the example in Figure 3.1 with terminals $a$, $b$ and $c$. In that example we can see that by choosing two of the edges with cost 100 we are able
to connect all the nodes we intended to connect. Nonetheless, by using the Steiner node \( s \), we can achieve a tree of lower cost 174. Intuitively, the computational complexity of this problem comes from not knowing which Steiner nodes need to be used to achieve the cheapest tree.

![Suboptimal solution with cost 200](image1)  
![Optimal solution with cost 174](image2)

Figure 3.1: Example of two Steiner trees for the same graph

The Steiner Tree problem has been well studied because of its applications in computer networks, VLSI design or other network applications. The surveys by Du and Hu [78] and Winter [219] demonstrate this. Nevertheless, in many of these applications, the STP is not pure. In most cases there are side constraints that change the topology of the solution in different situations. For instance, in some cases we might want the terminal nodes to be leafs of the tree and not internal nodes (as they might be electrical components). In some other cases, we might want to have a tree that minimizes costs, but also maximize reliability: since in a tree removing an edge “breaks down” a network (like a water or electrical network) we need to choose the tree in such a way that we only use nodes with some degree of connectivity (described by Agrawal et al. [5]). We can also imagine situations where nodes have weights (representing latency issues) or negative weights (representing some kind of reward). A great number of side constraints can be found in real world applications that turn the original STP into a more difficult problem.

### 3.2 State of the Art

Because the STP is a well known NP-complete problem, it is understandable that there has been a lot of work on this problem and extensions of it. To our knowledge, the state
of the art in pure Steiner Tree problems was reached by Polzin and Daneshmand [171], and little room for improvement was left by their work. Nonetheless, their ideas focused on the pure Steiner Tree Problem. Indeed, several of the techniques they used to solve the problem (called reductions in their paper) are only valid for the pure Steiner Tree. These reductions decrease the size of the problem by removing edges or nodes based on the fact that they cannot be part of the minimum Steiner tree. In our case, we are searching for a Steiner tree that is minimum given a set of side constraints. It may very well be the case that, because of some side constraints, we need to remove a set of edge such that the only way to keep the tree connected is through some very expensive edge that a reduction would have removed. In short, we cannot remove edges or nodes simply based on their weight. We can only remove the ones violating our constraints.

It is worth noting that the Steiner Tree Problem resembles the Minimum Spanning Tree (MST) problem. It is nonetheless substantially different in terms of computational complexity. The NP-completeness of this problem comes from not knowing which nodes we need to span. If we did, we could apply Kruskal’s algorithm [127]. A naive approach would therefore choose a set of nodes and apply an MST algorithm to enumerate all the possible Steiner Trees. This would obviously have a prohibitive cost. On the other hand, if all the nodes are terminals, then it is trivial to solve the STP. But in the general case, this problem cannot be tackled as an MST. There is in fact an MST propagator by Dooms and Katriel [70] dedicated for the MST specific case. In Chapter 5, we will show how to build a propagator to handle the specific situation where all the nodes must be spanned (known as WST). As we will see then, and in similar works such as the ones by Dooms and Katriel [71] and Régis et al. [183], the fact that all nodes are to be spanned allows stronger propagation.

Contrary to the above mentioned MST or WST problems, the Steiner Tree problem has had substantially less attention from the CP community. In fact, most of the previous work done with exact algorithms for the STP used Mixed Integer Programming [13, 52]. This approach performs extremely well, as we will see later (because we actually use ideas from MIP in our propagators), but is less flexible. In many cases, some of the side constraints that appear may be hard to encode as an Integer Program, or may drasti-
cally affect the performance of the MIP solver. For example, in scheduling problems, CP has proven to be more effective than MIP [117]. Constraint Programming may therefore be more suitable for problems combining constraints that are hard for MIP solvers (like scheduling ones) and Steiner Tree problems. Trees and Steiner trees can appear in other combinatorial problems usually tackled in CP. Thus this research helps improve state of the art CP solvers by adding new propagators. We believe our approach using CP is more flexible and reusable. Using global constraints we specialize the solver to tackle the problem in a much more efficient way than using composition of elementary constraints.

All the other work solving STP with variations rely on approximation algorithms for each variation (e.g. [39, 95, 121, 147, 187, 222]).

3.3 Applying Constraint Programming

For all the subsections in this section, we define \( G = (V, E) \) to be the graph variable, subgraph of \( G = (V, \mathcal{E}) \), that needs to be a tree. As the search progresses, \( G \) will become closer to a tree. As before, we note \( v_V \) and \( v_E \) the sets of Boolean variables representing the choice of nodes and edges in \( G \), respectively. We denote by \( b_x \) the Boolean variable for a specific object \( x \) (node or edge) in \( v_V \) and \( v_E \). Recall from Section 2.2 of Chapter 2, that the graph variable can be any subgraph of \( G \). In order to specify a set of nodes or edges that need to be in \( G \), the user can set some of the variables in \( v_V \) and \( v_E \) to true. Those will be the mandatory nodes or edges.

3.3.1 Basic Graph Propagation with Explanations

In order for \( G \) to be a valid graph, we need to make sure that when an edge \( e = (u, v) \) is selected to be in \( G \), the endnodes \( u \) and \( v \) of that edge are also in \( G \). Otherwise \( G \) is not coherent with its semantics. Thus, whenever a node is removed or an edge is added, we must make sure that \( G \) is still sound. To do so, we implement a propagation rule that we call \textsc{Coherent}(n) \) applied to a node \( n \). This rule enforces two propagations explained as follows:
3.3 Applying Constraint Programming

1. If a node \( n \) becomes forbidden (i.e. is removed from the domain of \( G \)), then all the edges involving that node must be forbidden too. The explanations is, for any edge \( e \in \text{outgoing}[n] \): \( \neg b_n \Rightarrow \neg b_e \)

2. If an edge \( e = (u, v) \) becomes mandatory (i.e. must appear in \( G \)), then both its endnodes must also appear in \( G \). The explanations are: \( b_e \Rightarrow b_u \) and \( b_e \Rightarrow b_v \)

If the solver we use has graph variables (like CHOCO does), this propagation can be done within the variable. In our case, we do it as a graph propagator from which all the other propagators involving graphs will inherit.

3.3.2 A Connectedness Propagator with Explanations

We define the connected constraint as follows:

**Definition 3.2.** The connected global constraint enforces a graph variable \( G \) encoded with two sets of Boolean variables \( v_V \) and \( v_E \) to be a connected subgraph of a given graph \( \mathcal{G} \). It is noted:

\[
\text{connected}(v_V, v_E, \mathcal{G})
\]

In this section we present our propagator for the connected constraint. This propagator will wake whenever a variable in \( v_V \) becomes true (a node becomes mandatory) or a variable in \( v_E \) becomes false (an edge becomes forbidden). In the first case, we need to make sure that the new node is reachable from the other already existing mandatory nodes. In the second case, we must be sure that the removed edge did not cause a disconnection of the graph. When an edge is added, it can’t affect the connectedness of the graph, so nothing has to be done. When a node is removed, the edges reaching that node will be automatically removed, thus leading to the second situation. When the propagator is woken-up, the algorithms \text{REACHABLE} and \text{ARTICULATIONS} are executed to ensure connectedness.
3.3.2.1 The \texttt{REACHABLE} Algorithm

Given a mandatory node $n$, the \texttt{REACHABLE}(n) algorithm ensures that $n$ can be connected through mandatory or unfixed edges to the other mandatory nodes (if any). If $n$ is not reachable from some other mandatory node, then $n$ cannot be part of $G$ as it would not be a connected graph. In this situation the propagator will raise a conflict. Note how Quesada et al. [174] already introduced a reachability constraint that can be implemented with this algorithm, although in their case they used a transitive closure approach to ensure reachability of all nodes.

Figure 3.2 gives an example of two unreachable mandatory nodes. As it can be seen, the nodes $o$, $c$ and $d$ were already in $G$ when $n$ was added. Since $e_3$ and $e_4$ are forbidden edges, there is no way of building a connected graph containing the 4 nodes.

![Figure 3.2: Example of unreachable nodes: $n$ was added to $G_n$. Edges $e_3$ and $e_4$ being out-edges, we cannot reach $o$.](image)

First, to detect the failure, we run a depth first search (DFS) starting at $n$, that uses only mandatory and unfixed edges. We will mark all the nodes visited as blue. This gives us the reachable region from $n$. Any node marked in blue is reachable from $n$ and vice-versa. Then we look for any non-blue mandatory node $o$. If such a node $o$ exists, this node is unreachable from $n$ and we can trigger a conflict.

Explaining this conflict requires finding a minimal set of forbidden edges such that if any of them was available (i.e. unfixed or mandatory), there would be a path from $n$ to $o$. To find those edges, we run another DFS from the found target node $o$ marking all the nodes reached as pink. During this DFS we allow traversal through all edges except the ones having one blue endnode. Figure 3.3 illustrates the idea.

Let $F$ be the set of forbidden edges encountered during the second DFS that have one blue extremity (we do not cross them). If at least one of them was allowed to be used, the pink DFS would have reached the blue nodes thus showing that $G$ could still
Figure 3.3: Example of the blue (on the left, ‘ ‘) and pink (on the right, ‘ ‘) DFS to detect failure. The zigzag edges (‘ ‘) explain failure. Note that \( e_8 \) is not needed in the explanation for it to hold.

be connected. Therefore, this set of edges explains the un-reachability of \( o \) from \( n \). The final explanation is:

\[
(b_n \land b_o \land \bigwedge_{e \in F} \neg b_e) \Rightarrow \text{fail}
\]

That is, the fact that \( n \) and \( o \) are mandatory and the set of edges in \( F \) are forbidden causes a conflict.

Looking at the example in Figure 3.3, the mandatory nodes on the left cannot be connected to the mandatory nodes on the right. There are three ways in which the two regions could have been connected. The first would be to use \( e_5 \), the second to use \( e_7 \) and the third to use both \( e_6 \) and \( e_8 \). Because \( e_5 \) and \( e_7 \) are not available, the only option would be to use \( e_6 \) and \( e_8 \). But both need to be available at the same time, so either of them missing is enough to cause this disconnection. The explanation is therefore \( b_n \land b_o \land \neg e_5 \land \neg e_7 \land \neg e_6 \Rightarrow \text{fail} \). And alternative explanation would be \( b_n \land b_o \land \neg e_5 \land \neg e_7 \land \neg e_8 \Rightarrow \text{fail} \). Note how, if both \( e_6 \) and \( e_8 \) were used in the explanation, it would be correct but less general, and therefore less reusable. Indeed, the explanations that we generate with our algorithm are minimal: it suffices that exactly one of the edges becomes available for the conflict to disappear. Algorithms 3.1 and 3.2 below present the pseudo-code for this algorithm.

Note that it may be the case that another mandatory node \( o' \) is also unreachable from \( n \), and restoring any of the edges that make \( o \) reachable from \( n \) won’t necessarily make \( o' \) reachable from \( n \). Nonetheless, \( o \) being unreachable from \( n \) is reason enough to cause
Algorithm 3.1 Auxiliary DFS functions

1: procedure BLUEDFS(n,visited[]) ⊳ DFS starting in $n$, avoiding forbidden edges
2:     visited[n] ← true
3:     for all $e = (n, o) \in$ outgoing[n] do
4:         if $b_e \neq false \land \neg$visited[o] then
5:             BLUEDFS(o)

1: procedure PINKDFS(n, visited[], blue[], F) ⊳ DFS starting in $n$, avoiding blue nodes.
2:     visited[o] ← true
3:     for all $e = (n, o) \in$ outgoing[n] do
4:         if $\neg$visited[o] then
5:             if blue[o] then $F \leftarrow F \cup \{e\}$
6:             else PINKDFS(o, blue, F)

Algorithm 3.2 Algorithm to ensure connectivity in a graph

1: procedure REACHABLE(n) ⊳ $n$ must be a mandatory node
2:     BLUEDFS(n,[false | $n \in V$])
3:     $F \leftarrow \emptyset$
4:     for all $o \in V$ do
5:         if $\neg$blue[o] then ⊳ Not visited: run second DFS
6:             PINKDFS(o,[false | $n \in V$],blue,F)
7:     FAIL($\bigwedge_{e \in F} b_e \land b_u \land b_v$)
8:     break

a conflict, and the explanation must involve, at least, the forbidden edges between these two nodes. For that reason, when one pair of disconnected nodes is found, we stop and do not look further.

3.3.2.2 The ARTICULATIONS Algorithm

To guarantee the connectedness of $G$, we must propagate that any articulation or bridge that is in the (unique) path between two mandatory nodes $u$ and $v$ is mandatory, otherwise $u$ and $v$ could be disconnected. We assume that the REACHABLE algorithm succeeded as there would be no solution otherwise. Figure 3.4 shows an example of graph with articulations and bridges.

To find the articulations, we modify Tarjan’s algorithm for finding biconnected components [210] starting at a mandatory node. Recall Tarjan’s algorithm performs a DFS in the graph while marking nodes with their depth and their “lowpoint”. The lowpoint of a node $n$ is the deepest node that has been reached from the recursive calls of the DFS start-
ing at \( n \). By lemma 5 in Tarjan’s paper, if \( i \) \( u \) and \( v \) are adjacent, \( ii \) \( u \) is the parent of \( v \) in the DFS traversal, and \( iii \) the depth of the lowpoint of \( v \) is greater or equal to the depth of \( u \), then \( u \) is an articulation. Formally, if the condition \( u = \text{parent}(v) \land \text{depth}(\text{lowpoint}(v)) \geq \text{depth}(u) \) is true, then \( u \) is an articulation.

In our version, the DFS will start at a mandatory node and will only use available edges (i.e. no forbidden edges). Also, we are only interested in articulations/bridges that are in the path between two mandatory nodes. To identify only these we use a stack \( S \) to which we push all the mandatory nodes reached whilst performing the DFS. If after a recursive call in the DFS we detect an articulation \( a \) and the top of \( S \) is different to what it was when we reached \( a \), then \( a \) is a required articulation. Let \( a_b \) be the top of the stack \( S \) before the recursive calls at node \( a \), and \( a_a \) be the top of the stack after the recursive calls at node \( a \). Then, \( a_b \) is on one side of \( a \) and \( a_a \) on the other, and the only path from \( a_b \) to \( a_a \) goes through \( a \) (because \( a \) is an articulation). As \( a_b \) and \( a_a \) are mandatory (only mandatory nodes are recorded in the stack), \( a \) is required to guarantee the connectivity of \( G \). We can therefore propagate \( b_a = \text{true} \). Additionally, if a node \( a \) is an articulation (with a mandatory node on each side) and has only one edge on one of its sides, then that edge is a bridge can we can propagate it to be mandatory.

To explain these articulations (resp. bridges) we need the two mandatory nodes \( a_b \) and \( a_a \) that required the articulation (resp. bridge) \( a \) to exist to maintain \( G \) connected. We extract those nodes from \( S \) after finding \( a \) is an articulation during the DFS (let \( a_a \) be the top of \( S \), that is, the mandatory node found later than \( a \)). We also need all the forbidden edges that could have connected \( a_b \) and \( a_a \) if they were not forbidden. These are “broken alternative paths”. Indeed, if those edges were available, then we would not have found articulations or bridges. We do this in two steps after Tarjan’s DFS. First, we run a DFS
from $a$ that does not go through forbidden edges, nor the articulation (resp. bridge). This gives a set $R$ of reachable nodes from $a$. Visually, $R$ is the set of nodes “after” $a$. Then we run a second DFS from $a$ this time allowing to cross forbidden edges, but still not the articulation (resp. bridge). We add any forbidden edge that contains a node not in $R$ but visited during Tarjan’s DFS to a set $F$. Figure 3.5 illustrates this process. The explanations is

$$\quad (b_{a} \land b_{a} \land \bigwedge_{e \in F} \neg b_{e}) \Rightarrow b_{a} \quad (3.2)$$

where $a$ is either the articulation or the bridge we found.

In our example in Figure 3.5 (starting at $a$) we would find the following propagations. First, $b$ and $h$ being mandatory, we need $d$ to be mandatory as well because it is an articulation between them. If either $e_9$ or $e_{10}$ were available, we could have reached $b$ from $h$ without using $d$. Thus, we explain the propagation as $b_{b} \land b_{h} \land \neg b_{e_{9}} \land \neg b_{e_{10}} \Rightarrow b_{d}$. Second, since $c$ and $b$ are mandatory, and $e_{10}$ is forbidden, the only way of getting from $b$ to $c$ is through $e_{3}$. In other words, $b_{b} \land b_{c} \land \neg b_{e_{10}} \Rightarrow b_{e_{3}}$.

Below we show pseudocode for this algorithm. Algorithm 3.3 shows helper functions to perform the DFS that we need for the explanations.

The more delicate function is the modification of Tarjan’s algorithm to find articulations, presented in Algorithm 3.4. To simplify the code, we use two stacks instead of the aforementioned one ($S$, in the previous paragraphs). The first stack, $s_{m}$, stores the mandatory nodes as they are seen. The second stack, $s_{e}$, stores all the edges in the order of visit.
3.3 Applying Constraint Programming

Algorithm 3.3 Auxiliary DFS functions

1: procedure REACHABLEDFS(n, visited[], r) ▷ DFS starting at n avoiding r and forbidden edges.
2: visited[o] ← true
3: for all e = (n, o) ∈ outgoing[n] do
4:     if b_e ≠ false ∧ ¬visited[o] then
5:         if r ≠ o ∧ r ≠ e then ▷ Object r could be an edge or a node
6:             REACHABLEDFS(o, visited, r)

1: procedure BROKENEDGESDFS(n, visited[], r, R, V, F) ▷ DFS starting in n, not using r. Stops recursion when reaching a node
2:     visited[o] ← true ▷ n such that n ∈ R ∧ n ∈ F
3:     for all e = (n, o) ∈ outgoing[n] do
4:         if o ∈ V ∧ o ∈ R then
5:             F ← F ∪ {e}
6:     else if ¬visited[o] then
7:         if r ≠ o ∧ r ≠ e then ▷ Object r could be an edge or a node
8:             BROKENEDGESDFS(o, visited, r, V, F)

When an articulation is found (line 12), the stack of edges is popped until we pop the edge that we most recently traversed from the current node (in the code, e). If only one edge was popped, then that edge must be a bridge. This is because, by definition, the endnodes of a bridge are articulations, thus the stack was already popped at the latest recursive call (in the code, when FINDBRIDGES was called with v as a parameter, which must also be an articulation). If the top of stack s_n is different than it was at the beginning of the current recursion it must be the case that a mandatory node was visited after the current one (u in the code). Thus b_u = true must be propagated (as well as b_e = true, if e is a bridge). Let a_u be the current top of s_n. We can store the b_u and b_e part of the explanation in Equation 3.2 in the set partExpl. After all the recursive calls, the traversal of the graph “below” u finishes. At this point, if node u happens to be mandatory, then u is the node a_u missing from the explanation. We store the triplet of nodes involved in Equation 3.2 in a set props. Note that in this algorithm, when we test b_x ≠ true any b_x assigned to false or unfixed would pass the test.

Finally, we can combine all the algorithms to build the final Algorithm 3.5 that performs the propagation with explanations. First it runs Algorithm 3.4 to find the articulations/bridges and the nodes that will be needed to explain them. Then, Algorithm 3.3 is used to collect the edges need in the explanations.
Algorithm 3.4 Algorithm to propagate articulations and bridges for connectivity

1: procedure FINDBRIDGES($u$, $visited[]$, $depth[]$, $low[]$, $parent[]$, $partialExpl$, $props$, $s_n$, $s_e$)
2:     $visited[u] ← true$
3:     $depth[u] ← low[u] ← count ← count + 1$
4:     if $b_u = true$ then $s_n$.PUSH($u$)  \(\triangleright\) Add mandatory nodes to stack
5:     $prevTop ← s_n$.TOP ; $curTop ← nil$
6:     for all $e = (u, v) ∈ \{\text{outgoing}[u] | b_e \neq false\}$ do
7:         $s_e$.PUSH($e$)
8:     if $¬visited[v]$ then
9:         $parent[v] ← u$
10:        FINDBRIDGES($v$, $visited$, $depth$, $low$, $parent$, $partialExpl$, $props$, $s_n$, $s_e$)
11:       $curTop ← s_n$.TOP
12:      if $low[v] ≥ depth[u]$ then  \(\triangleright\) $u$ is an articulation
13:          $lastEdge ← nil$ ; $counter ← 0$
14:          while $lastEdge \neq e$ do  \(\triangleright\) Unfold the stack of edges to detect a bridge
15:              $lastEdge ← s_e$.TOP ; $s_e$.POP()
16:          $counter ← counter + 1$
17:          if $curTop \neq prevTop$ then  \(\triangleright\) Found a mandatory node after $u$ in the recursions
18:              if $counter = 1 ∧ b_u \neq true$ then  \(\triangleright\) Unfixed bridge
19:                  $partialExpl ← partialExpl \cup \{(e, curTop)\}$
20:            if $b_u \neq true$ then  \(\triangleright\) Unfixed articulation
21:                $partialExpl ← partialExpl \cup \{(u, curTop)\}$
22:        while $s_n$.TOP $\neq prevTop$ do $s_n$.POP()  \(\triangleright\) Recover previous state
23:          $low[u] ← min(low[u], low[v])$
24:      else  \(\triangleright\) Normal Tarjan execution
25:          $low[u] ← min(low[u], depth[v])$
26:      if $b_u = true$ then  \(\triangleright\) Recover articulations found after $u$ and the partial explanations
27:          for all $(req, a_u) ∈ partialExpl$ do
28:              $props ← props \cup \{(req, a_u, u)\}$
29:            $partialExpl ← \emptyset$
30:        else if $curTop \neq nil$ then  \(\triangleright\) Notify above level that we hit a mandatory node
31:          $s_n$.PUSH($curTop$)

Algorithm 3.5 Algorithm to explain propagation on articulations and bridge

1: procedure ARTICULATIONS($n$)
2:     $V ← \{false | n ∈ V\}$
3:     $h, s ← \text{STACK}()$
4:     $P ← \emptyset$
5:     FINDBRIDGES($n, V, [−1 | n ∈ V], [−1 | n ∈ V], [nil | n ∈ V]\emptyset, P, h, s$)
6:     for all $(a, a_x, a_y) ∈ P$ do
7:         $R ← \{false | n ∈ V\}$
8:         $F ← \emptyset$
9:         REACHABLEDFS($a_y, R, a$)
10:        BROKENEDGESDFS($a_y, [false | n ∈ V], a, R, V, F$)
11:       PROPAGATE($b_{a_y} ∧ b_{a_x} ∧ \wedge_{e ∈ F} ¬b_e ⇒ b_a$)
3.3 Applying Constraint Programming

3.3.3 An Acyclicity Propagator with Explanations

We define the acyclic constraint as follows:

**Definition 3.3.** The acyclic global constraint enforces a graph variable \(G\) encoded with two sets of Boolean variables \(v_V\) and \(v_E\) to be an acyclic subgraph of a given graph \(G\). It is denoted:

\[
\text{acyclic}(v_V, v_E, G)
\]

In this section we present our propagator for the acyclic constraint. This propagator is similar to the nocycle constraint by Caseau and Laburthe [38] although theirs is used in directed graphs via DFS (we will use connected components instead) for subtour elimination specifically for the Traveling Salesman Problem.

Our acyclic propagator will wake whenever a variable in \(v_E\) becomes mandatory. When an edge is added, it may be the case that it creates a cycle, in which case the propagator must raise a conflict. But we can also prevent these cycles. When an edge is added, a connected component of mandatory edges grows by one edge, thus opening the possibility of new cycles to appear later on. When the propagator is woken up, the algorithms CYCLEDetect and CYCLEPrevent are executed to ensure acyclicity.

3.3.3.1 The CYCLEDetect and CYCLEPrevent Algorithms

Given a new mandatory edge \(e = (u, v)\), the CYCLEDetect algorithm ensures that \(e\) does not form a cycle in \(G\). If it does, there must be a pre-existing path \(p\) from \(u\) to \(v\). If \(p\) exists, we trigger a conflict with reason

\[
(b_e \land \bigwedge_{e' \in p} b_{e'}) \Rightarrow \text{fail}
\]

Furthermore, given a node \(n\), the algorithm CYCLEPrevent removes any edge adjacent to \(n\) that would form a cycle if it was added to \(G\). We can safely propagate the decision that they must be forbidden as \(G\) would contain a cycle otherwise.

To perform this operation, we simply remove any edge \(e = (n, o)\) such that \(n\) and \(o\) are connected by mandatory edges. The minimal reason required for this removal the
same as before: the set of mandatory edges forming a path $p$ from $n$ to $o$ in $G$. That is:

$$\bigwedge_{e_i \in p} e_i \Rightarrow \neg e$$

The `CYCLEPREVENT` operation must be done from all the nodes in the connected component to which the new edge $e$ belongs, as the new potential cycle can appear anywhere in the connected component. The algorithms are presented with pseudo-code in Algorithm 3.6.

**Algorithm 3.6 Algorithms for cycle detection and prevention**

1: procedure `CYCLEDetect(e = (u, v))`  \>
   \hspace{1cm} $\triangleright$ $e$ is a mandatory edge
2: \hspace{1cm} path $\leftarrow$ `PATH(u, v)`  \>
   \hspace{1cm} $\triangleright$ Retrieve path from $u$ to $v$, if any
3: \hspace{1cm} if path $\neq \emptyset$ then
4: \hspace{1cm} \hspace{1cm} `FAIL(e \bigwedge_{e' \in \text{path}} e')`
5: \hspace{1cm} else
6: \hspace{1cm} \hspace{1cm} `UNITE(u, v)`  \>
   \hspace{1cm} $\triangleright$ Store that there is a direct path from $u$ to $v$

1: procedure `CYCLEPREVENTAtNode(u)`  \>
   \hspace{1cm} $\triangleright$ $u$ is a mandatory node
2: \hspace{1cm} for all $e = (u, v) \in \text{outgoing}[u]$ do
3: \hspace{1cm} \hspace{1cm} path $\leftarrow$ `PATH(u, v)`
4: \hspace{1cm} \hspace{1cm} if path $\neq \emptyset$ then
5: \hspace{1cm} \hspace{1cm} \hspace{1cm} `PROPAGATE(\bigwedge_{e' \in \text{path}} b_{e'} \Rightarrow \neg b_e)`

1: procedure `CYCLEPREVENT(e = (u, v))`  \>
   \hspace{1cm} $\triangleright$ $e$ is a mandatory edge
2: \hspace{1cm} `nodes \leftarrow NODESINCC(u)`  \>
   \hspace{1cm} $\triangleright$ Get all the nodes in the mandatory connected component of $u$
3: \hspace{1cm} for all $n \in \text{nodes}$ do
4: \hspace{1cm} \hspace{1cm} `CYCLEPREVENTAtNode(n)`

### 3.3.3.2 Rerooted-Union-Find (R-UF): Efficiently Retrieving Paths Between Nodes

We saw above that in order to ensure and explain acyclicity we need to retrieve paths between nodes. To do so efficiently, we will use a modified version of the classic union-
find data structure (also known as disjoint-set). We call this the Rerooted-Union-Find (abbreviated R-UF).

The typical union-find data structure (UF) builds a directed forest of nodes when the method \textsc{Unite}(u,v) is applied. Then we can retrieve the root of each tree by using the method \textsc{Find}(u). In our implementation, we will have a method \textsc{Path}(u,v) that will return the nodes in the path from \(u\) to \(v\) in order (or an empty list if they are not connected).

To do so, we modify the \textsc{Unite}(u,v) procedure: we first make \(v\) become the root of its own tree (by inverting some of the parenthood relations in the trees), then we make \(u\) the parent of \(v\). To retrieve the path we modify the \textsc{Find}(u) method to return the nodes it goes through. We then can map pairs of nodes into edges to get a proper path of edges. Calling this method on two nodes \(a\) and \(b\) in the same connected component allows us to find the path between those nodes. The worst case complexity of this query is linear in the number of nodes in the graph, although in practice it is much closer to the length of the path between the two nodes queried.

Pseudocode for this data structure can be found in Algorithm 3.7. The directed forest is represented by parenthood links in an array \textit{parent}, where \textit{parent}[i] represents the parent node of node \(i\), and if \textit{parent}[i] = \(i\), then \(i\) is the root of its own tree. This is the same as the classic implementation of the Union-Find data-structure.

An example is provided in Figure 3.7. On the left, we have the R-UF after having united nodes 0, 1 and 2, and nodes 4, 5 and 6. We then apply \textsc{Unite}(2,5) and the result is shown on the right-hand side. First, 5 becomes the root of it’s own tree. Then the parent of 5 becomes 2. This way, both trees are connected, thus we can answer queries to detect connections. But most importantly, we can now query what are the nodes in the path from 1 to 6, for example. It suffices to traverse the parent relationships from 6 and 1 until we find their least common ancestor. All the nodes seen in that traversal participate in the connections. We can then map this sequence of nodes to edges.


**Algorithm 3.7 Algorithms for R-UF**

1: procedure **MAKEROOT**(u)
2:   i ← u
3:   parent ← parents[i]
4:   last ← i
5: while parent ≠ i do
6:   x ← parent
7:   parent ← parents[parent]
8:   parents[i] ← last
9:   last ← i
10: i ← x
11: parents[i] ← last

1: function **CONNECTED**(u,v)
2: return FIND(u)[0] = FIND(v)[0]
3: end function

1: function **NODESPATH**(u,v)
2: ▷ Assumes u and v are connected
3: l ← []
4: r ← []
5: s1 ← FIND(u)[1]
6: s2 ← FIND(v)[1]
7: while s1 > s2 do
8:   l ← [u] ++ r
9:   u ← parent[u]
10: s1 ← s1 − 1
11: while s2 > s1 do
12:   l ← l ++ [v]
13:   v ← parent[v]
14: s2 ← s2 − 1
15: while u ≠ v do
16:   l ← [u] ++ r
17:   l ← l ++ [v]
18:   u ← parent[u]
19:   v ← parent[v]
20: return l ++ [u] ++ r
21: end function

Note that as a consequence of this implementation, we cannot use path compression as in the classical Union-Find. This is because would immediately lose the path information.
3.3 Applying Constraint Programming

3.3.4 A Tree Propagator with Explanations

We define the tree constraint as follows:

**Definition 3.4.** The tree global constraint enforces a graph variable $G$ encoded with two sets of Boolean variables $v_V$ and $v_E$ to be a connected, acyclic subgraph of a given graph $G$. It is noted:

$$\text{tree}(v_V, v_E, G)$$

Obviously, this global constraint can simply be encoded with a decomposition as

$$\text{tree}(v_V, v_E, G) \equiv \text{connected}(v_V, v_E, G) \land \text{acyclic}(v_V, v_E, G)$$

Nonetheless, for performance reasons, we implemented this propagator as a stand-alone one, in CHUFFED. Of course, it uses the same algorithms as described for connectedness and acyclicity, but because some traversals of the graph need to be done several times, we implement it as a standalone propagator to reuse the result from previous traversals and thus gain in efficiency.

This propagator will wake whenever a variable in $v_V$ or $v_E$ is fixed, meaning that a node (resp. edge) becomes a mandatory node or a forbidden node (resp. mandatory/forbidden edge). In each of these four different situations, we apply a set of rules in order. Table 3.1 summarizes these rules. To summarize, REACHABLE and ARTICULATIONS ensure that $G$ is connected, CYCLEDetect and CYCLEPrevent ensure that $G$ is acyclic, and COHERENT simply ensures that the graph is indeed a valid graph.

3.3.5 Steiner Tree Global Constraint

We define the steiner_tree constraint as follows:

**Definition 3.5.** The steiner_tree global constraint enforces a graph variable $G$ encoded with two sets of Boolean variables $v_V$ and $v_E$ to be a connected, acyclic subgraph of a given graph $G = (V, E)$, of weight at most $w$ given a weighting function $w_f$ over the set of edges $E$. It is denoted:

$$\text{steiner_tree}(v_V, v_E, G, w_f, w)$$
Event Rules

<table>
<thead>
<tr>
<th>Event</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node addition $n$</td>
<td>1. REACHABLE($n$)</td>
</tr>
<tr>
<td></td>
<td>2. ARTICULATIONS($n$)</td>
</tr>
<tr>
<td>Node removal $n$</td>
<td>1. COHERENT($n$)</td>
</tr>
<tr>
<td>Edge addition $e = (u, v)$</td>
<td>1. COHERENT($e$)</td>
</tr>
<tr>
<td></td>
<td>2. CYCLEDETECT($e$)</td>
</tr>
<tr>
<td></td>
<td>3. $\forall n \in CC_G(e), CYCLEPREVENT(u)$</td>
</tr>
<tr>
<td></td>
<td>4. UNITE($u, v$)</td>
</tr>
<tr>
<td>Edge removal $e = (u, v)$</td>
<td>1. REACHABLE($u$) (This could also be done on $v$ with the same result)</td>
</tr>
<tr>
<td></td>
<td>2. ARTICULATIONS($u$)</td>
</tr>
</tbody>
</table>

Table 3.1: List of algorithms applied in our tree propagator. $CC_G(e)$ is the connected component of $G$ containing $e$.

Once again, a decomposition approach could be possible:

$$steiner\_tree(v_V, v_E, G, w_f, w) \equiv tree(v_V, v_E, G) \land \sum_{e \in E} b_e \cdot w_f(e) \leq w$$

Nonetheless, we develop a lower bounding approach that can be used to speed-up the solver by using branch-and-bound. Indeed, with that encoding, the solver would only do branch-and-bound based on the summation part (as in Example 2.6 in page 13). But because we have more specific knowledge of the problem (we know it is a Steiner Tree Problem), we can create a better lower bound. Two techniques will be discussed in subsection 3.3.6.

### 3.3.5.1 Additional Propagation for Steiner Trees

We present here an additional propagation that can only be done when we know which nodes we want to connect. In the Steiner Tree Problem, we are given a set of terminals $\tau$ that we want to connect. If the other nodes in $G$ (the Steiner nodes) are there only to connect the terminals, we know that a Steiner node of degree 1 is of no use, since it is not helping connect two terminals.

Let the available degree of a Steiner node $n$ (noted $deg_a(n)$) be the number of edges incident to $n$ that are not forbidden. Let $R_n = \{ e | e \in \text{outgoing}[n] \land \neg b_e \}$ be the set of for-
bidden edges outgoing from \( n \). Given a node \( n \):

- if \( \deg_A(n) = 1 \), we can set \( b_n \) to false with the following reason: \( \lor_{e \in R_n} \neg b_e \Rightarrow \neg b_n \)

- if \( \deg_A(n) = 2 \) we can safely propagate that the solution will only contain \( n \) if both edges are also in \( G \). Let \( e_1 \) and \( e_2 \) be the only two remaining available edges. We force them in \( G \) giving the explanation: \( \forall b \in \{ b_{e_1}, b_{e_2} \}, (b_n \land \lor_{e \in R_n} \neg b_e) \Rightarrow b \)

Once again, this propagation rule is not generalizable to all the problems involving trees, but if the user knows it applies to the problem they are trying to solve, such rules can be beneficial. If would be applied by the \textit{steiner_tree} propagator when a Steiner node \( n \) becomes mandatory, or as a preprocessing step to remove Steiner nodes until none of degree 1 is left.

### 3.3.6 Lower-bounding for Steiner Trees

We now introduce two lower bounding techniques that we use to perform branch-and-bound in the search space. Given a solution of cost \( K \), a lower bound allows us to prune the search space by proving that no better solution exists in a branch. This is the main idea of branch-and-bound described in Section 2.1.2.

In this section we present two approaches to compute lower bounds to the Steiner Tree Problem that will allow us to prune the search space. Notice that these lower bounds can be applicable to any variation of the STP. Indeed, a solution to the STP with side constraints can’t have a lower cost than a solution to the STP alone.

#### 3.3.6.1 Shortest-Paths Based Lower Bound (SPLB)

First, given a graph variable \( G \) and the decisions made so far in the search, we define a contraction function \( \mu : (V, E) \mapsto (V', E') \) that contracts all the mandatory nodes connected by mandatory edges into one new mandatory node and removes all the forbidden edges. In other words, the connected components of \( G_s \) are contracted and the forbidden edges removed. Clearly \( E' \) contains only unfixed edges. By analogy, we call \textit{mandatory} the nodes of \( G' \) built from mandatory nodes of \( G \).
Consider the graph $G'$ obtained by applying the $\mu$ function to $G$. Let $S$ be the set of mandatory nodes in $G'$. We claim that, in $G'$, the following is a lower bound for the STP.

$$
LB(G') = \begin{cases} 
\frac{1}{2} \sum_{u \in S} spc_{G'}(u), & \text{if } |S| \text{ is even} \\
\frac{1}{2} \sum_{u \in S} spc_{G'}(u) - \min_{u \in S} spc_{G'}(u), & \text{otherwise}
\end{cases}
$$

(3.3)

where $spc_{G'}(u)$ is the weight of the shortest path between $u$ and its closest mandatory node in $G'$. The following Lemmas and Theorem prove that this is a correct lower bound in the following paragraph.

**Lemma 3.1.** Given a graph $G = (V, E)$ with an even size set of terminals $\tau$ ($|\tau| = k \geq 2$) and a Steiner Tree $T = (V_T, E_T)$ that spans all the terminals, there exists a path $p_{u,v}$ between two terminals $u$ and $v$ such that i) $p_{u,v}$ contains only two terminals (namely $u$ and $v$) and ii) $p_{u,v}$ contains at most one node of degree more than 2 in $T$.

**Proof.** We prove this by induction.

Basis: When $G$ contains only two terminals $u$ and $v$, $T$ only contains nodes of degree 2 and two leaves ($u$ and $v$ of degree 1). In this case, the only path between $u$ and $v$ verifies the 2 conditions of the lemma.

Inductive step: Assume the lemma is valid for a tree with $k' \geq 2$ terminals. We prove it for $k = k' + 2$ (i.e. we added two terminals). Let us arbitrarily choose two terminals $u$ and $v$, and let $p_{u,v}$ be some path in $T$ between these two terminals. If $p_{u,v}$ satisfies the 2 conditions of the lemma, then we finish. Otherwise, either $p_{u,v}$ contains a third terminal $w$ or $p_{u,v}$ contains two nodes $x$ and $y$ of degree greater than 2.

In the first case, $w$ is in between $u$ and $v$ in $p_{u,v}$ (i.e. the path is somewhat like $u-w-v$). The subtree containing both $u$ and $w$ but not $v$ has $k'' < k$ terminals and the lemma
holds for it (by hypothesis). Thus, the lemma holds for \( k \) terminals.

In the second case, when nodes \( x \) and \( y \) of degree more than 2 in \( T \) are in the path between terminals \( u \) and \( v \), we can apply a similar reasoning. Here, \( x \) and \( y \) are in between \( u \) and \( v \) in \( p_{u,v} \) (i.e. the path is somewhat like \( u \rightarrow x \rightarrow y \rightarrow v \)). Then the third edge incident to \( x \) that is not in \( p_{u,v} \) must lead to some other terminal \( w \). Thus, the subtree containing both \( u \) and \( v \) has \( k'' < k \) terminals and the lemma holds for it (by hypothesis). Thus, the lemma holds for \( k \) terminals.

**Lemma 3.2.** Given a graph \( G = (V, E) \) with an even size set of terminals \( \tau (|\tau| \geq 2) \) and a Steiner tree \( T = (V_T, E_T) \) that spans all the terminals, there exists a path \( p_{u,v} \) (let \( E_{p_{u,v}} \) be the set of edges in \( p_{u,v} \)) between two terminals \( u \) and \( v \) such that there is a Steiner Tree \( T' = (V_T', E_T') \) for \( G' = (V, E \setminus E_{p_{u,v}}) \) with terminals \( \tau' = \tau \setminus \{u, v\} \) and \( E_T' \cap E_{p_{u,v}} = \emptyset \).

**Proof.** We choose \( p_{u,v} \) satisfying Lemma 3.1. Then if we remove \( E_{p_{u,v}} \) from \( G \), we are left with the graph \( G' \) and terminals \( \tau' \). Let’s prove that \( T' \) exists in \( G' \).

Because \( p_{u,v} \) has only one node of degree greater than 2, the connectivity of \( T \) does not rely on any edge in \( E_{p_{u,v}} \). Therefore, removing \( E_{p_{u,v}} \) from \( E \) does not disconnect any of the terminals of \( \tau' \) in \( T \). For this reason, if \( T \) was a valid Steiner Tree connecting the terminals in \( \tau \), \( G' \) contains a tree \( T' \) that does not use the edges in \( E_{p_{u,v}} \) that is connected and spans all the terminals in \( \tau' \).

**Theorem 3.1.** \( LB(G') \) is a valid lower bound for the Steiner Tree problem in \( G' \).

**Proof.** Let \( T \) be the Steiner tree of minimum cost \( K^* \) in \( G' \) (i.e. \( T \) contains all the nodes in \( \tau \)). We prove that \( LB(G') \) is indeed a lower bound by verifying that \( LB(G') \leq K^* \).

If \( |\tau| \) is even: Because \( T \) is connected, there is a path from every node in \( \tau \) to any other node in \( \tau \). We name \( p_{a,b} \) the path in \( T \) between two terminals \( a \) and \( b \). We construct a set of paths \( \rho \) as follows: we choose a path \( p_{u,v} \) as in Lemma 3.1 (for some pair of nodes \( \{u, v\} \in \tau^2 \)) and remove all its edges from \( T \) (and \( G' \)) and add \( p_{u,v} \) to \( \rho \). By Lemma 3.2, the remaining tree is still a Steiner tree and so we can do this until \( T \) does not contain any node in \( \tau \). We can indeed pair them because \( |\tau| \) is even. Note that \( \rho \) is a set of edge-disjoint paths (since we choose a path after removing the previous ones, thus making the edges unavailable).
Clearly, after having done this, $T$ is either empty or contains some edge that would have been needed to connect the paths in $\rho$ to each other in such a way that they form a tree. Therefore, the sum of the costs of the paths in $\rho$ is lower than $K^*$. Let $E_{\rho}$ be the set of all the edges in the set of paths $\rho$: $E_{\rho} = \bigcup_{p_{u,b} \in \rho} \text{edges}(p_{u,b})$.

$$\sum_{e \in E_{\rho}} w(e) \leq K^* \quad (3.4)$$

We now choose the paths in $\rho$ to be of minimal weight (i.e. the shortest paths). The Equation 3.4 still holds as the cost of the shortest paths can only be lower or equal to the cost of the paths chosen in the previous step.

For ease of computation, we compute the shortest paths from each mandatory node to some other mandatory node using $\text{spc}_{G'}(u)$. For this reason, every edge in $\rho$ is counted twice using this computation. Thus:

$$\frac{1}{2} \sum_{u \in \tau} \text{spc}_{G'}(u) \leq \sum_{e \in E_{\rho}} w(e) \quad (3.5)$$

By dividing the cost by 2, we get the same lower bound.

**If $|\tau|$ is not even:** In this case, we choose some terminal $v$ that we ignore (we subtract the cost of the path containing it). Let $G'' = (V' \setminus \{v\}, E' \setminus \text{outgoing}(v))$. This new graph $G''$ has an even number of terminals. Let $K''^*$ be the cost of the best Steiner tree for $G''$. Then, by applying the even case on $G''$, $LB(G'')$ is a valid lower bound for $G''$ (i.e. $LB(G'') \leq K''^*$). Because we ignore one terminal, $K''^* \leq K^*$. And since $LB(G') \leq LB(G'')$ (because we subtract a non-negative value to it), $LB(G') \leq K^*$. Therefore $LB(G')$ is also a valid lower bound.

We extend this lower bound to a lower bound of the STP in the current graph by adding the weight of the mandatory edges that were contracted. We call this lower bound $\text{SPLB}$.

$$\text{SPLB}(G) = LB(\mu(G)) + w_m$$
3.3 Applying Constraint Programming

where \( w_m = \sum_{e \in E} w_f(e) \)

Recall \( w_f \) is the weighting function as described in Definition 3.1 (page 37).

3.3.6.2 Computing SPLB

Because computing this lower bound can be expensive if we do it every time, we implemented it in an incremental way. In order to avoid having a contracted version of the graph, we will consider that all mandatory edges have weight zero and we will only compute the shortest paths between a representative mandatory node from each connected component of mandatory nodes in \( G \).

Let \( sp_C \) be a map from representatives to the cost of its shortest path to another representative and \( sp_E \) a map from representatives to the edges used in that shortest path. We compute SPLB by adding the following steps after the previously described propagations.

**Node Addition**  A new node may change all the shortest paths between connected components of \( G \) (as it might itself be a new component on its own), so we recompute all the shortest paths. We use Dijkstra’s algorithm starting at a representative while recording the paths and update \( sp_C \) and \( sp_E \).

**Edge Removal**  All paths using the removed edge (recorded in \( sp_E \)) must be recomputed with Dijkstra’s algorithm.

**Edge Addition**  A new edge merges two connected components of \( G \), so we use the shortest path of the two of them as the new shortest path of the resulting connected component (other than the path between them). We also add the weight of the edge to a variable \( w_m \). Eventually, the lower bound will be the sum of \( w_m \) and the sum of all the costs recorded in \( sp_C \).
3.3.6.3 Explaining SPLB

The lower bound will prune the search space by making the solver fail and backtrack. As with other propagations, we need to explain this conflict.

All mandatory edges in $G$ must be part of the explanation since they bring weight to the lower bound. They correspond to the $w_m$ part of the sum. We must also include the forbidden edges that have been in a shortest path at some stage. Indeed, if we remove an edge from a shortest path, we find a second shortest path of higher weight. Therefore, deleting those edges causes the lower bound to increase. We record them in a set $sp_R$ whenever we remove them. The resulting explanation is:

$$\left( \lceil w < K \rceil \land \bigwedge_{e \in E} b_e \land \bigwedge_{e \in sp_R} \neg b_e \right) \Rightarrow \text{fail}$$

where $\lceil w < K \rceil$ is the literal stating that we want a tree of cost less than $K$ (which is the cost of the best solution found so far). This explanation states that no better solution can be found given the edges in $G$ and the forbidden edges that could have lowered the weight of the solution.

3.3.6.4 Linear Program Lower Bound (LPLB)

Previous work in the pure STP used a linear program (LP) lower bounding technique to compute the solution to the problem [171]. This lower bound remains valid for any variant of the STP.

Cut Formulation of the STP  Any cut of $G$ that separates the nodes in two partitions $V = \{W, \overline{W}\}$ such that both contain at least one terminal must have at least one edge crossing from $W$ to $\overline{W}$ that is part of the solution. From this observation derives the cut formulation of the STP introduced by Aneja [13]:

$$\text{minimize} \sum_{e \in E} w(e) \cdot b_e \text{ such that:}$$
∀ W, \sum_{e \in \delta(W)} b_e \geq 1

where \( \delta(W) \) is the set of edges with exactly one endnode in \( W \).

The linear relaxation of this problem, which we call LPLB\((G)\), makes use of real variables \( x_e \in [0,1] \) for each edge \( e \) instead of the Boolean variables \( b_e \) and yields a lower bound to the pure STP.

### 3.3.6.5 Computing LPLB

Following the work of Polzin and Daneshmand [171], we implemented this lower bound using row generation and we solve it using a MIP solver (in particular, we used CPLEX12.4). Row generation is the idea of not adding all the constraints at once, but only when they are needed. Indeed, if we were to create the above model, we would need to enumerate all the cuts of the graph, and add all these cuts to the model, thus yielding a huge number of equations (of which most likely we won’t need many) that will slow down the solver. Instead, in row generation, the model is initialized with a small number of equations only. It is then solved and a separate, more efficient, algorithm looks for violations of the equations that were not in the model. If none are found, the solution is a solution to the original problem. If at least one is found, it is added to the model to ensure that the current solution is removed. This process is repeated until no violations are found.

In order to generate the rows (i.e. the equations), we use the Edmonds-Karp’s maximum flow algorithm [80]. Each run of this algorithm gives us a minimum cut that we add to the Linear Program. We also implemented the DUALASCEND algorithm described by Wong [220] to choose the initial set of equations to be added to the LP. In order to make this incremental, we run the flow computation and re-optimize LPLB\((G)\) after all the rules described for the tree propagator (Section 3.3.4) are applied.

### 3.3.6.6 Explaining LPLB

As before, part of the explanation must be the mandatory edges as they raise the weight of \( G \). Also any fixed (mandatory or forbidden) edge in \( R = \{ e \mid rc(x_e) \neq 0 \} \) that has non-


zero reduced cost (noted $rc$) is part of the explanation. In short, for a linear program, the reduced cost indicates how much the objective function coefficient of a variable must be reduced before the variable will be positive in the optimal solution. This is because edges with zero reduced cost would not change the value of the lower bound if they changed theirs, so they do not contribute to the lower bound. Let $R_f = R \cap \{b_e = false \mid e \in E\}$ and $R_t = R \cap \{b_e = true \mid e \in E\}$. The explanation is:

$$\lceil w < K \rceil \land \bigwedge_{e \in E} b_e \land \bigwedge_{e \in R_f} \lnot b_e \land \bigwedge_{e \in R_t} b_e \Rightarrow fail$$

3.4 Experimental Evaluation and Results

We modeled the pure STP and two variations in MINIZINC and solved them with the CHUFFED solver. We used the latest CHOCO [172] solver as a comparison since it includes the most up to date implementation of the CP(Graph) framework [72].

We ran all our tests with the SPLB and LPLB lower bounds as well as without lower bound (NOLB), a decomposition of the tree constraint instead of a tree propagator (noted NOPROP) and the LPLB with no learning (labeled n.l.) to compare the benefits of the lower bounds and learning. We also tested a version called SP+LPLB where SPLB runs first and if it does not prune, we run LPLB.

The benchmarks used in this study are from the SteinLib [124]. Note that a number of benchmarks have been solved to optimality in the pure STP but not with side constraints. We used the test-sets ES10FST (15 instances of 12 to 24 nodes), ES20FST (15 instances of 27 to 57 nodes) and B (11 instances of 50 to 75 nodes).

We used the same search strategy in all the implementations. The order of the variables is: edges sorted by decreasing weight, then nodes in arbitrary order. The value strategy is: try assigning the values $false$, then $true$ to each variable. This is the strategy that gave the best results. The idea is to remove heavy edges until an edge is required to keep the connectivity (and it will be the cheapest such edge).

All tests were run on a Linux 3.16 Intel® Core™ i7-4770 CPU @ 3.40GHz, 15.6GB of RAM machine. We used 5 hours as the time-out for all the tests and the geometric
average (including timed-out instances) to summarize the results. Time is indicated in seconds and the number of unsolved instances (if any) appears in parentheses in the tables. Subsections 3.4.1 and 3.4.2 present the models for the two variants followed by the results tables.

Table 3.2 shows the results for the pure STP. As can be seen, LPLB almost always gave the best results in terms of time. This is unsurprising, as for the pure STP Mixed Integer Programming is the state of the art.

<table>
<thead>
<tr>
<th>Dataset Configuration</th>
<th>Conflicts</th>
<th>Nodes</th>
<th>Propagations</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES10FST</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLB</td>
<td>12</td>
<td>15</td>
<td>123</td>
<td>0.01</td>
</tr>
<tr>
<td>LPLB</td>
<td>11</td>
<td>14</td>
<td>101</td>
<td>0.01</td>
</tr>
<tr>
<td>LPLB (n.l.)</td>
<td>12</td>
<td>22</td>
<td>112</td>
<td>0.01</td>
</tr>
<tr>
<td>SP+LPLB</td>
<td>10</td>
<td>13</td>
<td>99</td>
<td>0.01</td>
</tr>
<tr>
<td>NOLB</td>
<td>13</td>
<td>138</td>
<td>134</td>
<td>0.01</td>
</tr>
<tr>
<td>NoPROP</td>
<td>10686</td>
<td>1079</td>
<td>797998</td>
<td>1.75 (4)</td>
</tr>
<tr>
<td>CHOCO</td>
<td>130</td>
<td>138</td>
<td>198</td>
<td>0.01</td>
</tr>
<tr>
<td>ES20FST</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLB</td>
<td>729</td>
<td>816</td>
<td>6383</td>
<td>1.35 (2)</td>
</tr>
<tr>
<td>LPLB</td>
<td>492</td>
<td>534</td>
<td>3723</td>
<td>0.49</td>
</tr>
<tr>
<td>LPLB (n.l.)</td>
<td>632</td>
<td>1080</td>
<td>4636</td>
<td>0.47</td>
</tr>
<tr>
<td>SP+LPLB</td>
<td>477</td>
<td>520</td>
<td>3626</td>
<td>0.74</td>
</tr>
<tr>
<td>NOLB</td>
<td>746</td>
<td>831</td>
<td>7016</td>
<td>0.67 (2)</td>
</tr>
<tr>
<td>NoPROP</td>
<td>604055</td>
<td>604200</td>
<td>43168067</td>
<td>1065 (11)</td>
</tr>
<tr>
<td>CHOCO</td>
<td>45434</td>
<td>46175</td>
<td>67780</td>
<td>1.10 (3)</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLB</td>
<td>55979</td>
<td>83120</td>
<td>531033</td>
<td>22.25 (2)</td>
</tr>
<tr>
<td>LPLB</td>
<td>8097</td>
<td>9157</td>
<td>53731</td>
<td>5.17</td>
</tr>
<tr>
<td>LPLB (n.l.)</td>
<td>18000</td>
<td>37096</td>
<td>110711</td>
<td>6.99 (1)</td>
</tr>
<tr>
<td>SP+LPLB</td>
<td>8102</td>
<td>10372</td>
<td>68814</td>
<td>7.11</td>
</tr>
<tr>
<td>NOLB</td>
<td>114538</td>
<td>192343</td>
<td>1460071</td>
<td>17.80 (2)</td>
</tr>
<tr>
<td>NoPROP</td>
<td>177364998</td>
<td>179890423</td>
<td>18203144792</td>
<td>18000 (11)</td>
</tr>
<tr>
<td>CHOCO</td>
<td>80406257</td>
<td>80406288</td>
<td>152270771</td>
<td>3339.66 (5)</td>
</tr>
</tbody>
</table>

Table 3.2: Results for the pure Steiner Tree problem

### 3.4.1 The Grade of Service STP (GoSST)

In computer networks, computers have bit-rate requests that need to be matched by the network. This has important real world applications (e.g. video distribution described by Maxemchuk [142]). Moreover, networks are not unlimited: each edge has a maximum capacity $\kappa$. We call the capacity of a path $p$ the minimum of the capacities of the edges in $p$. Let $d_n$ be the demand of a terminal node $n$. 
The goal of the GoSST [78] is to find a minimum Steiner tree network such that for each pair of terminals, there is at least one path of capacity higher than the minimum of the demand of the two terminals. The state of the art in this problem is an approximation algorithm [116]. Figure 3.9 is an example of a network not satisfying the GoSST constraints.

Figure 3.9: Example of a non-satisfiable GoSST. Capacities are indicated next to the edges. No path reaching b has a minimum capacity 8. One solution would be to increase the capacity of edges with capacity 7, to at least 8.

To model this problem, we use Boolean variables $p_{i,j}^k$ indicating whether an edge $k$ is in the path from $i$ to $j$. Formally, we have the following constraints:

$$\forall \{u, v\} \in \mathcal{T}, \forall n \in \mathcal{V}, \sum_{e \in \text{outgoing}[n]} p_{u,v}^e \in \{0, 2\} \quad (3.6)$$

$$\forall \{u, v\} \in \mathcal{T}, \sum_{e \in \text{outgoing}[u]} p_{u,v}^e = 1 \quad (3.7)$$

$$\forall e \in E, \forall \{u, v\} \in \mathcal{T}, \left( -b_e \lor (\kappa_e < \min(d_u, d_v)) \right) \Rightarrow \neg p_{u,v}^e \quad (3.8)$$

Equation 3.6 enforces that a node $n$ must provide either no edge or two edges to each path. Equation 3.7 enforces that all terminals must contribute with one edge to each path of which they are an extremity. Lastly, Equation 3.8 enforces that, for any pair of terminals, forbidden edges or edges with lower capacity than their demand cannot be in the path connecting them.

Table 3.3 shows the results for this problem. SP+LPLB seems to dominate in the size
of the search, but in time it is not so clear: SPLB, LPLB and LPLB (n.l.) perform similarly. It is clear, nonetheless, that having a lower bounding technique is beneficial and NOLB is slower than the others. Moreover, the decomposition approach (NoPROP) is substantially slower than our approach. CHOCO could not keep up with our results. By investigating further, it seems that the tree propagator used in the version 3.3.0 of CHOCO that we tested does no propagation, and only checks that the solution is indeed a tree.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Configuration</th>
<th>Conflicts</th>
<th>Nodes</th>
<th>Propagations</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>E8</td>
<td>10FST</td>
<td>SPLB</td>
<td>5</td>
<td>1124</td>
<td>7423</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LPLB</td>
<td>4</td>
<td>1173</td>
<td>6635</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LPLB (n.l.)</td>
<td>4</td>
<td>1178</td>
<td>6599</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SP+LPLB</td>
<td>4</td>
<td>1121</td>
<td>6428</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOLB</td>
<td>5</td>
<td>1178</td>
<td>7839</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NoPROP</td>
<td>5</td>
<td>1178</td>
<td>7839</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CHOCO</td>
<td>65</td>
<td>69</td>
<td>67358</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CCHOCO</td>
<td>3373</td>
<td>3433</td>
<td>22745514</td>
</tr>
</tbody>
</table>

Table 3.3: Results for the GoSST

3.4.2 The Terminal Steiner Tree Problem (TSTP)

The Terminal STP [138] is a small variation of the original problem used in VLSI and phylogenetic studies. In such environments, we might need a terminal to be a leaf. This only affects the degree of the terminals and can be achieved by using the following constraint:

$$\forall t \in T, \sum_{e \in \text{outgoing}[t]} b_e = 1$$

We only use the testset B because most benchmarks in the other sets were proven unsat-
isfiable too fast to show any significant result. Again, all the work existing in this problem is in approximation algorithms [41, 76].

The results for this problem clearly show the value of having a propagator like ours instead of a simple decomposition. Lazy Clause Generation also proved to be useful in this problem.

### 3.5 Concluding Remarks

We can clearly see that both our propagator and the explanations are greatly beneficial to solve the problems faster and with fewer nodes (i.e. a smaller search space). Also, LPLB is overall the best in time, although SP+LPLB is usually better in all the other measures. This is because having both lower bounding techniques in the same propagator has a higher runtime cost when the weaker lower bound (SPLB) is not good enough to stop the search and we need to run LPLB.

The contributions of this chapter are:

- A new propagator for the connected constraint, with explanations
- A new propagator for the acyclic constraint, with explanations
- A new propagator for the tree constraint, with explanations
- A novel lower-bounding technique for the steiner.tree constraint (SPLB), with explanations
- Explanations for an already existing lower bound for the STP (LPLB)
3.5 Concluding Remarks

In Chapter 4 we will see a real-world application of this work in a problem called the Relational-To-Ontology Schema Mapping.
Chapter 4
Steiner Trees Applied to the Relational-To-Ontology Schema Mapping Problem

In this chapter we present an application of our work on Steiner Trees in CP to help solve a data integration problem. The problem of integrating heterogeneous data sources into an ontology is of high relevance in the database field. Several techniques exist to approach this problem. Nonetheless, side constraints on the data integration cannot be easily integrated with these tools and thus the results may be inconsistent. The approach presented in this chapter combines Machine Learning and Constraint Programming techniques, by modeling the data integration problem as a Steiner Tree Problem in a weighted graph. Our experiments show that through this approach we can achieve better precision, recall and speed compared to state-of-the-art approaches. We provide a comprehensive set of experiments supporting our findings.

4.1 The Rel2Onto Problem

Relational data sources are still one of the most popular ways to store enterprise or Web data. However, the issue with relational schema is the lack of a well-defined semantic description. A common ontology provides a way of representing the meaning of a relational schema and can facilitate the integration of heterogeneous data sources within a domain. Indicating semantic correspondences manually might be appropriate if only few data sources need to be integrated, however, it becomes tedious with the growing number of heterogeneous schemata.

Automatically integrating heterogeneous data sources is a long standing issue in the database research field and is of high relevance in many real-world domains [66, 175, 69].
A standard approach to tackle this problem is to design a common ontology and to construct source descriptions which specify mappings between the sources and the ontology [68]. In this chapter, we approach the data integration problem consisting in automatically mapping a new relational data source onto a user provided ontology. To do so, we develop a new system that automatically builds a semantic model which describes a relational data source in terms of concepts and relationships defined by an ontology [209].

**Example 4.1.** Consider the situation where we have some simple relational database tables with columns \(\langle \text{Surname}, \text{Event}, \text{Date} \rangle\), and \(\langle \text{Company}, \text{Festival}, \text{Address} \rangle\). We do not a priori know if “Date” is a date of birth of the person or the start date of the event (c.f. Figure 4.1); or whether “Address” refers to the name of the place where the festival is located, or the email address of the company. Given an ontology like the one in Figure 4.1, we would like to automatically map the new data sources to the ontology.

We use Machine Learning techniques (ML) to learn mapping rules from previously mapped instances. To this end we formulate the Relational-To-Ontology Mapping Problem (REL2ONTO) as a Steiner Tree Problem with side constraints, as seen in Chapter 3. To outline our approach, firstly, we build a graph which includes attributes (i.e. columns) from the new source as well as ontology classes and properties. We name it the integration graph. It uses information gathered from the previously mapped data sources, and is further extended with information derived from the ontology. Secondly, we apply Machine Learning techniques to create a weighting function \(w_I\) over its set of edges. Lastly, we use Constraint Programming to find a minimum cost Steiner Tree in the graph. The goal is to design \(w_I\) in such a way that the resulting Steiner Tree is a valid and coherent semantic model for the new source.

Section 4.1.1 formally states the problem and introduces the required vocabulary. In Section 4.2 we present how we convert learnt data into a useful data representation. That section will also have a brief overview of Machine Learning to ease the reading of this chapter. Section 4.3 shows how we model the problem as an STP, whereas Section 4.4 presents our implementation in CP. Section 4.5 shows our results. Section 4.6 compares
with the previous work done to achieve this task (this is presented at the end as it is easier to understand the differences between other work and ours once our approach has been presented).

4.1.1 Problem Statement and Vocabulary

Figure 4.1: Example of ontology (‘’ are Classes, ‘’ means “has an object property”, ‘’ means “has a data property” and ‘’ means “subclassOf” ). Example by Taheriyan et al. [207].

In our work we consider that an ontology \( O \) includes basic elements such as classes (which represent concepts), literal values, individuals (members of classes) and properties [201]. Properties are classified into object properties, which relate two individuals, and datatype properties, which relate individuals to literal values. Figure 4.1 gives an example of an ontology. Here, an individual of the class “Organisation” can be related to an individual of the class “Person” via object properties “ceo” or “worksFor”, and it can have a data property “name”. We also consider a special “subclassOf” (or “is-a”) type of object properties. For example, “Cities” and “States” are both “Places”. A subclass inherits all the properties of the parent class.

A semantic model \( m \) is a directed graph with two types of nodes: class nodes and data nodes. We denote them as \( C_m \) and \( D_m \), respectively. \( C_m \) corresponds to classes in the ontology whereas \( D_m \) corresponds to data properties. Edges in the semantic model correspond to properties in the ontology, as shown in Figure 4.2 (notice the same edges appear in Figure 4.1). The semantic model may have several instances of the same ontology class, that is why class nodes are enumerated. Semantic models can be formalized using a standard
mapping language such as R2RML [216].

Figure 4.2: Examples of two semantic models (‘∅’ are class nodes, and ‘∈’ are data nodes).

In our setting we work with relational sources. Hence, a data source \( s \) is a \( n \)-ary relation with a set of attributes \( A_s = (a_1, \ldots, a_n) \). We want to map them to the target ontology \( O \).

Following the traditional data integration framework [68], we decompose the problem into two parts: schema matching and schema mapping. Schema matching, which we refer to as semantic labeling, finds correspondences between attributes from data sources and data nodes of the target ontology. In the schema mapping part we want to generate the semantic models of data sources by identifying the connecting paths for the matched data nodes.

An attribute mapping function \( \phi : A_s \mapsto D_m \) is a function which maps the attributes of the source \( s \) into the nodes of the semantic model \( m \). It can be a partial mapping, meaning that only some of the attributes are connected to the nodes of \( m \). The attribute mapping function addresses the first part of the problem (i.e. the semantic labeling).

We define a source description as a triple \( \delta = (s, m, \phi) \), where \( s \) is a source, \( m \) is a semantic model, and \( \phi \) is an attribute mapping function. Our problem can hence be stated as follows. We have an ontology \( O \) and a set of source descriptions \( \Delta_T = \{ (s_1, m_1, \phi_1), \ldots, (s_l, m_l, \phi_l) \} \). Given a new source \( s^* \), we want to build a semantic model \( m^* \) and an attribute mapping function \( \phi^* \) such that \( \delta^* = (s^*, m^*, \phi^*) \) is an appropriate source description. We use the term “appropriate” since there might be many such triples which are well-formed source descriptions, but only one or a few will capture the intended meaning of the source. Our goal is to automatically build \( \delta^* \) such that it maximizes the precision and recall between the semantic model \( m^* \) and the semantic model \( m^\dagger \) that the user considers correct.
Example 4.2. As an example of this problem, consider the situation where we have the ontology in Figure 4.1 and the three tables below, in Table 4.1:

<table>
<thead>
<tr>
<th>S.O.B.</th>
<th>C.O.B.</th>
<th>D.O.B.</th>
<th>Employee</th>
<th>Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Organizer</th>
<th>President</th>
<th>Title</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Table 4.1: Example of tables to be integrated in the ontology.

Assume we were be able to identify semantic models for the two top tables. These semantic models are in fact shown in Figure 4.2 and could have been built by hand: each column of the two tables corresponds to a data node of the shown semantic models. For instance “S.O.B.” maps to “State1.name” in the first one, and “President” maps to “Person2.name” in the second one.

We would like now to integrate the third table ⟨Name, Date, At⟩ in the ontology. By just looking at the column titles and the first row, one could think that this is referring to the date of birth of a person that lives in a certain street (in the case of the first row, Mr. Hanson was born on an 18th of May, and lives at 221B Baker Street). Nonetheless, by looking at the second row, it seems more likely that the table represents the date and venue of some concert (indeed, Hanson was a music band, and 221B Baker St could very well be a concert room named after Sherlock Holmes’ address). The semantic model we would like to find for this new table is shown below in Figure 4.3.

![Figure 4.3: Expected semantic model.](image)

Our task is indeed to identify that semantic model as being the most likely semantic model for the new table, given the two known semantic models for the other two tables.
4.2 Machine Learning for Training on Source Descriptions

4.2.1 Overview on Machine Learning

Machine Learning is out of the scope of this thesis, and therefore we will not go into much detail in this section. Nonetheless, we will introduce the basic concepts for the reader to understand this chapter. A much more thorough understanding on machine learning can be gained from the book by Kelleher et al. [119].

In short, Machine Learning (ML) is the task of predicting a quantity or class for an object. Simple examples are predicting the price of a house in 5 years, or classifying a tumor as benign or cancerous. There are two types of problems in ML: regression and classification problems. The former tries to predict a continuous value (like a price) and the latter a discrete value (like a label). In this chapter we use classification.

In order to do these predictions, ML algorithms are given a set of data called the training set. A simple example would be to provide 50 pairs \((\text{price}, \text{m}^2)\) and try to create a function \(\psi : \mathbb{R} \rightarrow \mathbb{R}\) that, accurately predicts the price of a house given its size. Of course, one could imagine much more complex scenarios where more attributes are taken into account to perform the prediction, such as the criminality in the neighborhood, the orientation, the number of bathrooms or the number of rooms. These are called features.

4.2.1.1 Regression

Assume we only have one feature, the size of a house, and we want to predict its price. We would obtain data from 15 houses and plot it as in Figure 4.4a. In a regression problem we are trying to design function \(\psi\) such that it fits the data as much as possible (without over-fitting it, as that can result in bad recall afterwards). One way of approaching this problem is to design function \(\psi\) using a linear function, and minimize the mean square error between the function and the training data. This typically performs poorly, so one idea is to increase the polynomial of the function. Figure 4.4a shows a cubic regression that fits the data relatively well. Another approach, which is what we will be using in this chapter, is that of random forests. The idea of a random forest, for regression problems,
is to create a step function that gives a value for each interval. This is shown in Figure 4.4a in green. The name random forest comes from the fact that during the training of the model, we create decision trees where the decisions are on the values of the features. An example of such a tree is given in figure 4.4b.

Each tree is built using only part of the training set. Therefore, a random forest is composed of several trees that were built from different data from the same dataset. By analogy to polynomials, keeping just one tree would be like keeping a linear function for the regression, whereas averaging several trees provides a more flexible function to match the data (just as a higher polynomial function).

When a prediction needs to be done (in this case the price of a house given its size), the predicted value will be the average of the predicted values of each tree.

### 4.2.1.2 Classification

The problem of classification is the equivalent of regression, but for a discrete domain to be predicted. We can imagine a set of features for tumor classification such as: age of the patient, size of the tumor, do they smoke, location of the tumor, etc. Based on
those features, a model would predict if the tumor is benign or cancerous. Of course, it is possible to have more than two outcomes in a classification problem.

For classification problems, a random forest will create trees where the internal nodes are decisions on the value of the features and the leaves are predictions. A tree similar to the one shown in Figure 4.4b will be created. In this case the internal nodes would be, for example, \( \text{size} < 1 \text{cm}^3 \), \( \text{location} = \text{lung} \) or \( \text{age} > 55 \), and the leaves would be \( \text{Benign} \) or \( \text{Cancer} \).

In classification problems, the classifier provides the best fitting class for the prediction, and a confidence value (a probability). In the specific case of a random forest, each tree will provide a vote for each class, and the majority vote will be the final class selected. For example if 2 trees classify a tumor as cancer and 498 others classify it as benign, the final outcome would be benign. We will see how for the semantic labeling part of our problem we use a random forest classifier and the confidence values as weighting functions for Steiner Trees.

### 4.2.2 Semantic Labeling

The semantic types \( \mathcal{L}_O = \{l_1, l_2, ..., l_p\} \) of an ontology correspond to all pairs \( l_i = (c_i, d_i) \), where \( c \) is a Class in \( O \), and \( d \) is a data property of that class (including inherited properties). For example, from the ontology in Figure 4.1, we would get (City, name) and (State, name), among others.

The first step to model the semantics of a new source \( s^* \) is to recognize the semantic types present in the source. We call this step semantic labeling, which assigns a confidence value to a match of an attribute from \( s^* \) to a type \( l \in \mathcal{L}_O \). Typically semantic labeling techniques encounter several problems. Firstly, there can be naming conflicts [169], including those cases where users represent the same data in different ways. Secondly, semantically different attributes might have syntactically similar content, for example, “startDate” versus “endDate” for the Event class. Thirdly, there may be a considerable number of attributes which do not have any corresponding property in the ontology, either by accident or deliberately.

We formulate the problem of semantic labeling as a multi-class classification problem.
The known source descriptions $\Delta_T$ provide us the training sample. We compute a feature vector for each attribute in a data source and associate the known semantic type with the corresponding feature vector. The feature vector includes, among others, characteristics such as a number of whitespaces and other special characters, statistics of values in the column. A full list can be found in Tables 4.2 and 4.3.

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num-unique-vals</td>
<td>A simple count of unique entries of a column.</td>
</tr>
<tr>
<td>prop-unique-vals</td>
<td>The proportion of unique entries of a column.</td>
</tr>
<tr>
<td>prop-missing-val</td>
<td>The proportion of all missing/empty entries from all entries in a column.</td>
</tr>
<tr>
<td>ratio-alpha-chars</td>
<td>The average of the proportion of alphabetic characters from each entry.</td>
</tr>
<tr>
<td>prop-numerical-chars</td>
<td>The average of the proportion of numeric characters from each entry.</td>
</tr>
<tr>
<td>prop-whitespace-chars</td>
<td>The average of the proportion of whitespace characters from each entry.</td>
</tr>
<tr>
<td>prop-entries-with-at-sign</td>
<td>The proportion of entries with an '@' sign.</td>
</tr>
<tr>
<td>prop-entries-with-hyphen</td>
<td>The proportion of entries with a '-'</td>
</tr>
<tr>
<td>prop-range-format</td>
<td>The proportion of entries which follow some numerical range format ([0-9]+)-(0-9]+)</td>
</tr>
<tr>
<td>is-discrete</td>
<td>A binary indicator if the column entries are discrete.</td>
</tr>
<tr>
<td>entropy-for-discrete-values</td>
<td>The entropy of the entries of a column.</td>
</tr>
<tr>
<td>shannon-entropy</td>
<td>Shannon’s entropy for the vocabulary specified in the description of the feature “char-dist-features”</td>
</tr>
</tbody>
</table>

Table 4.2: Feature extractors that produce a scalar value

One of the important features characterizing information content of an attribute is Shannon’s entropy. Shannon’s entropy (or information entropy [141]) of a string $X$ is defined as $H(X) = - \sum_i p_i \log_2(p_i)$, where $p_i$ is the probability of a character, whose index in character vocabulary is $i$, to appear in $X$, and the summation ranges over all characters in the vocabulary. To evaluate $p_i$ in Shannon’s entropy, we evaluate the normalized character frequency distribution $\text{chardist}$ of an attribute, as character counts in concatenated rows of the attribute, normalized by the total length of the concatenated rows. The vocabulary of all characters consists of 100 printable characters (including \n). Finally, we
<table>
<thead>
<tr>
<th>Feature name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inferred-data-type</td>
<td>Produces a vector which contains the inferred data type. Only a single element is non-zero in the vector (1-of-K format). The set of inferred types are in float, integer, long, bool, date, time, datetime, string.</td>
</tr>
<tr>
<td>char-dist-features</td>
<td>Produces a vector for character distribution of column values; considered characters are printable characters</td>
</tr>
<tr>
<td>stats-of-text-length</td>
<td>Computes the mean, median, mode, min, max of string lengths from a columns entries.</td>
</tr>
<tr>
<td>stats-of-numerical-type</td>
<td>Computes the mean, median, mode, min, max of entries from a column of numerical type.</td>
</tr>
<tr>
<td>prop-instances-per-class-in-knearestneighbours</td>
<td>This finds the $k$ columns from the training set with most similar names. It then produces a vector of class proportions from these $k$ neighbours. Parameters: “num-neighbours”: The number of neighbours to find ($k$).</td>
</tr>
<tr>
<td>mean-character-cosine-similarity-from-class-examples</td>
<td>For some instance $t$, computes the character distribution from this instance’s entries, and produces a vector of the average cosine similarity of $t$ with training instances of each class.</td>
</tr>
<tr>
<td>min-editdistance-from-class-examples</td>
<td>For some instance $t$, produces a vector of the minimum column name edit distance between $t$ and all training instances of each class. Parameters: “cache-size”: size of cache to store edit distances of words; “max-comparisons-per-class”: max number of training instances to compare for each class.</td>
</tr>
<tr>
<td>min-wordnet-jcn-distance-from-class-examples</td>
<td>For some instance $t$, produces a vector of the minimum column name JCN WS4J distance between $t$ and all training instances of each class. Parameters: “cache-size”: size of cache to store wordnet distances of words; “max-comparisons-per-class”: max number of training instances to compare against for each class.</td>
</tr>
<tr>
<td>min-wordnet-lin-distance-from-class-examples</td>
<td>For some instance $t$, produces a vector of the minimum column name LIN WS4J distance between $t$ and all training instances of each class. Parameters: “cache-size”: size of cache to store wordnet distances of words; “max-comparisons-per-class”: max number of training instances to compare against for each class.</td>
</tr>
</tbody>
</table>

Table 4.3: Feature extractors that produce a vector value. Some of these are computed together to reduce computation time. Others are values that are computed for all classes.

add the 100-dimensional vector of $p_i$ to the attribute feature vector.

We also compute a set of features based on similarity metrics inspired by state-of-the-art works by Pham et al. [167] or Ritze and Bizer [186]. Among others, we compute mean cosine similarity for character distributions of attribute values and string similarity metrics for attribute names. We train a random forest on the obtained sample.
In this way, we learn the mapping $\psi : A_s \times L_O \mapsto [0..1]$, where $\psi(a_i, l_j)$ indicates the confidence that the attribute $a_i$ is mapped to the semantic type $l_j$. For example, if a column contains the ‘@’ sign once in every row, the column is likely to be an email address. Conversely, if a column contains only numbers, it is unlikely that it refers to the name of a person. Note that we keep all the matches, regardless of the confidence of the match. This is an important difference between our system and other approaches such as the one by Taheriyan et al. [209] that remove some of the matches based on heuristics in order to simplify the task of finding the semantic model.

### 4.2.3 Alignment Graph

To provide an integrated view over the known source descriptions $\Delta_T$, we need to align their semantic models as well as all considered semantic types. This is achieved by constructing an alignment graph.

The alignment graph is a directed weighted graph $G_O = (V_O, E_O)$ built on top of the known semantic models and expanded using the semantic types $L_O$ and the ontology $O$. Similar to a semantic model, $G_O$ contains both class and data nodes. The links correspond to properties in $O$ and are weighted [209].

The algorithm we use to construct the alignment graph $G_O$ was given by Taheriyan et al. [209]. Briefly, it has three parts 1) adding the known semantic models to $G_O$, 2) adding the semantic types learned for the target source to $G_O$ and 3) expanding the graph using the domain ontology.

An example alignment graph is illustrated in Figure 4.5, following the scenario outlined in Example 4.2. Black links correspond to the links which are supported by the known semantic models. Blue and orange links are inferred from the ontology $O$ (the orange ones correspond to the ones added to connect Classes of the ontology that never appeared in the semantic models).

Note how the alignment graph contains data nodes that correspond to semantic types. For instance, it could contain two nodes “City.name” and “State.name” rather than just one node “name” connected to two nodes “City” and “State”. We say these nodes are induced into the alignment graph by the semantic types. We note them $D_{G_O}$ and call them...
Figure 4.5: Example of alignment graph ("\( \phi \)" are class nodes, and "\( \psi \)" are data nodes) built from the semantic models in Figure 4.2 and the ontology in Figure 4.1. We omit weights for clarity.

the data nodes of the alignment graph. Class nodes are noted \( C_G \).

The graph is weighted by a function \( w_O : E_O \rightarrow \mathbb{R} \) such that edges which are present in the known semantic models have lower weights than those which are inferred from the ontology. In the next section we will see that this makes edges from semantic models more attractive. Taheriyan et al. [209] provide details on the weighting function.

### 4.2.4 Frequent Graph Pattern Mining

Certain patterns of connections can be prevalent in the domain. For example, in Figure 4.2 both semantic models have class nodes “City”, “Organization” and “Person”. According to the ontology in Figure 4.1 there are multiple ways to connect these nodes. However, if we know that the “Person” works for the “Organization”, then based on the known semantic models (Figure 4.2), “City” is more likely to be the birth place of the “Person” rather than the location of the “Organization”. To increase the coherence of the generated semantic models, we would like to discover such patterns of connections.

As an extension to the problem, we consider using pattern mining techniques in order to identify patterns in the training set of semantic model that repeat themselves often. The hope is that with such information we can encourage the semantic model \( m^* \) for the new source to use groups of edges that appeared often together in training semantic models.

In our context, patterns are directed graphs. We mine these patterns from the set of
4.3 Steiner Tree Formulation

Up to this stage, we have created an alignment graph that takes into account all the information in the training set (semantic models) as well as the ontology. We are now given the alignment graph and a new data source $s^*$ to be integrated and we need to find a semantic model for the table $s^*$, which will consist in a matching between the attributes of $s^*$ and the data nodes $D_{O}$ of the alignment graph as well as a subgraph of the alignment graph connecting the matched data nodes. To do so, we formulate the REL2ONTO schema mapping problem as a Steiner Tree Problem (STP) for a new source $s^*$, we construct the integration graph noted $I_{O}^s = (V_{O}^s, E_{O}^s)$, where $V_{O}^s = V_{O} \cup A_{s^*}$. This will be the graph in which we will look for a Steiner Tree.

The set of edges $E_{O}^s$ is constructed by using all the edges in the alignment graph, and edges connecting each attribute of $s^*$ to the nodes in the alignment graph induced by the semantic types (i.e. the set of nodes in $D_{O}$). We call this last set of edges $M_{O}^s$ (for “matches”). Thus, $E_{O}^s = E_{O} \cup M_{O}^s$.

We associate a weighting function $w_T : E_{O}^s \mapsto \mathbb{R}^+$ to the integration graph. For an edge $e \in E_{O}$, $w_T(e) = w_O(e)$. For an edge $e \in M_{O}^s$ connecting attribute $a_i$ to the node $l_j$ induced by the semantic types, $w_T(e) = -\ln(\psi(a_i, l_j))$, making unlikely matches have a
higher weight.

**Example 4.3.** An example of an integration graph can be found in Figure 4.6, following the scenario outlined in Example 4.2.

![Class nodes of Alignment Graph C\*\textsubscript{\text{GO}}](image)

Figure 4.6: Example of integration graph. (\textbullet{} are attribute nodes). We omit weights for clarity. Data node are the same as in Figure 4.5.

Due to lack of space, weights are not shown. In this example we could expect that the edge from “Place2.name” to “At” could have a weight of 0.1 whereas the edge “Event1.endDate” to “At” would have a much higher weight, such as 0.9, indicating that it is more likely that column “At” contains the name of a place rather than a date.

Note that, although the alignment graph is directed, we disregard the direction of edges for the STP model. We disambiguate the direction and semantics of edges via their weights. We can further improve this step by using patterns.

The goal is to build a subgraph $T^* = (V^*, E^*)$ of $I^{s^*}_O$ for the new source $s^*$. The solution $T^*$ will be used to build the source description $\delta^*$. In particular, $(V_O \cap V^*, E_O \cap E^*)$ corresponds to the semantic model, and $(A_{s^*}, \mathcal{M}_{s^*}^O \cap E^*)$ corresponds to the attribute mapping function.

The solution $T^*$ must satisfy the following constraints:

1. $T^*$ must be a subgraph of $I^{s^*}_O$
2. $T^*$ must be a tree
3. $\forall a \in A_{s^*}, a \in V^*$
4. $\forall a \in A_{s^*}, \text{degree}(a) = 1$
4.3 Steiner Tree Formulation

5. \( \forall n \in D_{\mathcal{G}_O} \cap V^*, \text{degree}(n) = 2 \)

6. \( V^* \cap \mathcal{C}_{\mathcal{G}_O} \neq \emptyset \)

It is therefore natural to model this problem as a STP with side constraints. By designing the weighting function \( w_T \) through ML techniques, as shown in Section 4.2, our expectation is that the minimum cost Steiner Tree is a valid and coherent semantic model for the new source.

4.3.1 Using Patterns

As explained in Subsection 4.2.4, we also use graph patterns in order to incentivise the solution tree \( T^* \) to contain subgraphs of the alignment graph that have been frequently seen in the training set. To do this, we use the support of each of the obtained patterns as a prize. If the tree contains a pattern, then the objective function is automatically reduced by the value of the support of that pattern. We will see in the next section how this information is integrated in the model.

4.3.2 Unmatched Attributes

It is common that the data sources to be integrated will have columns that simply cannot be matched to the ontology. This can happen when a column of a source table contains some information that is uninteresting to the user, or because the ontology has not been properly designed. Examples of these situations can be found in domain specific data [167] or HTML tables [186]. In current systems, these columns are removed in a pre-processing step as a manual effort.

For this reason, we add two artificial Class nodes to the integration graph: \textit{unknown} and \textit{root}. The latter will be connected to every other Class node in \( V^*_{\mathcal{O}} \), including \textit{unknown}. We also add a set \( U = \{ unk_1, ..., unk_{|A_s|} \} \) of \( |A_s| \) data nodes to the integration graph, each connected to exactly one node in \( A_s \) and to the \textit{unknown} Class node.

If an attribute \( a \) is matched to \( unk_a \), then \( unk_a \) will be linked to Class node \textit{unknown}. The rest of the attributes can then be matched normally and build a semantic model as
usual. To maintain connectedness of the Steiner tree, the unknown node and one of the other Class nodes in $T^\star$ will both be connected to the root node. Note that if all attributes find a match, then both the unknown and root nodes will not be selected in the Steiner tree and the normal behavior will take place. The weights of the edges between these special nodes are assigned the same way as for the rest of the edges.

**Example 4.4.** Assume we have the ontology of Figure 4.1, and a new table to be integrated $\langle$Name, Office, Dietary Allergies$\rangle$. In this case, the last column of the table cannot be integrated with the ontology. Either this column is irrelevant to the designer of the ontology, or the design of the ontology is erroneous. Either way, our system would match this last column to “unknown” and show it to the user to reconsider the design of the ontology if necessary.

### 4.4 Modeling in Constraint Programming

In order to model the REL2ONTO problem we used the MINIZINC language and the solver CHUFFED, because this is where we implemented our steiner_tree constraint.

Because attributes must be connected to exactly one node of the alignment graph, and that node will be in $D \cap V^\star$, then the part of the problem between attribute nodes and the alignment graph is actually a matching problem. Each attribute must match exactly one node of $D$. Note that not all nodes in $D$ need to match to an attribute, as not all of them are part of $T^\star$.

Because there are global constraints in CP specialized in matching [179], we split the problem into two parts: the steiner_tree global constraint will only deal with the part of the integration graph that corresponds to the alignment graph, and the alldifferent global constraint will deal with the matching part of the problem.

As in Chapter 3, we will have a graph variable $G = (V, E)$. Again, $G$ is encoded with two sets of Boolean variables $v_{V}^O$ and $v_{E}^O$. We denote $b_x$ the Boolean variable for a specific object $x$ (node or edge) in $v_{V}^O$ and $v_{E}^O$. The graph variable $G$ represents the part of $T^\star$ that overlaps with the alignment graph, that is, the semantic model. We also have an array of variables match indexed by the set of attributes of $s^\star$. The semantics of this array is that $match[a] = d$ (for $a \in A_s$ and $d \in D_{GO}$) means that the edge $(a,d) \in M^s_O$ is
part of $T^*$. This array represents the part of $T^*$ that corresponds to the attribute mapping function.

Additionally, for a given set of patterns $P$ with a support function $w_P : P \rightarrow \mathbb{R}$, we have a set of Boolean variables $c_p, \forall p \in P$ that tell us whether a pattern $p$ appears in $T^*$ or not. The model is presented below.

Minimize $w_{STP} + w_{ADIFF} - w_{PAT}$ such that

\begin{align}
\text{steiner\_tree}(v_{V_O}, v_{E_O}, G_O, w_O, w_{STP}) \\
\forall d \in D_{G_O}, \text{deg}(d) \leq 1 \\
\forall d \in D_{G_O}, b_d \iff \text{deg}(d) = 1 \\
\forall a \in A_{s^*}, \text{match}[a] \in \{d \mid (a, d) \in M_{s^*}^O\} \\
\text{alldifferent}(\text{match}) \\
\forall a \in A_{s^*}, b_{\text{match}[a]} = \text{true} \\
w_{ADIFF} = \sum_{(a,d) \in M_{s^*}^O} w_T((a,d)) * \mathbb{1}[\text{match}[a] = d] \\
\forall p \in P, (\forall e \in \text{edges}(p), b_e = \text{true}) \iff c_p \\
w_{PAT} = \sum_{p \in P} w_P(p) * c_p \\
b_{\text{unknown}} \Rightarrow (b_{\text{root}} \land b_{(\text{unknown,root})})
\end{align}

Equation 4.1 is the objective function: we are minimizing the cost of $T^*$ while collecting prizes for each pattern we use. Equation 4.2 enforces that the graph variable $G$ is indeed a tree of total weight $w_{STP}$. This automatically implies that $T^*$ will be a tree. Equations 4.3 and 4.4 ensure that if a data node of the alignment graph is selected, then at most one edge reaches it (from the side of the alignment graph) and otherwise it is disconnected. Equations 4.5 and 4.6 ensure that there is exactly one data node matched to an attribute. Equation 4.7 ensures that if a data node of the alignment graph has been mapped to some attribute, then that data node must be in the solution tree, and vice-versa. Equation 4.8 computes the cost $w_{ADIFF}$ of the matches. Equation 4.9 indicates that
a pattern is used if and only if all its edges are selected in the tree. Equation 4.10 computes the prizes collected by using patterns. Equation 4.11 ensures that if the unknown class node is used, then it is connected to root. Notice there is no further requirement for unmatched attributes, as the unk data nodes behave like normal data nodes, and the connectedness requirement will make sure that root is connected to the rest of the tree (i.e. the semantic model). Note how this CP model is easy to adapt to more classic settings where no patterns are used (by dropping Equations 4.9 and 4.10), or without unknown nodes (by dropping Equation 4.11).

We choose a search strategy that will first try to match attributes using the cheapest edges in $M^*_s$. Then, it will try to use the cheapest unfixed edge $e \in E_O$ that fills a pattern $p$ (i.e. $\forall e_p \in p, e_p \neq e, b_{e_p}$) such that $w_p(p) > w_O(e)$ or simply the cheapest edge if none fill a pattern.

4.5 Experimental Evaluation and Results

4.5.1 Experimental Setup

We call our system SERENE if no patterns are used in the model, and SERENEPATS otherwise. We run experiments on two domains: museum and soccer (see Figure 4.4 for statistics). The museum dataset [209] contains 29 sources which are mapped to the EDM domain ontology. The soccer dataset [167] is much smaller with only 12 data sources, and its domain ontology is an extension of the schema.org ontology.

<table>
<thead>
<tr>
<th>Domain</th>
<th># data sources</th>
<th># semantic labels</th>
<th># attributes</th>
<th># unknown attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>museum</td>
<td>29</td>
<td>20</td>
<td>443</td>
<td>159</td>
</tr>
<tr>
<td>soccer</td>
<td>12</td>
<td>18</td>
<td>138</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 4.4: Description of data sources

We choose KARMA [209] as our baseline. This system also phrases the REL2ONTO problem as STP and decomposes it further into two parts. However, it uses heuristic algorithms both for the matching and for the STP parts. It solves the problems sequentially, i.e., once it produces a set of candidate mappings for attributes into the ontology, it fixes
this set and moves onto the STP part. Additionally, it does not consider unmatched attributes in the sources. To ensure that KARMA also handles such attributes, we change its semantic labeling model, SemanticTyper [176], to ours and we add a special unknown ontology which gives specification for root and unknown nodes of the alignment graph. The first modification also ensures the fairness of evaluation since both KARMA and SERENE will have the same matches for attributes.

The performance of these systems is estimated in terms of precision and recall. Assuming \( m^* \) is the predicted semantic model and \( m^\dagger \) is the correct semantic model, then:

\[
\text{prec} = \frac{|\text{rel}(m^*) \cap \text{rel}(m^\dagger)|}{|\text{rel}(m^*)|}, \quad \text{recall} = \frac{|\text{rel}(m^*) \cap \text{rel}(m^\dagger)|}{|\text{rel}(m^\dagger)|}
\]

where \( \text{rel}(m) \) is the set of triples \((u, e, v)\) with \( e \) being an edge from the vertex \( u \) to the vertex \( v \) in the semantic model \( m \). Note that unmatched attributes as well as unknown and root nodes are not part of these sets. We also perform a modification to the ground truth semantic models: if the true semantic label of an attribute \( a \) is not present in any of the previously mapped data sources, we substitute it with \( \text{unk}_a \) and map it to the unknown class node. This way we can validate how well the systems can detect earlier unseen cases.

To illustrate that our semantic labeling model is better suited for the matching task, we compare it against the state-of-the-art model DSL [167] which was shown to perform even better than SemanticTyper. We use mean reciprocal rank (MRR) to evaluate semantic labeling models. This measure is useful to estimate how highly the true semantic label is ranked among the top \( k \) suggestions. It is calculated the following way:

\[
\text{MRR} = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\text{rank}_i},
\]

where \( \text{rank}_i \) is the rank of the correct semantic label for the attribute \( a_i \) among the top \( k \) predictions and \( n \) is the number of attributes in the data source.

We perform an evaluation strategy outlined by Taheriy et al. [209]. Let \( M_j \) be the set of \( j \) known semantic models. For each data source \( s_i \) in the domain we perform experiments \( t - 1 \) times, where \( t \) is the total number of data sources in the domain and each experiment has a different number of known semantic models \( M_1, M_2, \ldots, M_{t-1} \). The case with \( M_{t-1} \) known semantic models corresponds to leave-one-out validation strategy. For example, in the soccer domain for the source \( s_1 \) we run experiments 11 times using
$M_1 = \{m_2\}, M_2 = \{m_2, m_3\}, \ldots, M_{11} = \{m_2, m_3, \ldots, m_{12}\}$. We repeat the procedure for other data sources in the domain and then average the results. This procedure ensures that each source is at least once in the training and testing datasets.

From a CP perspective, the instances are fairly large: the typical alignment graph provided to the \texttt{tree} constraint is about 150 nodes and 4000 edges. The matching part of the problem, given to the \texttt{alldifferent} constraint is much smaller, averaging 10 variables to be matched. Nonetheless, the instances are not particularly hard for a CP solver because finding feasible trees is quite easy in these graphs, and optimizing the solution is fairly easy given the distribution of the weights of the edges.

We run all our experiments on a Dell server with 252 GB of memory, 2 CPUs with 4 cores each. In all our experiments we use a timeout threshold of 15s for \texttt{CHUFFED}, which runs on a single core (thus easy to deploy on any user’s machine).

4.5.2 Experimental Results

To evaluate our new system \texttt{SERENE}, we show that its semantic labeling model produces more accurate matches for attributes and that the CP formulation leads to better semantic models in terms of precision and recall.

First, Table 4.5 provides evidence that our new semantic labeling model is better suited for the task when there are unmatched attributes in domains. DSL uses heuristic measures to capture the similarity of attributes within the same class, but unmatched attributes are clearly dissimilar from known semantic types, thus similarity measures may be unsound in the presence of unmatched attributes. Our intuition as to why our approach performs better is that we have incorporated features which are derived directly from attribute values and are not based on the notion of similarity.

<table>
<thead>
<tr>
<th>Model</th>
<th>MRR scores</th>
<th>Train time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>museum</td>
<td>soccer</td>
</tr>
<tr>
<td>DSL</td>
<td>0.560</td>
<td>0.618</td>
</tr>
<tr>
<td>\texttt{SERENE}(PATS)</td>
<td>0.866</td>
<td>0.827</td>
</tr>
</tbody>
</table>

Table 4.5: Average performance of semantic labeling models for leave-one-out strategy

We now evaluate \texttt{SERENE} and its modification \texttt{SERENEPATS}, which uses graph pat-
4.5 Experimental Evaluation and Results

Figure 4.7: Performance for the museum and soccer domains. Average size of integration graphs for the museum domain: 88 nodes and 1129 edges. Average run time for the museum domain: SERENE 0.5s, KARMA 0.5s, SERENE-PATS 1s.

Patterns, against the state-of-the-art system KARMA. All three systems have access to the same set of matches produced by our semantic labeling model to ensure a fair comparison. Figure 4.7 shows their performance on the museum and soccer domains. We report average precision and recall for the systems with regard to variable number of semantic models in the training set. For SERENE and SERENE-PATS we report the first solutions found by CHUFFED. All three systems find solutions in less than a second on average.
We use default parameters for KARMA which were shown to yield the best results [209].

As we can see, SERENE produces on average the best semantic models in terms of precision while SERENEPATS generates slightly better models in terms of recall. If we consider results separately for each instance and not on average, there are few instances where KARMA generates more precise models, however, our new approaches produce much better models in terms of recall in all considered instances.

In Figure 4.8 we show how the performance of SERENE changes across sequentially found solutions. CHUFFED successively finds better solutions in terms of the objective function of the model, Equation 4.1. However, that does not translate directly into more precise semantic models. For instance, in the scenario outlined in Example 4.2, CHUFFED could first report the solution corresponding to the date of birth and address of a person (which is wrong in that example), and only later report a solution corresponding to the information of a music concert (which is the correct one in this case). On average, however, we have observed that SERENE yields very good results already with the first solution found, and plateaus after the third solution.

Additionally, we investigate how the performance is influenced by introducing a weighting scheme on the patterns, i.e., costs of patterns are scaled by a factor. In Figure 4.7 we use a scaling factor 1 for pattern costs. When trying scaling factors 5, 10 or 20 for pattern costs from the museum domain we can generate semantic models which are almost 90% in precision and recall. This is a 20% improvement in precision compared to the first solution found with a scaling factor of 1, and a 10% improvement both in precision and recall for the first solution from SERENE. Contrarily to SERENE, SERENEPATS with scaled factors produced better solutions in terms of precision over time. However, the sequential solutions are not always better, and hitting the optimal solution may take hours.

We have also observed that SERENE and SERENEPATS manage to find the ground truth on many instances among its sequential solutions. This property of our approach can be used to configure various parameters for the model, e.g., scaling factors for pattern costs or match score.
4.6 Related Work

The majority of approaches to solve the REL2ONTO mapping problem are based on heuristic rules and alignment of constraints specified within relational schemata and ontologies. A very comprehensive overview and comparison of existing mapping generation tools based on this approach is given by Pinkel et al. [169] and Spanos et al. [201]. As examples, there are BootOX [112], MIRROR [58] and ontop [87]. Crudely speaking, these tools first apply a default direct mapping specified by the W3C. Further, the default ontology is enriched by using explicit and implicit schema constraints. Finally, ontology alignment techniques are applied to match the default ontology to the target ontology. The main advantage of these systems is the ability to run in a fully automatic setting. Our approach is complementary to these and at its current stage is semi-automatic. However, Constraint Programming offers a convenient framework to incorporate integrity constraints specified either within relational or ontological schema as additional constraints to govern the search for the solution. This opens an interesting direction for further research.

A major issue with fully automatic systems is that constraints may be inconsistent or absent completely, e.g., data from Web services or tables on the Web. To overcome
this issue, we can apply ML techniques. Limaye et al. [137] design a system to annotate web tables with entities for cell values, semantic labels for attributes and relationships for binary combinations of attributes. As in our approach, they decompose the process of mapping into two main stages: semantic labeling and finding relationships between matched semantic labels. Limaye et al. [137] enrich their data sources by using YAGO Knowledge Base [205]. Mulwad et al. [150] extend this approach by leveraging information from Wikitology Knowledge Base. Venetis et al. [214] develop a scalable approach to recover the semantics of Web tables by incorporating data from the “isA” Knowledge Base. Ritze and Bizer [186], on the other hand, use DBPedia as their Knowledge Base. Hence, these approaches are limited to domains well represented in those knowledge bases. Also, they are not able to find the relation between attributes in the table if there is no direct connection between the attributes. Our approach, on the other hand, allows a model to be trained on any data and can infer complex semantic paths which might exist between attributes. However, it could be further bootstrapped by leveraging external knowledge bases. This is especially beneficial at the start when the system does not have sufficient training data.

As mentioned above, the approaches for mapping Web tables also perform semantic labeling. They design various similarity metrics for attribute names and values. However, they disregard the attributes which are not matched to the ontology (the so-called unknown set of attributes) which are especially abundant on the Web [186, 167]. It is clear that we cannot directly speak about similarity for unmatched attributes, since they are rather dissimilar from known semantic types. Our approach differs from previous work in that we incorporate an efficient method to handle the unknown class, compared to the state-of-the-art approach DSL [167].

We build upon the work of Taheriyan et al. [209] and use their ideas for the construction of the alignment graph. The difference at this step is that we introduce the unknown and root class nodes and as many unknown data nodes as there are attributes in the modeled data source. These nodes serve to capture the unmatched attributes from the source. Though we have modified KARMA to treat these additional nodes as well, our approach outperforms KARMA since we have additional constraints for these nodes and use an ex-
act algorithm to solve the Steiner Tree Problem. Taheriyan et al. [209] treat the matching and STP parts of REL2ONTO independently and use heuristic algorithms for both. We, on the other hand, use exact algorithms for both parts and address them within a unified CP model.

In their follow up work, Taheriyan et al. [208] suggest using graph patterns to boost the performance of their system. To this end, they had to revise their algorithm by introducing additional heuristics. However, in our case we only had to add pattern variables and to modify the objective function in the MINIZINC model. No changes to the solver were required. This makes our system very convenient and opens directions for validating various additional constraints.

Another relevant problem within the Semantic Web community is ontology matching, or alignment [197]. This problem is similar to schema matching since it finds semantically related entities across different ontologies. Yet, it is different from REL2ONTO since data sources in our case are not bound to any ontology. We think that our approach might be further extended to incorporate ontology matching techniques.

4.7 Concluding Remarks

In this chapter we have presented our approach to the REL2ONTO problem implemented in the system we called SERENE (and its extension that uses frequent graph patterns, SERENEPATS). Our experiments demonstrate that SERENE generates on average more consistent semantic models in terms of precision and recall compared to the state-of-the-art approach KARMA. SERENEPATS produces, on average, better semantic models in terms of recall, and its precision can be enhanced by adding a scaling factor for pattern costs.

Furthermore, our approach is extremely flexible and easy to extend with arbitrary side constraints thanks to the use of CP rather than a specific heuristic algorithm.

Our contributions are:

- A novel modeling framework for the REL2ONTO problem which seamlessly incorporates techniques from the areas of Machine Learning, Knowledge Representation
and Constraint Programming.

- An efficient approach to handle attributes which cannot be matched to the ontology, through a new semantic labeling model and a new STP model with additional nodes in the integration graph.

- A flexible and extensible approach to the problem, which is easy to reuse and adapt to specific situations. This is achieved thanks to CP, which allows introduction of additional constraints with no changes to the solver.
Chapter 5
Spanning Trees in Constraint Programming

In Chapter 3 we saw how to implement a constraint for trees in general and Steiner trees specifically. In this chapter we will see a special case of Steiner trees where all the nodes are terminals, known as Minimum Spanning Trees.

Minimum Spanning Trees (MSTs) are ubiquitous in optimization problems in networks. Even though fast algorithms exist to solve the MST problem, real world applications are usually subject to constraints that do not let us apply such methods directly. In these cases we confront a version of the MST called the “Weighted Spanning Tree” (WST) in which we look for a spanning tree in a graph that satisfies other side constraints and is of minimum cost. In this chapter we implement this constraint using a lower bound and no-good learning to accelerate the search and thus reduce the solving time. We show that having this propagator is tremendously beneficial for solvers and we show the benefits of clause learning.

5.1 The Minimum Spanning Tree Problem

Given a connected weighted graph $G = (V, E)$, the Minimum Spanning Tree (MST) $T$ of $G$ is a connected acyclic sub-graph of $G$ that contains all the nodes in $V$ and is of minimum weight. This is equivalent to the Steiner Tree with terminals $\tau = V$. Finding the MST of a graph can be done using Kruskal’s algorithm (among others) which is $O(|V| \log(|E|))$. Nevertheless, many interesting variants of the MST are NP-hard. In these variants, there are side constraints that make these algorithms unusable.

Some examples where side constraints make the MST problem NP-hard are the capacitated MST [40, 158] (a mixture of maximum flow and the minimum spanning tree), the
degree-constrained MST [152] (where nodes have a limit on their degree), the min-degree MST [6] (where internal nodes must have at least a certain degree), the constrained MST [177] (that adds a knapsack-like constraint to the MST) or the diameter-constrained MST [2] (where every pair of nodes can be separated by at most \( k \) other nodes). These and other variants can be found in the real world. For instance, cable layout for offshore wind farms [123] combines the capacitated MST, the degree-constrained MST and an extra constraint disallowing cables to cross.

We define the \( wst \) constraint as follows:

**Definition 5.1.** The \( wst \) global constraint enforces a graph variable \( G \) encoded with one set of Boolean variables \( v_E \) to be a connected, acyclic subgraph of a given graph \( G = (V, E) \), of weight no greater than \( w \) given a weighting function \( w_f \) over the set of edges \( E \). It is noted:

\[
wst(v_E, G, w_f, w)\]

It is equivalent to the previously defined \( \text{steiner} \text{.tree} \) constraint as follows:

\[
wst(v_E, G, w_f, w) \equiv \text{steiner} \text{.tree}(\{\text{true} \mid n \in V\}, v_E, G, w_f, w)\]

Indeed, this is just a special case of the Steiner Tree constraint. Nonetheless, note how, because we know we need to span all the nodes in \( V \), there is no need to define the Boolean variable set of nodes \( v_V \).

In this chapter we present our implementation of the \( wst \) constraint in CHUFFED. We use clause learning to accelerate the search and show that the explanations on this global constraint are tremendously beneficial in practice. As a comparison with another CP system, we run the same experiments on CHOCO.

We illustrate the use of this constraint on the Diameter-Constrained MST (DCMST) problem, because it has been recently addressed in Constraint Programming by Noronha et al. [154] and has a large number of applications in wireless network routing [14], telecommunications [217], distributed mutual exclusion in computer networks [178] and data compression [35]. For this problem there has been work on both approximation and exact algorithms. In approximations, Gruber and Raidl [103] presented an approach us-
ing Variable Neighbourhood Search, followed by another heuristic approach [104]. For exact solutions dos Santos et al. [74] presented a Mixed Integer Programming formulation of the problem that was later improved by Gruber and Raidl [102]. The latest exact algorithm was presented by Noronha et al. [154] using CP and it outperforms all other approaches. Although CHUFFED is a CP solver just as the one used by Noronha et al. [154] (IBM OPL), both are quite different, and thus comparisons (especially in time) should be considered with care.

In addition to this example, we examine the problem of finding the cheapest spanning tree in an Euclidean graph where no edges can cross as it gives us a good framework to evaluate the power of learning. We call this the NCMST problem.

5.2 State of the Art

Initially, Dooms and Katriel [70] presented a constraint to force a tree to be a minimum spanning tree, called \textit{mst}. It is the case that a graph may have multiple spanning trees of same cost, and thus multiple minimum spanning trees. The \textit{mst} constraint enforced the solution to be one of these minimum spanning trees. Although similar, this is not the constraint we are implementing. Our \textit{wst} constraint is more flexible as it allows side constraints to eliminate the originally minimum spanning tree and we would still find a minimum spanning tree given the side constraints. Of course, we can implement their constraint with ours by simply computing the MST with Kruskal’s algorithm and forcing the weight of the solution to match the weight of the MST.

The first appearance of the \textit{wst} constraint was presented by the same authors, Dooms and Katriel [71], and was originally called “Not-too heavy“ spanning tree. Their work was followed up by Régin [181] with a simpler algorithm for propagation maintaining the same strength for the propagator. The constraint was renamed \textit{wst} then. In that paper by Régin, the propagation was proved to be arc-consistent. He used a data structure named \textit{ccTree} to retrieve paths between nodes, just as our R-UF (Section 3.3.3.2). Later, Régin et al. [183] improved the \textit{ccTree} data-structure to decrease the complexity of the algorithms. The complexity of our R-UF and their \textit{ccTree} is the same, yet ours is simpler
to implement, thus we reused it here.

The contribution of the above papers are the filtering algorithms they provide, but no implementation or experiments are reported. Nonetheless, in Constraint Programming, constant factors in the complexity are crucial and the asymptotic complexity of their algorithms gives only partial information on performance. Also, no previous work explored the use of explanations in this useful constraint.

5.3 Global Constraint for the Weighted Spanning Tree

As mentioned earlier, the graph variable $G$ that will represent the spanning tree only needs to be represented with a set of Boolean variables $v_E = \{b_1, \ldots, b_{|E|}\}$ for the edges. For each edge $x \in E$, we denote $b_x$ the Boolean decision of its membership to $G$ (true if the edge is in the solution, false otherwise). In this chapter we will call $M$ be the set of mandatory edges and $F$ the set of forbidden edges.

**Definition 5.2.** A substitute edge of an edge $e$ in a spanning tree $T = (V, T_E)$ is any edge $e'$ such that $(V, T_E \{e\} \cup \{e'\})$ remains a tree spanning $V$.

**Definition 5.3.** Given a tree $T$ and a non-tree edge $e = (i, j)$, the edge $e'$ of maximum cost in the path from $i$ to $j$ in $T$ is the support of $e$. This was already defined by Régis [181].

We first discuss out lower bounding technique, which is the main contribution of this chapter. We later discuss the propagation rules that we used in our implementation. These are taken from the paper by Régis [181] and are not our contribution, although the explanations for these propagations are.

5.3.1 Lower Bound with Explanations

Given a solution of cost $K$, we want to detect whether a subtree of the search space will not have any solution lower than $K$ before proceeding to explore that entire subtree. Just as we saw for Steiner Trees (Section 3.3.6), this will help the branch-and-bound algorithm to stop the search earlier.
The most accurate lower bound for \( w \) in the \( w_{st} \) constraint is naturally the MST of the graph given the decisions so far. That is, the tree \( T^* = (V, E^*) \) of minimum weight \( W_{T^*} \) such that \( M \subseteq E^* \) (all mandatory edges are in \( T^* \)) and \( E^* \cap F = \emptyset \) (no forbidden edges appear in \( T^* \)). It is easy to see that applying Kruskal’s algorithm where the edges in \( M \) have been pre-added and the edges in \( F \) are not used yields \( T^* \).

After computing \( T^* \), if \( W_{T^*} \geq K \) then no solution of cost lower than \( K \) exists in the current search space, and we can interrupt the search in the current subtree. A trivially correct explanation is \( \bigwedge_{e \in F} \neg b_e \land \bigwedge_{e \in M} b_e \land [w < K] \Rightarrow \text{fail} \). This explanations claims that because of the specific choice of mandatory and forbidden edges at the current stage of the search, no solution better than \( K \) exists. We will show that this explanation contains more information than strictly required and we will build minimal explanations.

Let \( F_c \) be the set of forbidden edges \( e_F \) such that \( T^* \cup \{ e_F \} \) forms a cycle where \( e_F \) is not the most expensive edge. Let \( M_S \) be the set of edges \( e \in M \) having some substitute \( e' \) such that \( ws(e') < ws(e) \). Let \( S_S \) be a mapping \( M_S \mapsto E \) from each edge in \( M_S \) to the substitute of minimum weight for that edge. We then select a subset \( M_H \subseteq M_S \) such that the inequality \( \sum_{e \in M_H} (ws[S_S[e]] - ws[e]) + W_{T^*} \geq K \) holds. This is a set of edges such that, if removed from the best current tree \( T^* \) and substituted by their cheapest alternative, the tree \( T^* \) is still heavier than \( K \). Note that multiple such sets \( M_H \) may exist.

A more reusable explanation is given by Theorem 5.1.

**Theorem 5.1.** A correct explanation for the failure of \( w_{st}(v, G, w_f, w) \) is:

\[
\bigwedge_{e \in F_c} \neg b_e \land \bigwedge_{e \in M \setminus M_H} b_e \land [w < W_{T^*}] \Rightarrow \text{fail}
\]

Note that in the case of failure, \( W_{T^*} \geq K > w \), thus this explanation is more general than using \([w < K]\).

**Proof.** Forbidden edges: Clearly, \( F_c \subseteq F \). Let \( e = (u, v) \in F \setminus F_c \). By definition of \( F_c \), \( e \) is the most expensive edge in the cycle formed by \( T^* \cup \{ e \} \). Because the queue in Kruskal’s algorithm is sorted in increasing order, the path \( p_{u,v} \) between \( u \) and \( v \) in \( T^* \) was already built before considering \( e \). Therefore, whether \( e \) is forbidden or not does not affect the cost of \( p_{u,v} \) and consequently does not affect \( W_{T^*} \) and the explanation holds.
Mandatory edges: By construction, $M_H$ is a set of edges that, when removed and substituted by the best possible edge available, the cost of the tree is still higher than $K$. Therefore, the edges in $M_H$ do not need to be in the explanation for it to hold.

Of course, the same explanations can be used to explaining propagation over the variable $w$:

$$\bigwedge_{e \in F} \neg b_e \land \bigwedge_{e \in M \setminus M_H} b_e \Rightarrow \| w \geq W_{T^*} \|
$$

Note that because several sets $M_H$ may exist, different explanations can be computed. Evaluating which explanation is better than another is highly dependent on the instance of the problem. We ran different tests and could not determine a way of choosing $M_H$ that dominated others in all cases. In our final implementation we start by putting the cheapest edges in $M_H$.

The algorithm to compute the explanation is Algorithm 5.1. To construct explanations, we use the Rerooted-Union-Find (R-UF) data structure described in section 3.3.3.2. Lines 6 to 8 pre-add all the mandatory edges. Lines 9 to 19 follow the classic Kruskal’s algorithm with some modifications. Inside these, line 16 adds to the explanation any forbidden edge that should have been used. Line 14 computes the cheapest substitute for each mandatory edges (if any). Once the tree $T^*$ is computed, we build $M_H$ in line 22, leaving all the other mandatory edges (that have substitutes) in the explanation in line 24. The algorithm returns the set of literals for edge variables involved in the explanation and the cost of the lower bound. Depending on the cost, the explanation computed in $X$ can be used to trigger a conflict or for propagation over variable $w$, simply by combining it with the literal $\| w < W_{T^*} \|$ in the first case or $\| w \geq W_{T^*} \|$ in the second. The complexity of the algorithm is $O(|\mathcal{E}|(|\mathcal{V}| + \log(|\mathcal{E}|)))$.

In the example of Figure 5.1, the upper bound of $w$ is 27 (the cost of the best solution found so far). We can already anticipate that $W_{T^*}$ is 31, so the propagator must raise a conflict. Following is the execution of the algorithm. First, all the mandatory edges are added. We now consider all the edges in the order that Kruskal’s algorithm would have considered them: increasing weight. First, we consider edge $e_1$. The fact that it is mandatory causes no trouble, as there is no other substitute to this edge that would
Algorithm 5.1 Computing the lower bound with explanation

1: procedure WST\text{LB}(K)
2: \hspace{1em} Q \leftarrow \text{SORT}(\mathcal{E})
3: \hspace{1em} UF \leftarrow \text{R-UF}(|V|)
4: \hspace{1em} c \leftarrow 0, \text{cost} \leftarrow 0
5: \hspace{1em} X \leftarrow \emptyset, \text{sub} \leftarrow [\text{nil} \mid e \in \mathcal{E}]
6: \hspace{1em} \textbf{for all} e = (u, v) \in M \hspace{1em} \triangleright \text{Pre-add mandatory edges}
7: \hspace{2em} UF.\text{UNITE}(u, v)
8: \hspace{2em} c \leftarrow c + 1; \text{cost} \leftarrow \text{cost} + w_f(e)
9: \hspace{1em} \textbf{for all} e = (u, v) \in Q \hspace{1em} \triangleright \text{(in order)}
10: \hspace{2em} \text{if} \neg \text{UF.\text{CONNECTED}}(u, v) \land e \in F \hspace{1em} \triangleright \text{Should add } e \text{ but it is forbidden}
11: \hspace{3em} X \leftarrow X \cup \{-b_e\}
12: \hspace{2em} \text{else if} UF.\text{CONNECTED}(u, v) \hspace{1em} \triangleright \text{ } e \text{ would be cheaper}
13: \hspace{3em} \textbf{for all} ep \in \text{PATH}(u, v)
14: \hspace{4em} \text{sub}[ep] = \min_{\text{sub}[ep], e}
15: \hspace{4em} \text{if} w_f(ep) \geq w_f(e) \land e \in F \hspace{1em} \triangleright e \in M_H
16: \hspace{5em} X \leftarrow X \cup \{-b_e\}
17: \hspace{2em} \text{else if} c < |V| - 1 \land \neg \text{UF.\text{CONNECTED}}(u, v) \hspace{1em} \triangleright e \not\in M_H
18: \hspace{3em} UF.\text{UNITE}(u, v)
19: \hspace{3em} c \leftarrow c + 1; \text{cost} \leftarrow \text{cost} + w_f(e)
20: \hspace{1em} \textbf{for all} e \in M \land \text{sub}[e] \neq \text{nil} \hspace{1em} \triangleright e \in M_H
21: \hspace{2em} \text{if} \text{cost} - w_f(e) \geq K \hspace{1em} \triangleright e \in M_H
22: \hspace{3em} \text{cost} \leftarrow \text{cost} - w_f(e) + w_f(\text{sub}[e])
23: \hspace{2em} \text{else if} w_f(\text{sub}[e]) \neq w_f(e) \hspace{1em} \triangleright e \not\in M_H
24: \hspace{3em} X \leftarrow X \cup \{b_e\}
25: \hspace{1em} \text{return} (X, \text{cost})

connect h. Edges \(e_2\) and \(e_3\) are added normally following Kruskal’s algorithm. When we consider \(e_4\), we must skip it because it is forbidden, which means that we will use a more expensive edge to reach \(c\) (here \(e_8\)). Thus \(\neg e_4\) is added to the explanation. Edge \(e_5\) is added normally (as it has no substitute). When considering \(e_6\), \(e\) and \(g\) are already connected by a path containing the mandatory edges \(e_7, e_9\) and \(e_{10}\) and the unfixed edge \(e_3\). Therefore \(e_6\) is a substitute to all of them. This is stored in the array \text{sub}. The main loop finishes after adding \(e_8\) (the other mandatories were already added). We later compute that: \(w_f(e_7) + w_f(e_6) = 30 > K\), then \(30 - w_f(e_9) + w_f(e_6) = 28 > K\) and lastly \(28 - w_f(e_{10}) + w_f(e_6) = 21 < K\). Therefore, the explanation will be \(\neg b_{e_4} \land b_{e_{10}} \land \lceil w < 31 \rceil \Rightarrow \text{fail}\).
5.3.2 Propagation Rule with Explanations

We use the propagation rule exposed in Proposition 3 from the paper by Régin [181], that is: given the best possible tree $T^*$ and an upper bound for the solution $K$ such that $W_{T^*} < K$, for any non-tree edge $e^*$ of support $e'$, $e^*$ can be part of the solution if and only if $W_{T^*} - w_f(e') + w_f(e^*) < K$. That is, $e^*$ is a valid substitute of $e'$ that does not increase the weight of the tree beyond the upper bound. If this is not the case, we must remove $e^*$ from the possible edges since using it would increase the weight of $T^*$ above $K$. It is easy to see that the previous explanation applies as well in this case. Let $M'_H = ((M \setminus M_H) \setminus \{e'\}) \cup \{e^*\}$.

We execute this rule after the previously described algorithm in the case where no failure is detected.
5.4 Experimental Evaluation and Results

5.4.1 The Diameter Constrained Minimum Spanning Tree

The DCMST is formally defined as follows: given a graph \(G = (\mathcal{V}, \mathcal{E})\) find a sub-graph \(G' = (\mathcal{V}', \mathcal{E}')\) of \(G\) such that \(G'\) is a tree, \(\mathcal{V}' = \mathcal{V}\) and the longest distance between any two nodes in \(G\) is at most \(D\), called the diameter of \(G\). Here, the distance between two nodes \(u\) and \(v\) is the number of nodes in the path from \(u\) to \(v\).

5.4.1.1 Modeling DCMST

This problem is separated in two cases depending on whether \(D\) is even or odd. If \(D\) is even, then there exists a node \(r\) that is the root of \(T\) and the height of all the other nodes has to be at most \(\lfloor D/2 \rfloor\). If \(D\) is odd, there exists an edge \(e = (a, b)\) that acts as the root of the tree (\(e\) is therefore in the tree), meaning that the height of \(a\) and \(b\) is zero and all the other nodes must have at most height \(\lfloor D/2 \rfloor\). Notice that \(r\), \(a\) and \(b\) are not given in the input: these are variables.

We used the same model as Noronha et al. [154] with the only addition of our propagator. For the DCMST-specific constraints, we use an array of heights of nodes \(h\), and an array of parenthood of nodes \(p\). Two variables \(a\) and \(b\) are the end-nodes of the edge that acts as root in the odd case, or are both the root in the even case (in that case, \(a = b\)). The model is:

\[
\text{Minimize } w \text{ such that} \quad w_{st}(v_\mathcal{E}, G_\mathcal{G}, w_f, w) \quad (5.1) \\
D \mod 2 = 0 \iff a = b \quad (5.2) \\
(h[a] = 0 \land p[a] = b) \land (h[b] = 0 \land p[b] = a) \quad (5.3) \\
\forall n \in N \backslash \{a, b\}, \ h[n] = h[p[n]] + 1 \quad (5.4) \\
\forall n \in N \backslash \{a, b\}, \ p[n] \in \text{adj}[n] \quad (5.5) \\
\forall e = (u, v) \in \mathcal{E}, \ b_e \iff p[u] = v \lor p[v] = u \quad (5.6)
\]
Equation 5.1 is the objective function, minimizing the weight of the tree. Equation 5.2 is the call to our global constraint. Equation 5.3 states that in the even case \(a\) and \(b\) are the same node (the root \(r\)). Equation 5.4 forces \(a\) and \(b\) to be at height 0 and be each others parents. Equation 5.5 makes every node (other than the root(s)) be one level below its parent. Equation 5.6 forces each node to chose a parent that is adjacent to it. Finally, Equation 5.7 links the edge variables of the graph with the parenthood relations.

Although our main intention is to compare the improvement that the \(wst\) propagator and explanations bring to the solver, we also compare our work to the one of Noronha et al. [154] (we name their results NRS) as their results are the state of the art in DCMST as far as we can tell. They used a Pentium 4, 2.8GHz and 2GB of RAM and IBM CP Optimizer as their solver.

For better comparison, we implemented the exact same search strategy they describe in their paper (Section 3, Figure 2 in their paper). First, for each node \(n\) we compute the sum \(s_n\) of the shortest paths from \(n\) to any other node. Then, we associate to each pair of nodes \((a, b)\) the minimum of \(s_a\) and \(s_b\), noted \(s_{(a,b)} = \min(s_a, s_b)\). The search is as follows. Start by taking each pair of nodes \((a, b)\) in increasing order of \(s_{(a,b)}\). Then for each possible value of the height (from 1 to \(\lfloor D/2 \rfloor\)), remove that value from the domain of all the nodes (when possible) taking the nodes in decreasing order of the shortest path to either \(a\) or \(b\). Here “shortest path” is in weight of the edges.

They use a dominance rule in the search, which we converted into a dominance-breaking constraint [45] in our model, for ease of implementation: \(\forall \{e_1 = (u,v), e_2 = (u,y)\} \in E^2, w_f(e_1) < w_f(e_2) \land h[v] \leq h[y] \Rightarrow p[u] \neq y\). This states that if it is cheaper to connect \(u\) to \(v\) than to \(y\) and the height of \(v\) is lower than the height of \(y\), we can connect \(u\) to \(v\) with a lower cost. This is because if using \(e_2\) does not violate the diameter constraint, neither does \(e_1\).

### 5.4.1.2 Experimental results

We run our experiments on a Linux 3.16 Intel® Core™ i7-4770 CPU @ 3.40GHz, 15.6GB of RAM machine. We used 5 minutes as the time limit. The results from NRS are extracted from their paper. Benchmarks can be found online [60]. We give different versions in
5.4 Experimental Evaluation and Results

105

C HUFFED: N O P ROP uses learning but does not use our propagator, N O E XPL uses our
propagator without learning, E XPL uses the propagator with the explanations as computed by Algorithm 5.1, and N AIVE E XPL uses our propagator with naive explanations
(i.e. all fixed elements in the graph are in the explanation). All use the same strategy.
Instance
|V | |E | D
15 105
4
15 105
5
15 105
6
15 105
7
15 105
9
15 105 10
20 190
4
20 190
5
20 190
6
20 190
7
20 190
9
20 190 10
25 300
4
25 300
5
25 300
6
25 300
7
25 300
9
25 300 10
20 50
4
20 50
5
20 50
6
20 50
7
20 50
9
20 50
10
40 100
4
40 100
5
40 100
6
40 100
7
40 100
9
Total

N O P ROP
Nodes Time
6825
0.45
35322
2.28
133259
10.31
258317
22.91
493166
39.80
550536
40.93
192965
20.39
1869837 186.76
2585912
300
2661381
300
2433234
300
2628419
300
1898689
300
2415919
300
2262702
300
2045173
300
1929801
300
1961836
300
14548
0.48
55748
2.58
52217
2.34
66676
3.45
274583
16.59
310688
18.93
3426079
300
3261615
300
4836734
300
4709441
300
3646022
300
47017644 4868.2

N O E XPL
Nodes
Time
784
0.23
1921
0.39
5997
0.63
5873
0.54
6049
0.47
24259
1.95
2651
1.86
9387
4.85
49673
14.97
16690
4.67
157236
34.43
315618
52.61
20202
58.53
86662
93.57
402861
300
449104
210.13
462886
300
620555
261.71
1219
0.05
307392
10.87
68384
0.75
25043
0.68
14016
0.33
410
0.01
45199
6.27
9596291
300
8161773
300
5979528
300
4468371
300
31306034 2530.50

N AIVE E XPL
Nodes Time
448
0.2
1003
0.39
2235
0.48
2312
0.41
1968
0.21
2872
0.39
1261
1.70
4452
4.56
9018
4.55
4252
2.04
6336
1.71
9050
3.96
6166
17.10
32787
80.62
16150
17.66
76272
87.72
21195
18.66
21453
11.86
558
0.05
2258
0.26
1574
0.10
1381
0.12
1117
0.06
564
0.03
13766
4.41
36496
14.22
10708
2.59
38153
11.03
88837
25.12
414642 312.21

E XPL
Nodes Time
447
0.19
1001
0.38
2101
0.45
2221
0.41
1731
0.19
2831
0.29
1266
1.69
4432
4.48
8462
4.08
4288
1.97
5972
1.60
8645
3.56
6217
17.1
26547
68.88
15147
15.99
61098
63.68
19724
17.43
21565
11.50
558
0.05
2227
0.24
1475
0.08
1238
0.09
1261
0.06
564
0.02
13901
4.30
26970
9.69
5037
0.84
18504
4.49
36572
7.08
302002 240.81

C HOCO
Nodes
Time
6256
1.92
301269
44.36
160445
37.06
2182510
300
2623006
300
898948
156.63
200651
82.51
2064050
300
862115
300
1857850
300
1738525
300
1067170
300
592738
300
1553235
300
847691
300
1448142
300
1270399
300
586552
300
4489
0.61
426762
40.91
41892
5.28
1389117
133
3820792
300
329333
42.87
1180714
300
3196955
300
1851687
300
2989047
300
2873734
300
38365804 6245.15

NRS (IBM OPL)
Nodes
Time
1044
0.08
2850
0.22
6960
0.28
8240
0.38
11743
0.47
11830
0.41
3143
0.20
18283
1.06
35383
2.03
19142
0.97
119906
5.01
151969
6.08
28842
1.48
37608
2.83
534222
39.14
812957
56.06
2655810 114.14
1126130
55.47
389
0.05
3611
0.17
2678
0.13
1975
0.14
13040
0.45
17937
0.64
130480
5.44
161961
7.31
91022
4.72
778669
34.38
769161
40.16
7556985 379.84

Table 5.1: Comparison in time (seconds) and nodes for the DCMST models
As we can see in Table 5.1, the use of the propagator is absolutely beneficial. The total time is improved by 48.02% when using the propagator without explanations against
no propagator. Furthermore, our version with explanation (E XPL) is 90.5% faster than
the version without explanations (N O E XPL) and 95.1% faster than the version with no
propagator at all. Also, our total time is 36.6% shorter than NRS. Most of the tests with
N O P ROP and C HOCO got to the optimal solution, but timed-out when proving optimality. This also illustrates the need for this propagator.
In CP, the number of nodes represents the size of the search space explored before
proving optimality. Here we see an obvious dominance of E XPL as it almost always has
less nodes than other versions. It also has an improvement on the total number of nodes


for all benchmarks of 99.4% over NOPROP and 99.0% over NOEXPL. Additionally, it has an improvement of 96% over NRS.

The comparison between EXPL and NAIVEEXPL shows that computing our explanations is worthwhile. The NAIVEEXPL uses the same algorithms as described throughout this paper, only the explanations contain all the fixed $b_u$ and $b_v$. This makes the explanations more strict and thus less reusable and this also makes the explanations much longer: the average length in the explanations for NAIVEEXPL is 128.88 literals, whereas the length of our explanations is 73.18 literals in average. We see the consequences of this in the Table 5.1: naive explanations most often slow down the solving step. The version EXPL is 22.9% faster and has 27.2% less nodes.

We observe that our propagator dominates in particular when the diameter is big. This is because in that case, the lower bound is more accurate as it violates fewer diameter constraints. When the diameter is small, Algorithm 5.1 is not aware of it and just computes an MST thus rapidly violating the diameter constraints.

### 5.4.2 Non-crossing Edges MST

A topological graph is a graph $G$ that is accompanied by a layout: for every node there is a point in the plane, and the layout describes how to draw the edges. In our case, to simplify and without loss of generality, edges will be straight segments between two nodes.

The non-crossing edges MST (NCMST) is defined as follows: given a topological graph, find a minimum spanning tree such that no edges cross each other.

Note how in the simple case where the weight of the edges correspond to their lengths, any algorithm would work (assuming that the graph contains all the edges from Delaunay’s triangulation [132] of the nodes, which is the case for complete graphs). In the more general case, this is not true. Consider for instance the example in Figure 5.2.

Finding a non-crossing minimum spanning tree in a topological graph was proven to be NP-hard by Kratochvíl et al. [126] (see their corollary 5.3 derived from their Theorem 5.1). As the authors mention, it is a common problem in VLSI, but our main motivation came from the Offshore-Wind-Farm Cable Layout problem described by Klein et al. [123]
5.4 Experimental Evaluation and Results

Figure 5.2: Example of a graph where a greedy algorithm would fail to find the tree of minimum weight. Edges marked with ▼ are added to the tree, edges marked ◀ are skipped because they cross other edges. The cost is 33, but a cheaper tree is \{\text{e}_2, \text{e}_3, \text{e}_4, \text{e}_6, \text{e}_8\} (of cost 29).

and Bauer and Lysgaard [17].

Modeling this problem is nonetheless very easy since crossing edges can be precomputed. Let \(C\) be the set of pairs of edges \((\text{e}_i, \text{e}_j)\) that cross each other. The model is:

\[
\text{minimize}(w) \quad \text{such that:}
\]

\[
\text{wst}(v, G, w, w) \\
\bigwedge_{(\text{e}_i, \text{e}_j) \in C} (\neg b_{\text{e}_i} \lor \neg b_{\text{e}_j})
\]

5.4.2.1 Experimental results

We modeled this problem using the well known Solomon benchmarks for the Vehicle Routing Problem with Time Windows. We used the sets of benchmarks C101, C201, R101 and RC101 (others had the same coordinates for the nodes). For each of them we created complete graphs with the first 25, 30, 35 and 40 nodes. For each one of these 16 graphs, we generated 6 other benchmarks as follows: we chose a random cost \(r_e\) for each edge \(e\) and we created benchmarks with the cost of each edge being \(w_f(\text{e}) = \text{coef} \cdot r_e + (1 - \text{coef}) \cdot r_e\) where \(\text{coef}\) takes the values \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}. Therefore the first benchmark is always simple to solve, and they become more and more random when \(\text{coef}\) grows. The total number of benchmarks is 96. We used the same machine as in Section 5.4.1.2, with
a time-out of 5 hours this time. Our search strategy tries to add the cheapest edges first, like Kruskal’s algorithm.

We used this benchmarks to compare the benefits of learning. We present these comparisons using plots in Figures 5.3, 5.4 and 5.5. We define the gain of a version \( A \) over a version \( B \) by the ratio \( (B - A)/B \). Figure 5.3 presents the gain in number of nodes of EXPL over NOEXPL for all the tests. Figure 5.4 presents the gain in time. These plots represent the percentage of gain that we get out of learning. Note that the two timed-out instances (R101 and RC101 with 40 nodes and \( coef \geq 0.8 \)) are not plotted.

We can observe in the plots that learning starts to be beneficial once the problem gets hard enough. Benchmarks where \( coef \) has a small value are very similar to the pure MST, and therefore the search space is very small as we rapidly hit the best solution. On the contrary, when \( coef \) grows, the problems get harder and more time and nodes are required before reaching the optimal solution. In this case, learning is tremendously beneficial as we can see. In all cases we spend less than half the time and around 75% less nodes thanks to the explanations.

Since no results exist in the literature for the NCMST problem, we compared our work to CHOCO as it has a dedicated propagator for weighted trees that results in an equivalent propagator to ours (although their implementation is simpler) without explanations. The results are shown in Figure 5.5 as the gain of EXPL over CHOCO. We used 24 instances of 15 nodes (105 edges) so that no benchmark would reach the time limit.

We can see that for easy instances, both propagators behave similarly in terms of nodes. For more difficult instances, ours outperforms CHOCO by more than 90% (reaching almost 100%). Our propagator is never slower, and in harder cases much faster (always above 80%).
Figure 5.3: Gain in nodes provided by explanations. The closer the gain is to 1, the more beneficial explanations were.
Figure 5.4: Gain in time provided by explanations. The closer the gain is to 1, the more beneficial explanations were.

(a) 25 node, 300 edges.
(b) 30 nodes, 435 edges
(c) 35 nodes, 595 edges
(d) 40 nodes, 780 edges

Figure 5.5: Comparison between our propagator against CHOCO

(a) Gain in nodes.
(b) Gain in time
5.5 Concluding Remarks

In Chapter 3 we created a propagator for trees and lower bounds for the Steiner Tree problem. In this chapter we have specialized that work in the case where all the nodes need to be spanned. The topology of the tree can still be done with the propagations presented in Chapter 3, but the lower bounding can be substantially improved for this particular case, specially with the use of explanations.

We have shown with our experiments that this specialization was worth it. We have also demonstrated that the use of explanations for this propagator produces little overhead and can be greatly beneficial to accelerate the solving time.

Our contributions are:

- An efficient algorithm to produce explanations for the already known lower bound for the $w_{st}$ constraint.

- Explanations for the propagation rules developed by Régis [181].

- A suite of experiments demonstrating the value of this lower bounding technique and the explanations.
Chapter 6
Graph Paths in Constraint Programming

Path finding is an ubiquitous problem in optimization and graphs in general, for which fast algorithms exist. Yet, just like for trees in the previous chapters, in many cases side constraints make these well known algorithms inapplicable. In this chapter, we study constraints that enforce a graph to be a path of a certain length (or weight), on a weighted directed graph. These findings can be applied to Constraint Programming models involving paths with arbitrary side constraints. To implement this, we use the conjunction of two directed tree constraints to model the path, and a bounded path propagator to take into account the weights of the arcs. We show how to implement these constraints with explanations so that we can make use of powerful Constraint Programming solving techniques using learning. We give experiments to show how the resulting propagators substantially accelerate the solving of complex path problems on directed graphs.

6.1 Path Problems in Constraint Programming

Path-finding is an important task in (directed) networks. It arises in tasks such as graph layout [79], metabolic networks [215] or collaborative path-finding in video-games [199], among other examples. In many cases, though, side constraints make these problems highly combinatorial and no efficient algorithms exist.

In this chapter, we focus on path-finding with distances. In order to do so, we go through preliminary steps to build two propagators from which we build a path propagator that works on the topology of the graph. These propagators are actually directed versions of the connected and tree propagators seen in Chapter 3. Once we have this path propagator, we construct a bounded path propagator that takes into account the
weights of the arcs to propagate distances.

**Definition 6.1.** The *dreachable* global constraint enforces a graph variable $G$, encoded with two sets of Boolean variables $v_V$ and $v_E$, subgraph of a given directed graph $G$, to contain a directed path from a given node $r$ (the root) to all its other nodes. It is denoted:

$$dreachable(v_V, v_E, r, G)$$

This is very similar to the *connected* constraint from Definition 3.2 from Chapter 3. In that case, all the nodes needed to have a path to each other. In this case, since $G$ is directed, we choose a node $r$, which we call *root*, from which all paths to other nodes need to start. This does not mean that there are only directed paths from $r$ to other nodes: there could be directed paths between other nodes. Nonetheless, this constraint forces that all nodes are *reachable* from $r$ following a directed path.

**Definition 6.2.** The *dtree* global constraint enforces a graph variable $G$ encoded with two sets of Boolean variables $v_V$ and $v_E$, subgraph of a given directed graph $G$, to be a directed tree with root $r$ (see Definition 2.31 on page 22). It is denoted:

$$dtree(v_V, v_E, r, G)$$

The *dtree* constraint is clearly a special case of *dreachable*, where all the nodes are reachable from $r$ following directed paths, but in which there are no (directed or undirected) cycles.

**Definition 6.3.** The *path* global constraint enforces a graph variable $G$ encoded with two sets of Boolean variables $v_V$ and $v_E$, subgraph of a given directed graph $G$, to be a simple path from $s$ to $d$ (see Definition 2.22 on page 21). It is denoted:

$$path(v_V, v_E, s, d, G)$$

**Definition 6.4.** The *bounded path* global constraint enforces a graph variable $G$ encoded with two sets of Boolean variables $v_V$ and $v_E$, subgraph of a given directed graph $G$, weighted by a given
function \( w_f \), to be a simple path from \( s \) to \( d \) of weight at most \( w \). It is denoted:

\[
\text{bounded_path}(v_V, v_E, s, d, G, w_f, w)
\]

Note that, as we said in Chapter 2, the weight of the path (often called the length in the literature) is the sum of the weights of the edges in the path. Also, for our approach \( w \) can be a variable.

The goal of this chapter is to show how to build the \( \text{bounded_path} \) propagator with explanations. The other propagators listed above are intermediate stages to achieve this goal. We present two novel explanations for already existing propagation rules. Furthermore, we introduce a new stronger propagation technique with explanations as well. The explanations for this propagator and its new version are the main contributions of this chapter.

As a side effect of this work, we also develop a propagator for a constraint that forces a graph to be a Directed Acyclic Graph (DAG, see Definition 2.32 in page 22). We name it \( \text{dag} \). This is not necessary to build paths as the other constraints, but it derives very easily from the \( \text{dreachable} \) constraint. For that reason we present it here.

**Definition 6.5.** The \( \text{dag} \) global constraint enforces a graph variable \( G \) encoded with two sets of Boolean variables \( v_V \) and \( v_E \), subgraph of a given directed graph \( G \), to be a DAG rooted at a node \( r \). It is denoted:

\[
\text{dag}(v_V, v_E, r, G)
\]

Note that, by definition, DAGs do not have a root node. The constraint we show in Definition 6.5 has a root argument to explicitly show that this constraint is a special case of \( \text{dreachable} \). We can provide the user with a \( \text{dag}(v_V, v_E, G) \) without a root. In that case, the modeling language would internally add a root node and edges from that node to all the other nodes in the graph. Both definitions are equivalent: \( \text{dag}(v_V, v_E, (V, E)) \equiv \text{dag}(v_V, v_E, r, (V \cup \{r\} \cup \{(r, n) | n \in V\})) \).
6.2 State of the Art

Paths in Constraint Programming

Finding a simple path (no node repetitions) is a classic graph problem with wide applicability. The usefulness of the constraint arises when there are interesting side constraints. Our path propagator is based on the Ph.D. thesis by Fages [84], which showed how to model the path constraint as a conjunction of dtree constraints. We will show this later in this chapter in detail.

There exist other approaches to finding paths by using circuit-style propagators [89]. We compare these two approaches where both use explanations.

Directed Reachability, Directed Trees and DAGs in Constraint Programming

The three first constraints defined in the introduction were first introduced as part of CP(Graph) [72] in 2005, using a decomposition approach, rather than propagators.

Later, Quesada et al. [173] implemented the first dreachable propagator and used it as a path constraint in their paper, by adding degree constraints. Quesada et al. [173] made use of simple propagation rules based on depth-first traversals of the graph and on the use of dominator nodes (i.e. nodes that appear in all paths). Nonetheless the asymptotic complexity of their algorithms is substantially greater than ours or those of Fages and Lorca [85] since they use a brute-force algorithm to compute dominator nodes, instead of the algorithm by Lengauer and Tarjan [133]. Note that the algorithm by Alstrup et al. [8] can also be used to find dominators and, although its worst-case complexity is worse than Lengauer-Tarjan’s algorithm, in practice it performs better. Nonetheless, we used the algorithm by Lengauer and Tarjan [133] for our implementation.

Since reachability (or directed-connectedness) also appears in trees, the constraints for trees and forests developed by Beldiceanu et al. [22], Beldiceanu et al. [23] and Fages and Lorca [85] face similar challenges as the ones by Quesada et al. [173]. Although focused on forests, their work used better algorithms improving the work of Quesada et al. [173] to make each individual tree connected. For our implementation, we base our
propagation rules on the ones described in the aforementioned papers. The explanations for the \textit{dtree} constraint are an adaptation for directed graphs of the explanations for \textit{tree}, shown in Section 3.3.4 of Chapter 3. We describe these directed versions in Section 6.3 of the current chapter.

The \textit{dag} constraint is novel as far as we can tell. It is closely related to the directed tree constraint \textit{dtree}. It is applicable to the well studied Maximum Directed Acyclic Subgraph problem, for which only approximations exist \cite{27, 83, 107}. We give the first CP approach to such problems of which we are aware.

\textbf{Weighted Paths}

Path finding with distances is one of the most well-studied graph problems, for which very well known fast algorithms exist. Many specific algorithms that handle some form of side constraint are also known. For instance, paths with resource constraints have been very well studied for electrical cars \cite{203} and for bike routes \cite{204}. Another application is the Generalized Shortest Path queries \cite{184, 185} where a person needs to do a series of tasks during their journey and choose among different places to do them (e.g., withdrawing money can be done at any ATM). The \textit{bounded.path} constraint allows us to specify “shorter path” problems with arbitrary side constraints. It was introduced by Sellmann \cite{194} and Sellmann et al. \cite{195} with some propagation rules. Our work adds explanations to their propagation rules and improves on their work.

\section{From a Directed Reachability Propagator to a Path Constraint}

Because this chapter deals with directed graphs, we will refer to the directed edges as \textit{arcs} of the graph. Recall from Chapter 2 that an arc goes from its tail \( t \) to its head \( h \), and is drawn ‘\( t \rightarrow h \)’.

As in other chapters, we use two sets \( v \) and \( e \) of Boolean variables to represent the nodes and arcs of a graph variable \( G \). For each individual Boolean variable, we will use the notation \( b_x \) where \( x \) refers to either an arc or a node. We will also use the notion of the \textit{inverted graph} \( G^{-1} \) of \( G \), defined in Definition 2.18, on page 20.
6.3.1 A Directed Reachability Propagator with Explanations

The \texttt{dreachable} constraint guarantees that all nodes in the subgraph $G$ are reachable from a given node $r$. Quesada et al. [173] first proposed this propagator, although our algorithm is substantially improved by making use of the Lengauer-Tarjan algorithm to find dominators in a directed graph [133]. Fages and Lorca [85] also used this algorithm in their work.

For this propagator, the algorithms are analogous to their undirected version presented in Section 3.3.2 of Chapter 3, although adapted to the directed case.

6.3.1.1 Detecting and Explaining Failure

In order to detect that the current assignment of arcs and nodes in $G$ is invalid, we need to check that all the mandatory nodes are reachable from the given node $r$.

Algorithm 3.2, presented in page 44, can achieve exactly this. To do so, the parameter $n$ will always be the node $r$. Recall the algorithm uses two DFS passes. The first one, to detect the failure, will take the directed edges in the normal orientation in $G$. The second one, called \texttt{PINKDFS}, shown in Algorithm 3.1 (page 44) will only need one change: it will take the arcs in the opposite orientation, that is in $G^{-1}$. No further change is required.

6.3.1.2 Propagating and Explaining Dominators

During the search, we also make inferences that will accelerate the search. This is what we did in Section 3.3.2 of Chapter 3 for articulation nodes. Nonetheless, articulations are a notion of undirected graphs, and the algorithm we used to find and propagate them is not valid anymore. For directed graphs we use dominator nodes, defined in Definition 2.26 (page 21). Recall these are nodes contained in all paths between two other nodes. In our case, we will be interested in dominators between $r$ and any mandatory node in $G$. For \texttt{dreachable}, dominators must be mandatory, otherwise some mandatory node $t$ would not be reachable from $r$.

Finding dominators in a graph can be done using the Lengauer-Tarjan algorithm [133] in $\mathcal{O}(|E|\alpha(|E|,|V|))$ where $\alpha$ is the inverse Ackerman function. Their algorithm builds
an array representation of a so-called *dominator tree* where the parent of a node is its immediate dominator (see Definition 2.27, page 21). This dominator tree can easily be represented as an array $\text{idoms}$ where $\text{idoms}[x]$ is the immediate dominator of node $x$. For our purposes, we apply this algorithm to the available nodes and arcs (that is, all $\mathcal{G}$ except for forbidden arcs and nodes).

We assume that the reachability has been ensured and thus all mandatory nodes are reachable from $r$. To enforce dominators to be in $G$, we build a queue containing all the mandatory nodes and iterate through the queue until it is empty. When a node is popped from the queue, we make its immediate dominator mandatory (if they are not already) and enqueue it. This way, all the nodes in the path between $r$ and some mandatory node $t$ in the dominator tree become mandatory.

Explaining this inference is almost identical to explaining the propagation of articulations for undirected graphs, seen in Section 3.3.2 of Chapter 3. First, we compute a partition of nodes $R$ from which $t$ can be visited without using forbidden arcs or going through the immediate dominator $d$ that we are trying to propagate. This is achieved using a DFS on the inverted graph $\mathcal{G}^{-1}$, from $t$. We then need to obtain the forbidden arcs that, if made available, would allow $r$ to reach $t$. For that, we perform another DFS in $\mathcal{G}^{-1}$ starting in $t$, this time allowing the use of forbidden arcs. Whenever a forbidden reversed arc $e^{-1} = (t, h)$ has its tail in the set $R$ but its head is a node outside of $R$ we know that $e$ would have allowed us to short-circuit $d$ if it was not forbidden. Let $F_d$ be the set of such arcs. The explanation for making a dominator mandatory is $b_r \land b_t \land \bigwedge_{e \in F_d} \neg b_e \Rightarrow b_d$, which is analogous the one in Equation 3.2 in page 46.

The algorithm can be seen in Algorithm 6.1. Both procedures REACHABLEDFS and BROKENEDGESDFS where already described in Algorithm 3.3 in page 47. They only need to be changed to take the arcs in the opposite direction. The example in Figure 3.5 (page 46) is valid for this directed version as well.

### 6.3.1.3 Propagating and Explaining Bridges

Additionally, if any mandatory node $n$ (other than $r$) has only one incoming arc that is not forbidden, that arc can be set as mandatory (if it is not already). This is because lacking
Algorithm 6.1 Enforcing dominators to be in the graph

1: procedure DOMINATORS($G, r, \nu_G, \nu_F$) 
2: $V \leftarrow \emptyset$ \hspace{1cm} \triangleright V is the set of nodes reachable from $r$
3: idoms $\leftarrow$ LENGAUERTARJAN($G, r, \nu_G, \nu_F, V$) \hspace{1cm} \triangleright V is set by this function
4: $Q \leftarrow$ new Queue($\{n \mid n \in V, b_n = \text{true}\}$) \hspace{1cm} \triangleright Q is a queue of mandatory nodes
5: while $\neg Q.$EMPTY do
6: $n \leftarrow Q.$TOP 
7: $Q.$POP() 
8: $R \leftarrow [\text{false} \mid n \in V]$ 
9: $F_d \leftarrow \emptyset$ 
10: REACHABLEDFS($t, R, d$) \hspace{1cm} \triangleright Adapted to take arcs inverted.
11: BROKENEDGESDFS($t, [\text{false} \mid n \in V], d, R, V, F_d$) \hspace{1cm} \triangleright Adapted to take arcs inverted.
12: PROPAGATE($b_r \wedge b_n \wedge \forall e \in F_d \neg b_e \Rightarrow b_{\text{idoms}[n]}$) 
13: $Q.$PUSH(idoms[$n$]) 

that arc would make that node unreachable. That arc is called a bridge. The explanation for including this arc $e$ in $G$ is trivial: $b_n \wedge \bigwedge_{b \in \text{incoming}[e]} \neg b_e \Rightarrow b_e$, where $\text{incoming}[n]$ is the set of arcs incident to $n$. Similarly, if $n$ no longer has any incident arcs available, we have to set it to $\text{false}$, or fail if it is mandatory, explained by $\bigwedge_{b \in \text{incoming}[n]} \neg b_e \Rightarrow \neg b_n$.

6.3.2 A Directed Tree Propagator with Explanations

Trees are connected graphs, therefore $\text{dtree}$ is special case of $\text{dreachable}$. Additionally, trees cannot contain any cycle (whether it is directed or not). Maintaining this condition is the task of $\text{dtree}$.

This acyclically propagation can be achieved directly with the propagations from the acyclic constraint seen in Section 3.3.3. The algorithms do not even need to be adapted, they simply need to ignore the orientation of the arcs and consider them like undirected edges. The explanations generated by those algorithms are also valid for the directed case.

6.3.3 DAG Propagator with Explanations

Very similarly to trees, DAGs are connected graphs, therefore $\text{dag}$ is special case of $\text{dreachable}$. Additionally, DAGs cannot contain directed cycles.

As seen above, the way to avoid cycles in trees is through the propagations of acyclic.
The core element in that propagator was the Rerooted-Union-Find data-structure (R-UF, Section 3.3.3.2). This is what informs us of whether a cycle happening and provides us with the list of edges in that cycle (those edges are the explanation for failure or propagation in acyclic).

To build dag we only need to substitute the R-UF with another data-structure. We call it Incremental Transitive Closure (ITC).

**Incremental Transitive Closure**

The ITC will have the exact same interface as the R-UF, that way a simple substitution of data-structures converts a dtree propagator into a dag propagator. In order to efficiently implement this, we use sparse sets [36, 92] which guarantee constant time access and addition to the set. They are also easy to use in CP solvers as restoring their state upon backtracking is trivial by trailing their size [92].

The ITC contains, for each node $u$, the set of nodes that are reachable from $u$ (the “successors” $succ[u]$) and a set of nodes from which $u$ is reachable (the “predecessors” $pred[u]$). Additionally, we have a mapping $through[u] : V \mapsto V$ for each node $u$ that maps each node $s_u \in succ[u]$ to the next node $l_u$ in the path from $u$ to $s_u$. We call this node $l_u$ the leading node. Thanks to this, it is easy to reconstruct the path leading from $u$ to $s_u$ by simply following the leading nodes. We use this for explanations.

**Algorithm 6.2 Algorithms for ITC**

1: procedure **UNITE**($u,v$)
2: if **CONNECTED**($u,v$) then return
3: for all $p \in pred[u]$ do
4: $succ[p] \leftarrow succ[p] \cup \{v\}$
5: $through[p][v] \leftarrow through[p][u]$ 
6: $pred[v] \leftarrow pred[v] \cup \{p\}$
7: for all $s \in succ[v]$ do
8: $succ[u] \leftarrow succ[u] \cup \{s\}$
9: $through[u][s] \leftarrow v$
10: $pred[s] \leftarrow pred[s] \cup \{u\}$
11: for all $p \in pred[u]$ do
12: $succ[p] \leftarrow succ[p] \cup \{s\}$
13: $through[p][s] \leftarrow through[p][u]$
14: $pred[s] \leftarrow pred[v] \cup \{p\}$

1: function **CONNECTED**($u,v$)
2: return $u \in succ[v] \land v \in succ[u]$
3: end function

1: function **NODES_PATH**($u,v$)
2: $\triangleright$ Assumes $u$ and $v$ are connected
3: $p \leftarrow []$
4: $c \leftarrow u$
5: while $c \neq v$ do
6: $next \leftarrow through[u][v]$
7: $p \leftarrow p + + [next]$
8: $c \leftarrow next$
9: return $p$
10: end function
Recall how for acyclic all we needed was to update the R-UF every time a new edge \((u, v)\) was added, calling \textsc{Unite}(u,v). To detect cycles, we used the \textsc{Connected}(u,v) method of the R-UF, and to retrieve paths, the \textsc{NodePaths}(u,v) method. For the ITC we have the exact same methods. The \textsc{Unite}(u,v) method will update the sets in the transitive closures to convey the information that there is an edge \((u, v)\) (directed from \(u\) to \(v\)) that has been added. To know if there is a cycle for failure (a directed cycle), it means that two nodes are in the successor set of each other. Retrieving the path is trivial by using the \texttt{through} maps.

With only this change, the algorithms shown in Algorithm 3.6 (page 50) can be reused to detect and prevent cycles and it will only detect directed cycles and effectively implement \texttt{dag}.

### 6.3.4 A Path Global Constraint

To summarize, the last step is to build a path constraint. For this, we will not build a specific propagator, instead we use the approach by Fages and Lorca [85] that consists in a conjunction of \texttt{dtree}:

\[
\text{path}(v_V, v_E, s, d, G) \iff \text{dtree}(v_V, v_E, s, G) \land \text{dtree}(v_V, v_E, d, G^{-1})
\]

This states that a path from \(s\) to \(d\) is the intersection of a subtree of \(G\) rooted at \(s\) and a subtree of \(G^{-1}\), the inverse graph of \(G\) (see Definition 2.18 on page 20), rooted at \(d\).

### 6.4 A Distance-Bounded Path Propagator with Explanations

As we will see in the experiments, the decomposition of the path constraint as two trees (Equation 6.1) is not competitive for solving shortest path problems when compared to the alternative circuit-based formulation by Francis and Stuckey [89]. For this reason, we needed a bounded path propagator to enhance optimality proving.
In this section we develop the \texttt{bounded\_path} propagator from Definition 6.4:

$$\texttt{bounded\_path}(v_V, v_E, s, d, \mathcal{G}, w_f, w)$$

where \(s\) is the fixed source of the path, \(d\) is the fixed destination of the path, \(w\) is the length of the path, and \(w_f\) is a weighting function over \(\mathcal{E}\). As a requirement for our algorithms, the function is of the form \(w_f : \mathcal{E} \mapsto \mathbb{R}^+\). That is, all the weights must be non-negative.

The propagations in Sections 6.4.1 and 6.4.2 were already introduced by Sellmann [194] and Sellmann et al. [195], without explanation. As we will see in the experimental section, the explanations greatly improve these propagations.

### 6.4.1 Propagating Simple Distances

This constraint fails when there is no path from \(s\) to \(d\) of cost at most equal to \(w\). This property naturally extends to all nodes in the path: the distance from \(s\) to any node \(n\) in \(G\) must be no more than \(w\). The best correct lower bound for this is obviously the shortest path from \(s\) to \(n\): if the shortest path is longer than \(w\), then no path of cost less than or equal to \(w\) exists using this node \(n\).

We compute the shortest path from \(s\) to every available node in \(G\) using Dijkstra’s algorithm while avoiding forbidden arcs. This yields the shortest available path from \(s\) to all nodes. If the cost of the path to a node \(n\) is greater than \(w\), we can forbid node \(n\) as it cannot be in \(G\) (it would form a path of length greater than \(w\)). This reasoning can be applied in both directions: \(d\) cannot be further than \(w\) from any node \(n\) in the path. For this reason, we also apply this rule starting Dijkstra’s algorithm at \(d\) on the inverted graph. Of course, if the node \(n\) to be removed is mandatory, this is not a propagation but a failure instead.

To explain this inference we need to find (at least a superset of) the arcs that made the path to \(n\) too long. Let \(F_f\) be that set of arcs (initially empty). Let \(\delta_x\) be the shortest available path from \(s\) to some node \(x\), and \(\delta_x^{-1}\) the shortest available path from \(x\) to \(d\). Any arc \(e = (u, v)\) such that \(\delta_u + w_f(e) + \delta_v^{-1} \leq w\) can be used to connect \(s\) to \(d\) in no more than \(w\) (we say \(e\) is in a short-enough path). We keep these arcs in a set \(S\). We can easily
keep track of those arcs since both runs of Dijkstra’s algorithm yield $\delta_u$ and $\delta_v^{-1}$. When such an arc is forbidden, a feasible path is removed from the graph. We then add $e$ to $F_f$ in constant time. Eventually, $F_f$ contains all the arcs causing $n$ to be too far from either $s$ or $d$.

Figure 6.1 provides an example of these explanations. Here, all the edges belong to $S$ (as they provide a path shorter than the limit $w = 10$), except for $e_3$ (of weight 9) which can never participate in any path of length less than 10. If any edge $e$ in $S$ becomes forbidden, then $e$ is added to $F_f$ and used in the explanations for all the propagations.

![Figure 6.1: Example of basic explanations for bounded paths. The distances starting at $s$ (‘\(\triangleright\)’) and $n$ (‘\(\triangleleft\)’) to all nodes are given by two passes of Dijkstra’s algorithm.](image)

Observe how, by that construction, $F_f$ does not just contain the arcs making $n$ be further than $w$ from the source. $F_f$ actually contains all the arcs that were part of a path that was shorter than $w$ from $s$ to $d$. This explanation is therefore correct, but not targeted to a specific node $n$. For example, in Figure 6.1, if $e_1$ and $e_2$ are forbidden, we don’t actually need both of them in the explanation. We will address this flaw and give an example in Figure 6.3 later on.

We have the following Theorem:

**Theorem 6.1.** Let $[\delta_d \leq w]$ be the literal stating that $\delta_d$ (i.e. the length of G) should be less than or equal to $w$ ($w$ is typically a variable). Then,

$$[\delta_d \leq w] \land \bigwedge_{e_j \in F_f} \neg b_{e_j} \Rightarrow \neg b_n$$

is a valid explanation for why $n$ cannot be in $G$.

The explanations in Theorem 6.1 can be used to explain failure too.
These explanations can be computed very efficiently by storing a function giving constant time access to whether an arc has been in a short-enough path. Upon removal of an arc $e$, we add it to $F_f$ if $e$ has been in a short-enough path.

### 6.4.2 Propagating Combined Distances

The previous rule removes any node that is too far from the source or too far from the destination to be in $G$, or detects failure. In addition, we can consider nodes through which a path from $s$ to $d$ would be longer than $w$ and filter them. We could also filter arcs contained only in paths from $s$ to $d$ that are longer than $w$.

**Proposition 6.1.** Let $\delta_u$ be the cost of the shortest path from $s$ to $u$, and let $\delta_u^{-1}$ be the cost of the shortest path from $u$ to $d$. If $\delta_u + \delta_u^{-1} > w$, then $u$ cannot be in the path from $s$ to $d$ of cost less than or equal to $w$.

**Proposition 6.2.** Let $e = (u, v)$ be an arc of cost $w_f(e)$. Let $\delta_u$ and $\delta_v^{-1}$ be the cost of the shortest paths from $s$ to $u$ and $v$ to $d$ respectively. If $\delta_u + w_f(e) + \delta_v^{-1} > w$, then $e$ cannot be in a path from $s$ to $d$ of cost less than or equal to $w$.

We use these observations to filter out nodes and arcs that cannot participate in the path from $s$ to $d$.

To explain these propagations, we note that if the filtered element (either node or arc) was mandatory, we would have to fail. Thus the explanations are the same as given in Theorem 6.1 (applied to the node or the arc we are propagating here). The explanations are:

**Prop. 6.1:** $[\delta_d < w] \land \bigwedge_{e_f \in F_f} \neg b_{e_f} \Rightarrow \neg b_u$

**Prop. 6.2:** $[\delta_d < w] \land \bigwedge_{e_f \in F_f} \neg b_{e_f} \Rightarrow \neg b_e$

where $F_f$ is the same set of forbidden arcs described in Section 6.4.1. These explanations can be used for failure if either $u$ or $e$ is mandatory.
6.4.3 Stronger Bounding Using Dynamic Programming

Although the implementation of \texttt{bounded} \texttt{path} explained above proves to be useful, the bound is too weak if there are many intermediate nodes. For this reason, we developed a dynamic programming (DP) lower bound. If the previous one does not prune, we run a more expensive DP algorithm to find the shortest path from \textit{s} to \textit{d} containing all the mandatory nodes. We call the set of all mandatory nodes at any stage of the search \textit{M}.

The algorithm is similar to Dijkstra’s, but our priority queue stores more information. Each entry of the queue is a tuple \((u, m_p, \gamma)\): a node \(u\), the set of mandatory nodes \(m_p\) visited in some path \(p\) leading to \(u\) and the length of \(p\), called \(\gamma\). As usual, the priority is on the length.

We associate a hash-table to each node \(n\) that maps sets of mandatory nodes (encoded as bit-sets in our implementation) to the length of the path visiting those nodes before reaching \(n\). Formally the tables are functions \(\text{tables}[n] : (M' \subseteq M) \mapsto \mathbb{N}\).

Then, when a tuple \((u, m_p, \gamma)\) is retrieved from the queue, the algorithm considers each available arc \((u, v)\) from \(u\). For each neighbor node \(v\), we first check if there is a set \(m'_p\) in its table such that \(m'_p = m_p\). If \(m'_p\) exists and its associated cost is greater than \(\gamma + w_f((u, v))\), we update the entry on \(v\)'s table, and enqueue \((v, m_p \oplus v, \gamma + w_f((u, v)))\), where \(\oplus\) is an operator that adds \(v\) to \(m_p\) if and only if \(v\) is mandatory, and returns \(m_p\) otherwise. If such \(m'_p\) does not exist, we add that same entry to \(v\)'s table and enqueue it. We do not need to enqueue or update any table if \(m'_p\) exists and its associated cost is less than \(\gamma + w_f((u, v))\).

The cost of the shortest path to \(d\) containing all the nodes will be found in \(d\)'s table. If such a path does not exist, we simply return an error code and fail with the naive explanation (all the fixed arcs and nodes). In practice, this rarely happens.

Notice that this algorithm does not give simple paths, and therefore it does not give an exact lower bound. Indeed, if we did, we would need to keep track of all the nodes in the paths in the tables, making the state space grow too quickly. Instead we only keep track of the mandatory nodes visited.

As a starting point, we use the same explanation for pruning as in Theorem 6.1. We only need to add the conjunction of \(b_n\) for all the mandatory nodes \(n\). We will see
Algorithm 6.3 Shortest path from $s$ to $d$ containing all mandatory nodes $M$

\begin{algorithm}
1: procedure $DP\text{BOUND}(G, s, d, v_v, v_e, w_f, M)$
2: \hspace{1em} $Q \leftarrow \text{new PriorityQueue}()$
3: \hspace{1em} $Q.PUSH((s, \{s\}, 0))$
4: \hspace{1em} $tables[s][\{s\}] \leftarrow 0$  \CommentOne{One table per node}
5: \hspace{1em} while $\neg Q.\text{EMPTY}$ do
6: \hspace{2em} $(u, m_p, \gamma) \leftarrow Q.\text{TOP}$
7: \hspace{2em} $Q.\text{POP}()$
8: \hspace{2em} if $m_p \in tables[u] \land tables[u][m_p] < \gamma$ then
9: \hspace{3em} continue;
10: \hspace{2em} for all $e = (u, v) \in E$ do
11: \hspace{3em} if $\neg b_e$ then
12: \hspace{4em} continue;
13: \hspace{3em} if $m_p \not\in tables[v] \lor tables[v][m_p] > \gamma + w_f(e)$ then \CommentOne{The entry exists}
14: \hspace{4em} $tables[v][m_p] \leftarrow (p, \gamma + w_f(e))$
15: \hspace{2em} $Q.\text{PUSH}(v, m_p \oplus v, \gamma + w_f(e))$  \CommentOne{$S \oplus v$ adds $v$ to set $S$ iff $v \in M$}
16: return $tables[d][M]$
\end{algorithm}

later how to improve this explanations for this DP algorithm. Note that the asymptotic complexity of this algorithm is $O(n \cdot 2^{|M|} M \cdot \log(|V|))$, hence the state may grow prohibitively. We will study solutions to this issue in the next subsection. Nonetheless, as we will see in the results, this explosion rarely happens since, usually, the higher the number of mandatory nodes, the smaller the choice in arcs.

6.4.3.1 Limiting state explosion in the DP propagation

Strongly Connected Components: Some basic inference we can take into account to reduce the state explosion is based on strongly connected components (SCC) of the current graph.

Definition 6.6. A Strongly Connected Component (SCC) of a directed graph $G = (V,E)$ is a subgraph $G'$ of $G$ where all the nodes contain a directed path to each other.

Our first observations is that there is no point for the DP algorithm to take an arc leaving SCC $A$ if it has not yet visited all the mandatory nodes in $A$. All the work done when leaving $A$ before visiting all the mandatory nodes of $A$ will be useless, as there will be no way of getting to $d$ with a path containing all the mandatory nodes.

We use Kosaraju’s algorithm [109] to compute SCCs. We then label the SCCs as fol-
Each node on the left graph is an SCC of the form of the right graph (from [173]). Each SCC contains 3 random mandatory nodes. All edges have weight 1.

Figure 6.2: Example of use of SCCs to accelerate Algorithm 6.3

...
of the clusters only as mandatory nodes (i.e., we have as many mandatory nodes as clusters, treating the non-centroid nodes as unfixed nodes). Because the centroids tend to be equidistant to the other nodes in the cluster, the DP tends to also use some of the other mandatory nodes, thus visiting more than just the centroids. Also, since k-means has some inherent randomness, we have different clusters every time, which is also beneficial for the lower bound.

This has huge performance effects, but is a double-edged sword: the DP gets faster but we prune much less often as the bound is not as high. In order to regulate this, we use a simple heuristic based on the time spent by the DP. If the DP algorithm with $C$ clusters takes less than $x$ seconds, we increase $C$ by 1, if it takes more than $y$ seconds, we decrease it by 1. For the experiments where we used clustering, we chose $x = 0.5s$, $y = 8.0s$ and started with $C = 5$.

### 6.4.4 Improving the Explanations of the Bounded Path Propagator

So far, the explanations for bounded path have been the set of forbidden arcs that were in a short-enough path at some point in the search, noted $F_f$ in Section 6.4.1. One problem with these explanations is that they are not targeted to specific nodes. It is easy to see that some of the arcs in the explanations may have nothing to do with the fact that some specific node $n$ is too far from the source. We now provide better explanations for the previous propagations.

#### 6.4.4.1 Simple and Combined Distances Propagation

First, during the propagation we use Dijkstra’s algorithm on the available graph. This leaves a label on each node indicating how far it is from the source. These labels are noted $\delta_n$, $\forall n \in V$. Nodes not visited have label $\delta_n = \infty$.

Let $n$ be a node that is at distance $\delta_n$ more than the limit $w$ from the source. Algorithm 6.4 returns a set of forbidden arcs that explain why $\delta_n > w$.

Algorithm 6.4 mimics Dijkstra’s starting at $n$ in $G^{-1}$. For any arc $e = (v, u)$ of weight $w_f(e)$ we know $\delta_v$ (obtained during propagation). We also have the distance from $u$ to $n$
Algorithm 6.4 Explaining why \( n \) is at distance more than \( w \) from the source

1: procedure EXPLAINDIST(\( G, s, n, \{ \delta_u|u \in V \}, w \)) \( \triangleright \) We consider all arcs in \( G \)
2: \( Q \leftarrow \text{new PriorityQueue}() \)
3: \( Q.PUSH((n,0)) \)
4: \( X = \emptyset \)
5: \( \text{cost} = [\infty|v \in V] \)
6: while \( \neg Q.\text{EMPTY} \) do
7: \( (u, \delta_u^{-1}) \leftarrow Q.\text{TOP} \) \( \triangleright \delta_u^{-1} : \text{cost of the shortest path from } n \text{ to } u \)
8: \( Q.\text{POP}() \)
9: if \( u = s \) then
10: \( \text{break} \) \( \triangleright \text{Reached start } s \)
11: for all \( e = (v,u) \in \{ e^{-1}|e \in E \} \) do
12: if \( \neg b_e \land \delta_v + w_f(e) + \delta_u^{-1} \leq w \) then
13: \( X = X \cup \{ e \} \)
14: else if \( \text{cost}[v] > \delta_u^{-1} + w_f(e) \) then
15: \( \text{cost}[v] = \delta_u^{-1} + w_f(e) \) \( \triangleright \text{Update cost} \)
16: \( Q.\text{PUSH}((v,\text{cost}[v])) \) \( \triangleright \text{Overwrites previous instances of } v \text{ in } Q \)
17: return \( X \)

(the cost of the last current node in the loop, line 7). Let \( X \) be the initially empty set of arcs explaining why \( n \) is at distance more than \( w \) from \( s \). When considering a forbidden arc, if \( \delta_v + w_f(e) + \delta_u^{-1} \leq w \), \( e \) participates in a path from \( s \) to \( n \) no longer than \( w \). Therefore we add it to the explanation \textit{and we do not cross it.} Otherwise, we can cross it. Once we dequeue \( s \) we finish since all other paths are no shorter than \( \delta_s^{-1} > w \). See Figure 6.3 for an example of explanation.

![Figure 6.3](image-url)  

Figure 6.3: Example of improved explanations for bounded paths. The labels for propagation (‘\( \triangleright \)’, from Dijkstra’s algorithm) and explanation (‘\( \triangleleft \)’, from Algorithm 6.4) are given next to each node.

**Theorem 6.2.** The clause \( [\delta_d \leq w] \land \land_{e_f \in X} \neg b_{e_f} \Rightarrow \neg b_n \) computed by Algorithm 6.4 is a correct and \textit{minimal} explanation for why \( n \) is too far from \( s \) to be in \( G \).

**Proof.** Let \( F \) be the set of forbidden arcs at the time of explanation. At any stage of Al-
Algorithm 6.4, let $F_p$ be the set of forbidden arcs not yet considered (initially $F$), $X$ the arcs in the explanation, $d_{G'}(u,v)$ the shortest distance from $u$ to $v$ for any $G' \subseteq G$, and $u$ the top of $Q$. Let $G_R = G \setminus (F_p \cup X)$. We ensure correctness and minimality by preserving the following invariants: (1) $d_{G_R}(s,n) > w$, (2) for all $(v',u') \in F_p$, $d_{G_R}(u,n) \leq d_{G_R}(u',n)$, and (3) for all $e \in X$, $d_{G_R \cup \{e\}}(s,n) \leq w$.

The three invariants hold initially: $G_R = G \setminus F = G$, so (1) is the bound to be explained, (2) holds because $n$ is initially the head of $Q$ and all weights are non-negative, and (3) holds because $X$ is initially empty.

At each iteration, we remove $u$ from $Q$ and process each arc $e = (v,u) \in E$ (removing all forbidden arcs $(v,u)$ from $F_p$, preserving (2) as nodes are processed in order of distance from $n$). We add arcs such that $\delta_v + w_f(e) + \delta_u^{-1} \leq w$ to $X$ (preserving property (3)). Other forbidden arcs are now made available in $G_R$.

We show how adding these arcs to $G_R$ maintains the invariants. Note $d_{G_R}(x,n)$ values for previously processed nodes $x$ remain unchanged as any newly introduced path from $n$ must be at least as long as $\delta_u^{-1}$. Newly available arcs may, however, decrease $d_{G_R}(x,n)$ for some $x$ which has not yet been processed. However, if $d_{G_R}(x,n)$ decreased as a result of $(v,u)$ becoming available, then the shortest path from $x$ to $n$ must pass through $u$. But, $x$ is not yet processed, so still $d_{G_R}(u,n) \leq d_{G_R}(x,n)$, preserving property (2). If the shortest path from $s$ to $n$ were to be reduced because now $\delta_x + d_{G_R}(x,n) < \delta_n$, there is a contradiction since this path goes through $(v,u)$ meaning the arc should have been added to $X$ and be unavailable. Hence property (1) is preserved. Adding arcs to $G_R$ can only make paths shorter, hence property (3) is preserved.

Once $s$ is popped from $Q$, our explanation is $X$. By (1), $F_p \cup X$ is a valid explanation; but by (2), no arc $e$ remaining in $F_p$ may be on a shorter path from $s$ to $n$ (as either $n$ is unreachable via $e$, or the head of the arc is distance no less than $\delta_s^{-1}$ from $n$), so $e$ may be omitted from the explanation. Thus $X$ is also a valid explanation. Removing arcs from $F_p$, thus adding them to $G_R$ preserves property (3). By (3), omitting any element of $X$ introduces a path from $s$ to $n$ of length no greater than $w$, so $X$ is also minimal.

Clearly, Algorithm 6.4 runs in $O(|E| + |V| \log(|V|))$, like Dijkstra’s algorithm. We can use it to explain Propositions 6.1 and 6.2 as follows. For Proposition 6.1, we first obtain
$X_1 = \text{EXPLAINDIST}(G, s, u, \{\delta_v | v \in V\}, w - \delta_u^{-1})$, the explanation for $u$ being at distance $w - \delta_u^{-1}$ from $s$. The call to Algorithm 6.4 also yields the distance $\delta_u^*$ from $s$ to $u$ in $G$ that is still greater than $w - \delta_u^{-1}$. Let $X_2$ be the explanation for $d$ being at distance $w - \delta_u^*$ from $u$. The final explanation is $X = X_1 \cup X_2$. The same idea can be used for Proposition 6.2, using the head and tail of the arc to be removed.

### 6.4.4.2 DP-based Propagation

We can also improve the explanations for the DP-based propagation. Similarly to the simple propagation, Algorithm 6.3 leaves a table on each node stating the cost of visiting some subsets of mandatory nodes before getting to that node. If $d$ is not reachable in less than $w$ (included) visiting all mandatory nodes, we fail and explain the failure. To do so, we run the same Algorithm 6.3 starting at $d$ on the inverted graph allowing forbidden arcs (similarly to Algorithm 6.4).

Let $e^{-1} = (v, u)$ be some reversed forbidden arc of cost $w_f(e)$. During the propagation pass of Algorithm 6.3, we associated a hash-table to node $u$ (the tail of $e$ in the original graph). Each row of the table is a pair $(m_u, \gamma_u)$ as defined in Section 6.4.3.

Symmetrically, node $v$ contains a table where each row $(m_v, \gamma_v)$ indicates the mandatory nodes visited from $d$ to $v$. If there exists an entry $(m_u, \gamma_u)$ in $u$’s table and an entry $(m_v, \gamma_v)$ in $v$’s table such that $m_v \cup m_u = M$, then $e$ is an arc that could be used in a path from $s$ to $d$ containing all nodes in $M$. If additionally, $\gamma_v + w_f(e) + \gamma_u \leq w$, that path would be a valid path. Therefore, $e$ being forbidden explains why we can’t reach $d$ visiting all mandatory nodes in no more than $w$. This corresponds to substituting the if-condition in line 11 of Algorithm 6.3 with a call to EXPLAIN from Algorithm 6.5. Following the notations in Algorithm 6.3, the call would have the arguments as EXPLAIN($(v, u), \gamma_v, m_v)$.

**State explosion for explanations:** The explanation algorithm needs to use the same mandatory nodes as the propagation. Therefore, if we clustered, the same clustering is given to this algorithm. Also, we cannot use SCC levels here (other than the ones computed at the root) since we need to traverse forbidden arcs whether or not they skip
entire mandatory SCCs as there may be other forbidden arcs leading to the skipped SCCs later.

A major problem with these explanations is that we need to traverse forbidden arcs. In dense graphs, this can be slow as there may be many possible paths to consider. For this reason, we use a simple stopping condition. Let \( t_p \) be the time it takes to run Algorithm 6.3 for propagation. If explaining is taking more than \( x \times t_p \) (we choose \( x \) arbitrarily) we switch to the version of EXPLAIN in Algorithm 6.6 which corresponds to the basic explanations described in Theorem 6.1. We say that we interrupt the explanation when this change happens.

Algorithm 6.5 Better explanations

1: function EXPLAIN\( (e, \gamma_v, m) \)
2: for all \((m_h, \gamma_h) \in \text{table}[\text{head}(e)]\) do
3: if \( m_h \cup m_v = M \) then
4: if \( \gamma_h + w_f(e) + \gamma_v \leq w \) then
5: explanation.add\( (\neg b_e) \)
6: return true
7: return false

Algorithm 6.6 Avoiding state explosion

1: function EXPLAIN\( (e, \gamma, m) \)
2: \( \triangledown \text{was short}(e) = \text{true} \iff e \text{ was in a short-enough path at some point.} \)
3: if \( \neg b_e \land \text{was short}(e) \) then
4: explanation.add\( (\neg b_e) \)
5: return true
6: return false

6.5 Experimental Evaluation and Results

In this section we test our propagators on different problems (all benchmarks available [59]). We implemented all our work in CHUFFED. All tests reported were run on a Linux 3.16 Intel® Core™ i7-4770 CPU @ 3.40GHz, 15.6GB of RAM.

We annotate the tests EXPL when learning is enabled using the explanations from Section 6.4.1, EXPL* for the improved explanations from Section 6.4.4 and NOEXPL if both are disabled. We name PATH the version of CHUFFED using the tree decomposition for the path constraint, BPATH the version with the bounded_path propagator without the DP algorithm, and DPBPATH when using the DP algorithm. We call CIRCUIT the circuit-based version of a path propagator developed by Francis and Stuckey [89]. We compare the number of conflicts, nodes in the search and the time in seconds.

We found it beneficial to add an array of successors constrained as \( b_e \Leftrightarrow \text{succ} [\text{tail}(e)] = \text{head}(e) \). Definitions of all search strategies are given in MINIZINC language [153].
6.5.1 The Maximum Directed Acyclic Subgraph Problem (MDSP)

We first test our DAG propagator. The Maximum Directed Acyclic Subgraph Problem (MDSP) is formulated as follows. Given a graph $G = (V, E)$, the solution to the MDSP is a subgraph $G' = (V', E')$ of $G$ which is a DAG with maximum cardinality of $E$. This is also known as the Minimum Feedback Set problem, which was proven to be NP-complete by Karp [115]. This problem has never been tacked in CP as far as we know, although it has been widely investigated with approximation algorithms [27, 83, 107].

We compare the use of our dag propagator against a decomposition of the problem. The input is a graph $G$. Since our dag propagator requires a root node, we add an artificial root node and arcs from that node to all the other ones (we fix them to be in $G$ in the model). In the decomposition we use an integer variable $d_n$ for each node $n$ with domain $d_n \in [0..|E|]$ representing the distance from the root $r$. The decomposition is shown below.

Maximize $\sum_{e \in E} b_e$ such that
\begin{align*}
  d_r &= 0 \quad (6.3) \\
  \forall n \in V, d_n &= \max\{(d_{\text{tail}(e)} + 1) * b_e \mid e \in \text{incident}[n]\} \quad (6.4)
\end{align*}

Equation 6.3 forces the root to be at distance 0. Equation 6.4 computes the distance of each node $n$ as the maximum of the distance of the nodes leading to $n$ plus one.

| $|V|$ | $|E|$ | Method | Clause learning enabled | Clause learning disabled |
|-----|-----|-------|-------------------------|-------------------------|
|     |     |       | Fails | Nodes | Time(s) | Fails | Nodes | Time(s) |
| 25  | 52  | Propagator dag | 2071 | 3389 | 0.11    | 22705 | 46170 | 0.55    |
| 30  | 85  | Propagator dag | 82504| 120448| 5.70    | 1239883| 2487893| 43.57    |
| 25  | 52  | Decomposition | 71542| 106397| 6.92(2) | 139605 | 280559 | 12.67(3) |
| 30  | 85  | Decomposition | 2801015| 3612877| 478.03(30) | 3661220| 6731010| 497.60(29) |

Table 6.1: Comparison of propagator vs. decomposition for dag constraint. The number of timeouts in parenthesis. The number of arcs is an average among all the instances.

We generated 50 random instances of 25 nodes and another 50 of 30 nodes. The results are in Table 6.1. As we can see, this is quite a challenging problem even for small instances, many of them reach the limit of 30 minutes. The use of the dag propagator is crucial to solve this problems in practical time. Also, explanations are tremendously
6.5 Experimental Evaluation and Results

beneficial in this problem.

6.5.2 Node Constrained Shortest Paths

Here we compare our path propagators with the results from Quesada et al. [173] using their same benchmarks. The aim of these problems is to find the shortest path between two given nodes in a graph \( G = (V, E) \) passing through a given set of mandatory nodes \( M \). We present the results in Table 6.2 using `first_fail` on the `succ` variables as the search strategy.

| Benchmark   | \(|V|\) | Quesada et al. [173] | CIRCUIT | PATH | PATH +BPATH | PATH +DPBPATH |
|-------------|-------|----------------------|---------|------|-------------|---------------|
|             | Fails | Time(s)              | Fails   | Time(s) | Fails | Time(s) | Fails | Time(s) | Fails | Time(s) | Fails | Time(s) |
| Ham22       | 22    | 13                   | 4.45    | 24     | 0.00 | 139 | 0.03 | 19 | 0.01 | 16 | 0.01 |
| Ham22full   | 22    | 0                    | 1.22    | 2      | 0.00 | 19  | 0.01 | 15 | 0.01 | 15 | 0.01 |
| Ham52b      | 52    | 100                  | 402     | 112    | 0.01 | 1119 | 0.81 | 19 | 0.07 | 19 | 0.22 |
| Ham52full   | 52    | 3                    | 45.03   | 5      | 0.00 | 90  | 0.13 | 72 | 0.11 | 72 | 0.58 (C) |
| Ham52order_a| 52    | 16                   | 57.07   | 97     | 0.02 | 2203 | 2.54 | 189 | 0.45 | 76 | 3.80 |
| Ham52order_b| 52    | 41                   | 117     | 1      | 0.00 | 49  | 0.04 | 49 | 0.05 | 49 | 0.08 |
| Ham22       | 22    |                      |         |        |      | 202 | 0.02 | 34 | 0.01 | 22 | 0.01 |
| Ham22full   | 22    |                      |         |        |      | 35  | 0.01 | 27 | 0.01 | 13 | 0.74 |
| Ham52b      | 52    |                      |         |        |      | 17579 | 6.04 | 1523 | 0.76 | 21 | 4.03 |
| Ham52full   | 52    |                      |         |        |      | 328 | 0.12 | 264 | 0.12 | 264 | 0.59(C) |
| Ham52order_a| 52    |                      |         |        |      | 17438 | 7.93 | 1409 | 0.83 | 407 | 0.38 |
| Ham52order_b| 52    |                      |         |        |      | 83  | 0.03 | 83 | 0.03 | 83 | 0.13 |

Table 6.2: Comparison between the results reported by Quesada et al. [173], CIRCUIT, PATH, BPATH, DPBPATH, EXPL and NoEXPL. (C) indicates when clustering is used.

Details about the instances can be found in the paper by Quesada et al. [173]. In the name of the instances, “full” means that all the nodes are mandatory and “order” means that the some of the mandatory nodes must be visited in a given order.

We clearly see that we solved the benchmarks faster than Quesada et al. [173]. We also see how BPATH and DPBPATH improve the results obtained by PATH, which is the point of having a bounded_path propagator. We can also see that the explanations reduced the number of failures greatly, specially for the two instances with biggest search space (52b and 52order_a). We do not show EXPL* as they don’t improve on EXPL, because the search space is already very small, and EXPL* is more expensive than EXPL.

Although slow, the DPBPATH is still suitable for the Hamiltonian path of 22 nodes. For 52 nodes in such dense graph though, the state space explodes and we absolutely need to cluster.
We also compared against the circuit-based path propagator with explanations presented by Francis and Stuckey [89]. Their propagator is surprisingly fast on these benchmarks and requires little search. This is because their propagator has much better reasoning over the topology of the graph. The topological reasoning of our case is done by the path propagator, which is a combination of directed trees (Equation 6.1), whereas their propagator makes more inferences based on strongly connected components and starting the path at different nodes. This specific benchmarks are simple in terms of distance (all the arcs have the same weight), but hard in terms of topology, hence the advantage. It is worth noting that the work by Beldiceanu et al. [26] also tackled this problem by using the tree propagator described by Beldiceanu et al. [22] (already discussed in Section 6.2) in combination with precedence constraints, achieving 0 failures in all instances of this problem, due to the strong reasoning that we also witness in the circuit constraint.

The take-away from this experiment is that for graphs that are topologically hard, using our propagator might be a burden whereas using other propagators with strong topological reasoning as the one by Francis and Stuckey [89] might be a better approach.

### 6.5.3 Metabolic Networks

A metabolic network is a network of molecules and reactions. Biologists use this to understand how some molecules transform into others and cause some behavior in cells. For instance, this helps biologists understand how a protein behaves or how gene expression is regulated. This problem was modeled by Viegas and Azevedo [215] by creating a bipartite graph where molecules are in one partition of the nodes, and reactions in another partition. The arcs of the graph link the substrates and products participating in a reaction to the reaction itself.

Here, there is a set of mandatory nodes (because biologists are aware of their existence) and mutually exclusive nodes (corresponding to mutually exclusive reactions). Furthermore, each node is given a weight corresponding to its degree (this is to model highly connected molecules). The objective is to find a pathway from some given substrate to some given product minimizing the total weight of the path, where the weight is the sum of the degrees of the nodes.
Table 6.3 shows a comparison between our solver and the work by Viegas and Azevedo [215], which used the solvers GRASPER and CP(Graph) on an Intel Core 2 Duo 2.16GHz. Here BPATH stands for PATH + BPATH. There is one instance for each size.

| Value | Glycosis | Lysine | Heme | | | | | | |
|-------|---------|--------|------|---------|--------|---------|--------|----|
|       | GRASP   | CP(Graph) | PATH | BPATH   | GRASP   | CP(Graph) | PATH | BPATH   | GRASP   | CP(Graph) | PATH | BPATH   | GRASP   | CP(Graph) | PATH | BPATH   |
| 500   | 0.28    | 0.21    | 0.05 | 0.11    | 0.36    | 0.41    | 0.06 | 0.12    | 0.23    | 0.10    | 0.05 | 0.22    |
| 600   | 0.38    | 0.31    | 0.07 | 0.17    | 0.48    | 0.44    | 0.06 | 0.16    | 0.28    | 0.12    | 0.06 | 0.31    |
| 700   | 0.45    | 0.35    | 0.19 | 0.22    | 0.47    | 0.75    | 0.08 | 0.25    | 0.36    | 0.16    | 0.08 | 0.46    |
| 800   | 0.53    | 0.50    | 0.24 | 0.29    | 0.53    | 1.00    | 0.12 | 0.37    | 0.41    | 0.19    | 0.11 | 0.55    |
| 900   | 0.64    | 0.68    | 0.15 | 0.39    | 0.57    | 1.29    | 0.16 | 0.4    | 0.51    | 0.27    | 0.15 | 0.73    |
| 1000  | 0.77    | 0.84    | 0.18 | 0.51    | 0.60    | 1.37    | 0.18 | 0.46    | 0.62    | 0.32    | 0.18 | 0.95    |
| 1100  | 0.91    | 1.00    | 0.17 | 0.71    | 0.73    | 1.29    | 0.39 | 0.64    | 0.65    | 0.33    | 0.32 | 1.08    |
| 1200  | 0.96    | 1.08    | 0.20 | 0.75    | 0.86    | 2.23    | 0.23 | 0.79    | 0.80    | 0.41    | 0.21 | 3.62    |
| 1300  | 1.03    | 1.21    | 0.81 | 0.84    | 0.99    | 2.50    | 0.28 | 1.02    | 0.94    | 0.47    | 0.4  | 1.81    |
| 1400  | 1.23    | 1.56    | 0.71 | 1.05    | 1.12    | 2.84    | 0.30 | 1.17    | 1.11    | 0.51    | 0.4  | 2.1     |
| 1500  | 1.40    | 1.85    | 1.25 | 1.28    | 1.25    | 2.92    | 0.39 | 1.33    | 1.14    | 0.52    | 0.94 | 2.09    |
| 1600  | 1.67    | 2.14    | 0.75 | 1.49    | 1.30    | 2.97    | 0.43 | 1.36    | 1.35    | 0.61    | 0.74 | 2.55    |
| 1700  | 1.93    | 2.40    | 0.82 | 1.77    | 1.41    | 3.03    | 0.67 | 1.44    | 1.57    | 0.69    | 0.4  | 3.08    |
| 1800  | 2.11    | 2.77    | 1.01 | 2.01    | 1.53    | 3.69    | 0.49 | 1.69    | 1.72    | 0.77    | 0.45 | 3.69    |
| 1900  | 2.27    | 3.02    | 1.19 | 2.21    | 1.75    | 3.93    | 0.69 | 1.95    | 1.96    | 0.84    | 0.48 | 6.21    |
| 2000  | 2.40    | 3.14    | 1.33 | 2.3     | 1.96    | 2.18    | 0.64 | 2.39    | 2.18    | 0.91    | 0.51 | 4.86    |

Table 6.3: Solving metabolic pathways in real-world networks (same strategy as [215]).

The results show that BPATH slows PATH down. We interpret this as the effect of the overhead of bounded_path. Indeed, the instances are solved so quickly by PATH that BPATH has little to improve on. We also ran the same experiments with the VSIDS [148] search strategy. The times were very similar to those in Table 6.3 for PATH, but 3 benchmarks (1200, 1300 and 1900 nodes for Heme) were much slower (around 30 seconds). We tested the BPATH version on those three instances and noticed a big speedup (between 5 and 15 times faster). Nonetheless, note how BPATH is still faster than GRASPER and CP(Graph) in two thirds of the tests. From this we conclude that bounding is only worthwhile if the instances are hard to solve (i.e. there is a big search space to explore).

6.5.4 Task Constrained Shortest Path

In this problem, we are required to perform a set of tasks along a path. A task can be done at different nodes, and visiting a node where some task can be performed is enough, we do not need to visit more than one. As an example, consider on the drive home withdrawing money from an ATM, going to a carwash and buying some groceries. Any ATM, supermarket or carwash on the path is sufficient. This problem was studied by Rice and Tsotras [184, 185] using dynamic programming only.
Francis and Stuckey [89] used a circuit-based path propagator to solve a similar problem (minimizing the longest arc). We compare our implementation against theirs using the same instances (500 graphs of 20 nodes each) with the objective of minimizing the total length of the path. The aim of this experiment is to see if BPATH and DPBPATH can also improve the circuit-based path propagator.

In this experiment we compare the best runtimes of both approaches, even if they use two different strategies. Our best search strategy is smallest (i.e. branching on the succ variable with smallest domain) and their best search strategy is first_fail on the succ variables. Additionally, we combine our bounding propagator with theirs to see the benefits.

Table 6.4: Comparing BPATH and DPBPATH using PATH and CIRCUIT constraints. Geometric average over 500 instances of 20 nodes.

Table 6.4 gives the results, also showing the time (Opt) to find (but not prove) the optimal solution. The PATH version finds optimal solutions very fast, but takes time to prove optimality. On the other hand, the CIRCUIT is superior in both these aspects. Adding BPATH and DPBPATH improves both these versions. The CIRCUIT version does 89% less search when combined with DPBPATH (in its fastest version, using EXPL), and PATH does 98% less search when combined with DPBPATH (using EXPL*). This shows how bounded_path with explanations can be used in combination with both tree-based and circuit-based path propagators to enhance propagation.

6.5.5 Non-Overlapping Paths

In order to test the power of the version BPATH, we use the Non-Overlapping Paths problem devised by Beldiceanu and Lorca [21] (k Node-Disjoint Path Problem in their paper), also known as the Cooperative Shortest Path [199, 206], which occurs in AI in video-games, robotics and transportation. In this problem, a set of agents need to get to
their destinations while avoiding collisions (otherwise unwanted behavior like deadlocks or cyclic repetitions can happen).

For this series of tests, we used the London Underground system as $G (|V| = 309, |E| = 740)$. The source and destination of each agent were chosen randomly on opposite sides of the outer city. The paths followed by the agents cannot share nodes. The objective is to minimize the sum of the cost of each path.

<table>
<thead>
<tr>
<th>Nb. of agents</th>
<th>Fails</th>
<th>Nb. of agents</th>
<th>Fails</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 13039</td>
<td>21959</td>
<td>55 1115</td>
<td>2.92</td>
<td></td>
</tr>
<tr>
<td>3 8205</td>
<td>15500</td>
<td>173 1218</td>
<td>6.23</td>
<td></td>
</tr>
<tr>
<td>4 2080</td>
<td>4831</td>
<td>280 2454</td>
<td>6.00</td>
<td></td>
</tr>
<tr>
<td>5 1274</td>
<td>4292</td>
<td>211 1704</td>
<td>6.00</td>
<td></td>
</tr>
<tr>
<td>6 (unsat)</td>
<td>30</td>
<td>33 146</td>
<td>0.76</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: Cooperative shortest path finding (using EXPL, search: VSIDS).

As we can see in Table 6.5, the BPATH is tremendously beneficial in this problem. Notice how this propagator behaves very well when the problem is hard to solve. If the problem is too simple (like the unsatisfiable case with 6 agents), then the overhead of the lower bound is not beneficial to solve the problem.

### 6.5.6 Profitable Tourist Path

We introduce here a new problem (as far as we are aware) similar to the prize collecting TSP. Imagine you need to do a long layover during a trip and change airports. You might be interested in visiting the city while waiting for your connection flight. In this problem, we model every point of interest (POI) of a city with a minimum visit time (i.e. the least amount of time that a visit to some POI is worthwhile) and a profit (i.e. how much a person enjoys visiting some POI). The path can contain a node without necessarily visiting the corresponding POI, but in order to visit a POI the path must contain the corresponding node and spend the minimum visit time. The objective of the problem is to find the path with most profit such that the total time is less than a certain bound (i.e. the time we have available between connections). The total time is the cost of the path plus the time spent at each POI (either 0 or the minimum visit time).

We created two benchmarks, based on New York City (14 nodes, from LGA Airport to
JFK Airport) and London (12 nodes, from Heathrow Airport to Liverpool Street Station). We added two side constraints: for London, we require that the visit to the Tower Bridge (if it happens) takes place between two narrow time frames (which would correspond to times where the bridge opens to let ships go through); for NYC, the ferry to Liberty Island leaves every hour and so there might be a waiting time added to the total time (if the visit happens).

We used EXPL on all the tests to study the benefits of bounding. The results are in Table 6.6. Clearly, DPBPATH and BPATH largely improve PATH for this problem. Again, there was no need to cluster or interrupt explanations.

<table>
<thead>
<tr>
<th>Version</th>
<th>New York City (14 POI)</th>
<th>London (12 POI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fails</td>
<td>Nodes</td>
</tr>
<tr>
<td>PATH</td>
<td>&gt;5030898</td>
<td>&gt;5034046</td>
</tr>
<tr>
<td>PATH +BPATH (EXPL)</td>
<td>390985</td>
<td>391746</td>
</tr>
<tr>
<td>PATH +DPBPATH (EXPL)</td>
<td>44015</td>
<td>44606</td>
</tr>
<tr>
<td>PATH +BPATH (EXPL)’</td>
<td>360945</td>
<td>361971</td>
</tr>
<tr>
<td>PATH +DPBPATH (EXPL)’</td>
<td>2062</td>
<td>2690</td>
</tr>
</tbody>
</table>

Table 6.6: Profitable tourist path.

We used the search strategy smallest on the succ variables. Without explanations, though, NYC takes 1598s using DPBPATH and London takes 13s, making them substantially slower than with explanations.

### 6.6 Concluding Remarks

We have seen how to build a constraint for path using directed trees as suggested by Fages and Lorca [85]. We have seen how to add explanations to the propagators implementing that constraints.

The limit of the path constraint seems to be problems where topology is complex. In those cases, we showed that circuit-based approaches for path constraints are more suitable.

Furthermore, we looked at the problem of paths with a bound on their length, which is the most typical use of paths in graphs. For this constraint, some work had been done before, but clause learning had never been tried. We showed two ways of computing explanations for this propagator, and we implemented a new stronger dynamic-
programming based approach to bounding. Our results show that the use of learning is clearly beneficial, that the dynamic-programming approach scales well and state explosion can be avoided.

The contributions of this chapter are:

- Incremental work on explanations for `dreachable` and `dtree` constraints.
- Explanations for the propagations presented by Sellmann [194] and Sellmann et al. [195] for the `bounded_path` propagator.
- A new propagation technique for `bounded_path` using dynamic-programming, and subsequent techniques to avoid state explosion.
- Explanations for the dynamic-programming propagations.
- A new `dag` propagator with explanations and experiments showing its value.
Chapter 7
Discrete Optimization for Habitat Conservation

Modeling ecological connectivity is an area of increasing interest amongst biologists and conservation agencies. In the past few years, different modeling approaches have been used by experts in the field to understand the state of wildlife distribution. One of these approaches is based on modeling land as a resistive network. The analysis of electric current in such networks allows biologists to understand how random walkers (animals) move across the landscape. In this chapter we present a MIP model and a Local Search approach to tackle the problem of minimizing the effective resistance in an electric network. This is then mapped onto landscapes in order to decide which areas need restoration to facilitate the movement of wildlife.

This chapter is certainly an outlier from the previous chapters. While researching about Discrete Optimization applied to habitat conservation, we found this problem. We initially believed it was possible to model using path constraints in a CP model, but after some interaction with conservation experts, we understood the problem is more complicated than paths between habitats. It is also a very unconstrained problem, and therefore CP was not the best tool to tackle it. In this chapter we will see other Discrete Optimization techniques being used.

7.1 Habitat Conservation By Resistance Model

In the past decades, the natural habitat of different species across the globe have become disrupted and fragmented. This is considered to be a major threat to the conservation of biodiversity by biologists and conservation experts [189]. As pointed out by Rosenberg et al. [189] and Pimm et al. [168] among other experts, the isolation of groups of animals can easily lead to extinction. For this reason, restoration needs to be undertaken in order to maintain a suitable environment where wildlife can move, feed and breed. Landscape
discrete optimization for habitat conservation, which has been increasingly attractive for researchers in the CP and Operations Research communities as it presents interesting computational problems that happen to be NP-hard [67, 101, 131, 211, 218].

In ecology, patches of landscape are characterized by their resistance. This corresponds to how “hard” it is for wildlife to traverse a land patch. For instance, a low resistance value might be caused by soil of good quality that allows plants to spread more easily, or an open field might have high resistance for small rodents (that may be targets for birds of prey). Of course, this measure depends on the species being studied: some animals may be able to cross rivers more easily than others. Lower resistance means the land patch is more suitable for the species studied. Therefore, a suitable environment for wildlife would be one where their core habitats are connected by low resistance patches, so that animals can freely travel between core habitats to breed and feed.

7.1.1 Corridors Model

To improve landscape connectivity, ecologists have used the idea of corridors [18, 113]. A corridor connects core habitats of animals using uninterrupted paths through which the animals can move. A common measure of the quality of a corridor is know as the Least-Cost Corridor (LCC) [4, 19]. LCCs are chosen to minimize the total sum of the resistance in the corridor. There has been extensive work in the Constraint Programming, AI and OR communities in helping identify the best corridors to be built (e.g. [67, 131, 218]).

7.1.2 Multi-species Area-driven Model

Other work has addressed the problem of habitat conservation without enforcing connectivity. For instance, Crossman and Bryan [51] wrote a MIP model for habitat conservation. In their case, the intention was to minimize the number of sites to be restored while keeping a desired area for the animals and maintaining safe distances to roads and other dangers.
7.1.3 Resistance-based Model

Although LCC is a valid model that is broadly used, it has been criticized for over simplifying the actual movement of species [144]. McRae [143, 145] proposed the use of electric circuit theory to measure the total connectivity of a landscape. This model is called Isolation By Resistance (IBR). In particular, it was shown [144] how the IBR model better matches the empirical evidence of genetic distance amongst a distributed species measured with two standard statistical models (fixation index and a step-wise mutation model). The IBR model has since been used to study the effects of habitat loss in birds [10, 11], for instance.

In the IBR model, the land patches are modeled as nodes in an electric circuit. The transition between contiguous patches is modeled by a branch of the electric circuit carrying a resistor. The resistance of the resistor in Ohms gives the resistance of moving between adjacent patches. The circuit is then connected to a 1A current source in a core habitat, while another core habitat is connected to the ground (0V). The effective resistance between two habitats in the electric circuit is the measure used for connectivity in the IBR model. It physically corresponds to the real resistance between two points of the circuit. An example of such circuit with 25 land patches (i.e. 25 nodes) can be seen in Figure 7.1. The core habitats are labeled $s$ and $t$.

![Resistor network for a landscape](image)

Figure 7.1: Example of resistor network for a landscape. Habitats are $s$ and $t$. Colors indicate current intensity (red being the highest and green the lowest).

Experts use the tool Circuitscape [196] for this model. Its task is to compute the currents, voltages and effective resistance that are then viewable by experts in geographic visualization software. Nevertheless, Circuitscape does not make conservation decisions,
it only builds a linear system and solves it for the experts.

The model is justified by the fact that the commute time between two nodes $s$ and $t$ is given by $2mR_{st}$ where $m$ is the number of edges in the graph, and $R_{st}$ is the effective resistance between $s$ and $t$ in the underlying electrical network. This was proven by Lovász [140] (c.f. Theorem 4.1 in his paper). The goal is therefore to lower the effective resistance between core habitats, as this directly translates into decreasing the commute time between habitats. Previous work by Doyle and Snell [75] also proved this property, and gave an interpretation of current $i_{xy}$ through a branch $(x, y)$ of the underlying electric network. The current gives the net number of times a random walker would walk from $x$ to $y$ using that branch. This is exactly what is used by biologists to detect areas where wildlife concentrates most in the IBR model [145]. As an example, consider Figure 7.2. On the right side the landscape is plotted showing the conductance values in $\Omega^{-1}$ (inverse of resistance). On the left, a heatmap of the current at each patch of land. We can observe that, when moving out of their habitats, animals tend to walk using high conductance areas: indeed the $[0.2, 1]$ areas of the heatmap tend to coincide with high conductance areas on the conductance map.

Figure 7.2: Example of landscape. Left: The heatmap shows the current (in Amperes) in the electric circuit. The three darkest/red patches correspond to core habitats. This model predicts that animals will walk on the high current areas ($\geq 0.2A$, yellow/orange) often and less often in low current ($\leq 0.2A$, green) areas. Right: Map of conductance (in $\Omega^{-1}$). Darker/green areas are land where the resistance is low and easy for animals to move across.

For habitat and ecologic planning using the Isolation By Resistance model, we are
interested in minimizing the effective resistance between habitats. To do so, biologists need to decide where improving the habitat is more beneficial. An improvement could mean planning for reforestation, building highway bridges for animals or improving the quality of the soil, among other actions. Our goal here will be to choose the right spots for these investments, subject to a budget. Note how the technical term effective resistance is often substituted by resistance distance in the biology literature. Also, in a non-reactive circuit with no alternating current, effective resistance and equivalent resistance are equal. For consistency we will always refer to effective resistance.

A similar problem was addressed by Ghosh et al. [98], but their problem was continuous. The continuous variant does not apply in our habitat conservation planning, as it is not possible to invest an infinitesimal fraction of budget in one location. In the real world, reforestation areas are typically well defined discrete interventions.

7.2 Problem Formulation and Required Notions

7.2.1 Preliminaries

In this chapter we will denote $r_{xy}$ the value of a resistor connected between points $x$ and $y$. The conductance is defined as the inverse of the resistance, denoted $g_{xy} = r_{xy}^{-1}$. We will use this often to simplify the formulas. The effective resistance corresponds to the real resistance between two nodes in the circuit. During the rest of this chapter we will note $R_{xy}$ the effective resistance between $x$ and $y$.

In an electric circuit, the effective resistance can be computed by solving a system of linear equations [7, 73]. Such linear system can be obtained by doing nodal analysis or mesh analysis. A more systematic way of obtaining the same system of linear equations is by using the nodal admittance matrix [15], also known by graph theoreticians as the Laplacian matrix. Let $\text{adj}(n)$ be the set of nodes adjacent to a node $n$ in an electric circuit, and $g_{ij}$ the conductance of branch $(i, j)$. For an electric circuit of $n$ nodes, the Laplacian is
an $n \times n$ matrix defined as follows:

$$L_{i,j} = \begin{cases} -g_{ij} & \text{if } (i, j) \text{ is a branch of the circuit} \\ \sum_{k \in \text{adj}(i)} g_{ik} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

For any two nodes $s$ and $t$ in an electric circuit of Laplacian $L$, we can calculate the effective resistance $R_{st}$ between those nodes by simply solving the linear system given by $L_t v = f$ where $L_t$ is $L$ without the $t^{th}$ row and column, and $f$ is a vector with all its elements equal to 0 except for the $s^{th}$, which is equal to 1. The effective resistance we are after will be found in $v_s$.

This is strictly equivalent to nodal analysis. In classic nodal analysis, we would connect a source of 1A to $s$ and connect $t$ to the ground. Then we obtain an equation for each node of the form $\sum_{y \in \text{adj}(x)} i_{xy} = 0$ (flow conservation of the current), except for the node $s$ where the flow of current is 1 since it’s connected to the source, and the node $t$ where the flow is -1 for being connected to the ground. Ohm’s law gives us that, for any branch, $i_{xy} = g_{xy}(v_x - v_y)$ where $v_x$ and $v_y$ are the voltages at nodes $x$ and $y$ and $g_{xy}$ is the conductance of the resistor between $x$ and $y$. Then any node can be selected as reference or datum node, setting its voltage to 0. We select $t$ as the datum node. Then by substituting with Ohm’s law and the value of the datum node in the nodal analysis equations we obtain the same system as with the Laplacian. Solving the system yields the voltage of node $s$, among others. Since the source of current was 1A, Ohm’s law gives us that $R_{st} = (v_s - v_t) / 1 = v_s$. It is therefore only necessary to look at the $s^{th}$ component of the solution vector to obtain the effective resistance. This is exactly the method implemented in Circuitscape [196], software used nowadays for landscape planning by experts.

Note how the definition of effective resistance only allows the computation of one effective resistance at a time: the current source needs to be connected to exactly one node, $s$, and only one node, $t$, must be connected to the ground. Therefore, the effective resistance between three or more nodes is undefined. Instead, we will consider the total effective resistance for three nodes $a$, $b$ and $c$ to be $R_{ab} + R_{ac} + R_{bc}$. These have to be
computed by solving three different equation systems. In general, for \( f \) nodes, we will need \( f(f - 1)/2 \) linear systems.

### 7.2.2 Problem Formulation

We define the problem of finding the Minimum Effective Resistance in a Circuit with Binary Investments (MERCBI) as follows.

We are given an electric network \( G^\varnothing = (N, E, g^\varnothing) \) where \( N \) are the nodes, \( E \) are the edges (known as branches in electric circuit theory) and \( g^\varnothing : E \mapsto \mathbb{R}^+ \) is a function giving the value of the original conductances that are placed on each edge. We are also given a function \( g^E : E \mapsto \mathbb{R}^+ \) of new improved conductances for each edge. These functions are such that \( \forall e \in E, g^\varnothing(e) \leq g^E(e) \). Lastly, we are given a set of pairs of focal nodes (i.e. core habitats) \( P = \{ \langle s_1, t_1 \rangle, ..., \langle s_{|F|}, t_{|F|} \rangle \} \), a budget \( B \), and a cost function \( c : E \mapsto \mathbb{R}^+ \).

The pairs in \( P \) are the pairs of core habitats between which we want to improve the resistance. It typically contains all the possible pairs of habitats in the studied landscape, but not necessarily.

We note \( G^A \) for \( A \subseteq E \) the network given by \( G^A = (N, E, g^A) \) where \( g^A \) is a function defined by \( g^\varnothing \) for all \( e \notin A \) and \( g^E \) for all \( e \in A \). Similarly, \( R^A_{xy} \) will refer to the effective resistance between \( x \) and \( y \) in \( G^A \).

Our goal is to find a set \( S \subseteq E \) such that we minimize \( R^S = \sum_{i=1}^{|F|} R^S_{s_i, t_i} \) while keeping the sum of the investment costs below \( B \). We say that the edges in \( S \) are investments. The edges in \( E \setminus S \) are wild edges. Note that it may be the case that \( g^S \) forms an open circuit (i.e. there is no way for current to go from some \( s_i \) to some \( t_i \) due to 0-conductance branches). In such case, \( R^S = \infty \) by definition, which can happen if there is not enough budget to ensure connectivity.

Formally, the model is translated into a Mixed Integer Program with the following additional variables:

- \( b_e \) is a binary decision on whether the edge \( e \) of the circuit is part of the selected solution \( S \).
- \( v^{(s,t)}_x \) is the voltage at node \( x \) when \( s \) is connected to a 1A source and \( t \) to the ground.
• \( p^{(s,t)}_{(a,b),c} \) is an intermediate variable for the product \( g^S((a,b)) * v^{(s,t)}_c \).

The MIP formulation is:

\[
\text{Minimize } \sum_{i=1}^{|F|} v^{(s_i,t_i)}_{s_i} \text{ such that } \\
\sum_{e=0}^{\{E\}} b_e c(e) \leq B \tag{7.1}
\]

\[
\forall (s_i, t_i) \in P, \forall x \in N \setminus \{s_i, t_i\}, \sum_{y \in \text{adj}(s_i)} p^{(s_i,t_i)}_{(x,y),x} - \sum_{y \in \text{adj}(s_i) \setminus \{t_i\}} p^{(s_i,t_i)}_{(x,y),y} = 0 \tag{7.2}
\]

\[
\forall (s_i, t_i) \in P, \sum_{y \in \text{adj}(s_i)} p^{(s_i,t_i)}_{(s_i,y),s_i} - \sum_{y \in \text{adj}(s_i) \setminus \{t_i\}} p^{(s_i,t_i)}_{(s_i,y),y} = 1 \tag{7.3}
\]

\[
\forall (s_i, t_i) \in P, \forall e = (x, y) \in E, p^{(s_i,t_i)}_{(e,x)} = b_e g^E(e) v^{(s_i,t_i)}_x + (1 - b_e) g^O(e) v^{(s_i,t_i)}_y \tag{7.4}
\]

\[
\forall (s_i, t_i) \in P, \forall e = (x, y) \in E, p^{(s_i,t_i)}_{(e,y)} = b_e g^E(e) v^{(s_i,t_i)}_y + (1 - b_e) g^O(e) v^{(s_i,t_i)}_x \tag{7.5}
\]

Equation 7.1 is our objective: minimize the sum of effective resistances between focal nodes. Equation 7.2 is our budget constraint. Equation 7.3 constrains the flow of current to be 0 in all nodes, except the nodes directly connected to the source. Equation 7.4 indicates that the flow at the source nodes has to be 1. Note how the equations at the sinks with flow -1 have been removed as they are linearly dependant from the others. Equations 7.5 and 7.6 choose the value of the \( p \) variables based on the values of the Booleans.

In Equations 7.3 and 7.4, the first sum correspond to the diagonal terms in the Laplacian matrix, whereas the second sum is the non-diagonal terms of the Laplacian matrix.

Furthermore, Equations 7.3, 7.4, 7.5 and 7.6 are repeated for each pair of focal nodes in \( P \). Therefore, the equations obtained by nodal analysis are repeated \( C = |P| \) times, one per pair in \( P \). Nonetheless, these \( C \) systems are not equivalent, as the source of 1A is connected at different nodes, and the voltages are therefore different in each system. That is, it is not necessarily the case that \( v^{(s_i,t_i)}_x = v^{(s_i,t_i)}_y, \forall i, j, x, i \neq j \). We say that our model contains \( C \) circuits.
7.2.3 Complexity

We now prove that the MERCBI problem is NP-hard by reduction from the Steiner Tree Problem on graphs (STP). We remind the reader of the definition of the Steiner Tree Problem here (also available in Definition 3.1, page 37). As formulated by Karp [115] in his paper where he proved its NP-completeness, the STP is: Given a graph $G = (V, E)$, a set $\tau \subseteq N$, weighting function $w$ on the edges and positive integer $K$, is there a subtree of $G$ weight no greater than $K$ containing the set of nodes in $\tau$?

We apply the following reduction:

- The electric circuit is the graph $G$.

- The cost function is $w$.

- The original resistance of edges is infinite (i.e. $g^O(e) = 0, \forall e \in E$).

- The resistance upon investment of all edges is 1 (i.e. $g^E(e) = 1, \forall e \in E$).

- The budget is $K$.

- The set of pairs of focal nodes $P$ is the set of all pairs of distinct nodes in $\tau$, which can be built in $O(|\tau|^2)$.

Assume we have an algorithm to solve the MERCBI problem that gives a solution $S$. Clearly, by investing in the selected edges $S$ we will obtain a resistance $R^S \in \mathbb{R}$ iff there is enough budget $K$, and $R^S = \infty$ otherwise:

1. If the resistance is $\infty$, then there is no Steiner Tree of cost less than $K$, since we could not connect focal nodes with $1\Omega$ resistors.

2. If the resistance is 0 then we can obtain a graph $G^*$ by restricting $G^S$ to the edges that have been invested. Clearly, $G^*$ is of weight no more than $K$ and a subgraph of $G$ that connects all pairs in $P$. Because $P$ is the set of all pairs of nodes from $\tau$, this means that all nodes in $R$ are connected pairwise in $G^*$. Although $G^*$ may not be a tree, we can extract a tree $T$ from it by breaking any cycle $G^*$ contains while maintaining its connectivity. The tree $T$ is a Steiner tree of cost at most $K$. 
We have therefore shown that if we have a solver for the MERCBI problem, we have one for the STP. Thus, the MERCBI problem is NP-hard.

7.3 Solving the Minimum Effective Resistance Problem

7.3.1 Greedy Algorithm Approach

We first devise a greedy algorithm for the problem based on the following observation: increasing the conductance of an edge with low current has little impact in the overall effective resistance. This intuition is easily justifiable: if the current of an edge corresponds to the net number of times a random walker uses that edge, the lower that number, the less impact improving that area would have on the total commute time. Clearly edges near focal nodes or near low resistance areas tend to concentrate more current and be used by more random walkers. Thus increasing the conductance of those edges will likely have a stronger impact in lowering the effective resistance. Our greedy algorithm is presented in Algorithm 7.1.

Algorithm 7.1 Greedy algorithm

```
1: procedure GREEDY(\(G = (N, E), F, g^\emptyset, g^E, c, B\))
2: \(lp \leftarrow \text{BUILDMODEL}(G, F, g^\emptyset, g^E, c, B)\) \(\triangleright \) Build the model from Section 7.2.2
3: \(lp\text{-SETBINARIES}(E, \text{false})\) \(\triangleright \) Sets all the binaries to false
4: \(lp\text{-SOLVE()}\) \(\triangleright \) Solve the LP, with all binary variables fixed
5: \(i[e] \leftarrow 0, \forall e \in E\) \(\triangleright \) Array of currents
6: for all \((s_i, t_i) \in P\) do
7: for all \((x, y) \in E\) do
8: \(i[e] \leftarrow i[e] + |g^\emptyset(e) \ast (v^{(s_i,t_i)}_x - v^{(s_i,t_i)}_y)|\) \(\triangleright \) Ohm’s law
9: \(se \leftarrow \text{REVERSE(SORTBY}(E, i))\) \(\triangleright \) Array of edges sorted by decreasing current
10: \(S \leftarrow \emptyset\)
11: \(b \leftarrow 0\)
12: \(j \leftarrow 0\)
13: while \(b < B \land j < |E|\) do
14: if \(g^\emptyset(se[j]) < g^E(se[j]) \land b + c(se[j]) \leq B\) then
15: \(S \leftarrow S \cup se[j]\)
16: \(b \leftarrow b + c(se[j])\) \(\triangleright \) Select edges with high current.
17: \(j \leftarrow j + 1\)

return \(S\)
```

The algorithm solves the conductance LP once to find the voltages at each node, then
calculates the currents in each edge (with no investments being made), and greedily selects the edges with most current to invest in, that fit within the budget.

As we will see in the experiments, this algorithm performs surprisingly well despite not providing optimal solutions. In next sections we will try to obtain better solutions than the ones provided by this algorithm.

### 7.3.2 Mixed Integer Programming Approach

The MIP model expressed in Section 7.2.2 is a direct mapping of the nodal analysis performed in the electric circuit into a MIP model. In our implementation Equations 7.5 and 7.6 were implemented using four *indicator constraints* which are supported by the IBM ILOG CPLEX 12.4 solver as follows:

\[
\forall (s_i, t_i) \in P, \forall e = (x, y) \in E,
\begin{align*}
  b_e \implies p^{(s_i, t_i)}_{c_l} &= g^E(e)v_x^{(s_i, t_i)} \\
  -b_e \implies p^{(s_i, t_i)}_{c_l} &= g^\varnothing(e)v_x^{(s_i, t_i)}
\end{align*}
\]

\[
\begin{align*}
  b_e \implies p^{(s_i, t_i)}_{c_l} &= g^E(e)v_y^{(s_i, t_i)} \\
  -b_e \implies p^{(s_i, t_i)}_{c_l} &= g^\varnothing(e)v_y^{(s_i, t_i)}
\end{align*}
\]

#### 7.3.2.1 Finding Bounds for the Variables

To help CPLEX tackle the problem, we need to compute bounds on the variables. We apply basic circuit analysis to find these bounds. To do so, we need an initial assignment for the binary variables: this could be assigning all to *false*, or to the value of some initial solution that respects the budget constraint (either a random solution, or obtained with Algorithm 7.1). Without loss of generality, let us assume that all the binaries are set to *false*, thus \( g^\varnothing \) gives the conductance for all edges.

When we are computing the effective resistance between nodes \( s \) and \( t \) of a circuit, we connect a current source between \( s \) and \( t \) as in Figure 7.3. This can be converted into an equivalent circuit as seen in the center of the figure. The value of \( R_{st} \) is actually the value of the effective resistance when all our binary variables are fixed. To obtain bounds for the voltages at the nodes marked ●, we observe that the circuit in the center is a Norton circuit, thus we can apply Thévenin’s theorem [37] to it to obtain an equivalent
Figure 7.3: Conversion of electrical circuit to compute upper bounds, and application of Thévenin’s theorem.

Circuit with a voltage source instead of a current source (circuit on the right). Because the current in that Norton’s circuit is 1A, the voltage of Thévenin’s equivalent source will be $V_{Th} = I_{No} R_{st} = R_{st}$. Thus, the upper bound of $v_s$ is $R_{st}$. Since $t$ is connected to the ground, $v_t = 0V$. All the voltage at points between $s$ and $t$ are necessarily bounded by the voltages at these points, because voltage can only drop. Therefore, the bounds for all voltage variables are: $\forall \langle s_i, t_i \rangle \in P, \forall x \in N, \ 0 \leq v^{(s_i,t_i)}_x \leq R_{s_it_i}^\emptyset$. From these bounds it is easy to derive bounds for the $p$ variables: $\forall \langle s_i, t_i \rangle \in P, \ \forall e = (x, y) \in E, \forall z \in \{x, y\}, \ g^D(e) v^{(s_i,t_i)}_z \leq p^{(s_i,t_i)}_{e,z} \leq g^E(e) v^{(s_i,t_i)}_z$.

7.3.2.2 Motivation for Local Search Approaches

We attempted to solve the MIP model using IBM ILOG CPLEX 12.4. The major challenge we discovered while running our experiments was that the linear relaxation of the model performed poorly. The gap between the LP relaxation and the best integer solution found in less than 5 hours was usually bigger than 40% even for small $10 \times 10$ grid-like networks, as reported by CPLEX. As a small example, consider a $3 \times 3$ grid with $\forall e, \ g^D(e) = 0.01$ and $g^E(e) = 1$, a budget of 2 and 2 core habitats. The proven optimal solution gives a resistance of 49.15, whereas the linear relaxation achieves a resistance as low as 0.50. With such a big gap, the linear relaxation cannot help pruning branches. We observed these gaps both when using indicator constraints and without them (using an alternative encoding of the product of continuous and binary variables as linear inequalities). The main reason for such a bad linear relaxation is that the only true constraint is the budget. It is actually an almost pure optimization problem, as all the other constraints...
only exist to compute the effective resistance.

Since the linear relaxation is so weak, we decided to try a Local Search approach.

### 7.3.3 Local Search Approach

Although solving the MIP problem is time consuming, once all binary variables are fixed, the problem becomes a Linear Program, which can be solved efficiently. The idea of local search is to repeatedly destroy part of the solution and build a new one from the remaining solution. This was explained in Section 2.3.2 of Chapter 2.

Our Local Search (LS) moves are based on two stages: first choose areas where we invested but we are no longer interested in investing, then choose new areas where we would like to invest. In this section we will discuss different techniques we use to make moves in the LS.

#### 7.3.3.1 Destroying Investments

We start from a solution $s$ where we have selected a set $S$ of edges to be investments. We call $d$ the destruction rate of investments.

To destroy investments, we select a subset $S_d$ of $S$ such that $|S_d| = d$. We will convert these $d$ investments into wild edges, thus freeing part of the budget to be used elsewhere. We implemented 3 ways of selecting those $d$ edges:

1. **INVRAND**: Choose $d$ invested edges randomly.
2. **INVLC**: Choose the $d$ invested edges with lowest current in $s$.
3. **INVLCP**: Choose $d$ invested edges based on a probability distribution that favors low current edges in $s$ being selected.

The way to compute the current of an edge is by simply using Ohm’s law, as in line 8 of Algorithm 7.1. Because we are solving $C$ linear systems for $C$ circuits at once, the current across an edge will be the sum of the currents of that edge in the $C$ different circuits.
As for the probability distribution, we chose that the total current across an edge is proportional to the inverse of its probability. Therefore edges with low current will have a higher tendency to be chosen than edges with high current: \( \forall e = (x, y) \in E, \ Pr(e) \propto (\sum_{(s_i, t_i) \in P} |g^S(e) \ast (v^x_{s_i,t_i} - v^y_{s_i,t_i})|)^{-1}. \)

### 7.3.3.2 Making New Investments

Let \( s' \) be the solution \( s \) with investments \( S \setminus S_d \), that is all investments of \( s \) except those selected for destruction. After destroying investments \( S_d \), we have recovered a part \( b = \sum_{e \in S_d} c(e) \) of our budget back, so we are ready to choose new places to invest. To do so, we have 4 different strategies to choose new edges to invest in:

1. **WilRand**: Choose a set \( W \) of wild edges randomly.

2. **WilBFS**: Choose a set \( W \) of wild edges by doing a Breath-First Search (BFS) in the graph. The origin of the BFS is chosen according to a probability distribution that favors high current nodes. This ensures that the chosen edges are close to each other.

3. **WilHC**: Choose a set \( W \) of wild edges that have the highest current in \( s \).

4. **WilHCP**: Choose a set \( W \) of wild edges based on a probability distribution that favors high-current wild edges.

For all these strategies, we ensure that \( \sum_{e \in W} c(e) = b - \epsilon \) and we select those edges greedily to get the smallest slack \( \epsilon \).

The second strategy, WilBFS, requires computing the current at a node. We define it as \( c_x = \sum_{(s_i, t_i) \in P} \frac{1}{2} \sum_{y \in \text{adj}(x)} |g^S((x, y)) \ast (v^x_{s_i,t_i} - v^y_{s_i,t_i})|. \) That is, the sum of the accumulated current of the surrounding edges of the node.

As for the probability distributions used by both WilBFS and WilHCP, we could not use a distribution as simple as for InvLCP. The reason is that, even though the probability of choosing a particular node (or edge) of high current is much higher than the probability of choosing particular node (or edge) of low current, because there are many more nodes and edges with low current, the probability of choosing a low current node
or edge would be much higher.
To overcome this, we will only allow the choice of nodes or edges that have a current higher than 10% of the current of the highest node or edge, plus a certain number $\gamma$ of low current elements (i.e. below that 10%). We choose $\gamma$ to be equal to the number of nodes or edges above 10% of the maximum current. The probability of choosing an element with a current below 10% of the maximum is now the same as the probability of choosing an element with a current above that threshold.

7.3.3.3 Initial solution

Local Search needs to start with an initial solution. Our approach was to take the output of the greedy algorithm as the initial solution for the LS.

7.3.3.4 Using Simulated Annealing

We used simulated annealing, as described in 2.3.2.2 of Chapter 2.

Even using simulated annealing, it is easy to see how the combined usage of INVLC and WILHC can perform the exact same destruction over and over once it hits a local minimum. To avoid that, when we are using these two destruction techniques together and we don’t accept a solution, we switch to WILHCP for one iteration, which will put us in another neighborhood to explore (like a restart).

7.4 Experimental Evaluation and Results

All the following experiments use IBM ILOG CPLEX 12.4 on a Linux 3.16 Intel® Core™ i7-4770 CPU @ 3.40GHz, 15.6GB of RAM.

7.4.1 Instances

We could not obtain real-world instances. Nonetheless we generated artificial landscapes while trying to keep them realistic. In the literature where the IBR model is used on
empirical data [10, 11], high resistance areas have an approximate resistance between 10 and 100 times bigger than areas with low resistance.

Our instances were built in the form of grids with 4 neighbor nodes (except at the border of the landscape where nodes have 2 or 3 neighbors). The values of conductances (i.e. inverse of resistances) are chosen following a probabilistic beta-distribution with parameters $\alpha = 5$ and $\beta = 80$.

Using this distribution, most conductances are in the desired range, still allowing the unlikely possibility of having some high conductances scattered on the map. Then, for each instance of size $n \times n$ we select $n$ nodes, and their neighbors, to be oases. All the edges coming out of oasis nodes have a conductance of 1, that is the same as if we had invested. This is to account for areas that are not considered core habitats but are friendly to the animals: wild trees, fresh water lakes, etc. Figure 7.2 shows, on the right, an example of such map in a $50 \times 50$ grid (the scale being in $\Omega^{-1}$).

Regarding the budget cost, we created instances with homogeneous costs (labelled HOMO) where each edge costs 1. The reason for homogeneous costs is that the main application for our problem is in wild land and specifically on National Parks, where rehabilitation costs are typically fairly uniform across a large area. Typically these kind of projects are used to recover land that has been damaged by bush fires, flooding or landslides or that has been fragmented by urbanization that cannot be torn down or purchased. We also constructed instances where the cost of an investments is a uniform random number between 0.1 and 10 (labeled RAND).

Furthermore, the locations of core habitats are selected from a uniform distribution. For each size of grid and number of habitats, we generated 20 different instances, totalling
500 instances. Values reported are arithmetic means.

### 7.4.2 Improvement Over the Initial Landscape

To measure the quality of a solution, we look at the ratio between the effective resistance in the solution and the original resistance when no investment is made. Table 7.1 shows the average of these ratios. All the results reported on that table had a budget equal to twice the side of a grid (e.g., for a grid of 50×50, we have a budget of 100). The ratio of the budget to the number of edges where we could invest is shown in the third column. The destruction rate here is fixed to 10 for instances of size 20×20 and 25×25, and 25 for bigger instances. The number of iterations the LS does for these tests is 200. We do not show the results obtained by random selections for destruction as they perform worse than the ones shown here. The initial temperature for simulated annealing is chosen such that a solution 50% worse than the initial solution is accepted with a probability of 10%. The cooling rate is 0.98 for all experiments.

![Table 7.1](image)

Table 7.1: Averages of the ratios of updated resistance over original resistance. Budgets are always twice the length of the side of the grid.

Our first observation is that even with a very low budget, proportionally to the mass
of land, we manage to have significant drops of resistance. We infer that our approach is placing the investments in the right places. The greedy algorithm performs very well too. The Local Search is able to reduce the resistance of the greedy algorithm by up to 11%.

We can observe how in general the INVLIC approach to destroy bad investments outperforms INVLCP. Regarding the selection of new investment locations it seems there is no clear winner, although WILBFS and WILHC tend to choose investments in places nearby or directly connected, and that seems to give them the advantage.

![Heatmap showing the places of investment and the best investments aligned to connect low resistance areas.](image)

Figure 7.5: Example of a solution obtained with INVLIC+WILHC for the same instance as shown in Figure 7.2.

As an example, Figure 7.5 shows the solution found to the instance of Figure 7.2 On the left, the heatmap shows the places of investment (marked lines) which redirect most of the animals, as these become more suitable for them. On the right, we see how the best investments tend to be aligned to connect low resistance areas that already existed. Notice how they do not force a complete corridor: some investments may not be used to maintain connectivity and are instead moved to areas where they might be more useful.

7.4.3 Optimality Gap

In this section we look at the difference between the solution obtained and the proven optimum for some instances. We could not obtain the optimal value for instances of the size given in the previous subsection. For this reason, we ran this experiment on $10 \times 10$
grids. These took, in some cases, up to 8 hours to prove optimality, even given the small size. It is clearly impractical to use the pure MIP formulation.

<table>
<thead>
<tr>
<th>Instance Name</th>
<th>(10\times 10) Instances</th>
<th>(50\times 50) Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>002</td>
<td>1.01</td>
<td>1.03</td>
</tr>
<tr>
<td>003</td>
<td>1.10</td>
<td>1.09</td>
</tr>
<tr>
<td>004</td>
<td>1.01</td>
<td>1.05</td>
</tr>
<tr>
<td>005</td>
<td>1.17</td>
<td>1.08</td>
</tr>
<tr>
<td>006</td>
<td>1.34</td>
<td>1.01</td>
</tr>
<tr>
<td>007</td>
<td>1.00</td>
<td>1.02</td>
</tr>
<tr>
<td>008</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>009</td>
<td>1.10</td>
<td>1.15</td>
</tr>
<tr>
<td>010</td>
<td>1.04</td>
<td>1.07</td>
</tr>
<tr>
<td>011</td>
<td>1.01</td>
<td>1.03</td>
</tr>
<tr>
<td>012</td>
<td>1.00</td>
<td>1.02</td>
</tr>
<tr>
<td>013</td>
<td>1.08</td>
<td>1.10</td>
</tr>
<tr>
<td>014</td>
<td>1.05</td>
<td>1.06</td>
</tr>
<tr>
<td>015</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>016</td>
<td>1.04</td>
<td>1.17</td>
</tr>
<tr>
<td>017</td>
<td>1.06</td>
<td>1.18</td>
</tr>
<tr>
<td>018</td>
<td>1.11</td>
<td>1.05</td>
</tr>
<tr>
<td>019</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>020</td>
<td>1.06</td>
<td>1.11</td>
</tr>
</tbody>
</table>

Table 7.2: Average (across 20 instances) of ratio between LS and proven optimum for 10\(\times\)10 instances (200 iterations). Exact optimal solutions are marked with *.

As it can be seen in Table 7.2, the LS approach does not deviate much from the optimum. Once again we see no clear domination between the different techniques applied, although \(\text{INVLC}+\text{WILBFS}\) seems slightly worse than others. The LS even managed to find optimal solution of two instances.

### 7.4.4 Number of Iterations

We now study the effect of the number of iterations. We are interested to know how quickly we get improvements in our solutions. To evaluate this, we looked at the 50\(\times\)50 instances with homogeneous investment cost and a budget of 100 (same instances as in Table 7.1). We compute the ratio between the best solution found so far at the \(x^{th}\) iteration to the non-investment resistance. We average this ratio across the 60 HOMO 50 \(\times\) 50 instances (2, 3 and 4 habitats) for each iteration. The results are plotted in Figure 7.6. Based on the results seen in Table 7.1 and this plot, we can see that \(\text{INVLCP}+\text{WILBFS}\)
and INVLCP+W1HCP are not only the ones that perform the worse, but we also only find better solutions very rarely. The INVL+H1HC shows the quickest drop, but then slows down. Since the drop happens in the first 20 iterations, the temperature is still high, thus suggesting that it does not get trapped in a local minimum, instead it is most likely moving towards the global optimum (as we also saw on the optimality gap results). All other techniques have a slower slope.

![Graph showing the ratio between the best solution found and the value of the initial resistance over the number of iterations for 6 destruction strategies.](image)

Figure 7.6: Ratio between the best solution found and the value of the initial resistance over the number of iterations for 6 destruction strategies (50×50 HOMO grids, 2 to 4 habitats).

Regarding the time spent to perform these 200 iterations: no instance needed more than 5 minutes in our machine. Therefore this number of iterations is also suitable for use by habitat planners during their work.

### 7.4.5 Destruction Rate

In this subsection we will look at the effects of the choice of destruction rate. Recall, the destruction rate corresponds to how many edges are converted from investments into wild edges for the next iteration of the LS. We ran these experiments on the 50×50 instances for 2, 3 and 4 habitats with a budget of 100 and homogeneous costs. Table 7.3 reports the average ratio between our solution and the original resistance across the 60 instances depending on the destruction rate. The number of iterations is once again 200.

As we can see in Table 7.3, the choice of destruction rate only marginally affects the quality of the solution. This shows that our local search approach is robust and no matter what parameter is used (within a reasonable range), the algorithm is likely to give
7.5 Concluding Remarks

In this chapter we have discussed a new problem, the MERCBI problem, in habitat conservation. This is the only problem in this thesis that is not tackled by Constraint Programming. The reason for this is that it is an extremely under-constrained problem. It is trivial to find a solution, but it is hard to find a good one. This is typically the kind of problems where CP techniques do not excel.

For that reason, we were motivated to tackle the problem using MIP. Unfortunately, MIP failed at solving this problem. The reason is somewhat similar to the reason why CP is not suitable: the problem is almost a pure optimization problem. The only constraint is on the budget, and the other constraints are used to compute the value of the objective function. This leads to a very poor linear relaxation and MIP cannot solve the problem.

This prompted us to tackle the problem using Local Search. We have provided a series of heuristics used to implement it. The results show that this approach yields good quality solutions.

The contributions of this chapter are:

- The introduction of the MERCBI problem and its formal definition.
- Proof of NP-completeness of the MERCBI problem.
- A Local Search algorithm for the MERCBI problem.

Table 7.3: Ratio of solution found over initial landscape resistance for 50 × 50 grids with different destruction rates, over 200 iterations.

<table>
<thead>
<tr>
<th>Destruction Rate</th>
<th>INVLCS</th>
<th>INVLHC</th>
<th>INVLHCP</th>
<th>INVLBS</th>
<th>INVLHC</th>
<th>INVLHCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.40</td>
<td>0.38</td>
<td>0.40</td>
<td>0.47</td>
<td>0.39</td>
<td>0.45</td>
</tr>
<tr>
<td>10</td>
<td>0.40</td>
<td>0.38</td>
<td>0.40</td>
<td>0.48</td>
<td>0.39</td>
<td>0.46</td>
</tr>
<tr>
<td>15</td>
<td>0.40</td>
<td>0.39</td>
<td>0.39</td>
<td>0.48</td>
<td>0.40</td>
<td>0.46</td>
</tr>
<tr>
<td>20</td>
<td>0.40</td>
<td>0.39</td>
<td>0.40</td>
<td>0.48</td>
<td>0.41</td>
<td>0.47</td>
</tr>
<tr>
<td>25</td>
<td>0.41</td>
<td>0.39</td>
<td>0.41</td>
<td>0.48</td>
<td>0.42</td>
<td>0.47</td>
</tr>
<tr>
<td>30</td>
<td>0.42</td>
<td>0.40</td>
<td>0.41</td>
<td>0.48</td>
<td>0.43</td>
<td>0.47</td>
</tr>
<tr>
<td>35</td>
<td>0.43</td>
<td>0.40</td>
<td>0.42</td>
<td>0.48</td>
<td>0.44</td>
<td>0.47</td>
</tr>
</tbody>
</table>
• An extensive series of experiments showing that this problem can be solved satisfactorily using LS, and that the LS can be robust.
Chapter 8

Graphs for Constraint Programming: MDDs and d-DNNFs

In this last chapter of this thesis, we no longer look at problems involving graphs, but rather graphs within a CP solver. Graphs are an extremely useful data structure, and as such, they can be used to accelerate CP solvers. In this chapter we look at the problem of automatically compiling part of a problem into a graph-like structure, to accelerate the solving process.

In CP, it is often the case that reformulating a subproblem that involves only a few tightly constrained variables as a table constraint can improve solving time. Nevertheless, enumerating all the solutions of a subproblem into a table can be costly in time and space. In this chapter we propose using Multivalued Decision Diagrams (MDDs) and formulas in Deterministic Decomposable Negation Normal Form (d-DNNFs) rather than tables to compute and store all solutions of a subproblem. This, in turn, can be used to enhance the solver thanks to stronger propagation via specific propagators for these structures. We show how to precompile part of a problem into both these structures, which can then be injected back in the model by substituting the constraints it encodes, or simply adding it as a redundant constraint. Furthermore, in the case of MDDs, they can also be used to create edge-valued MDDs for optimization problems with an appropriate form. From our experiments we conclude that all three techniques are valuable in their own right, and we show when each one should be chosen over the others.

8.1 Compiling Constraint Programming Subproblems into Tractable Structures

Even when using modern high level constraint programming modeling approaches such as IBM OPL [213], MiniZinc or Essence [90], building a good model of a discrete optimization problem is challenging. Although a modeler may be able to easily write a model that correctly captures the problem at hand, the efficiency of this model may be far from
the best possible. Hence model improvement methods are valuable. Approaches such as detecting symmetries [146] or missing global constraints [135] are highly advantageous, but for many problems these refinements may not be applicable. Presolving [42, 81, 134], where the model is improved during compilation, is an effective approach to model improvement.

Another model improvement method is to replace a subproblem by a table constraint representing the solutions of this subproblem [62, 63]. We say the subproblem is tablified. This can be highly effective both in improving naive models and in improving models where some subproblem over a few variables is highly constrained.

Although tablifying a model has a lot of value, and can be the best way of presolving part of a problem, it comes with limitations. An obvious one is that the table can be huge, simply because there are too many solutions to the part of the problem being presolved. Furthermore, tables can contain a lot of repeated “suffixes”. For example, many rows of a 100-column table could contain identical values for the last 90 columns, and only the 10 first columns differ between those rows. In that case, finding a data structure that can compress all those rows into an object with only one common “suffix” for all the combinations of the first 10 columns could save up to 90% of memory. The problem is twofold. First, there is a clear redundant use of memory, and second, a presolver would actually need to generate all those almost-identical rows, which is slower than a data structure where rows could point to “commonly shared sub-rows”.

This prompts the question of what better data structure can be used. In the Constraint Programming community, Multivalued Decision Diagrams (MDD) are well known. But several works in model counting for SAT use variations of Negation Normal Forms (NNF) to store solutions and then perform model counting using this structure.

In this chapter we investigate alternatives to using Tables as solution stores for preprocessing parts of Constraint Programming models. We choose to compare with Multivalued Decision Diagrams (MDD) and Deterministic Decomposable Negation Normal Forms (d-DNNFs). We say that we mddfy or d-dnnffy part of a model. The goal of this compilation is to produce models with constraints that lead to stronger propagation without the user needing to construct this complex structures manually.
To achieve this, a key component will be subproblem-equivalence detection, or caching. We dedicate Section 8.3 to this in detail. In short, this allows us to detect that some subproblems are equivalent and reduce the amount of work to be done to collect the solutions in them. This is exactly what we mentioned earlier when tables may have common suffixes in multiple rows.

8.2 State of the Art

The most closely related work to this chapter is the work on tablifying parts of models by Dekker et al. [62, 63]. In this work, the authors introduce annotations to the MiniZinc language to allow the user to convert a predicate into a table constraint. We will mainly compare with this work, because our framework achieves the same results, but using alternative data structures to store the contents of those tables. In that work, a predicate is solved (all its solutions are determined) and converted into a table constraint that is then injected into the model at each call of the predicate. Their results clearly show how compiling predicates can yield better solving times for the model. While for the examples they examine, building the tables does not require much time, we will show examples where tablifying blows up, and hence the use of more compact representations can be crucial.

8.2.1 MDD Compilations

Pérez et al. [163, 161] presented efficient algorithms for MDD-operations with the goal of reducing the time to build MDDs. They show that using MDDs built with these operations can be competitive with dedicated algorithms. This motivates the idea of modeling complex constraints, as we will do in this chapter, with MDDs (or d-DNNFs) rather than having to create a specific propagator to achieve good propagation for some constraint. Additionally, the aforementioned authors summarize some approaches for building MDDs. These include building MDDs from a table constraint (originally presented by Cheng and Yap [44]), from an automaton (i.e. a regular constraint), a pattern or a trie. None of these approaches are what we are after because we want something as
modular as a table creation, but more robust in memory consumption (as we will see in the experiments, it not particularly hard to make the table approach explode in size). These papers by Pérez et al., although related, do not solve the problem we are tackling: we want to build the MDD as a preprocessing from a set of basic constraints that any modeler could write in a modeling language like MiniZinc, not from a specific form of constraint like the regular.

The work presented by Koriche et al. [125] is much more closely related to our own work here. Indeed, their algorithm for compiling constraint networks into their custom Multivalued Decomposable Decision Graphs (MDDG) language is quite similar to the algorithms we will see in this chapter. There are some important differences though. Their CN2MDDG compiler does not generalize to all global constraints. As they mention the paper, they only support three globals (alldifferent, linear constraints and element). Arguably this limitation can be solved by extending their compiler to other globals but this is a task that needs to be done every time a new global is defined and, as we have seen in recent times, the number of global constraints defined in the literature is constantly growing. In our case, we only need to update the caching capabilities for new globals. Another difference with that work, as we will see later in Section 8.4, is the way variables fixed by propagation are treated compared to their algorithm. Our approach for compiling parts of problems into MDDs is more flexible. We provide a larger number of experiments and comparisons with more competitive approaches in this chapter than they did in their paper (like tables and d-DNNFs, rather than just the original models).

Cheng and Yap [43] introduced a representation for general n-ary constraints called Constrained Decision Diagrams (CDDs). As they mention in their paper, their approach aims for compact representations (even more than canonical MDDs). The use they make of their data structure is to store solutions that can then be presented to a user to filter manually. A good example of this would be to solve a problem for a client, show the solutions and then the client can filter them out based on some criteria that may not have been present in the original model. Their compilation into CDDs is not intended as a preprocessing that can be used to enhance search like our work or that of Dekker et al. [63]. Although it could be used in this way, it would require a fair amount of work
since we would need to create a propagator for this CDD representation in our solver. Sadly, there are no efficient algorithms known for propagating these data-structures. On the other hand, MDD, cost-MDD and d-DNNF propagators (with explanations) are well understood [91, 93, 94] so we decided to limit our scope to these representations.

Hoda et al. [108] create a pure MDD-based CP solver. Given a problem, they construct an MDD that takes all the variables of the problem. Initially that MDD is of width 1 (i.e., a “stick”) that allows more solutions than the problem permits. Then, each constraint is propagated through the MDD to remove arcs that are inconsistent with the constraints. This provides a tractable data structure that models the original problem. The focus of their paper was the construction of the MDD by propagating constraints in the MDD. This technique is presented under the name compilation by separation by Bergman et al. [32]. Note that constraints are not propagated to a fixed point in that work; only two passes for each constraints are performed. Because they were mapping an entire problem into an MDD, the size of the latter could explode easily. For that reason, they consider approximate MDDs where nodes are merged to keep a predefined width. This of course allows for more solutions in the MDD than there are in the original problem. The approximate MDD is then used as a “domain store”.

Earlier work by Andersen et al. [12] constructs an MDD constraint store, much like Hoda et al. [108]. The main difference between these two approaches is that Andersen et al. [12] do not construct a pure MDD-based solver, but a hybrid one where the MDD starts by being of width one and as the search advances the MDD is refined by vertex splitting. Similarly, the MDD is always approximated to avoid size explosion.

The approach we use in this chapter differs from these two papers in that we do not build an MDD as a constraint store to capture an entire problem. Rather we focus on a subset of variables to do a preprocessing with the hope of enhancing the solving process. The goals are completely different. Our construction method for the MDD is also substantially different, as we do not use construction by separation. Furthermore, our resulting MDD is exact (although it only represents a part of the problem).

Work by Hadzic et al. [105] and Bergman et al. [29] also investigated compiling problems into MDDs. The initial algorithms used to construct the MDDs in these papers are
very similar to the algorithm we use: the MDD is built top-down using caching to detect equivalence of nodes. The caching used by Hadzic et al. [105] is specific to linear constraints and the \texttt{alldifferent} global (and detects fewer equivalences than the caching we use). The authors propose alternative ways of subsuming this problem with incremental algorithms and vertex-splitting approaches. The resulting MDD is approximated to avoid size explosion as well. Bergman et al. [29] apply the same algorithm to the specific case of Set-Covering, where the caching is properly constructed to detect more node equivalences for that specific problem. Even with efficient caching, the MDD needs to be approximated due to its size.

The main difference between these two publications and the work done in this chapter is the caching technique. We use the technique for subproblem dominance presented by Chu et al. [47]. This generalization solves several problems encountered in previous research. Specifically, it can be applied to any problem, rather than only problems with linear and \texttt{alldifferent} or set-covering constraints. The combination of using this caching technique and a reordering of the variables in our algorithms (as we will see) allows detecting much more equivalence than their work. Another major difference is that our MDDs are exact, and not approximate.

Other less closely related work [31, 49, 118] builds MDDs to use in Lagrangian Decompositions, or specifically for scheduling problems and bin-packing problems. The MDDs are not built in a generic way that could serve our purpose. Binary Decision Diagrams have also been used [28, 30, 33], but either no generic automatic compilation was used, or the compiled BDD was approximate and used only as a bounding technique for branch-and-bound.

\section{NNF Compilations}

d-DNNFs, as well as other forms of NNFs, are well known and widely used in model counting (\#SAT problems) and knowledge compilation work, but unlike MDDs they have had much less attention from the CP community. There is, nonetheless, a propagator for d-DNNF constraints with explanations [91] available in \texttt{CHUFFED}, which we will be using.
Darwiche [54] worked on compiling CNF to d-DNNF. The algorithm presented in his paper is somewhat similar to ours for d-DNNFs, but it is much simpler as its scope is limited to CNF formulae. Later on, Huang and Darwiche [110] worked on compiling propositional theories (again CNF formulae) into different tractable target languages, including d-DNNFs. This work differs from their previous work in that they used the exhaustive version of the DPLL algorithm from SAT solvers in order to construct a Free Binary Decision Diagram (FBDD) that is then converted into a d-DNNF. This research yielded the c2D compiler from CNFs into d-DNNFs. The work by Muise et al. [149] starts from that research and further improved it, yielding the DSHARP compiler.

Sang et al. [190] published an improvement to model counting using the ZCHAFF solver where they used caching and clause learning, just like we do (since CHUFFED also implements clause learning, and we used caching). With an extra effort, their work could be converted in a precompilation technique like ours, but this was not their goal: their objective was to perform model counting, so the solutions were never stored in a data structure. Furthermore, their model counting and caching was targeted to SAT problems, and not to Constraint Programming, therefore it could not have been used directly to achieve our task, due to the presence of global constraints in CP models.

Another interesting application of knowledge compilation by Jha and Suciu [111] looked into compiling database queries into d-DNNFs. Work in knowledge compilation has also been used for probabilistic reasoning and inference via model counting using d-DNNFs or related languages [129, 212].

A common component in all these works in knowledge compilation is the use of caching very similar to the one we use ourselves [47]. Nonetheless, because all this work is directed to propositional formulae and SAT, they did not need to deal with global constraints as we have to. Furthermore, we have not found any work that uses this compilation into d-DNNFs for the purpose we are using them, that is, preprocessing to enhance the solving process of a CP solver.
8.3 Subproblem Equivalence

A key component of the algorithm to build structures that can store all the solutions to a CSP efficiently is equivalence detection: we will need to detect when we reach a state where the “remaining subproblem” is equivalent to a previously seen “remaining subproblem”. If we detect this we can re-use parts of the data structures we constructed previously.

As far as we are aware, the state of the art in subproblem dominance detection in CP is the method of Chu et al. [47]. In their paper, the authors describe how to map a subproblem into a concise key that can be used to compare two subproblems. The goal of their work was detecting dominance between subproblems, in order to avoid exploring a subproblem that was already proved to contain no solutions. We use their dominance detection to efficiently construct our solution stores.

The key definition and proposition we use are as follows. Let \( P = (v, C) \) be a CSP and \( P' = (v, C') \) and \( P'' = (v, C'') \) be two CSPs (subproblems of \( P \)). Recall from Definitions 2.4 and 2.5 in Chapter 2 that the constraints of a CSP include domain constraints. Because \( P' \) and \( P'' \) are subproblems of the same CSP, they both have the same set of constraints except for the domain constraints, as the domains of the variables may be different in both subproblems. Let \( D' \) be the domain constraints of \( P' \) and \( D'' \) the domain constraints of \( P'' \), then \( C' = C \land D' \) and \( C'' = C \land D'' \).

Let \( \text{fixed}(P) \) be the set of fixed variables (i.e. their domain is a singleton) in any problem \( P \). We use \( \exists X Y \) to denote the existential quantification of every variable in a set \( X \) in a formula \( Y \). We say that \( P' \) dominates \( P'' \) if \( \text{fixed}(P') = \text{fixed}(P'') = F \) and \( \exists_f C \land D'' \Rightarrow \exists_f C \land D' \). We say that \( P' \) and \( P'' \) are equivalent if they dominate each other. Hence if we build an MDD/d-DNNF encoding \( P \), we can reuse the same structure to encode \( P' \). In order to efficiently check that \( \exists_f C \land D \Rightarrow \exists_f C \land D' \), we use Theorem 8.1.

**Theorem 8.1** (from [47]). Let \( P' = (v, C \land D') \) and \( P'' = (v, C \land D'') \) be subproblems of \( P = (v, C) \) where \( \text{fixed}(P') = \text{fixed}(P'') = F \) and \( U = v \setminus F \). Let \( D'_U \) and \( D''_U \) be the domains of unfixed variables in \( P' \) and \( P'' \) respectively. Then if \( \forall c \in C \) we have that \( \exists_f C \land D'' \Rightarrow \exists_f C \land D' \) and \( D'_U \Rightarrow D'_{U} \) then \( P \) dominates \( P' \).
What this means is that to detect that two remaining subproblems are equivalent we need to make sure that:

- The fixed variables are the same on both. We do not care about their value, only that they are fixed.

- The domains of the unfixed variables in both problems need to be the same.

- The projection of each constraint onto the unfixed variables has to be the same in both problems.

Thus, any subproblem $P$ can be projected into a key $(F, \{key(c, D) \mid c \in C\}, D_U)$. The construction of those keys is described in detail by Chu et al. [47], and thus we will only discuss here the ones that we will be using. In addition, we will present our key for the tree constraint, as this is novel to our work.

### 8.3.1 Key Construction

The examples that we will see in the experiments in Section 8.6 will use three types of constraints: binary, linear and the tree constraint. For the two first ones, Chu et al. [47] provided the keys as follows.

#### 8.3.1.1 Binary Constraints

Assuming that the propagator used in the solver has the property that once one of the two variables is fixed, the domain of the other variable is modified in a manner that the constraint will always be satisfied, then no key is needed for these constraints. This is the case because

- if neither variable is fixed, this information is already available in the part of the key representing the domains of the variables
- if neither variable is fixed, the binary constraint at hand is satisfiable (otherwise it would have caused a failure) and
- if at least one variable is fixed, the constraint is satisfied (by the propagation rules).

Thus, in all cases, the key would be the same, and therefore, no key is needed.
8.3.1.2 Linear Constraints

The key for linear constraints \( c \equiv \sum_{i=1}^{n} a_i x_i \leq a_0 \) is defined by \( a_0 - \sum_{i \in F_c} a_i x_i \) where \( F_c \) is the fixed variables involved in \( c \). The intuition behind this is that the gap between the right-hand-side and the current partial sum is what matters to detect equivalence between two subproblems. Thus, if the gap between the right hand side is 5, for example, it does not matter how the left hand side reached the value \( a_0 - 5 \), only the fact that it reached that value. Example 8.1 illustrates this.

Example 8.1. Consider the problem given by the constraint \( x_1 + x_2 + 2x_3 \leq 8 \) for variables \( \{x_1, x_2, x_3\} \subseteq [1..3]^3 \). The key for the subproblem \( P = (C, D) \) where \( D = \{x_1 = 1, x_2 = 2\} \) would be \( k_P = (\{x_1, x_2\}, [\text{true}, 8 - 1 - 2 = 5], \{x_3 \in 1..3\}) \). Now consider the subproblem \( P' = (C, D') \) where \( D' = \{x_1 = 2, x_2 = 1\} \). The key for \( P' \) is \( k_{P'} = (\{x_1, x_2\}, [\text{true}, 8 - 2 - 1 = 5], \{x_3 \in 1..3\}) \). Thus \( P \) and \( P' \) are equivalent, as it can be seen in Figure 2.7, page 33.

8.3.1.3 Tree Constraint

No key representation of this constraint was ever introduced, so we will be presenting it here. The motivation for choosing this global constraint to be represented as a key is that later on (in Section 8.5.1.2) we will see how we can split globals and we will use the tree constraint to demonstrate the value of this technique in the experimental section (Section 8.6).

Let \( G_1 \) and \( G_2 \) be two partial assignment to the graph variable in the tree constraint. The subproblems in the search tree below these partial assignments are called, respectively, \( P_1 \) and \( P_2 \). We want to create a key that allows us to identify whether \( P_1 \) and \( P_2 \) are equivalent.

Intuitively, we need all the mandatory nodes to be part of the key. This is because if \( G_1 \) contains node \( n \), and \( n \) is still unfixed in \( G_2 \), there may be solution of \( P_2 \) that does not involve \( n \), but those solutions will not appear in \( P_1 \). So this information needs to be in the key.

Furthermore, let \( G_1 \) have a set of mandatory edges connecting some mandatory nodes. This creates a connected component of mandatory nodes \( C_1 \). The edges involved in this
connected component are irrelevant. The intuition behind this claim is that if we compress $C_1$ into a “meta-node”, the problem remains the exact same: the rest of the tree needs to be connected regardless of what is inside $C_1$. For this reason mandatory edges do not need to appear in the key. This implies that mandatory nodes should appear in the key, grouped by connected components, to be able to identify the “meta-nodes” both in $G_1$ and $G_2$. Both should have the same “meta-nodes” in order for $P_1$ and $P_2$ to be equivalent. We need to carefully represent a “meta-node” in a canonical way: we choose to represent each “meta-node” with a list $\langle |A| \rangle + + A$ where $A$ is the list of nodes in the connected component sorted by a unique integer identifier, $\langle |A| \rangle$ is a list with one element being the length of $A$ and $++$ is the concatenation operator for lists. The representation of each “meta-node” is then concatenated. An example of this can be seen in 8.1a, where the three nodes labeled 1, 2 and 3 become a “meta-node” regardless of how they were connected. Node 4 remains on its own. Clearly, the two problems on the left of subfigure 8.1a are equivalent. Their keys are $\langle 3, 1, 2, 3, 1, 4 \rangle$. The bold numbers correspond to the sizes of each connected component, for clarity.

The tree propagator removes edges that could form a cycle if they became mandatory. These edges do not need to be in the key. The reason is that, for an edge $e = (u, v)$ to potentially create a cycle, $u$ and $v$ need to be connected to each other via mandatory edges. That is, $u$ and $v$ are in the same connected component, which is already represented in the key. As we said before, we do not care about what edges are used in a connected component, since we can regard it as a “meta-node”. Following the same reasoning we do not care about which edges are not in the connected component.

Nonetheless, if $G_1$ has a forbidden edge $e$ that would not have formed a cycle (this edge could not have been removed by the tree propagator given the propagation rules described in Chapter 3), and $G_2$ still has $e$ available, there may be solutions to $P_2$ that do not exist in $P_1$. Therefore we need to keep this information in the keys. To represent this edges in the key, we use unique identifiers for them (positive integers) that we negate to distinguish them for the part of the key corresponding to nodes. For example, in Figure 8.1b, the key for the top figure is $\langle 3, 1, 2, 3, 1, 4, -5 \rangle$, but for the bottom one is $\langle 3, 1, 2, 3, 1, 4, -6 \rangle$. So, even if the rest of the key is the same, these two problems are
deemed not to be equivalent.

To sum up, the final key follows is a list matching \( \langle (|A| + + A)^* + + (-I)^* \rangle \) where 
\( I \) are integer identifiers for edges, and \( A \) are sorted lists of integer identifiers for nodes
that are connected by mandatory edges.

(a) Connected components need to be repre-
resented in the key through the nodes they
contain, not the edges involved in them.

(b) Removed edges (edge 5 or 6) that do not
form a cycle need to appear in the key to
distinguish these two cases.

Figure 8.1: Example of key for the tree constraint.

8.4 CP-to-MDD and CP-to-Cost-MDD Compilations

Given a problem \( P = (v, C \land D) \), Algorithm 8.1 describes our compilation of an MDD for that problem. For simplicity, we separate the constraints of the problem in two parts: \( C \) for the non-domain constraints, and \( D \) for the domain constraints. The function \textsc{MDDify} takes three arguments. The first argument \( C \) is the set of constraints being considered for
building the MDD. The second, \( D \), is the current domain of all variables. The third argu-
ment, \( X \subseteq v \), is an ordered list of the variables in the order in which they should appear
in the MDD. The algorithm returns an MDD which represents the formula \( \exists v \setminus X C \land D \).

For clarity, assume that the variable \texttt{problem.store} is a global hash-table (initially empty).

The operations \& and | over MDDs are the classical \textit{conjoin} and \textit{disjoin} operations
defined for this data structure. The constructor \textsc{MDD} for an MDD simply takes a mathe-
matical expression and builds an MDD for it.

The algorithm works by keeping a map from keys for subproblems to already com-
puted MDDs for the remaining subproblem, or filling that same map if the remaining
subproblem has never been encountered before.
Algorithm 8.1 Constructing MDD via propagation

1: procedure MDDIFY(C, D, X)  
2:     D ← PROPAGATE(C, D)  \(\triangleright\) Propagate constraints C to fixed point  
3:     if D is a false domain then return FALSE  
4:     fbp ← fixed(D) ∩ X  \(\triangleright\) Variables fixed by propagation at this level  
5:     p ← KEY(C, D)  \(\triangleright\) Hash the remaining subproblem  
6:     if p ∈ problem store then  
7:         m ← problem store[p]  \(\triangleright\) MDD representing remaining subproblem  
8:         m ← m & MDD(D_fbp)  \(\triangleright\) Conjoin m with a “stick” MDD of fixed variables  
9:         return m  
10:     Y ← X \ fbp  \(\triangleright\) Remove fixed variables  
11:     if Y = [] then m ← TRUE  
12:     else  
13:         x ← head(Y), R ← tail(Y), m ← FALSE  
14:         for all d ∈ D(x) do  
15:             m ← m | ( MDD(x = d) & MDDIFY(C, D ∧ \{x = d\}, R) )  
16:         problem store[p] ← m  \(\triangleright\) Associate the remaining subproblem with the MDD  
17:         m ← m & MDD(D_fbp)  \(\triangleright\) Conjoin fixed variables  
18:     return m

First, it propagates the last decision made (or propagates root-level information in the first call). If this propagation results in unsatisfiability, the corresponding MDD is FALSE. Otherwise, the problem is mapped into a key. This is done by using the domains of variables and the constraints; details of this are presented by Chu et al. [47]. If the key matches a problem in the table, then the MDD associated to that remaining subproblem is returned (line 9). If the key does not match a subproblem (line 10), then we choose the next unfixed variable x in X. If there are no more unfixed variables, then the remaining subproblem is empty and we return the MDD TRUE. If not, we branch on each possible value of x and create an MDD that is a disjunction of the MDDs corresponding to each assignment. Note that the MDD returned by the recursive call only considers the variables in R.

After branching on all values for x we store the MDD encoding the remaining subproblem in the map.

There is one important detail in the algorithm regarding fixed variables. The cache key KEY(C, D) only considers which variables are fixed by D and not their values (except how they affect the constraints C). When variables are fixed by propagation, they may be fixed out of the sequence of variables X we are using to build the MDD, but eventually the MDDs returned must constrain these variables to their fixed values, respecting the
variable order (by the definition of an MDD). The MDD \( m \) we attach to \( \text{KEY}(C, D) \) represents the solutions for the remaining subproblem (and hence only considers the variable sequence \( Y \) that are not fixed). To return an MDD that considers the whole sequence \( X \) we build a “stick” MDD representing the fixed value for each newly fixed variable (line 17), and conjoin this with the MDD for the remaining subproblem.

It may not be obvious but the algorithm does not require that \( X \) includes all variables in \( v \). In this case the algorithm produces an MDD that encodes \( \exists_v \neg X \land D \), where \( \exists \) is the quasi-projection introduced by Chu and Stuckey [46]. Now \( \exists_v \neg X \land D \Rightarrow \exists_v \neg X \land D \), hence the resulting MDD does not remove any solutions of the original problem, and hence can be safely added to the original model. It is also guaranteed that \( \exists_v \neg X \land D \Rightarrow c \) for all \( c \) where involving only variables in \( X \), so when adding the quasi-projection we can remove all constraints all of whose variables appear in \( X \) since they will be made redundant by the MDD of the quasi-projection.

A more straightforward approach to constructing \text{MDDIFY} is to attach the returned MDD to \( \text{KEY}(C, D) \). But if we do this we may miss opportunities to reuse MDDs as we will see in Example 8.2.

**Example 8.2.** Consider a subproblem with \( C = \{ -x_1 + x_3 = 2 \} \) and \( D_0 = \{ x_1 \in 1..2, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2 \} \) using the order \( X = [x_1, x_2, x_3, x_4] \).

The initial call \text{MDDIFY}(C, D_0, X) finds no propagation, and calculates key \((\emptyset,,, D_0)\), because binary constraints have no key [47], which has no entry in problem_store. We choose \( x_1 = 1 \) and make a recursive call \text{MDDIFY}(C, \{ x_1 = 1, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2 \}, [x_2, x_3, x_4])

Now propagation computes \( D_1 = \{ x_1 = 1, x_2 \in 1..2, x_3 = 3, x_4 \in 1..2 \} \) and \( fbp = \{ x_3 \}; \) the key is \( k_1 = ([x_1, x_3], [x_2 \in 1..2, x_4 \in 1..2]) \) which again has no entry. The remaining unfixed variables are \( Y = [x_2, x_4] \). We then choose \( x_2 = 1 \) and make a recursive call \text{MDDIFY}(C, \{ x_1 = 1, x_2 = 1, x_3 = 3, x_4 \in 1..2 \}, [x_4]) \). After examining both possibilities for \( x_4 \)
we build the MDD shown in Figure 8.2a attached to key \( k_2 = ([x_1, x_2, x_3], [x_4 \in 1..2]) \). We then choose \( x_2 = 2 \) and the recursive call \text{MDDIFY}(C, \{ x_1 = 1, x_2 = 2, x_3 = 3, x_4 \in 1..2 \}, [x_4]) \) reuses the MDD attached to \( k_2 \). After exploring all values of \( x_2 \), the MDD of Figure 8.2b is attached to \( k_1 \), but the returned one is shown in Figure 8.2c, with the fixed value for \( x_3 = 3 \). When trying \( x_1 = 2 \), the recursive call \text{MDDIFY}(C, \{ x_1 = 2, x_2 \in 1..2, x_3 \in 3..4, x_4 \in
8.4 CP-to-MDD and CP-to-Cost-MDD Compilations

Figure 8.2: Construction of an MDD with caching

\[(a)\ \text{MDD for } [x_4]\]

\[(b)\ \text{MDD for } [x_2, x_4].\]

\[(c)\ \text{MDD for } [x_2, x_3 = 3, x_4].\]

The key is \(\{(x_1, x_3), [], \{x_2 \in 1..2, x_4 \in 1..2\}\}\), which matches \(k_1\), thus returning the MDD in Figure 8.2b to which the value \(x_3 = 4\) is inserted (line 8, with \(fbp = \{x_3\}\)).

If we instead did not delay inserting the fixed variable \(x_3\) we would not discover a cache hit, and indeed the MDD of sub-figure 8.2c is not reusable.

The example clearly shows how delaying the insertion of propagated variables is key to reuse MDDs. This optimization can result in important differences in MDD construction (e.g. \(\approx 481k\) vs. \(\approx 1\) million search nodes for one instance).

Note that the algorithms by Bergman et al. [29] and Hadzic et al. [105] are not able to exploit this equivalence. First, their keys for \(D_1\) and \(D_2\) would not have matched. Secondly, if they had used the more effective keys we use, their algorithms would need to be modified as ours, since the stored MDDs could not have been reused (the fixed variables having possibly different values).

The key benefit of mddification compared to tablification [63, 62], is that we have to explore much less search space because of caching. Indeed, to construct a table, all solutions must be retrieved, thus a much broader region of the search space (all regions leading to satisfiable solutions at least) must be explored. In our approach, the stored MDDs represent parts of solutions that can be “plugged-in” to other partial assignments. As we saw in Example 8.2 we never had to explore \(x_1 = 1 \land x_2 = 2\) or \(x_1 = 2\) to retrieve the 6 solutions in those subspaces. In Section 8.5 we will see we also have this same advantage for the compilation of d-DNNFs.
8.4.1 Compiling a Cost-MDD

We can reuse Algorithm 8.1 for compiling a Cost-MDD under certain circumstances. Given an objective expression \( o = \sum_{x \in X, d \in D(x)} (w_{xd} \times [x = d]) \) if we build an MDD for the problem \((C, D)\) for variable sequence \(X\), we can convert this to a cost-MDD for calculating the value of \(o\) by adding weight \(w_{xd}\) to each edge labeled \(x = d\) in the resulting MDD.

8.5 CP-to-d-DNNF Compilation

The algorithm to build d-DNNFs from a set of constraints and variables is similar to the one to build MDDs. The major differences is that a fixed order of variables is not required (unlike in the case of MDDs). Therefore we do not need to worry about reinserting the set of variables \(f_{bp}\) in the data structure as we did in Algorithm 8.1. The variables fixed by propagation will instead just be immediately added to the d-DNNF.

Algorithm 8.2 provides pseudocode for the compilation of d-DNNFs from a set of variables and constraints. The arguments are the same as for Algorithm 8.1. The algorithm constructs a d-DNNF node and returns it. First, we propagate the current decisions. If this produces a false domain, then there is no solution, and the returned node is simply FALSE. If this was conjoined to some other formula, then the conjoined formula would obviously become false as well. The variables that have been fixed at the current decision level are then retrieved and directly injected in the d-DNNF called node in the pseudo-code. For MDDs we needed to inject them afterwards (in lines 8 and 17 of Algorithm 8.1). Here, the propagated assignments are just conjoined to the node in line 7.

Similarly to the MDD compilation, a lookup is done in the problem cache, and if found, the corresponding d-DNNF is returned, as it was already constructed. Notice how the returned d-DNNF is first conjoined with the node d-DNNF, which at this point may contain the assignments of variables fixed by propagation (in line 11).

Otherwise, a new variable \(x\) is chosen to be branched on. The choice of variable is wide open here, as there is no strict ordering in d-DNNFs. If no variable remains, then node can be returned. Otherwise, we iteratively assign a different value to \(x\) and recurse
Algorithm 8.2 Constructing d-DNNF via propagation

1: procedure DDNNFFY(C, D, X)
2: \( \text{node} \leftarrow \text{TRUE} \) \hfill \triangleright \text{Propagate constraints } C \text{ to fixed point}
3: \( D \leftarrow \text{PROPAGATE}(C, D) \)
4: \textbf{if } D \text{ is a false domain then return FALSE} \hfill \triangleright \text{Variables fixed by propagation at this level}
5: \( \text{fbp} \leftarrow \text{fixed}(D) \cap X \)
6: \textbf{for all } f \in \text{fbp} \text{ do}
7: \( \text{node} \leftarrow \text{node} \land \{ f = \text{val}(f) \} \) \hfill \triangleright \text{Append leaves enforcing propagated values}
8: \( p \leftarrow \text{KEY}(C, D) \) \hfill \triangleright \text{Hash the remaining subproblem}
9: \textbf{if } p \in \text{problemstore} \text{ then}
10: \( d \leftarrow \text{problemstore}[p] \) \hfill \triangleright \text{d-DNNF representing remaining subproblem}
11: \( \text{node} \leftarrow \text{node} \land d \)
12: \text{return } \text{node}

else
13: \( x \leftarrow \text{NEXTVAR}(X) \) \hfill \triangleright \text{Next variable to branch on}
14: \( \text{children} \leftarrow \text{FALSE} \)
15: \textbf{if } x = \bot \text{ then}
16: \text{return } \text{node}
17: \textbf{else}
18: \textbf{for all } d \in D(x) \text{ do}
19: \( \text{children} \leftarrow \text{children} \lor (\text{DDNNFFY}(C, D \cup \{ x = d \}, X) \land [x = d]) \)
20: \( \text{problemstore}[p] \leftarrow \text{children} \)
21: \text{return } \text{node} \land \text{children}

in the construction of the d-DNNF. The d-DNNF corresponding to the disjunction of recursive calls is stored for possible reuse. The returned d-DNNF is that same disjunction conjoint with the fixed variables (already in node).

Example 8.3. Consider the same problem as in Example 8.2: \( C = \{ -x_1 + x_3 = 2 \} \) and \( D_0 = \{ x_1 \in 1..2, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2 \} \). For simplicity we will use the same order \( X = [x_1, x_2, x_3, x_4] \), but note that this is not necessary.

The initial call DDNNFFY\((C, D_0, X)\) finds no propagation, and calculates key \((\emptyset, [], D_0)\), which has no entry in problemstore (just as earlier). We first branch on \( x_1 = 1 \) and make a recursive call DDNNFFY\((C, \{ x_1 = 1, x_2 \in 1..2, x_3 \in 3..4, x_4 \in 1..2 \}, X)\). Now propagation computes \( D_1 = \{ x_1 = 1, x_2 \in 1..2, x_3 = 3, x_4 \in 1..2 \} \) and \( \text{fbp} = \{ x_3 \} \). The partially constructed d-DNNF is shown in Figure 8.3a (the recursive call will attach d-DNNFs to the edge shown in dashes); the key is \( k_1 = (\{ x_1, x_3 \}, [], \{ x_2 \in 1..2, x_4 \in 1..2 \}) \) which again has no entry.

We then choose \( x_2 = 1 \) and make a recursive call DDNNFFY\((C, \{ x_1 = 1, x_2 = 1, x_3 = 3, x_4 \in 1..2 \}, X)\). After examining both possibilities for \( x_4 \) we build the d-DNNF shown in Figure 8.3b attached to key \( k_2 = (\{ x_1, x_2, x_3 \}, [], \{ x_4 \in 1..2 \}) \). When the recursive call ends,
Figure 8.3: Construction of d-DNNF with caching

this d-DNNF is conjoined to \([x_2 = 1]\) yielding the circled part of Fig. 8.3c. We then choose \(x_2 = 2\) and the recursive call \(\text{DDNNFFY}(C, \{x_1 = 1, x_2 = 2, x_3 = 3, x_4 \in 1..2\}, X)\) reuses the d-DNNF attached to \(k_2\). This is shown in Fig. 8.3c by a gray dashed arrow. After exploring all values of \(x_2\), the d-DDNF of Figure 8.3c is attached to \(k_1\), but the returned one is shown in Figure 8.3a, with the fixed value for \(x_3\) (the dashed line connects to the root of Fig. 8.3c). When trying \(x_1 = 2\), the recursive call \(\text{DDNNFFY}(C, \{x_1 = 2, x_2 \in 1..2, x_3 = 3..4, x_4 \in 1..2\}, [x_2, x_3, x_4])\) computes \(D_2 = \{x_1 = 2, x_2 \in 1..2, x_3 = 4, x_4 \in 1..2\}\) by propagation. The key is \((\{x_1, x_3\}, [], [x_2 \in 1..2, x_4 \in 1..2])\), which matches \(k_1\), thus returning the d-DNNF in Figure 8.3c to which the value \(x_3 = 4\) is conjoined.

The final d-DNNF will have a \(\lor\) root with two \(\land\) children, each of them with 3 children, two of which are the values of \(x_1\) and \(x_3\) on each branch and the third being the d-DNNF in Figure 8.3c.

Notice that similar behavior as for MDDs, where the d-DNNF stored for reuse is not necessarily the same as the one returned by the algorithm.

### 8.5.1 Splitting Subproblems for d-DNNFs

Because d-DNNFs are decomposable, it is natural to try to decompose the problem. Imagine a CSP composed of two completely independent parts: then the solution to the CSP is the conjunction of the solutions of all the parts. d-DNNFs are the perfect structure for this kind of behavior, if we can detect this independence of problems.
Example 8.4. Looking back at Example 8.3, variables $x_2$ and $x_4$ are completely independent from the rest of the problem. Therefore there are 3 independent sets of variables to this problem $\{x_1, x_3\}$, $\{x_2\}$ and $\{x_4\}$. The d-DNNF we would want for this would be as shown in Figure 8.4.

8.5.1.1 Detecting Independent Subproblems

A simple way to detect subproblems as being independent is to simply build the constraint graph after each decision. The Constraint Graph is a graphical representation of the Constraint Problem at hand. Nodes are variables of the problem, and edges connect variables that are related through constraints in the problem. This is commonly used in the compilation of d-DNNFs that we have seen in Section 8.2.

In our implementation, we consider all propagators and clauses to be keyable objects. All keyable objects produce a key (possibly empty) for subproblem equivalence (as seen in Section 8.3). In addition, we make all keyable objects update a disjoint-set data structure (also known as union-find) of variables at the time of producing a key. This introduces little overhead, since both tasks can usually be performed at the exact same time. By default, all keyable objects will unite all variables involved with the keyable itself. For instance, for a clause, all boolean variables involved in the clause are united. Similarly, all variables in each propagator are united. In addition, as our implementation in CHUFFED uses no-good learning, integer variables are also united with the literals associated to them. Thanks to this, we build an internal constraint graph after each decision.

The goal is then to consider each connected component of the Constraint Graph separately, as they are indeed independent problems. The expected d-DNNFs will look similar to the one in Figure 8.4, where the independent subproblems are all children of an “∧” node.
8.5.1.2 Splitting Global Constraints

Unlike CNFs or SAT problems, CP has global constraints. Sometimes, after some decisions are made, it is possible that some global constraints can split their variables into independent connected components of the constraints graph. Because these constraints are "opaque" they cannot be split directly as we do with other constraints. That is, we can separate them from each other, but it is not obvious how to split the inside of a global constraint.

Example 8.5. Consider the constraint $\text{alldifferent}([a, b, c, d, e])$ where $D(a) = D(b) = [0..5]$, $D(d) = D(e) = [6..10]$ and $D(c) = [0..10]$. If we make the decision that $c < 5$ then the constraint can be split into two globals: $\text{alldifferent}([a, b, c]) \land \text{alldifferent}([d, e])$.

This is something that each global needs to implement, as it cannot be generalized for all globals.

Fages et al. [86] formalized the idea of splitting globals. They showed it is very useful even simply for reducing the propagation time, arguing that propagating on many "smaller" constraints is faster than propagating on few "big" constraints. They showed how to split the $\text{alldifferent}$ constraint as well as the $\text{cumulative}$ constraint. In their paper, though, their algorithm does not require that the splitting is done in independent set of variables. That is, they can split the variables involved in an $\text{alldifferent}$ in a set of sets $G = \{S_1, S_2, ..., S_n\}$ such that $\exists i, j, S_i \cap S_j \neq \emptyset$. In our case we can’t have this, since the d-DNNF needs to split into independent set of variables. But this is a minor difference.

We have implemented the global splitting for three globals, as follows.

1. $\text{alldifferent}$ [86]: the variables are split into groups by overlapping domains. That is, if the intersection of the domain of two variables is empty, then the variables are not united at this stage (note that some other constraint may unite them).

2. $\text{minimum}$: this constraint enforces that a variable $y$ takes the value of the smallest variable in an array $x$ of integer variables. The split here is also trivial: only the variables in $x$ whose domain overlaps the domain of $y$ are united.
3. tree (as seen in Chapter 3): this constraint enforces a graph variable to be a tree. As we saw in Section 3.3.2 of Chapter 3, articulations are nodes that, if removed, split the graph in two disconnected “induced” graphs. The solution to a tree constraint with an articulation node is the conjunction of the solutions to the tree constraint on each “induced” subgraph. Therefore, the nodes in each “induced” subgraph are connected to each other, but not across subgraphs.

As an anecdotal result, for one alldifferent constraint with 10 variables where the domains of the even-numbered variables is 1..5 and the domain of the odd-numbered variables is 6..10 except that the first variable had domain 1..10 we saw a huge difference in size. The d-DNNF built using splitting had 635 nodes, whereas the one without splitting the alldifferent global had 5461 nodes! In the first case, the d-DNNF split in 4 rapidly (first in 2, depending on the value of the 1..10 variable, and once that decision was made, there are two independent conjoined d-DNNFs). In the second case, the d-DNNF was much more intricate and hard to visualize.

Once we have this ability to split a problem into smaller subproblems, we can also cache them independently, as shown in Algorithm 8.3. Notice in this version, there is a loop through all the independent problems, and each one of them is treated individually (each has a key and its own set of variables). The call to KEY(C, D) was changed to a call to a SPLIT(C, D) function that, in turn, calls a function for each propagator that produces a key and updates a union-find structure. We saw no overhead from updating the union-find. The results section backs this claim.

8.5.1.3 Variable Selection

As said in the introduction, d-DNNFs are a well known data structure in the fields of model counting. The notion of splitting the problem into independent subproblems is also well known in that field. It is therefore natural that researchers have investigated search strategies to be able to enhance this splitting of problems.

We implement the Variable State Aware Decaying Sum (VSADS) strategy, which is a combination of Variable State Independent Decaying Sum (VSIDS) and Dynamic Largest
Algorithm 8.3 Constructing d-DNNF via propagation while splitting subproblems

1: procedure DDNNFFY(C, D, X)
2:     node ← TRUE
3:     D ← PROPAGATE(C, D)  ▷ Propagate constraints C to fixed point
4:     if D is a false domain then return FALSE
5:     fbp ← fixed(D) ∩ X  ▷ Variables fixed by propagation at this level
6:     for all f ∈ fbp do
7:         node ← node ∧ [f = val(f)]  ▷ Append leaves enforcing propagated values
8:     pbs ← SPLIT(C, D)  ▷ Split into independent subproblems
9:     for all pb ∈ pbs do
10:        if pb.key ∈ problem_store then
11:            d ← problem_store[pb.key]
12:            continue;  ▷ No more variables
13:        x ← NEXTVAR(pb.vars)  ▷ Next variable to branch on for this subproblem
14:        if x = ⊥ then
15:            problem_store[pb.key] ← children  ▷ Append leaves enforcing propagated values
16:            return node
17:        else
18:            children ← FALSE
19:            for all d ∈ D(x) do
20:                children ← children ∨ (DDNNFFY(C, D ∪ {x = d}, X) ∧ [x = d])
22:            node ← node ∧ children
23:            return node

Combined Sum (DLCS). It was introduced by Davies and Bacchus [55] and a thorough explanation can be found there. All our experiments used this strategy to generate the d-DNNFs (using the same strategy meta-parameters as Davies and Bacchus [55]). The call to NEXTVAR() in Algorithm 8.2 uses this strategy.

8.6 Experimental Evaluation and Results

We present here our results. The experiments were done on a Linux 3.16 Intel® Core™ i7-4770 CPU @ 3.40GHz, 15.6GB of RAM, with the CP solver CHUFFED [48].

The table propagator implemented in CHUFFED is a decomposition in clauses of the table. The propagators for MDDs, cost-MDDs and d-DNNFs were described by Gange et al. [93, 94, 91]. Note that the results shown for tablification also use CHUFFED for preprocessing (that is building the table) and for solving the problem.

We first describe all the problems that we tackled in Section 8.6.1. Section 8.6.2 will an-
analyze the compilation speed of the different approaches, Section 8.6.3 analyzes the merit of splitting globals for the construction of d-DNNFs, and Section 8.6.4 will compare the net benefit of the three compilation approaches.

8.6.1 Problem Descriptions

8.6.1.1 Black-Hole

The well known Black-Hole problem [97, 159] can be stated as follows. We are given 17 piles of 3 cards, and the Ace of Spades as the starting point of the “black-hole”. The player needs to move the top of a pile onto the black hole until all the cards are on the black-hole. The card to be moved must be one less or one more than the current top of the black-hole, regardless of suit. The predicate ensuring that two cards are adjacent is shown in Figure 8.5.

```plaintext
predicate adjacent(var 1..52: a, var 1..52: b) =
((a-b) in {13*i+1 | i in -4..3} union {13*i-1 | i in -3..4});
```

Figure 8.5: MINIZINC Predicate for Black-Hole

For this problem, we separate the predicate from the problem and build an MDD or d-DNNF that ensures two cards are adjacent. The variables of our solution store are the parameters of the predicate. The body of the predicate is then substituted by the call to the global corresponding to either encoding. The MDD/d-DNNF has two variables of domains 1..52. We used 21 instances from Dekker et al. [63].

8.6.1.2 Block Party Metacube

The Block-Party Metacube Problem [63] can be defined as follows. Cubes have icons on each corner of each side, which have 3 attributes: shape, color and pattern. A metacube is an arrangement of 8 of these cubes that forms a cube (i.e. $2 \times 2 \times 2$ cubes). A block-party metacube is formed when the 4 icons on the center of each face of the metacube are all identical (in all 3 attributes) or have all attributes different. The full model for this
problem can be found in the paper by Dekker [62]. As part of the model, a predicate is
defined to link the identifier of each cube to the three icons of that cube that are on the
center of a face of the metacube, for all the possible rotations of each cube. The predicate
was presented and tablified by Dekker et al. [63]. The annotated variables are the argu-
ments, so there will be 4 variables in the MDD/d-DNNF, each with a domain 0..63 (there
are 64 possible symbols). We used 14 instances from [63].

```plaintext
predicate link_cube_and_symbols(array [1..4] of var 0..63: cs) =
  let { var 1..24: pos, var int: cube = cs[1],} in
  forall ( i in 1..3 ) ( data[cube, pp[pos, i]]==cs[i+1] );
```

Figure 8.6: MÌNIÈZINC Predicate for Block-Party

### 8.6.1.3 Fox Geese Corn (FGC)

This is a generalization of the famous Fox-Goose-Corn puzzle. In this version, a farmer
wants to transport $f$ foxes, $g$ geese and $c$ bags of corn from the west to the east side of
a river. She has a boat with a capacity available for her to move some of the goods at
once while the rest remain on shore. She can go back and forth to bring as many goods
as she wants to the east. Nonetheless, some rules apply to the goods that are not being
supervised on either side while the farmer is on the boat: i) if only foxes and bags of
corn are sitting on a shore, then a fox dies by eating a bag of corn; ii) if there are foxes
and geese, and the foxes outnumber the geese, one fox dies; iii) on the other hand, if the
geese are not outnumbered, each fox kills one goose; iv) if there is no fox, and the geese
outnumber the bags, a goose dies and one bag is eaten; v) on the other hand, if the corn
is not outnumbered, each goose eats a bag.

The farmer must maximize the profit (there is a price for each good) from the surviv-
ing goods on the east. Although she could do any number of trips, there is an optimal
number of trips $t$ after which it is worth abandoning goods on the west and continue her
journey with the goods on the east. The natural way of modeling this is by defining a
predicate for the above rules and apply the predicate for trips before $t$. It will update the
number of foxes, geese and corn bags for the next time slice. After time $t$, the predicate
is not applied, as she is not crossing the river anymore. Thus the predicate is reified by a condition dependent on \( t \). This is precisely the model we used. The precompiled predicate has 7 variables (3 for the state before the travel, 3 for the state after the travel, and a Boolean to turn on and off the constraint). Their domains will be highly dependent on the instances: we use 11 instances with domains ranging between 5 and 27 for each variable. The predicate’s declaration is in Figure 8.7.

```plaintext
predicate alone(var bool: reif, var 0..f: fox0, var 0..f: fox1,
   var 0..g: geese0, var 0..g: geese1, var 0..c: corn0, var 0..c: corn1) =
   if reif == true then
     if fox0 = 0 \ geese0 = 0
       fox1 = fox0 /
       if geese0 > 0 then
         fox1 = fox0 - 1 /\ geese0 = geese0 - fox0
       else
         fox1 = 0
       endif
     elseif fox0 > 0 \/ geese0 > 0 then
       fox1 = fox0 - 1 /\ geese1 = geese0 - fox0
     elseif geese0 = 0 \ fox0 > 0 \ geese0 > 0 then
       fox1 = fox0 - 1 /\ geese1 = geese0 - 1
     else
       fox1 = 0
     endif
   else
     true endif;
```

Figure 8.7: MINIZINC Predicate for Fox-Geese-Corn

### 8.6.1.4 Water Bucket

This is another classic puzzle. Given a set of buckets of water (some initially filled, some not), and a target water level for each bucket, transfer the initially contained water into other buckets to reach the target levels. The difficulty lies in the fact that we cannot stop pouring water anytime we please, but only when either the receiving bucket is full or the pouring bucket is empty. The objective is to achieve the final levels in as few transfers as possible.
The model uses a predicate that represents the transition of states of the buckets at a
given time when choosing which buckets to transfer, Figure 8.8. There will be $2 \times j + 2$
variables (where $j$ is the number of buckets). The domains depend on each instance, but
the biggest domain for any variable across all 4 instances we used is 0..12 with up to 4
buckets.

```plaintext
predicate transfer(array[BUCKET] of var PINT: state_b, array[BUCKET] of var PINT: state_a,
                   var BUCKET: from, var BUCKET: to) =
  (state_b = final \ state_a = final \ from = 1 \ to = 1) \/
  (forall(b in BUCKET where b != from \ b != to)
    (state_a[b] = state_b[b]) \/
    [state_a[from], state_a[to]] = pour(state_b[from], capacity[from], state_b[to], capacity[to]));

function array[1..2] of var PINT: pour(var int: from_b, var int: from_cap,
                   var int: to_b, var int: to_cap) =
  let { var PINT: amount = min(from_b, to_cap - to_b); } in
  [from_b - amount, to_b + amount];
```

Figure 8.8: MINIZINC Predicate for Water-Bucket

8.6.1.5 Progressive Graph Coloring (PGC)

We introduced this problem for our tests. Given a graph of $n$ nodes, an initial coloring
with $c$ colors and a target coloring, change the color of at most $k$ nodes at each step to
reach the target coloring in as few steps as possible maintaining a valid coloring at all
times. A valid coloring is one where any two adjacent nodes are not assigned the same
color.

This is modeled with a “valid coloring” predicate, called at each step. The variables
taken in the predicate will be one for each node, all with domains 1..c. We build the
graphs using Erdős-Rényi’s model [82] with the probability of an edge existing being 0.5
for 15-node graphs, and 0.2 for 25-node graphs.

```plaintext
predicate valid_coloring(array[NODES] of var COLORS: coloring,
  forall (e in EDGES) (coloring(xs[e]) != coloring(ys[e]));
```

Figure 8.9: MINIZINC Predicate for Valid Coloring
8.6 Experimental Evaluation and Results

8.6.1.6 Shift Scheduling

This problem was first introduced by Demassey et al. [64]. It consists in allocating \( n \) workers in 15 minute shifts to \( a \) activities such that all activity has the minimum required number of workers at all times. The objective is to minimize the number of shifts worked. The constraints are: i) workers must work on a task at least 1 hour, and cannot switch tasks without a 15 minute break; ii) part-time workers work between 3 and 5.75 hours, with one 15 minute break; iii) full-time workers work 6 to 8 hours, with one hour for lunch, and 2 breaks (one before and one after lunch); iv) workers can only be working after the first activity is started, and before the last activity finishes.

This problem can be modeled using a grammar constraint as described by Gange et al. [94]. We used their exact same model, with 5 variations to it:

- \textsc{ShiftReg} uses regular constraints (implemented by MDD propagators) to enforce the “shape” of a valid shift.
- \textsc{ShiftDec} uses a decomposition into clauses to enforce valid shifts.
- \textsc{ShiftNNF} and \textsc{ShiftGCC} use grammar propagators [114] to enforce valid shifts.
- \textsc{ShiftWRG} uses cost-regular constraints (implemented by cost-MDD propagators) to enforce the shape of shifts, and compute their costs (the objective).

\textsc{ShiftGCC} uses a gcc constraint [180] to ensure the demand of workers is met at each shift, whereas the other models simply use a linear for this. The cost function is modeled with a linear constraint in all cases (except \textsc{ShiftWRG}).

For this problem, we choose the variables of the MDD/d-DNNF to be the task allocation of all workers in windows of 3 adjacent shifts. That is, one for each 45 minutes. The ordering of the variables puts first the allocations of the first worker for three shifts, then the allocations of the second worker for the same three shifts, and so on. We used 8 instances with up to 2 activities and 5 workers.
8.6.1.7 All-Trees

To study whether splitting globals is worthwhile or not, we created a very simple model to collect in a d-DNNF all the solutions to the Steiner Tree Problem in a set of graphs.

Here we are not concerned with the Steiner Tree Problem per se, instead what we want is to simply collect all the Steiner Trees of a graph given a set of terminals into a d-DNNF. This will help us understand whether splitting the tree constraint was worthwhile. The d-DNNF can then be used in a variety of problems involving Steiner Trees.

We used instances from the SteinLib [124], from the dataset ES10FST and ES20FST (15 instances each).

8.6.2 Compilation Effort

We first compare the results of the first 5 problems against the tablification approach [63].

Let us first compare the total presolving time for each problem (all instances summed up) in Table 8.1. For Black-Hole, the same precomputation could be recycled for all instances. All the others are instance dependent.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Instances</th>
<th>Table construction</th>
<th>MDD construction</th>
<th>d-DNNF construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Hole</td>
<td>21</td>
<td>0.01s</td>
<td>0.02s</td>
<td>0.02s</td>
</tr>
<tr>
<td>Block-Party</td>
<td>14</td>
<td>0.30s</td>
<td>0.28s</td>
<td>0.28s</td>
</tr>
<tr>
<td>FGC (reif)</td>
<td>11</td>
<td>(+∞) 9.06s</td>
<td>(6.40s) 0.22s</td>
<td>(6.65s) 0.19s</td>
</tr>
<tr>
<td>FGC (no reif)</td>
<td>11</td>
<td>0.60s</td>
<td>0.27s</td>
<td>0.21s</td>
</tr>
<tr>
<td>Water Bucket</td>
<td>4</td>
<td>173.15s</td>
<td>0.45s</td>
<td>0.44s</td>
</tr>
<tr>
<td>PGC (n = 15, c = 6)</td>
<td>30</td>
<td>10.89s</td>
<td>7.06s</td>
<td>7.74s</td>
</tr>
<tr>
<td>PGC (n = 25, c = 5)</td>
<td>25</td>
<td>(+∞) 437.68s</td>
<td>(90.24s) 77.67s</td>
<td>(121.63s) 105.03s</td>
</tr>
<tr>
<td>Total</td>
<td>122</td>
<td>636.1s</td>
<td>85.97s</td>
<td>113.93s</td>
</tr>
</tbody>
</table>

Table 8.1: Time spent compiling Tables, MDDs or d-DNNFs

There are two things worth noting from this table. First building a table can behave poorly when there are too many solutions to a problem. Indeed, building a table requires finding all the solutions to a problem. We noticed how for some of the instances of PGC with 25 nodes, there are 65k solutions, including an instance with more than 13M solutions. That explains the result in the last row.

More importantly, we had to rewrite the model for FGC. In the original model, with the reified predicate, building the table is simply impossible. For 8 of the 11 instances
used, the compilation step ran out of memory in our machine. The reason is simple: if the reifying Boolean is turned off, all valuations of the other 6 arguments are valid. The memory explosion happens even with relatively small domains (6 to 8 fox, geese or corn bags), but it does not happen when building MDDs or d-DNNFs. In the case of the MDD this is because the MDD under the decision “turn off boolean” is simply a multi-edged stick (much like the one in Figure 8.2b) where all problems are equivalent. In the case of the d-DNNF, the part under the “turn off boolean” decision is compressed into multi-edged stick with intermediate layers for the assignments of each variable. This can be seen in Figure 8.10.

\[
\begin{align*}
\land & \land f_0 = 0 & f_0 = 4 \\
\lor & \land f_1 = 0 & f_1 = 4 \\
\end{align*}
\]

Figure 8.10: d-DNNF for the FGC example when the boolean in the reification is set to false (dashed lines link to other “∧” nodes, not shown for lack of space)

The time to build the MDDs and d-DNNFs of the 3 instances of FGC that could be tablified is shown in the table (the time for all 11 instances is in parenthesis). The rewritten model for FGC is basically the same as the first one, except that we remove the reification argument from the predicate and apply the constraint to the leftover foxes, geese and corn after the optimal number of trips. The same problem occurred with six of the bigger instances of the PGC problem.

This problem also arises if we construct our data structures without the use of the equivalence keys shown in Section 8.3. For example, for a FGC instance with domains 0..50, it took \(\approx 16.5\) hours to construct the MDD without using cache keys, but only 61 seconds when using them. Both yielded the same MDD, of course. The same behavior
happens for d-DNNFs.

From these experiments we see that building compact structures as MDDs or d-DNNFs can be substantially faster than building the tables, and is rarely slower. More importantly, it is safer in terms of resource consumption. This is why compiling a table and then converting it into an MDD, for example, is not as good as building an MDD via caching: the table needs to be built anyway, which is the bottleneck in this case.

### 8.6.3 Splitting Globals for d-DNNFs

Here we investigate whether the effort made for splitting subproblems is actually worthwhile. Recall that, from what we saw in Section 8.5, this comes practically for free, so we don’t expect any noticeable loss in time. Table 8.2 shows a comparison of the construction of d-DNNFs.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Time</th>
<th>Average d-DNNF Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Using Splitting</td>
<td>No Splitting</td>
</tr>
<tr>
<td>Black-Hole</td>
<td>0.02s</td>
<td>0.02s</td>
</tr>
<tr>
<td>Block-Party</td>
<td>0.28s</td>
<td>0.42s</td>
</tr>
<tr>
<td>FGC (reif)</td>
<td>6.65s</td>
<td>8.18s</td>
</tr>
<tr>
<td>FGC (no reif)</td>
<td>0.21s</td>
<td>0.22s</td>
</tr>
<tr>
<td>Water Bucket</td>
<td>0.44s</td>
<td>0.44s</td>
</tr>
<tr>
<td>PGC (n = 15, c = 6)</td>
<td>7.74s</td>
<td>84.24s</td>
</tr>
<tr>
<td>PGC (n = 25, c = 5)</td>
<td>121.63s</td>
<td>280.67s</td>
</tr>
<tr>
<td>Total</td>
<td><strong>136.97s</strong></td>
<td><strong>374.19s</strong></td>
</tr>
</tbody>
</table>

Table 8.2: Comparison of compilation time and size of d-DNNFs with or without splitting independent subproblems.

Looking at each individual instance, the splitting was not useful for all instances. In fact, for some instances, there were no independent subproblems found at all. This is likely due to the order of the decisions. Indeed, how often a problem can be separated into independent subproblems is very much determined by the order in which decision are made (some decisions induce more splitting).

However, we can clearly see in the PGC problem how beneficial splitting can be. This saved us around 65% of the time and produced d-DNNFs almost 4 times smaller in average.

We could not identify any cue as to when splitting will or not happen, apart from
obvious cases where a single variable unites two very distinct problems. For example, looking at one instance of the PGC problem with 25 nodes that takes 39s with splitting but 93s without it, we could not see anything special about it compared to instances that took similar times with or without splitting. We believe that the VSADS search can sometimes pick the right variables to branch on to create more splits, but as usual with search strategies, they don’t always work perfectly on all instances.

We look now at the task of collecting all Steiner Trees of a graph into a d-DNNF. To do so, we write a MINIZINC model using the tree constraint and we d-dnnffy the binary variables corresponding to edges of the graph. Table 8.3 shows the results for all the instances we tested.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Vars.</th>
<th>Time</th>
<th>d-DNNF Nodes</th>
<th>Ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td>es10fst01</td>
<td>20</td>
<td>0.01s</td>
<td>416</td>
<td>0.90 1.00</td>
</tr>
<tr>
<td>es10fst03</td>
<td>20</td>
<td>0.01s</td>
<td>330</td>
<td>0.60 1.00</td>
</tr>
<tr>
<td>es10fst04</td>
<td>20</td>
<td>0.01s</td>
<td>206</td>
<td>0.87 1.00</td>
</tr>
<tr>
<td>es10fst06</td>
<td>20</td>
<td>0.01s</td>
<td>474</td>
<td>0.56 0.50</td>
</tr>
<tr>
<td>es10fst08</td>
<td>28</td>
<td>0.29s</td>
<td>3277</td>
<td>0.06 0.19</td>
</tr>
<tr>
<td>es10fst09</td>
<td>29</td>
<td>0.30s</td>
<td>3087</td>
<td>0.02 0.02</td>
</tr>
<tr>
<td>es10fst10</td>
<td>21</td>
<td>0.01s</td>
<td>78</td>
<td>0.37 0.21</td>
</tr>
<tr>
<td>es10fst13</td>
<td>21</td>
<td>0.01s</td>
<td>310</td>
<td>0.56 0.55</td>
</tr>
<tr>
<td>es10fst14</td>
<td>32</td>
<td>0.57s</td>
<td>7419</td>
<td>0.05 0.04</td>
</tr>
<tr>
<td>es10fst15</td>
<td>18</td>
<td>0.01s</td>
<td>150</td>
<td>0.86 1.00</td>
</tr>
<tr>
<td>es20fst04</td>
<td>83</td>
<td>&gt;3600.00s</td>
<td>— —</td>
<td>— —</td>
</tr>
<tr>
<td>es20fst05</td>
<td>77</td>
<td>&gt;3600.00s</td>
<td>— —</td>
<td>— —</td>
</tr>
<tr>
<td>es20fst07</td>
<td>59</td>
<td>857.64s</td>
<td>664054</td>
<td>— —</td>
</tr>
<tr>
<td>es20fst08</td>
<td>74</td>
<td>&gt;3600.00s</td>
<td>— —</td>
<td>— —</td>
</tr>
<tr>
<td>es20fst09</td>
<td>42</td>
<td>1.84s</td>
<td>22523</td>
<td>0.18 0.20</td>
</tr>
<tr>
<td>es20fst10</td>
<td>67</td>
<td>&gt;3600.00s</td>
<td>— —</td>
<td>— —</td>
</tr>
<tr>
<td>es20fst11</td>
<td>36</td>
<td>0.02s</td>
<td>409</td>
<td>0.51 0.67</td>
</tr>
<tr>
<td>es20fst12</td>
<td>36</td>
<td>0.10s</td>
<td>2657</td>
<td>0.74 0.71</td>
</tr>
<tr>
<td>es20fst13</td>
<td>40</td>
<td>0.09s</td>
<td>1284</td>
<td>0.18 0.19</td>
</tr>
<tr>
<td>es20fst14</td>
<td>44</td>
<td>2.32s</td>
<td>14795</td>
<td>0.02 0.01</td>
</tr>
<tr>
<td>es20fst15</td>
<td>43</td>
<td>0.10s</td>
<td>1072</td>
<td>0.06 0.08</td>
</tr>
</tbody>
</table>

Table 8.3: Comparison of compilation time and size of d-DNNFs with or without splitting independent subproblems for a problem where the split is done in a tree constraint.

The table does not show instances where the resulting d-DNNFs are identical and therefore splitting was not beneficial. In those instances times were always identical (and the construction was actually immediate). As can be seen, there is never a loss from implementing this splitting technique within global constraints. For example, for the
instance es10fst14 we saved 96% of the time, and 95% of nodes. Big gains can also be seen in other instances like es10fst{08,09} or es20fst{09,13,14,15}. For instance es20fst07, the version with splitting the tree global constraint took 14 minutes, whereas the non-splitting version does not terminate within one hour.

This shows the value of implementing the independent subproblem detection inside global constraints. We conclude this is a good implementation choice since, when it pays off, it does so greatly, and when it does not, there is no loss.

8.6.4 Using the Compiled Structures

We now compare how much having compiled subproblems helps in solving the problem, depending on the data structure used. All instances ran with a time limit of one hour.

Figure 8.11 shows a comparison in solving time between the original model and having mddified or d-dnnffied a predicate (including compilation time). The plots are split in two groups of problems for clarity. The right panels shows results for the 4 sets of instances of PGC, and the left show the other tests.

For MDDs, as we see for the first 5 problems, compiling is clearly beneficial, as the solving time is generally smaller. For the PGC problem we notice a more scattered plot. An interesting point that we show is that there are two sets of instances of 25 nodes. The first one (marked *) averages 184450 nodes for each MDD, whereas the second averages 9402 nodes. This indicates that, despite a fast compilation step (total of 23.3 seconds for all 11 instances on the * set), the sizes of the MDDs are impractical for efficient propagation. This suggests that, after compiling, the user may want to avoid using the resulting MDD if it is too big. Since compiling takes less than \( \approx 3 \) seconds for each of these instances, it is reasonable to try mddifying and then ignore the result if it is too big, or simply abort the compilation. As a comparison, the number of nodes in d-DNNFs of the PGC instances with 25 nodes were, on average 50835 and 2488 for “*” instances and unmarked instances respectively.

Leaving aside these instances, most instances are solved quicker when the MDD is added, and the ones that are not tend to be solved in under 1 second in any case.

For d-DNNFs, the results are a bit less clear. For the first 5 problems, around 57%
Figure 8.11: Time to solve modified instances, including compilation time (y-axis) vs. time to solve original models (x-axis).

of the problems are solved faster using d-DNNFs. It is not clear that d-DNNFs would actually be useful in this case. On the other hand, looking at the hardest instances of PGC, we see that d-DNNF pays off compared to MDDs: most of the hard instances are now solved within 10 seconds with d-DNNFs whereas the original model would solve them in between 100 seconds and 1 hour. We conclude that d-DNNFs are probably more practical for hard problems. Indeed it seems like the biggest payoff (up to 3 orders of magnitude) appears only in the instances that originally took the longest to solve.

Figure 8.12 compares the solving time (including compilation time) when using an MDD or d-DNNF against using a table.

We notice that for all the Block-Party instances, a table is better. Overall the trend is that using an MDDs can save a lot of time, and occasionally lose little time: when
tablifying performs better, the gain is marginal or the instance was solved rather quickly anyway (under 1 second). When the MDD wins, it can make a difference of up to 3 orders of magnitude (c.f. Water Bucket, FGC and some of the 25 node PGC instances).

The results for d-DNNFs are more scattered, but we still see a similar pattern as before: for very big instances, d-DNNFs perform better than tables or original models. This can be clearly seen in the biggest PGC instances, as they are solved 3 to 4 orders of magnitude faster with d-DNNF than with tables.

We now look at a direct comparison between MDDs and d-DNNFs in Figure 8.13. Once again, we notice that MDDs perform better in general, but d-DNNFs seem to be more appropriate for very large instances (like the *d instances of the PGC problem).

Furthermore, out of the 122 instances, the mddified models solved 111, and the d-
8.6 Experimental Evaluation and Results

Figure 8.13: Time to solve mddified instances, including compilation time (y-axis) vs. time to solve d-dnnffied instances, including compilation time (x-axis).

dnnffied models solved 112. The tablified models solved only 87 (out of 108, since 14 of the 122 could not be tablified). The original models solved 108.

Overall, we notice two main results. First, mddifying can yield big MDDs that are impractical. Second, for easy instances, having the extra MDD propagators can be overkill. This technique is therefore more tailored for hard problems that do not result in huge MDDs. The user may also want to experiment with instance-dependent ordering of the variables to achieve more compact MDDs. Interestingly enough, it seems that the pitfall of MDDs is the strength of d-DNNFs: d-DNNFs seem to perform better than the other alternatives when the set of solutions is too big ($\geq 100k$). This brings the possibility of a more robust precompilation technique, where one of the 3 approached (tablifying, mddifying or d-dnnffying) is used depending on an estimate of the size of the solution set for the subproblem to compile. Such estimate could be obtained, for example, using a structural approach [165].

8.6.4.1 Using Compiled Cost-MDDs

In this section we show the results of the Shift Scheduling problem described earlier. We used 8 instances solved with the 5 models described earlier. We ran the original models, mddified models and cost-mddified (using the same MDDs as cost-MDDs) as well as d-dnnffied models.
Table 8.4: Comparison of conflicts, nodes and time for the shift scheduling problem.

The first 5 rows of Table 8.4 correspond to the use of MDDs constructed while all constraints were active (i.e. a quasi-projection). For the last 5 rows only the constraints ensuring the demand of workers is met were active upon mddifying. Columns labeled “Time” correspond to solving time; “Total” shows compilation plus solving time. The table shows geometric means of the ratios of conflicts, nodes and time of the compiled versions over the original version, and the total solving time.

As can be seen, the use of the Cost-MDD completely dominates the other options. The total solving time is enormously decreased, making it worth paying the overhead of constructing the MDDs. For 5 instances where SHIFTWRG performed extremely well (solved in \( \leq 3 \) s), cost-mddifying was not worth it (c.f. geometric means of “Total”), but it still paid-off when measuring the time to solve all instances. This indicates, once again, that for individual instances on which a given model performs well, this technique might not be so valuable. The Cost-MDD version always solved all the instances, whereas the other two versions failed to solve between 1 and 3 of them (depending on the model).
Comparing the first and last 5 rows shows that building MDDs for the quasi-projection reduces the solving time, although computing those MDDs is more costly as there are fewer equivalences. Tablifying this problem produced tables with more than a billion entries, making them impractical.

We believe this result to be very interesting. As we saw in the presentation of the models we used earlier, the SHIFTWRG uses a cost-MDD to enforce the shape of a shift. That is, there are already cost-MDDs present in that model, but one per worker. Thanks to our approach, we could create (cost-)MDDs for sets of days, across all workers. This is not trivial to do by hand, and it is not obvious that it would produce such a big advantage as it does, so a modeler might decide to not make that effort. Thanks to our approach, it was possible to build those MDDs and we show that they have a huge value for the model when used as cost-MDDs. It would be possible that this finding also translates to a weighted version of d-DNNFs, but sadly we do not have such a propagator with explanations available in CHUFFED. But this gives a very interesting direction for future work.

8.7 Concluding Remarks

In this chapter we have dealt with the use of graph-based structures in CP solvers as a resource to accelerate them. Namely, we have developed an automatic compilation from (parts of) CP models into MDDs and d-DNNFs.

Our experiments and those by Dekker et al. [63] show that compiling part of a problem to an MDD, d-DNNF or table can be beneficial for the total solving time of the problem. Indeed, these compilation into some data structures can be used by specific propagators (table, MDD and d-DNNF propagators) to produce stronger propagation, thus reducing the search space. Using our choice of data structures is not always better than using tables (there is certainly a role for tablification), but can be substantially more efficient.

We have identified the limitations of using tables as the resource for compilation, and proposed the use of precompilation with MDDs and d-DNNFs. We show how this
technique is flexible, since the MDDs can be reused as Cost-MDDs, as well as robust. Indeed, our approaches never ran out of memory, and the compilation times are better than with tables.

The result shows that for small problems, building tables is the right choice. For bigger problems, MDDs offer a scalable approach that, combined with a good MDD propagator, can be beneficial to the solving time. For the biggest instances, we saw that d-DNNFs are a better choice, as they are generally smaller.

Our contributions in this chapter are:

- The use of problem equivalence detection techniques to a new purpose: model compilations.
- New caching key for the tree constraint.
- A new way of handling fixed variables when compiling MDDs compared to previous work.
- An new algorithm to construct d-DNNFs from CP models that involve globals.
- Subsequently, a new approach to split globals, exemplified by an algorithm we introduce to split tree to reduce the size of the d-DNNFs.
- An extensive set of experiments comparing tablification, compilation into MDDs and d-DNNFs as well as an example with Cost-MDDs where we show the undeniable value of this technique.
Chapter 9
Conclusion

In this thesis we have contributed to the field of Constraint Programming by developing new algorithms for propagation rules and explanations for propagators. These algorithms allow for faster solving times when tackling combinatorial problems involving graphs with this technology. We have shown, through experiments, how no-good learning is extremely valuable for Constraint Programming solvers.

It is common to see the Steiner Tree Problem in computer network or VLSI design, as we have seen in Chapter 3. This is an NP-complete problem in itself, and although some efficient algorithms exist to simplify the problem or find approximations, these do not apply to more complex problems, where the Steiner Tree is only a subpart of the problem and side constraints participate in deciding the topology of the tree. Through new algorithms that ensure a graph variable is a tree, we have allowed Constraint Programming solvers to better reason about graph structures. Thanks to the work in this thesis, problems involving the design of a tree in a graph can now be approached better with Constraint Programming. We saw how specially no-good learning was key to achieve this performance. Furthermore, in most cases, graphs are weighted and the weight of the edges needs to be taken into account. We developed new lower bounding techniques, with no-good learning as well, to propagate inference about the weight of the resulting tree.

An application of the work on Steiner Trees was presented in Chapter 4 showing the immediate impact that this work can have in real life applications. This was an application-oriented research focused on solving the Relational-To-Ontology Mapping problem, which is quite common in industry (for example, during company acquisitions).
It is often the case that different organizations will have different representations of their data in relational databases. When this data needs to be shared across organizations or merged together, the user is confronted with heterogeneous, unnormalized data that needs to be merged. Through the use of Machine Learning and Constraint Programming we were able to develop a tool that could approach this problem and effectively predict where a new data source would fit in the set of already existing relational tables. This chapter shows that combining two usually independent research areas can yield the best results. The collaboration between the Machine Learning and Constraint Programming communities are perhaps too infrequent, and this research shows the value that both communities can get from each other.

A special case of the Steiner Tree Problem is the one where all the nodes in the graph need to be connected through the tree. This is quite a common problem, for example in electric networks because all the components need to be connected. We saw the example of offshore wind-farms, amongst others, in Chapter 5. This is known as the Weighted Spanning Tree problem, which is closely related to the Minimum Spanning Tree problem. The difference between the two is that more often than not, we want to find a tree of minimum weight given some restrictions. These restrictions are side constraints that typically make the problem NP-hard. These problems are a perfect target for Constraint Programming solvers. In order for the solver to perform well, we need to enhance it through specialized propagators for this constraint. We already had most of the components from Chapter 3 to ensure the graph was a tree, and previous work had produced propagation rules for this problem. What was missing was a key component: no-good learning. In this chapter we developed explanations for propagation rules on the weight of the tree. These proved to be crucial to achieve state-of-the-art performance, and were our main contribution.

The last graph propagator we worked on in this thesis was the bounded path propagator, in Chapter 6. Building this propagator required multiple other auxiliary ones, some of which were quite similar to propagators from Chapter 3 but adapted to directed graphs. We also developed a propagator ensuring that a graph is a DAG, which is not a particularly common problem, but was a corollary of our work. We evaluated our ap-
proach for paths, based on previous work by Fages [84] for the first time with no-good learning. This addition of explanations was our main contribution and was once again a key element to achieving competitive speed. The set of experiments we used was extensive and showed the flaws of our approach in that simple problems or topologically hard problems can be tackled better with other approaches. Yet we showed that for problems involving paths with distances where the feasibility of the problem relied heavily on the length of the path, our approach was the state-of-the-art thanks to no-good learning.

Our next motivation was to tackle a problem in computational sustainability where we could use Constraint Programming and some of our propagators. As it often happens in research, we looked for a problem and found another one. Chapter 7 presents a new problem in habitat conservation. It deals with making decisions as to where, on a landscape, a higher impact can be achieved for the survival of a species by some human action (such as reforestation or animal-crossing construction). This problem is related to graphs in that, surprisingly, biologists model the studied landscape as a graph. It is in fact an electric network of resistors that models the movement of the species across the land, where the value of the resistors serve as a measure of the difficulty of movement in a specific area. As this problem is hardly constrained and is almost a pure optimization problem, Constraint Programming was not the best tool. After unsuccessfully trying a Mixed Integer Programming approach, we created an ad-hoc Local Search algorithm to tackle this problem. We showed that this performed extremely well and the decisions made by the system could reduce the landscape resistance substantially. We also showed that the Local Search approach was robust and scalable.

The last chapter brought us back to Constraint Programming, but this time, instead of focusing on solving graph problems involving graphs, we investigated graphs as a tool for Constraint Programming solvers. Previous work had shown the value of preprocessing parts of Constraint Programming models to improve the overall runtime of the solver. These approaches, although valuable (as we also showed in our experiments) do not scale well. We therefore presented two new techniques for preprocessing the model by presolving parts of it and storing the sets of solution in compact structures. We chose Multi-valued Decision Diagrams and Decomposable Deterministic Negation
Normal Form Formulae, which both can be represented in the form of graphs. These data structures allowed for compact representation of the sets of solutions of subproblems and the propagators that already existed for them proved the value of this technique. Indeed, our experiments showed that presolving subproblems could drastically improve the overall performance of the solver. This was even more visible in the case of cost-MDDs, where the gains were truly remarkable. The experiments showed that this technique should not always be used, though. We gave estimate sizes of solution sets for which the previous work by Dekker [62] using tables, or ours using MDDs or d-DNNFs give the best performance.

Finally, this thesis has shown three main things. First, no-good learning is essential to achieve fast results in Constraint Programming, and is most definitely worth the implementation effort in solvers. Secondly, the collaboration of Machine Learning and Constraint Programming can be a huge success and should be considered more often, as should the combination of Constraint Programming with other research fields. And lastly, there are important problems in computational sustainability worth investigating due to their impact and because they can be extremely challenging.


[65] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. Towards a better understanding of the functionality of a conflict-driven SAT solver. In International


[128] Ailsa H Land and Alison G Doig. An automatic method of solving discrete pro-

Broeck, and Siegfried Nijssen. Combining stochastic constraint optimization and
probabilistic programming. In International Conference on Principles and Practice of


[131] Ronan LeBras, Bistra N Dilkina, Yexiang Xue, Carla P Gomes, Kevin S McKelvey,
Michael K Schwartz, Claire A Montgomery, et al. Robust network design for multi-

[132] Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay

[133] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding domi-
nators in a flowgraph. ACM Transactions on Programming Languages and Systems


[135] Kevin Leo, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Glob-
alizing constraint models. In Christian Schulte, editor, International Conference on
Principles and Practice of Constraint Programming, volume 8124 of Lecture Notes in

[136] Xiao Yu Li, Matthias F Stallmann, and Franc Brglez. A local search SAT solver using
an effective switching strategy and an efficient unit propagation. In International


[166] André Petermann, Martin Junghanns, and Erhard Rahm. Dimspan-transactional


[204] Sabine Storandt. Route planning for bicycles-exact constrained shortest paths made


[221] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the International Conference on Data Mining*, pages 721–724. IEEE, 2002.


Author/s: De Uña, Diego

Title: Discrete optimization over graph problems

Date: 2018

Persistent Link: http://hdl.handle.net/11343/217321

File Description: Discrete Optimization over Graph Problems

Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.