DYNAMICALLY WEAKENED CONSTRAINTS IN BOUNDED SEARCH FOR CONSTRAINT OPTIMISATION PROBLEMS

CRAIG REILLY

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

NOVEMBER 2020

Abstract

Combinatorial optimisation problems, where the goal is to an optimal solution from the set of solutions of a problem involving resources, constraints on how these resources can be used, and a ranking of solutions are of both theoretical and practical interest. Many real world problems (such as routing vehicles or planning timetables) can be modelled as constraint optimisation problems, and solved via a variety of solver technologies which rely on differing algorithms for search and inference.

The starting point for the work presented in this thesis is two existing approaches to solving constraint optimisation problems: constraint programming and decision diagram branch and bound search. Constraint programming models problems using variables which have domains of values and valid value assignments to variables are restricted by constraints. Constraint programming is a mature approach to solving optimisation problems, and typically relies on backtracking search algorithms combined with constraint propagators (which infer from incomplete solutions which values can be removed from the domains of variables which are yet to be assigned a value). Decision diagram branch and bound search is a less mature approach which solves problems modelled as dynamic programming models using width restricted decision diagrams to provide bounds during search.

The main contribution of this thesis is adapting decision diagram branch and bound to be the search scheme in a general purpose constraint solver. To achieve this we propose a method in which we introduce a new algorithm for each constraint that we wish to include in our solver and these new algorithms weaken individual constraints, so that they respect the problem relaxations introduced while using decision diagram branch and bound as the search algorithm in our solver. Constraints are weakened during search based on the problem relaxations imposed by the search algorithm: before search begins there is no way of telling which relaxations will be introduced. We attempt to provide weakening algorithms which require little to no changes to existing propagation algorithms.

We provide weakening algorithms for a number of built-in constraints in the Flatzinc specification, as well as for global constraints and symmetry reduction constraints. We implement a solver in Go and empirically verify the competitiveness of our approach. We show that our

solver can be parallelised using Goroutines and channels and that our approach scales well.

Finally, we also provide an implementation of our approach in a solver which is tailored towards solving extremal graph problems. We use the forbidden subgraph problem to show that our approach of using decision diagram branch and bound as a search scheme in a constraint solver can be paired with canonical search. Canonical search is a technique for graph search which ensures that no two isomorphic graphs are returned during search. We pair our solver with the Nauty graph isomorphism algorithm to achieve this, and explore the relationship between branch and bound and canonical search.

Acknowledgements

Initial thanks must go to my supervisor Professor Alice Miller for her support and advice throughout both my time as a PhD student and the writing of this thesis. I would also like to thank Dr Patrick Prosser, my second supervisor, for his words of encouragement; without taking his course during my Masters I would not have chosen to do a PhD. For their guidance and encouragement at my progression meetings I would like to thank Professor David Manlove, Professor Simon Gay, Dr Kitty Meeks and Dr Simon Rogers. I would also like to thank my examiners Dr Steven Prestwich and Dr Gethin Norman for their helpful comments and advice which certainly improved this thesis.

I forgive my brother Dr Colin Reilly for getting assigned the better email address before I did. Thanks goes to my dad for letting me explain so many of my coding woes to him and to my mum for putting up with this when it happened at the dinner table.

For professional reasons I should thank Ciaran McCreesh and Ruth Hoffmann for being paper coauthors, but I would rather thank them for being better friends and whisky connoisseurs. Still, it was nice of you two to let me be the one to go to San Francisco to present our work.

Thanks to Heather Laughland for the wine, hip hop and psych in the bouldering gym. To everyone with whom I played Dungeons and Dragons till the wee hours of the morning in some seminar room or another, I hope to play with most of you again.

Most of all I thank Murphy, Darcy and Lucey. Completing this thesis would have been even harder without you three.

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/M506539/1], and I am grateful to have received conference and summer school travel grants from AAAI and the School of Computing Science.

Contents

CONTENTS

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The focus of the work presented in this thesis is to adapt a decision diagram branch and bound, DDBB, search algorithm due to Bergman, Ciré, van Hoeve, et al. (2016) for use as the search scheme in a constraint optimisation solver. The existing algorithm uses width truncated decision diagrams to provide bounds during search when solving problems modelled using dynamic programming models. The DDBB search algorithm uses relaxed decision diagrams to provide upper bounds (when the objective is to maximise a cost function) and these rely on merging nodes to maintain a maximum allowed width. The approach to merging nodes (both the order in which to choose pairs and how to merge them) is provided by the modeller and must guarantee that the resulting relaxed decision diagrams provide an upper bound.

Our approach to adapting this algorithm for use in a constraint solver is to ensure that constraint propagators respect problem relaxations introduced by merging nodes. To achieve this we require an algorithm for each constraint to weaken it, based on problem relaxations introduced by a single algorithm we provide for merging nodes.

We implement two solvers to empirically evaluate our approach. A general purpose solver which is written in Go and a C++ implementation which is suitable for graph search problems.

In this chapter we begin by giving an informal overview of many of the core concepts used throughout the thesis. We leave formal definitions for later chapters and also defer more careful consideration of related work, handling this within later chapters.

We end this chapter with an overview of the structure of this thesis and a brief summary of the material included therein.

1.1 Combinatorial problems

In this section we give an overview of what combinatorial problems are, what combinatorial optimisation problems are, how these problems can be modelled and how they can be solved algorithmically.

1.1.1 What are combinatorial problems?

The goal when solving a combinatorial problem is to find an assignment of a discrete and finite set of objects which satisfies some given conditions. A combinatorial optimisation problem extends this idea by aiming to find an optimal assignment of all possible assignments which satisfy the given conditions (that is, the best assignment with respect to some ordering of solutions). Informally, a combinatorial problem involves a collection of resources and some constraints regarding how these resources can be used. As an example, these constraints could be that resource a must be consumed before resource b, but after resource c. A solution is then some choice of how these resources are used in a way which satisfies all of the constraints imposed. For example, we may have a problem where our resources are packages which have to be delivered within a certain time frame. Such problems are common in industry and everyday life. As an industrial example, there may be a need to move materials down a supply chain by boat in the petrochemical industry; Giles and van Hoove (2016) study such a problem and solved it using Constraint Programming, where the problem is to schedule the movement of boats such that each location in the supply chain is kept stocked with sufficient materials. This kind of combinatorial problem is called a satisfaction problem, where all solutions to the problem are equally valid. In some cases we might be interested in how many solutions there are to a problem, this is referred to as an enumeration problem. In this work we are most interested in combinatorial optimisation problems. Finding the "best" solution to a problem involves scoring solutions so that we can pick the solution which scores highest. Note that this often does not involve simply enumerating solutions and then ordering them to find the best answer to a problem. Instead many solver technologies will prune areas of search which have no chance of containing the best solution. Giles and van Hoove also study the optimisation variant of this supply chain delivery problem where the size of the fleet of boats is minimised. In everyday life such pick up and delivery problems are ubiquitous. An individual might take an Uber home from work to have Deliveroo bring them their dinner. O'Neill and Hoffman (2018) study these types of dynamic pickup and delivery problems, and compare multiple solver technologies against one another when solving them. The optimisation criteria for such problems could be that wait times for customers are kept to a minimum, or the fuel consumption of an entire fleet of delivery vehicles is minimised.

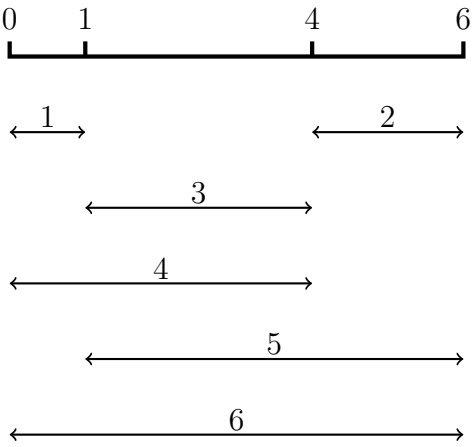

Figure 1.1: An optimal (and perfect) Golomb ruler of order 4.

As an example problem consider the problem of finding optimal Golomb rulers. A Golomb ruler is an ordered sequence of integers a_i which represent *marks* on an imaginary ruler. The distance between any pair of marks on the ruler must be distinct from the distance between any other pair. The number of marks n on the ruler is its *order* and the *length* of the ruler is the largest distance between any pair of marks. A Golomb ruler is *optimal* if no Golomb ruler of smaller length with the same order exists. Figure 1.1 shows the perfect, optimal Golomb ruler of order 4 which has length 6.

In this problem the resources are the marks and the spaces in which we can place them and the constraints are that no mark can share the same space as any other and the gap between all pairs of marks is unique. We will return to this problem when showing how constraint optimisation problems can be modelled.

1.1.2 Types of combinatorial problem

There are many different different classes of problems which all fit the definition of a combinatorial problem. Examples include constraint satisfaction problems (CSPs), integer programming problems, linear programming problems, mixed integer programs (MIP), boolean programs, pseudo-boolean programs, boolean satisfiability (SAT), quadratic programs, and more. We will focus on solving CSPs using constraint programming in this thesis, but we will introduce some of the other classes here.

A constraint satisfaction problem consists of a set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ which take values from a set of finite, non-empty domains $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ and a set of constraints $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$. Variables are assigned values from their domains and a complete assignment is one in which every variable is assigned. Each constraint is imposed over a (possibly improper) subset of the variables and defines which combinations of value

assignments the subset of variables that satisfies the constraint. A complete assignment which satisfies every constraint in C is a solution to the CSP. A more formal definition is given in Definition 2.1.6.

An integer programming problem is a CSP where the domains are restricted to be integers. Often integer programming is assumed to be integer linear programming, where the constraints and objective function (which allows solutions to be ranked) are linear. In the case where some of the domains are not discrete, a problem is known as a mixed-integer programming problem.

In a boolean satisfiability problem the domains are restricted to be boolean values. The constraints of the problem are expressed as a boolean formula (often expressed in conjunctive normal form). Each of these classes of problem is solved with a different solver technologies relying on a different set of underlying algorithms for search and inference.

1.1.3 What is constraint programming?

To solve a problem using constraint programming describes both the process of modelling a problem and solving it. Modelling a problem using constraint programming involves writing declarative code which describes the problem using variables, constraints between variables, an objective function (if one exists) and any search heuristics to guide search. The allowed types of variables and constraints can vary depending on the constraint solver which is to be used to find a solution to the problem. A constraint solver then takes this declarative model and uses a combination of search and inference algorithms to solve the problem. Constraint programming is often used as a catch all term for this entire process.

Some standard constraint modelling languages exist such as Minizinc (Nethercote et al., 2007a, Stuckey et al., 2014), Essence (Frisch, Grum, et al., 2007), NumberJack (Hurley and O'Sullivan, 2016) and JuMP (Dunning, Huchette, and Lubin, 2017). As well as these dedicated languages often constraint solvers will read models which are declared in programs written in general purpose computing language, with the model being declared in a format as set out by the solver's API. The Choco (Prud'homme, Fages, and Lorca, 2017) constraint solver is one such solver, which is written using the Java programming language.

Modelling a problem using constraint programming does not tie the modeller down to using a constraint solver to find solutions. For example if a problem only includes linear constraints on integer variables then it may be be more efficient to solve the problem using a linear programming solver. Most of the constraint modelling languages listed above can be used to interface with other solver technologies such as MIP solvers or SAT solvers.

1.1.4 Modelling a combinatorial problem

In previous sections we gave a short overview of various classes of combinatorial problems and a brief overview of constraint programming. In this section we model the problem of finding optimal Golomb rulers described in Section 1.1.1 as a constraint optimisation problem.

The optimal Golomb ruler problem of a given order n can be modelled by creating a variable x_i for each mark to be placed, and the domain of each variable is then a set of integers representing the spaces along the ruler in which the mark represented by x_i can be placed. We can then impose that each variable x_i must take a unique value to ensure that no two marks can be placed in the same place. To ensure that the gap between each mark is distinct, we can create a new variable $g_{i,j}$ to represent each gap and impose the constraint that $g_{i,j} = x_j - x_1$ for each i and j such that $1 \leq i < j \leq n$ and require that each $g_{i,j}$ takes a different value. To find the optimal ruler of order n we also need to ensure that the maximum value taken by any $g_{i,j}$ is minimised.

1.2 Solving combinatorial problems

In this section we focus on how combinatorial problems are solved, with a particular focus on Constraint Programming. In Constraint Programming problems are solved by alternating between search algorithms and inference algorithms (called constraint propagators) which determine if constraints are satisfiable or not. In the case where constraints are satisfied, these inference algorithms remove values from the domains of variables which can never be included any assignment.

1.2.1 Search

Search describes the process of systematically making a sequence of choices to solve a problem. Although it would be intractable, we could solve an instance of the optimal Golomb ruler problem by trying out every combination of placement of marks reviewing them to determine which satisfy all the constraints of the problem. This approach is known as generate and test and is wildly inefficient for a problem of anything but trivial size.

Another option for solving the problems is to take a small partial solution which we know to be good and add to that. For example, we could place the first mark on the ruler at position 0 and the next at position 1. We then attempt to place the next mark at the next available position, but placing it at position 2 breaks the constraint that all distances between marks are distinct. In this case we backtrack and try a new value for the placement of the third mark.

This backtracking continues if there are no more valid options to place a mark, and a new choice for the subsequent mark is tried. This process continues until a valid solution is found or there are no variable-value combinations are left to try. A key advantage of this approach is that we no longer generate assignments which are invalid; every solution presented by backtracking search satisfies all of the constraints imposed in a problem.

We can view search as a tree, with the root node containing the state of the problem where no variables have been assigned values. Each valid decision we make regarding a value assignment to a variable augments the state with this new assignment and creates a child node with the new state. When there are no variables remaining which have not been assigned a value then we are at a leaf node of the tree and have found a solution. The order in which the search tree is constructed is very important when solving problems.

Depth first search chooses the deepest node where all value assignments to a variable have not yet been tried and attempts to make the next value assignment to a variable. In constraint programming this means after we assign a variable a value we move on and attempt to assign another variable a value which does not conflict with any other previous assignments. In practice this is a very efficient way to conduct search as only the path to the current search node and a record of previous assignments at each node along the path are needed when conducting a depth first backtracking search.

Breadth first search (E. F. Moore, 1959) builds the search tree one layer at a time, choosing the shallowest node to the search tree to continue from. Again, assuming that the order we choose variables in is fixed, this means we try all possible value assignments for a variable before continuing with search. This approach is not as efficient as depth first search as it requires all the nodes in the deepest layer of the search tree to be retained in memory. This restriction means that it is almost always impractical to conduct search in this way.

An additional drawback to breadth first search is that it has to explore (almost) the whole search space before it finds any solutions (Kozen, 1992). In contrast depth first search finds its first solution at the first valid solution it arrives at. For satisfiable decision problems (problems where a solution exists) the behaviour of depth first search is clearly preferable. However for unsatisfiable (problems where no solution exists) problems the whole search space has to be explored to find that a solution does not exist, so it is only the memory requirements that lead to depth first search being preferable.

A common search scheme in constraint programming is forward checking (Haralick and Elliott, 1980), where each proceeds in a depth first manner and at each node in the search tree each constraint involved in the problem being solved is propagated. Constraint propagation removes values from the domains of unassigned variables if they cannot be included in any solution. We discuss constraint propagation in the following section.

A drawback to depth first search is that often choices made early on in search preclude a solution. Search then thrashes, backtracking to repeatedly try all variable value combinations deeper in the search tree while the conflicting assignments are higher up the tree. Prosser (1995) introduces conflict directed backjumping, which seeks to mitigate this problem by having search backtrack straight to the deepest conflicting assignment rather than to the previous search node. While this does not remove the prospect of thrashing it does mitigate the issue in many cases.

1.2.2 Inference and constraint propagation

Often we can do better than just searching by making a sequence of guesses. A success of constraint programming is that solvers do not treat the state at a search node as a black-box that can only determine if a variable can be given an assignment. Instead, it may be possible that there is scope for inference based on the constraints which are imposed on a problem and the current assignment of values to variables. Consider the problem modelled in Figure 1.2 which involves two integer variables and a single constraint between them. If we have assigned x the value 5, then we can tell immediately based on the constraint that the largest value y can take is 2 (while $x = 5$), without ever needing to attempt to assign y the values 3, 4 or 5 through search. This kind of inference which cuts down the search space can be a powerful tool when we try to solve problems as quickly as possible.

$$
\begin{aligned}
&x \in \{0, \ldots 5\} \\
&y \in \{0, \ldots 5\} \\
&x + y \leq 7
\end{aligned}
$$

Figure 1.2: An example problem.

While inference can be powerful, choosing the wrong constraints to model a problem can lead to increased search effort. Two equivalent models of the same problem which rely on different constraints can require different numbers of search nodes (steps where we have to make a decision about what to value to assign to a variable) to find a solution. Consider the example in Figure 1.3, which shows a problem on four binary variables which are all expected to take different values. It is clear that this problem is not satisfiable just by looking at it, there just are not enough values to go around. Before assigning any variables a value we can determine that the problem is unsatisfiable. However, while the model given in Figure 1.4 models the same problem as in Figure 1.3 the same level of inference is no longer possible. Before search proceeds the `not-equals` constraints do not allow any values to be pruned from the domains of x_1, x_2, x_3 and x_4.

$x_1 \in \{0, 1\}$
$x_2 \in \{0, 1\}$
$x_3 \in \{0, 1\}$
$x_4 \in \{0, 1\}$
$\texttt{alldiff}(x_1, x_2, x_3, x_4)$

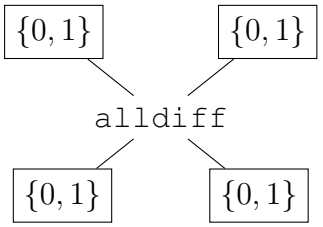

Figure 1.3: An example problem with a single `alldiff` constraint, which is not satisfiable.

$x_1 \in \{0, 1\}$
$x_2 \in \{0, 1\}$
$x_3 \in \{0, 1\}$
$x_4 \in \{0, 1\}$
$x_1 \neq x_2$
$x_1 \neq x_3$
$x_1 \neq x_4$
$x_2 \neq x_3$
$x_2 \neq x_4$
$x_3 \neq x_4$

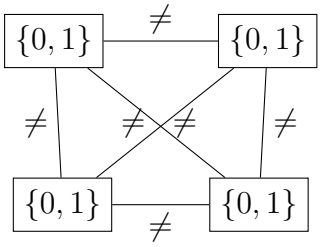

Figure 1.4: The same example problem where the `alldiff` constraint is decomposed.

1.2.3 Consistency and levels of consistency

In the previous section we demonstrated that propagating constraints to remove values from variable domains can be useful, but in practice the amount of effort that goes into propagation can have a large effect on the execution times of solvers.

An assignment of a value to a variable is consistent if if does not violate any constraints. Consider a simplified situation where there exists only binary constraints between pairs of variables x and y. A value v in the domain of a variable x has support if there exists a value w in y such that the assignment $x = v, y = w$ satisfies all constraints between x and y. In the case where every value in the domain of x and y has support, then x and y are said to be arc-consistent. A problem is arc-consistent if and only if each variable is arc-consistent with every other variable. This concept extends to constraints which involve any number of variables, and this case is called generalised arc-consistency.

Often it is advantageous to only keep track of the bounds of an integer variable (that is the smallest and largest values that it can be assigned), rather than record all of its possible values. While this has advantages regarding the amount of memory a solver needs to function, many constraint propagators also work by tightening the bounds of variables and do not remove all values without support from the variable domains. For example, in the problem shown in Figure 1.2 setting the value of x to be 5 means that we can immediately determine that the upper bound of y is 2. Two variables x and y are said to have interval support if there exists a pair of values s and t between the bounds of x and y respectively such that s and t satisfy all constraints between x and y. When support is only determined from the bounds of variables in this way a problem is said to be bounds consistent.

1.2.4 How hard is it to solve problems?

Combinatorial problems are difficult, both in theory and often also in practice. While we will not cover computational complexity (Goldreich, 2010) and (Garey and Johnson, 1979) could be used for reference. Many decision problems are NP-complete (Goldreich, 2010), which means that there (probably[1]) does not exist an efficient algorithm to solve these problems. Individual instances of such problems can become exponentially more difficult to solve as the number of variables involved increases. This could quickly lead to instances which take very very long to solve. In fact, linear programming and boolean satisfiability are standard NP-complete problems. To prove that a problem is NP-complete a reduction (a mapping from one problem to another) to one of these problems suffices.

[1] Unless P=NP

In general optimisation problems are even harder than decision problems. To determine if an assignment is a solution to a satisfaction problem all that is required is to check all of the values assigned to all of the variables satisfy each constraint involved in the problem. It might be very difficult to find this solution but it is simple to check that it is a solution to the problem being solved. This ease of verification does not extend to optimisation problems. While we can tell that any solution offered by a solver is in fact a solution, in general there is no reliable way to prove that it is is the best solution. So there is some trust on the part of the user that a solver does in fact explore the whole search space to determine which solution is optimal, or that it is correct when it chooses to prune areas of the search space where it expects to find no solutions which are better than current best solution found by the solver.

Some solver technologies overcome this issue of having to trust that the solver has correctly determined a solution to be optimal by producing a log during search. This log can then be checked to make sure that the solvers reasoning throughout its execution was sound. While this approach to proving the correctness of solvers is standard practice in SAT solvers, (Cruz-Filipe, Marques-Silva, and Schneider-Kamp, 2017),(Goldberg and Novikov, 2003) and (Heule, Hunt, and Wetzler, 2013), it is an currently a new area of research for constraint solvers (Gocht, McCreesh, and Nordström, 2020).

1.2.5 Heuristic choices during search

One method for combating the hardness of combinatorial problems is by using search heuristics. Search heuristics are choices which guide search in an attempt to find solutions more quickly. Best first search is a search method like depth first and breadth first approaches described in Section 1.2.1 where the next node to branch on is chosen by a heuristic which scores the viability of nodes of the search tree. Best first search branches on the node which is heuristically determined to be the most likely to lead to a good solution (that is, any solution for satisfaction problems and a high cost solution for optimisation problems).

In the case of constraint programming the search heuristics cover the order in which variables and values should be considered. An example variable heuristic could be to choose the variable with the smallest domain first (Haralick and Elliott, 1980). In practice this heuristic works well in many cases, and is one of two search heuristics suggested to be included in all solvers which implement a Flatzinc[2] parser (the other is to consider the variables in a specified static order).

Heuristics are not guaranteed to always lead to the best search order (the order which requires fewest search nodes to find a solution), but in practice a good choice of heuristic can have

[2]Models written in Minizinc are compiled to Flatzinc models which are then read by solvers.

a large affect on the time it takes for a solver to find a solution. For optimisation problems heuristics can effect the time it takes to find a good incumbent solution (the best solution found yet as search proceeds). Finding a better incumbent solution sooner can lead to more pruning of the search space.

1.2.6 Approximate methods of search

One approach to solving optimisation problems is to avoid complete exact search altogether. There are many incomplete and approximate search algorithms which trade always finding a solution, or the optimal solution for optimisation problems, with faster execution times than an exact search. One common incomplete approach to search is local search, which starts off with a random assignment of values to variables and makes changes to this assignment that improve its cost. The cost in of an assignment in this context is based on the number of constraints that it violates. GSAT (Selman, Levesque, and Mitchell, 1992), Tabu Search (Glover, 1990) and Simulated Annealing (van Laarhoven and Aarts, 1987) are examples of local search algorithms.

Another type of incomplete search algorithm is an approximation algorithm (Garey and Johnson, 1981). These algorithms seek solutions which are nearly optimal, which means that there are guaranteed bounds on how far from the true optimal value their answer will be.

Yet another approach to limiting the difficulty of problems is by Fixed Parameter Tractability (Downey and Fellows, 1995), which makes assumptions on the structure of the input data to a problem in an attempt to leverage problem structure.

1.3 Graphs

A graph is a set of vertices together with a set of edges which connect vertices. We will define graphs more formally in chapter 6. Problems can be modelled by graphs. For example if we wanted to find the largest group of people who are all friends with each other on a social media platform we could model each person as a vertex, each friendship as an edge between vertices. Finding the largest set of vertices such that each vertex is connected to every other vertex is a problem called the maximum clique problem.

Graphs are not only used to model problems, but can also be the objects that we consider when solving problems too. We will consider such a graph search problem in chapter 6.

1.4 Decision diagrams

This thesis is based around including a decision diagram branch and bound algorithm in a constraint solver. A binary decision diagram is a special type of graph, called a tree, which has a root node and two terminal nodes. Nodes in a binary decision diagram are arranged in layers and each node might have up to two children. Binary decision diagrams found great success through their use as a compact data structure to represent boolean formulae (Bryant, 1985). Figure 1.5 gives an example of a search tree and a BDD representing the possible combinations of a small boolean formula involving three variables, which evaluates to true if and only if example two of the variables take the value true.

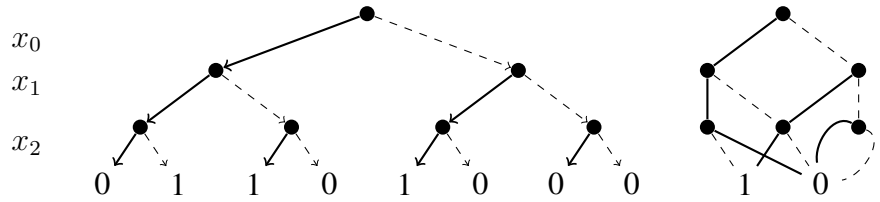

Figure 1.5: The tree and BDD associated with a boolean function on three variables which evaluates to true if exactly two variables are set to true. Solid lines represent the variable being assigned the value $\mathbf{1}$ and dashed $\mathbf{0}$.

1.5 Experimental setup

In this section we outline our setup for undertaking empirical evaluation of the performance of the solvers which we implement to test our algorithms. We cover our choice of both hardware and software, as well as our experimental workflow from instance generation to solution.

1.5.1 Solvers implemented

We take results from two solvers which we have implemented. One is written in Go and is a general purpose constraint solver, while the other is focussed specifically on graph search problems and is written in C++. Our Go solver makes use of Go version 1.12.5 and our C++ solver was implemented using C++14 and compiled using GCC 6.3.0 at optimisation level `-O3`. We defer in depth discussion of the implementation details of these solvers to chapters 5 and 6.

The use of Go as our choice of language to implement our solver means that we have to take care when running multiple instances of the solver concurrently on one machine. This is due

to Go's concurrent garbage collector. To ensure that multiple Go programs do not fight over resources we can limit the maximum number of processes which our solver can use.

1.5.2 Hardware used

We conducted our empirical analysis of our solvers using a cluster of machines. Each machine in the cluster is set up with two Intel Xeon E5-2697A v4 processors, 512GBytes of RAM and was running the Ubuntu 17.10 operating system. Each machine has 32 real processor cores and 64 hyper-threads. Hyper-threading is a feature on modern Intel processors where each real core is treated as two cores. When we present results obtained from running parallel versions of our solvers we often find that scaling drops off once the number of parallel workers out-strips the number of real cores available. This behaviour is to be expected, since hyper-threading can often only give an 20-30% improvement in performance (and sometimes gives none at all) (McCreesh, 2017). We did not make use of another feature of the Intel CPUs, i.e. turning off their "Turbo Boost" functionality. This feature allows the CPU to ramp up its clock speed when factors like cooling and system energy consumption allow. While this feature is useful in a desktop computing context it can skew our results when measuring the execution times of our algorithms.

1.5.3 Instance selection and format

All of the models which are solved by our Go solver are written in the Minizinc constraint modelling language (Nethercote et al., 2007b). Minizinc is a high level modelling language which allows for rich models to be expressed succinctly. The benefit of using the language is that it allows the modeller to only write a model once, rather than translating into multiple programming languages to produce programs which interface with solver specific APIs. Minizinc is not the only such high level language, with Essence (Frisch, Grum, et al., 2007) also gaining traction. We store data which varies by problem instances as `.dzn` files, the Minizinc data file format, and compiled problem instances to Flatzinc with the standard Minizinc compiler. Flatzinc is a solver input language which lists all constant values and decision variables, and arrays thereof, as well as all the constraints which define a problem instance. Although some solvers do implement their own Minizinc to Flatzinc compilers in an effort to speed up overall execution we have not taken this route. Where possible the models were also not written by us and we make use of some models from CSPLib (Jefferson et al., 1999).

Where we have generated random instances the goal was to make sure that, while none were trivially unsatisfiable, both satisfiable and unsatisfiable instances were generated. To remove

trivially unsatisfiable instances we discarded all instances which the Minizinc to Flatzinc compiler could prove were unsatisfiable.

1.5.4 Performance metrics

When we report execution times for solvers in this thesis we report the time it takes for search to be conducted. This allows us to focus on improving the performance of search instead of optimising the reading of models from files, which can be often be large[3]. Both of our solvers (written in Go and C++) use a monotonic clock when determining how long search takes. This clock is guaranteed to be monotonically increasing and is unaffected by changes to the system's wall clock during the execution of our solvers.

When we assess the size of the search space we do so by counting the number of search nodes used. We do not report the amount of memory used by our solvers. Our constraint solver implemented in Golang at times use a lot of system memory, requiring orders of magnitude more space in RAM than commercial solvers. We suspect this is partly down to our inability to write very efficient Go code, so we stick to reporting the size of the search space over the amount of memory used.

1.5.5 Presentation of results

We commonly use scatter graphs when comparing two methods directly. When reading such graphs points which lie below the $y = x$ line show that the method being represented on the y-axis is best, and points above this line show that the method represented on the x-axis is favourable.

1.6 Thesis outline

The goal of this work adapt an existing branch and bound algorithm which makes use of decision diagrams for use as the search scheme in a constraint solver. This thesis is arranged as follows:

1. In Chapter 2 we introduce decision diagram branch and bound search and show how we adapt it for use as the search scheme in a constraint solver. We give a single algorithm for merging search nodes during the execution of relaxed width truncated search. We

[3]Flatzinc models can involve lines which are too long for Go's buffio package's Scanner to parse.

show that using decision diagram branch and bound as the search scheme in our solver is not possible without allowing constraint propagators to take into account the problem relaxations introduced from merging nodes. To overcome this issue we propose a general strategy for dynamically weakening constraints during search to accommodate problem relaxations. We demonstrate by example the need for weakening algorithms for a number of constraints, and give these algorithms. We empirically evaluate the competitiveness of our approach by comparing against forward checking on the car factory scheduling problem and the maximum independent subgraph problem.

2. In Chapter 3 we continue to provide weakening algorithms for constraints, this time focusing on global constraints and constraints in the all different family of constraints. We again demonstrate by example the need for weakening algorithms for these constraints and give algorithms which allow them to be used in a constraint solver where the search scheme is based on decision diagram branch and bound. We use the problem of finding optimal Golomb rulers as an example problem to empirically verify the correctness of our approach to weakening `alldiff`.

3. In Chapter 4 we propose weakening algorithms for symmetry breaking constraints, again demonstrating by example the need for such algorithms before presenting them.

4. In Chapter 5 we provide a short overview of parallel combinatorial search before parallelising our own solver using an approach similar to one taken by an exiting parallel implementation of decision diagram branch and bound.

5. In Chapter 6 we tackle extremal graph problems using the constraint weakening algorithms given in earlier chapters and a graph search solver which includes the graph isomorphism library Nauty to facilitate canonical search.

6. In Chapter 7 we conclude by providing a summary of the outcomes of this work and suggestions for the direction of future work.

Chapter 2

Weakened constraints for bounding search

2.1 Introduction

In this chapter we introduce a new search scheme for constraint solvers which builds upon the work of Bergman, Ciré, van Hoeve, et al. (2016) who propose a branch and bound algorithm which makes use of two different width truncated decision diagrams to solve problems modelled using dynamic programming models. Inspired by this decision diagram based branch and bound search algorithm we show that a similar approach using width truncated search trees can be used in a constraint optimisation solver. An advantage of this approach is that it allows us to make use of existing constraint propagators. In order to use these existing propagators we introduce a novel approach of dynamically weakening constraints during search to allow relaxed width truncated search trees to return supersolutions ("solutions" whose variable-value assignments do not satisfy all the given constraints in a problem) which which represent upper bounds on the cost of the true optimal solution.

This chapter begins with an overview of the work which led to the development of the decision diagram based branch and bound algorithm, with a particular focus towards research on the use of decision diagrams throughout optimisation science. We follow this with a short and informal overview of how the existing algorithm works and in the remainder of the chapter we detail how we modify this algorithm for use as the search scheme in a constraint optimisation solver. In particular, we focus on a general scheme for constructing restricted and relaxed search trees which provide bounds during search. Exploring relaxed bounding search trees involves merging partial assignments to introduce problem relaxations during search. To respect the outcome of these merge operations we detail a general scheme for dynamically

weakening constraints. We illustrate our approach by applying it to a number of constraints. In subsequent chapters we will also apply this approach to a number of global constraints and symmetry breaking constraints.

2.1.1 Definitions and notation

This section introduces definitions and notations used in the remainder of this thesis.

Although we introduced the concepts of graphs, binary decision diagrams and constraint satisfaction problems informally in Chapter 1, we now provide more concrete definitions with appropriate mathematical notation. We begin by defining graphs and decision diagrams.

Definition 2.1.1. A *graph* is a pair $G = (V, E)$ where V is a set of vertices and E is a set of pairs of vertices called edges. Pairs of vertices in E are said to be *adjacent* to one another.

Graphs can have many additional properties, for example in a *digraph* (also called a *directed graph*) E is a set of ordered pairs of edges. That is, each edge has a direction which leaves one vertex and arrives at another. Graphs which allow multiple edges between pairs of vertices are called *multi-graphs*. Graphs which include no loops (where there is an edge (a,a) between a vertex a and itself) are called *simple graphs*. The vertices and edges in a graph might also have associated $labels$, taken from a set of symbols l (these are often letters or natural numbers). A vertex labelling of a graph G is a partial function from V to l. Similarly, an edge labelling of a graph G is a partial function from E to l. Note that a graph can be both vertex labelled and edge labelled. A *path* in a graph is a sequence of vertices such that each neighbouring pair in the sequence is connected by an edge. A path which begins and ends at the same vertex is known as a *cycle*, and graphs without cycles are known as *ayclic graphs*.

Note that when discussing graphs we use the terms vertices and edges, and when discussing BDDs we use nodes and arcs. This is in line with previous works regarding decision diagrams in optimisation.

Definition 2.1.2. A *binary decision diagram* (BDD) $\mathcal{B} = (N, A, l)$ is a directed acyclic multi-graph (N, A) with arcs labelled from the set l. The labelled arcs encode values given to binary variables. The set of nodes N is partitioned into layers, $L_1, L_2, \ldots, L_{n+1}$, where L_1 contains a root node ρ and L_{n+1} contains a single terminal node t. Each arc $a \in A$ is directed from a node in L_j to a node in the adjacent layer L_{j+1} and is labelled with a label from the set $\{0, 1\}$ representing the value assigned to a binary variable x_j. Arcs leaving the same node have unique labels, therefore each node in L_j has a maximum of two children in L_{j+1}.

For our purposes a path through a BDD from the root node to the terminal node represents a single solution to the optimisation problem being solved using the BDD. Notice that we only

include one terminal node, because when decision diagrams are used to model optimisation problems only paths which lead to feasible solutions are recorded in the BDD. BDDs typically have two terminal nodes, corresponding to the values True and False, but when solving optimisation problems we do not need to keep track of infeasible solutions, so we do not include a False terminal node.

In the previous Chapter we introduced informally some properties of decision diagrams and we state them more formally here, giving definitions for the width of a decision diagram, multivalued decision diagrams, weighted decision diagrams, and exact, relaxed and restricted BDDs.

Definition 2.1.3. A *multivalued decision diagram* (MDD) $\mathcal{M} = (N, A, l)$ differs from a BDD in that the set of labels l can be any finite set, and not only $\{0, 1\}$. If $l \in \{0, \ldots, n-1\}$ then each node in a level L_j can have up to n children in the level L_{j+1}.

When solving optimisation problems where variables are allowed to take more than one value multivalued decision diagrams can be used. The following definitions involving BDDs can also apply to MDDs.

Definition 2.1.4. For a BDD $\mathcal{B}$, whose set of nodes is partitioned into layers, the *width of a layer*, $|L_j|$, in a BDD is the number of nodes contained in the layer, and the *width of a BDD* is the width of the largest layer contained within the BDD.

Definition 2.1.5. A *weighted* BDD $\mathcal{B} = (N, A, l, v)$ is a BDD in which each arc $a \in A$ has an associated *length* $v(a)$. The length of a path through a weighted BDD is the sum of the lengths of the arcs along the path.

Although the algorithm on which we base our approach makes uses dynamic programming models to model problems, we model problems as constraint optimisation problems. Here we define more carefully constraint satisfaction problems, constraint optimisation problems and other required terminology.

Definition 2.1.6. A *constraint satisfaction problem* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ consists of a set of n variables $\mathcal{X} = (X_1, \ldots, X_n)$, each with a domain of values taken from a set of finite domains $\mathcal{D} = (D_1, \ldots, D_n)$ and a set of constraints $\mathcal{C} = (C_1, \ldots, C_m)$. Each *constraint* is a pair $(\mathcal{X}_i, \mathcal{R}_i)$ where $\mathcal{X}_i = X_{i_1}, \ldots, X_{i_n}$ is a subset of variables in $\mathcal{X}$, and $\mathcal{R}_i \subset \mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_n}$ is a relation which defines which values the variables in $\mathcal{X}_i$ are allowed to take. An assignment of values to variables which satisfies all of the constraints in $\mathcal{C}$ is a *solution* to the problem.

Definition 2.1.7. A *constraint optimisation problem* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f, m)$ is a constraint satisfaction problem together with an objective function $f : D_1 \times D_2 \times \ldots \times D_n \to \mathbb{R}$ (also

referred to as a cost function) which is used to score the quality of solutions, and a direction m which is to either minimise or maximise the cost of solutions under f. An *optimal solution* to a constraint optimisation problem has the best (either smallest or largest)) possible cost when compared to all other solutions.

Throughout this thesis we will assume that the goal of constraint optimisation problems is to maximise the cost of the objective function. This is done to simplify the presentation of the material, without loss of generality.

In practice, sometimes when modelling problems not all the variables used to model the problem need to be given a value to find a solution to a problem. When a variable must have its value set we call it a *decision variable*. When we branch during search each new job created will solve a subproblem of the original problem that was modelled as some variables will have their values fixed by assignment. We therefore now define partial assignments and completion problems.

Definition 2.1.8. Given a constraint satisfaction problem $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, a *partial assignment* of a $\mathcal{P}$ occurs when some but not all of the variables involved in $\mathcal{P}$ are assigned a value (that is they have singleton domain sets). A partial assignment of $\mathcal{P}$ defines a *subproblem* of $\mathcal{P}$, or *completion problem* where only a subset of the variables included in $\mathcal{P}$ are yet to be assigned a value.

Our approach to a creating relaxed width truncated search trees will be to create problem relaxations during search. To do this we will introduce in some search nodes values which are incompatible with the constraints being imposed in the problems we solve, and then weaken constraints to ensure that these values can be chosen as search proceeds. To ensure that this is the case we have to ensure that the all such values have support.

Definition 2.1.9. Given a constraint satisfaction problem $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and a pair of variables $x, y \in \mathcal{X}$, a value in the domain of x, $v \in D_x$, has *support* if there exists some value w in the domain of y, D_y, such that the assignment $x = v$ and $y = w$ satisfies all the constraints including x and y.

When BDDs are used to solve optimisation problems we are interested in exact BDDs which encode each solution to the problem, restricted BDDs which provide lower bounds on the cost of optimal solutions, and relaxed BDDs which provide upper bounds on the cost of optimal solutions. We now define these.

Definition 2.1.10. For an optimisation problem $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$, an *exact* weighted BDD $\mathcal{B}$ encodes as root,ρ, to the terminal node representing true, t, paths precisely the feasible solutions of $\mathcal{P}$, where the length of each path is the cost of the corresponding solution under f.

In Section 2.3 we will give an example of how an exact BDD can encode the solutions of a problem. In the previous section we noted the importance of width truncated binary decision diagrams in optimisation and specifically that relaxed and restricted decision diagrams provide bounds when used in optimisation. We now define restricted and relaxed BDDs.

Definition 2.1.11. A *restricted BDD* is a width truncated BDD which represents a subset of the feasible solutions of $\mathcal{P}$. Path lengths from ρ to t provide lower bounds on the best path through the corresponding exact BDD.

Definition 2.1.12. For a binary decision diagram $\mathcal{B} = (N, A, l, v)$ representing a binary optimisation problem $\mathcal{P}$, a *relaxed BDD* $\mathcal{B}' = (N, A, l, v)$ is a width truncated BDD which represents a superset of the feasible solutions of a binary optimisation problem $\mathcal{P}$. Path lengths from ρ to t provide upper bounds on the value of the cost of feasible solutions.

The width of restricted BDDs is typically kept under some maximum permitted width W by deleting nodes from any layer which is wider than W. The width of relaxed BDDs is typically kept under some maximum permitted width W by merging pairs of nodes in any layer which is wider than W until its width is less than W. When constructing a relaxed BDD to solve a problem $\mathcal{P}$, it is the job of the modeller to ensure that the strategy for merging nodes leads to a relaxed BDD as defined above (that is to ensure that an upper bound on the value of the cost of feasible solutions). Later in this chapter we will go into greater detail about how these decision diagrams are constructed, and how the deletion and merge operations work. A node in a relaxed decision diagram which is the result of a merge operation is known as a *relaxed* node (and nodes which are not are known as *exact* nodes).

2.2 Related work

In this section we cover the related work which led to the development of the decision diagram data structure; the use of decision diagrams in optimisation science; and constraint relaxation. Decision diagrams have been widely used throughout many fields in computing science and mathematics. Therefore, what follows is not an exhaustive review of the related literature but is focused on the path from the beginnings of decision diagrams to the work that directly precedes our own research.

The earliest use of an object similar to a binary decision diagram was by Lee (1959) who developed *binary decision programs* to represent switching circuits. A switching circuit is a network of switches, which each have only two states (on or off, true or false). The behaviour of a switch might depend on the state of switches which precede it in the network. Shannon's masters thesis (1940) is the seminal work on switching circuits, where Shannon gives an

algebraic method of modelling them. Lee's motivation for providing an alternative method of modelling switching circuits was to increase the ease with which computer programs which deal with switching circuits could be written. The first use of binary decision diagrams proper comes when Akers represents switching circuits (or more precisely the boolean formula they are equivalent to) in a graphical manner (1978). The binary decision diagrams presented in (Akers, 1978) are equivalent to Lee's binary decision programs, however they proved to be much more useful when developing and implementing algorithms for switching circuits due to their algebraic approach.

The single most relevant work in the field is (Bryant, 1985) which introduced several key concepts and algorithms for BDDs. Bryant's interest was in providing a compact data structure to represent boolean formulae (which can also by represented switching circuits). Bryant notes that a boolean formula can be represented by many decision diagrams, depending on the order in which the variables are considered. An ordered binary decision diagram is a decision diagram in which the order the variables is specified and each layer of the diagram corresponds to a single variable. Each boolean formula then has a single canonical binary decision diagram representation when isomorphic subtrees are superimposed. Figure 1.5 gives an example of how subtrees which lead to equivalent solutions can be superimposed. Such a binary decision diagram is called a reduced, ordered binary decision diagram, and Bryant provides efficient algorithms for reducing a given ordered binary decision diagram (Bryant, 1992). Many of the uses of binary decision diagrams rely on this reduction technique to encode information in a memory efficient manner.

Wegener (2000) provides a thorough survey of the applications of decision diagrams.

2.2.1 Decision diagrams in optimisation science

There has been substantial research effort regarding the use of decision diagrams in the field of optimisation. This spans multiple solver technologies such as Integer Programming and Constraint Programming, and also includes the use of decision diagrams as the basis for a branch and bound search algorithm.

An early use of binary decision diagrams in combinatorics is for enumerating solutions to combinatorial problems. To enumerate the solutions of a problem the approach involves mapping the constraints involved in the problem to a Boolean function, f, which evaluates to 1 if and only if a solution to f is feasible. A binary decision diagram can then be constructed which represents only the feasible solutions to the problem represented by f by discarding all paths which lead to 0. The number of solutions is then given by the number of paths through the binary decision diagram from the root node to the terminal node 1. These paths can be

counted with existing graph algorithms. This approach of only keeping paths which lead to a solution is common among most uses of decision diagrams in optimisation. However, the method of solution counting is usually not tractable as the size of these binary decision diagrams can grow exponentially with respect to the number of variables in a problem. Still the technique finds some uses as a stand alone approach (for example when enumerating the number of solutions to the forbidden subgraph problem (Nakahata, Kawahara, and Kasahara, 2018)) or when combined with other techniques such as backtracking search and divide and conquer methods (for example when enumerating the number of knight's tours (Schröer and Wegener, 1998)).

Another early use of binary decision diagrams for optimisation problems was by Lai, Pedram, and Vrudhula (1994). They provide an algorithm to construct binary decision diagrams from binary integer programs via Boolean functions, and an algorithm which combines these binary decision diagrams with a branch and bound approach in an effort to reduce execution times for search. The novel aspect of their approach is to first begin to solve a problem using existing solution techniques for integer programming before halting search after a number of branching decisions. The goal is then to create a binary decision diagram which encodes all feasible completions of the subproblem, from which the optimal solution can be obtained. More recent work in the field of constraint programming also applies a similar approach. In (de Uña et al., 2019) the authors also stop search, this time in a constraint solver, and use multivalued decision diagrams (where each node is allowed more than one child) to solve the resulting completion problem. Both of these approaches reported improvements in execution times for certain classes of problem.

Another use of binary decision diagrams in integer programming is given by Hadzic and Hooker (2006) who enumerate the optimal or near-optimal solutions of an integer programming problem. One of their reasons for enumerating not only the optimal solutions, but also all near-optimal solutions is to perform cost-based domain analysis. The goal of this analysis is to assess how the size of the domain of a variable grows as solutions are allowed to move further from the optimal. This tells the modeller which variables are fixed in all optimal solutions, Hadiz and Hooker use capital budgeting and network reliability as problems on which to test the effectiveness of their approach. An alternative view on this approach is that it could be used to show how far from the optimal solution a solution to a problem needs to be to provide robustness.

Hadzic and Hooker (2007) follow this work by addressing the issue of minimising the size of a BDD as problem size grows:. They also take the approach of only considering near-optimal solutions, under the assumption that these are of most use to the modeller. Armed with an objective function to score solutions they then prune arcs from the BDD when they are shown to lead to sub-optimal solutions. These BDDs which only consider nearly optimal solutions

(that is solutions whose cost is close to the cost of the optimal solution) are shown to be small compared to BDDs which consider all solutions when the authors conducted computational experiments conducted on randomly generated binary integer program examples.

In constraint programming Andersen et al. (2007) introduce the concept of limited width decision diagrams. The concepts introduced are critical for the work undertaken later in (Bergman et al., 2016), which is the main work on which this thesis builds. Anderson et al use decision diagrams to represent a superset of the set of feasible solutions to a constraint satisfaction problem. Instead of enumerating all solutions as in previous works, Anderson et al introduce relaxed decision diagrams. Relaxed decision diagrams have their size controlled by having a fixed maximum permitted width (where the width of a decision diagram is the number of nodes in its largest layer). A larger width in a relaxed decision diagram means a smaller set of infeasible solutions could be recorded and the closer the diagram is to a representation of the exact set of solutions. This work also introduces the concept of multivalued decision diagrams, where each node may have more than two children. The main goal of this work was to provide an alternative to the traditional domain store in constraint programming. In constraint programming the domain store records all of the possible domains of each variable. By replacing the domain store with a multivalued decision diagram Andersen et al. are able to keep a much smaller set of possible assignments than the domain store approach. The results given in their paper show that in certain cases this leads to orders of magnitude reduction of the execution times of a constraint solver. Hadzic, Hooker, et al. (2008) and Hoda, van Hoeve, and Hooker (2010) develop this approach further by providing a generic method for propagating constraints over a multivalued decision diagram constraint store. Their work showed that this general approach also causes orders of magnitude improvement in execution times.

Bergman, van Hoeve, and Hooker (2011) introduce a decision diagram based approach for computing bounds for set covering problems. The main contribution of this work that we are interested in is a top-down compilation method for the construction of relaxed multivalued decision diagrams. This work is critical for the algorithm Bergman et al propose for dynamic programming models of discrete optimisation problems, on which our work is based. We defer discussion of this algorithm to Section 2.3. The similarities between multivalued decision diagrams and dynamic programming models was the subject of a paper by Hooker (2013). Hooker views an multivalued decision diagram as being equivalent to the state transition graph of a dynamic program and shows that there is a unique canonical multivalued decision diagram for a given optimisation problem and variable ordering. Therefore a canonical multivalued decision diagram can be used as a smaller representation of the state transition graph.

2.2.2 Problem relaxations

This section details adjacent areas of AI where some of the terms given above already have an ingrained meaning. We include this section to avoid confusion caused by our use of terminology which is similar to that used in these related, but different fields.

A common question that can be asked if a problem does not admit a solution which satisfies all the constraints being imposed is: "how many constraints must be removed from the problem to make it satisfiable?". More specifically there is a type of optimisation problem which asks "what is the smallest number of constraints which must be removed to make a problem satisfiable?". When a problem has constraints removed from it problem relaxations are created. This kind of question arises particularly from real world problems, where it is often more economically advantageous to offer a solution to a problem relaxation than no solution at all. Gottlieb, Puchta, and Solnon (2003) answer this question for a number of instances of the car factory sequencing problem, which attempts to schedule the manufacture of cars in a factory. Gottlieb, Puchta, and Solnon give an upper bound on the number of constraints which must be removed from various unsatisfiable benchmark instances in order to output a "solution". While these problem relaxations are similar to our work, instead of removing constraints entirely we will seek to weaken them instead.

Problem relaxations are also used when solving mixed integer linear programming models, specifically when creating linear programming (LP) relaxations of MIP problems by removing integrally constraints (constrains which enforce that certain variables in the MIP problem take integer values). The LP relaxation can then be solved, and if all variables that were to take integer values in the original MIP problem are assigned integer values then the optimal solution to the MIP problem has been found. Otherwise search branches choosing a variable which does not have an integer assignment. In one branch the variable is constrained to take a value which is less than or equal to the floor its value before branching and greater than or equal to the ceiling of its value in the other branch. Search continues to branch and bound in this way until all variables which are required to have integer assignments are assigned integer values.

2.3 Decision Diagram Branch and Bound

In this section we give an overview of decision diagram branch and bound as it appears in the literature. In Section 2.4 we follow this overview with our own modifications to the algorithm which allow us to use it as the search scheme in a constraint optimisation solver.

Decision diagram branch and bound (DDBB) search (Bergman et al., 2016) solves problems

modelled using dynamic programming models. It does so indirectly, recursively constructing restricted and relaxed decision diagrams such that the entire state space of the dynamic program is not explored.

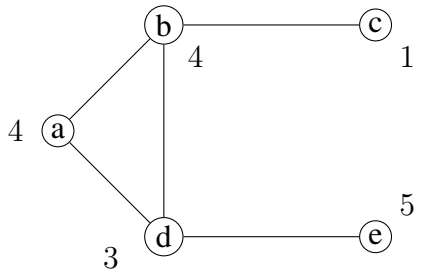

Figure 2.1: An instance of the MISP.

We motivate the algorithm by an example. Consider the instance of the maximum independent set problem shown in Figure 2.1. The goal of the maximum independent set problem (MISP) is to find a set of vertices in a weighted graph such that none of the vertices in the set are adjacent to one another (an independent set) and the sum of the weights in the set is the maximum possible for any independent set.

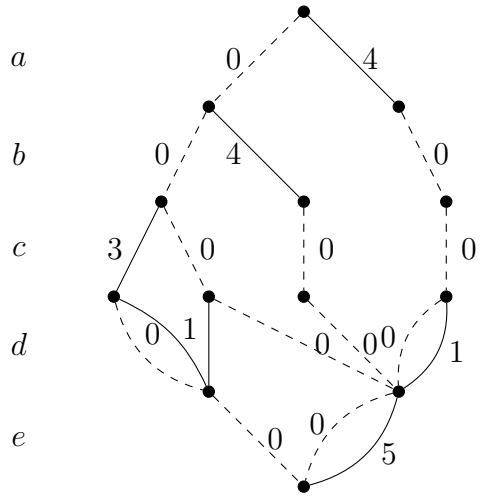

Figure 2.2: An exact decision diagram representing the instance of the MISP.

The exact decision diagram given in Figure 2.2 exactly encodes all of the independent sets of of the graph shown in Figure 2.1, where each solution is represented as a path from the root node to the terminal node. Each solid line represents including the vertex in an independent set, and each dashed line represents not including the vertex in an independent set. Not including a vertex in an independent set incurs a cost of 0, while including a vertex in an independent set costs as much as the weight of the vertex. In particular, the maximum independent set $\{a, d, e\}$ is represented by a path of weight 10. DDBB search constructs BDDs in a top down manner, starting at the root node and proceeding layer by layer. Each node, n, records all possible choices of variable-value assignments that are compatible with previous assignments

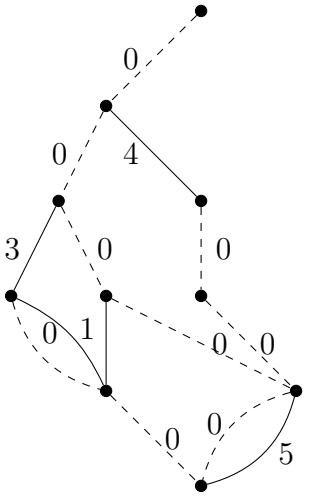

Figure 2.3: A restricted decision diagram.

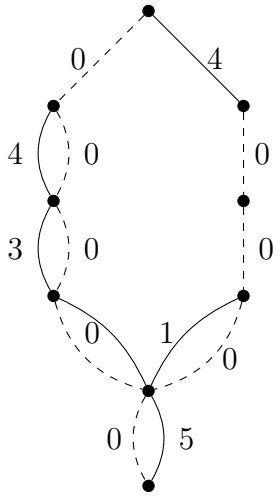

Figure 2.4: A relaxed decision diagrams.

on the path from the root node to n. Child nodes are created by assigning a single variable a value and these appear in the succeeding layer to the parent node. For example, in Figure 2.2 the construction of the BDD would start at the root node and proceed by creating a new node in the second layer corresponding to all possible choices of value assignment of a. In the following layers another variable is chosen and we create nodes which correspond to all possible value assignments to this variable in the subsequent layer.

For compactness we collapse equivalent nodes in the figures presented in this section, but in general the BDDs constructed while using DDBB search are not reduced in this way. As the size of problems increases (that is the number of variables and values involved in the problem grows) it becomes infeasible to construct exact decision diagrams, so DDBB search uses two types of width bounded decision diagrams to solve problems.

Restricted decision diagrams are constructed by deleting nodes from layers that become too wide while a BDD is being constructed. The choice of what width is chosen by the modeller. By removing nodes from the BDD not all solutions of the problem will be represented by the BDD. For problems where we are seeking to maximise some cost function (for example when we try to find the largest weight independent set in the MISP) restricted BDDs provide lower bounds on the maximum solution found by the corresponding exact diagram. Figure 2.3 shows a restricted decision diagram for this instance of the MISP presented in Figure 2.1, where the maximum weight independent found set is $\{b, e\}$, providing a lower bound of 9 on the size of the optimal solution.

Relaxed decision diagrams are constructed by merging nodes together when layers become too wide while the relaxed BDD is being constructed. DDBB search merges nodes by merging the state of future compatible variable-value assignments at each relaxed node. In the case of

the MISP, when nodes are merged the state at the merged node is the union of the states at the pair of nodes being merged. This allows for assignments which are incompatible with the problem originally being solved to be represented in a relaxed BDD. For example consider the relaxations introduced in Figure 2.4 which allow both b and d to be included in a set together because d is a viable option under leftmost node which is now merged with its neighbour. In Figure 2.4 we can see that a path representing the set $\{b, c, d, e\}$ is included in the BDD, which is not a valid independent set, but does provide an upper bound of 13 for the size of the optimum set.

In general when constructing relaxed BDDs nodes are merged in pairs, and the modeller provides a function $\oplus$ which prescribes how to merge two nodes together, as well as another function Γ which updates the cost associated with arcs arriving at the merged node. These functions must be chosen in such a way that they allow relaxed decision diagrams to include a path which has at least as large a cost as the optimal path found in the corresponding exact decision diagram built from the same root node — thereby ensuring that they satisfy the definition of relaxed decision diagrams given in Section 2.1.1. This is not intrinsically ensured by the approach of Bergman et al and has to be proven on an adhoc basis for each problem being solved. It is left to the modeller to ensure that the functions $\oplus$ and Γ allow relaxed decision diagrams to correctly provide upper bounds for the problem being solved. Since our approach does not rely on these functions $\oplus$ and Γ we omit in depth discussion here. In particular our approach does not require labels to be record on arcs, so we do not make use of Γ.

In the following sections we will give our approach to search which is heavily inspired by DDBB in detail. An informal overview of the approach taken by Bergman et al is:

1. Construct a restricted BDD, B, rooted at a node u and if a solution is found that is larger then the incumbent best solution yet found by DDBB search then it becomes the incumbent.

2. If B is not exact[1] then build a relaxed diagram B' rooted at u. If the upper bound found by B' is larger than the incumbent best solution then collect a cutset of exact nodes in B'. This cutset is a set of nodes n such that the path from u to n contains no relaxed nodes. This set of nodes could include all of the deepest such nodes which satisfy this criteria, such as the last layer in which there are no relaxed nodes, or any layer above that. Add each of these nodes to a queue Q.

3. While Q is not empty, chose a new node u to branch on and repeat from step 1.

[1]The width of some layer was reduced during its construction by deleting nodes.

2.4 DDBB search in constraint programming

In this section we show in detail how we adapt the DDBB search algorithm for use as the search scheme in a constraint programming solver for solving constraint optimisation problems. We move away from using decision diagrams and now use two different types of width truncated search trees. In his section we explain our approach to constructing restricted and relaxed width truncated search trees and then present a search algorithm which explores the search space of the problem being solved by recursively exploring restricted and relaxed search trees while making use of existing constraint propagators.

When using DDBB search as the approach to search in a constraint optimisation solver we no longer begin by modelling problems as dynamic programming models. We model problems as constraint optimisation problems (using a set of variables $\mathcal{X}$ with finite domains $\mathcal{D}$, a set of constraints $\mathcal{C}$ over these variables and a cost function f to rank solutions[2]) and solve them using a combination of search and constraint propagation using existing propagation algorithms. This change in how we model and solve problems has the following impacts:

- Instead of recording the state at each node and representing solutions as paths through the decision diagram we record the domains of variables at each node using a traditional domain store approach. In order for our constraint weakening approach to work we also record constraints at each node.
- The cost of a solution is determined by the assignments recorded at leaf nodes in restricted and relaxed width truncated search trees, rather than by annotating arcs in a BDD with lengths and searching for a maximum length path from the root node to the terminal node. This has the added benefit that we are no longer required to explicitly keep track of arcs between nodes in order to represent solutions.
- When continuing search from a node u in a layer L_j, we select a variable x and for each value a in the domain of x, D_x, we propagate the assignment of $x = a$ against a set of constraints $\mathcal{C}$. If propagation of the constraints in $\mathcal{C}$ does not lead to a contradiction we add a new node u' in L_{j+1} which includes the assignment $x = a$. The domains of the unassigned variables may change between u and u' due to the propagation stage.

As before, these changes mean that we move away from using decision diagrams explicitly, but given the close similarity between our approach to the existing algorithm we still refer to our approach as DDBB search when we use it as the search scheme in a constraint solver.

[2]Recall that to simplify the presentation of material we assume that the goal of each problem is to maximize the cost of solutions.

In the following sections we present algorithms for exploring restricted and relaxed width truncated search trees; for merging nodes while exploring relaxed width truncated search trees; and a branch and bound algorithm which uses these search trees to solve constraint optimisation problems.

2.4.1 Restricted width truncated search

Our approach to creating restricted width truncated search trees is given in Algorithm 2.1 and is very similar to the algorithm given by Bergman et al. which builds restricted decision diagrams, with changes made to reflect the move to solving constraint optimisation programs modelled as a constraint program.

The role of restricted width truncated search in our search scheme is twofold: these search trees determine which search nodes to branch on during search and also perform some limited search in the hope of improving the incumbent best solution found by our search algorithm. Search using Algorithm 2.1 begins at a root node r which records the variables and constraints of a problem $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$ and proceeds in a breadth first manner until some layer in search, which we will call L_c, exceeds the maximum permitted width W (the actual value of W is determined by the modeller). The nodes in this layer are candidates for branching and are returned by Algorithm 2.1 to be branched on in Algorithm 2.4. Search continues from L_c, but no more than W nodes from each subsequent layer are added to the search tree. When there are no longer any variables to branch on (either we have created h layers or run out of unassigned variables, which is checked on line 8), Algorithm 2.1 returns the layer of candidate nodes to branch on, the size of the best solution found, and the node containing the best solution found. Lines 12-14 are optional and are included to make use of existing variable and value ordering heuristics. Such heuristics attempt to place good solutions to the left of the search tree, and lines 12-14 allow us to keep the leftmost W nodes on each layer without having to compute each node in the entire layer before deleting the rightmost nodes. The for loop on line 11 takes each node in a layer L_c and attempts to create a new node in L_{j+1} for each possible value assignment from the domain of the branch variable x.

The procedure `Propagate` called on lines 14 and 16 assigns a value i to a variable x in a node u and propagates this assignment. This propagation stage works in the same way as in existing constraint solvers, where constraint propagation algorithms remove values from the domains of unassigned variables which cannot be compatible with any solution which includes the existing variable-value assignments. These propagation algorithms remove values from the domains of unassigned variables until either a fixed point is reached or some variable's domain is empty (proving that the current partial assignment cannot possibly be included in any solution to the problem).

The procedure `selectBranchVariable` seeks to find an unassigned variable across nodes in a layer, L_j, to assign values to when creating the next layer L_{j+1}. If no variable in any of the nodes in L_j are yet to be assigned a value the Algorithm 2.1 terminates, reporting the best solution found. The order in which this procedure returns variables depends on variable ordering heuristics. Similarly, the order in which nodes are added to L_{j+1} depends on value ordering heuristics (which is why we do not give a particular order to choose values on line 11 of Algorithm 2.1).

Proposition 2.1. Restricted width truncated search as given in Algorithm 2.1 explores a subset of the solutions of the (completion) problem $\mathcal{P}$ defined at the node r and a lower bound for the optimal solution of $\mathcal{P}$ is found by exploring this subset of solutions.

Proof. Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$ be a constraint optimisation problem, (or a completion problem of $\mathcal{P}$ where a subset of $\mathcal{X}$ are assigned values), which is solved using Algorithm 2.1. If the width of the search tree never exceeds W then the whole search space is explored by Algorithm 2.1 and value of lb will be the value of the optimal solution.

If the width of the search tree is truncated, then we are no longer guaranteed to find any solutions and those that are found cannot be guaranteed to be optimal as the entire search tree rooted at r has not been explored. The lower bound for the optimal solution of $\mathcal{P}$ is set to $-$ on line 24 of Algorithm 2.1 and if there is no solution found (that is L_l is empty) then $-$ is returned as the lower bound of $\mathcal{P}$, which is a lower bound for the cost of the optimal solution of $\mathcal{P}$. If L_l is not empty then this subset of solutions is compared on lines 26-29 and the highest cost solution is returned along with its cost, lb, on line 30. Since the width has exceeded W some search nodes have been dropped by line 23, and the whole search space is no longer explored. Therefore only a subset of the solutions to a problem $\mathcal{P}$ defined at the node r are explored and lb represents a lower bound for the optimal solution of $\mathcal{P}$. □

2.4.2 Relaxed width truncated search

Our approach to exploring relaxed width truncated search trees is again similar to the algorithm given by Bergman et al. We show here how this approach is used to solve problem using search and existing constraint propagation algorithms.

Our approach is shown in Algorithm 2.2. The role of relaxed width truncated search trees in our search scheme (which is given in Algorithm 2.4) is to provide upper bounds on the cost of the optimal solution of the subproblem defined at the input node r. To achieve this we merge nodes together during search using Algorithm 2.3 to introduce problem relaxations when a layer in search becomes too wide, rather than deleting nodes as in Algorithm 2.1. To ensure

Algorithm 2.1: Restricted width truncated search for a problem, $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$, recorded at a root node r.

```
1  solveRestricted (Node r) → Layer, int, Node
2  begin
3      L_1 ← {r}
4      b ← ⊥ // flag set if this search tree becomes restricted
5      h ← the number of unassigned (decision) variables at r
6      for l = 1 to h do
7          x ← selectBranchVariable(L_l)
8          if no variable is found then
9              break
10         L_{l+1} ←∅
11         forall u ∈ L_j and i ∈ D_x do
12             if b then
13                 if |L_{l+1}| < W then
                       // Propagate returns null if there is a
                           contradiction
14                     u' ← u.Propagate(x, i)
15                     L_{l+1} ← L_{l+1} ∪ u'
16                     break
               else
17                 u' ← u.Propagate(x, i)
18                 L_{l+1} ← L_{l+1} ∪ u'
19         while |L_{l+1}| > W do
20             if !b then
                   // current layer becomes jobs for branching
21                 L_c ← L_l
22                 b ← ⊤
23             L_{l+1} ← the leftmost W nodes in L_{l+1}
24     lb ← −∞
25     incumbent ← null
26     forall u ∈ L_l do
27         if f(u) > lb then
28             b ← f(u)
29             incumbent ← u
30     return L_c, lb, incumbent
```

that these problem relaxations are included in search, and lead to Algorithm 2.2 returning upper bounds, we have to take care with the procedures `mergeNodes` and `Propagate`. We require that `mergeNodes` creates a node m which defines a relaxed subproblem which admits an optimal solution at least as large as each of the nodes u or v being merged at line 13. We also require that when `Propagate` is called on m on line 17 that the relaxations introduced by `mergeNodes` are respected. In Section 2.4.3 we detail an algorithm for merging nodes and throughout this thesis we show how individual constraints can be weakened such that constraint propagation respects the problem relaxations introduced by merging nodes. The `selectNodesToMerge` procedure takes as input a layer and chooses which two nodes in the layer to merge together using the `mergeNodes` procedure.

Algorithm 2.2: Relaxed width truncated search.

```
1  solveRelaxed (Node r) → int
2  begin
3      L_1 ← {r}
4      h ← the number of unassigned decision variables at r
5      for l = 1 to h do
6          x ← selectBranchVariable(L_l)
7          if no variable is found then
8              break
9          while |L_j| > W do
10             u, v ← selectNodesToMerge(L_j)
11             L_j ← L_j \ {u, v}
12             m ← mergeNodes(u, v)
13             L_j ← L_j ∪ m
14         L_{l+1} ←∅
15         forall u ∈ L_j and i ∈ D_j do
16             u' ← u.Propagate(x, i)
17             L_{l+1} ← L_{l+1} ∪ u'
18     ub ← null
19     forall u ∈ L_l do
20         if f(u) > ub then
21             ub ← f(u)
22     return ub
```

2.4.3 Merging nodes in relaxed search trees

In the previous section we gave an algorithm for exploring relaxed width search trees, and stated that the `mergeNodes` procedure must merge two nodes together to provide a relaxation of the problem which admits a solution at least as large as either of the two input nodes.

Note that our approach differs from that of Bergman et al as we provide a single algorithm for merging nodes during search and then weaken individual constraints, while the existing DDBB algorithm requires that merge strategies be considered on a problem by problem basis.

Algorithm 2.3: Merging partial solutions with respect to a cost function f.

```
1  mergeNodes (Node u, Node v) → Node
2  begin
3      if the partial assignment at u has higher cost than the partial assignment at v, under f then
4          m ← u
5          foreach unassigned variable x recorded at v do
6              D_x^m ← D_x^m ∪ D_x^v, where D_x^m is the domain of the variable x as recorded at
                node m
7      else
8          m ← v
9          foreach unassigned variable x recorded at u do
10             D_x^m ← D_x^m ∪ D_x^u
11     foreach constraint c recorded at m do
12         relax c to ensure that all values in the domains of the variables recorded at m have
            support
13     return m
```

Our approach when merging two nodes u and v to create a new node m is to first identify which node, which we will assume for ease of presentation is v, contains the highest cost partial assignment, with respect to a cost function f. We calculate the cost of a partial assignment by evaluating the cost function over only the subset of variables which are assigned a value. The relaxed node m then inherits all of the variable assignments and domains of unassigned variables from v. To introduce a relaxation of the problem we modify the domains of the variables in m to also include all of the values in the domains of unassigned variables in the lower cost node u. We expect that any constraints that are implemented in a solver which uses DDBB as the search scheme has an associated weakening algorithm, to ensure that each value introduced in the domains of variables in m from u has support after the merge operation has been performed.

Since we might have assigned a value to some variables by propagation in one of nodes u and v but not the other this can lead to nodes which have been assigned a value in v becoming unassigned again in m. Note that this does not mean that search will fail to terminate, since when we choose a variable on line 6 of Algorithm 2.2 it is assigned a value in each node in the following layer.

Through lines $9-14$ in Algorithm 2.2, and in particular the use of Algorithm 2.3, we introduce problem relaxations in relaxed width truncated search trees. Throughout the remainder of this

thesis we detail how we weaken individual constraints in such a way that the additional values from u introduced to the domains of variables in m are supported. This is how we ensure that problem relaxations are introduced during relaxed truncated search trees.

$x_0 = 0$	$x_0 = 1$	$x_0 = 1$
$x_1 = \{1, 4\}$	$x_1 = 2$	$x_1 = \{1, 4, 2\}$
$x_2 = 2$	$x_2 = 5$	$x_2 = 5$
$x_3 = 2$	$x_3 = 1$	$x_3 = 1$
$x_4 = 2$	$x_4 = 1$	$x_4 = 1$
$\forall_{5 \leq i \leq 8}.\ x_i \in \{0..5\}$	$\forall_{5 \leq i \leq 8}.\ x_i \in \{1, 3\}$	$\forall_{5 \leq i \leq 8}.\ x_i \in \{0..5\}$
(a) Domains at u	**(b)** Domains at v	**(c)** A relaxation at m

Figure 2.5: Two partial solutions and the partial solution resulting from merging them together.

Figure 2.5 gives an example of two nodes u and v being merged together to create a new node m, when the cost function being considered is to maximise the sum of the values of all variables $\{x_0, \ldots, x_8\}$. The partial solution at v has higher cost than the partial assignment at u, so m inherits values for its assigned variables from v. The values in the domains of the unassigned variables in u are then introduced into the variables recorded at m. The goal of the constraint weakening algorithms we give in the remainder of this thesis is to ensure that all of these values introduced from u have support in m. That is, propagating constraints after a merge operation has taken place should remove no values from the domains of unassigned variables in m.

2.4.4 Limitation on cost functions

Our approach of providing a single algorithm for merging nodes, and then weakening constraints to allow the relaxations introduced to be respected, introduces a limitation on the cost functions that we can use when modelling problems. When we merge nodes we require that each completion of the new relaxed problem recorded at the node m has greater cost when it includes the variable assignments from v than those from u. Note that the highest cost completion of the new relaxed problem recorded at m might include assignments of values introduced to the unassigned variables in m from the domains of unassigned variables in u. Therefore, the optimal solution to this relaxed completion problem at m might not necessarily be a solution to the problem which is originally being solved (that is, this solution may violate one or more constraints as specified in the original problem). We will see exactly this behaviour in examples that we give to motivate weakening algorithms for constraints.

2.4.5 The DDBB based search algorithm

Our approach to search is again very similar to the algorithm presented in (Bergman et al., 2016), with small changes made to reflect our constraint programming approach. One change that we do make is that we use restricted diagrams to determine which nodes to branch on (while the existing DDBB search algorithm uses relaxed decision diagrams). This allows us to use the algorithm without creating relaxed width truncated search trees.

The search algorithm we use is given in Algorithm 2.4. We begin by creating a root node based on constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$. We call Algorithm 2.4 on the variables, constraints and cost function of this problem (as we assume that a variable x records its own domain D_x) and each node created during search records both the variables and constraints involved in $\mathcal{P}$. We add this root node to a queue of nodes. Algorithm 2.4 then explores restricted width truncated search rooted at r using Algorithm 2.1. If this width truncated search is not exact then Algorithm 2.1 returns a layer of nodes to branch on and we add these to the queue. If the restricted width truncated search finds an incumbent solution we update the incumbent and the size of the lower bound on the cost of the optimal solution z_{opt}.

We repeat this process for each node u in the queue until it is empty, with the additional step of exploring relaxed width truncated search trees to bound search once an incumbent solution has been found. If the upper bound on the size of the solution of the completion problem defined at u, found by Algorithm 2.2, is smaller than z_{opt} then we do not branch on u using Algorithm 2.1.

2.4.6 New heuristics introduced through using DDBB

By using Algorithm 2.4 as the search scheme for a constraint optimisation solver, we introduce new search heuristics to Constraint Programming to join the common variable and value ordering heuristics (these existing heuristics do still effect the execution times of our algorithm).

The most obvious heuristic choices that we have introduced are the maximum permitted widths of both restricted and relaxed width truncated search trees. We expect that wider search trees lead to tighter bounds and reduced execution times of the algorithm overall, but this is is not guaranteed. Further complicating matters, the maximum permitted width of restricted width truncated search effects which nodes Algorithm 2.4 branches on.

The order in which we remove nodes during restricted width truncated search and merge nodes during relaxed truncated search also matters. Our approach is to compliment the behaviour of existing variable and value ordering heuristics which attempt to place good solutions on the

Algorithm 2.4: Branch and bound search using width truncated search trees.

```
1  search (Variables 𝒱, Constraints 𝒞, Cost f) → Node
2  begin
3      r ← root node containing variables 𝒱 and constraints 𝒞
4      Q ← {r}, initalise a queue with the root node
5      incumbent ← null
6      z_opt ← −∞
7      while Q ≠ ∅ do
8          u ← select a node from Q
9          Q ← Q \ u
10         if z ≠ −∞ then
11             ub ← solveRelaxed(u)
12         if ub > z_opt then
13             L_c, lb, candidate ← solveRestricted(u)
14             Q ← Q ∪ L_c
15             if lb > z_opt then
16                 z_opt ← lb
17                 incumbent ← candidate
18     return incumbent
```

leftmost branch of the search tree. As a result we assume that nodes towards to leftmost side of each level are more likely to lead to good solutions. Therefore we only keep the leftmost nodes when creating restricted width truncated search trees and merge nodes to the right of a level when creating relaxed width truncated search trees.

The order in which nodes in the queue are visited is also important. We do this in a breadth first manner. The reasoning behind this is that we do a small amount of work each time we explore a restricted width truncated search tree to sample throughout the search space, in the hope of finding a good incumbent solution faster than forward checking.

2.5 Constraint weakening during search

In this section we outline a general approach to ensure that the problem relaxations introduced when merging nodes can be included in relaxed relaxed width truncated search. The first merge operation in Figure 2.4 illustrates this issue, because after this merge operation the resulting node corresponds to both b and c being included in the partial solution, however this conflicts with the constraints imposed (that at most one of each pair of adjacent vertices is included in a solution). When we take a constraint programming approach we have to take care that constraint propagators allow these problem relaxations.

Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$ and let α and β be two partial assignments of $\mathcal{P}$ recorded at nodes u and v which are merged using Algorithm 2.3 to obtain a partial assignment γ recorded at a new node m, which is a relaxed node resulting from the merge operation. To ensure that relaxed width truncated search trees return upper bounds we consider the completion problems for each of these partial assignments. Recall that the completion problem of a problem $\mathcal{P}$ is a subproblem where some of the variables in $\mathcal{X}$ have their values assigned. We require that the completion problem for γ admits a solution which is greater than or equal to any solution of $\mathcal{P}$ which includes the partial assignments α and β, with respect to the cost function f. To ensure that this is the case we require that each value which had support in the domains of the unassigned variables in α and β has support in γ. We achieve this requirement by weakening individual constraints in the set of constraints $\mathcal{C}$, to ensure that each value in the unassigned domains recorded at m has support. Recall that a value i in the domain of a variable x has support if there exists some value in the domains of all other variable which allows the value to be chosen as the assignment while satisfying $\mathcal{C}$. This constraint weakening is done dynamically during search as prior to merging it is not known which relaxations of $\mathcal{C}$ will need to be introduced when nodes are merged. This is unlike existing relaxation processes in constraint programming which allow some constraints to be removed from a problem in order to find a "good enough" solution where an exact solution might not exist, usually in some predefined order which is determined before search begins.

The completion problem defined at m should be as tightly constrained as possible. This means that no value which does not have support at u or v should have support at node m. We should also make use of existing constraint propagation algorithms with a minimum amount of modification, so that our new search scheme can be easily used with existing propagation algorithms.

We relax constraints in one of two ways:

- We modify the constraint itself in some way, for example changing the bound of a linear summation constraint.
- We modify the value seen by the propagator of the constraint when accessing the value of variables, for example to allow two variables in an all different constraint to be assigned the same value.

We refer the processes of modifying the value read by a propagator when accessing the value of an assigned variable as *masking* the variable. When we mask assigned variables with a new value, which we call the variable's mask or masked value, the new value is always distinct from the variable's assigned value. Whenever a propagator attempts to access the value of

$$a = 1$$
$$b = 2\ (6)$$
$$c = 3$$
$$d = \{2, 4, 5\}$$
$$e = \{2, 4, 5\}$$

Figure 2.6: An example completion problem, where an `alldiff` constraint is weakened by masking the value of the variable b with the value 6 to ensure that the value 2 has support in the domains of variables d and e.

a variable, if the variable were masked it instead reads the masked value. Note that cost functions ignore these masked values and read the true value being assigned to the variable. Figure 2.6 shows how we can apply this approach in the case of the *all different* constraint. In this example masking variable b with a value which is not in the domain of any other variable allows the unassigned variables to continue to be assigned the value 2, and ensures that each value in the domains of the unassigned variables d and e has support.

In the remainder of this chapter we introduce algorithms which weaken particular constraints over finite domain integer variables so that they can be used with the branch and bound based search scheme given in Algorithm 2.4. These constraints represent a subset of the built in constraints in the Flatzinc specification.

2.6 Weakening linear summation constraints

In this section we describe a how to weaken linear summation constraints. These constraints have the form

$$a_1x_1 + a_2x_2 + \ldots + a_nx_1 \bowtie b$$

where each x_i (for $1 \leq i \leq n$) is an integer variable, a_i is an integer constant, $\bowtie \in \{\leq, =, \geq\}$ and the bound b is an integer. Linear summation constraints are often used when modelling problems, so they are an important candidate for which to provide a weakening algorithm.

2.6.1 Weakening sum less than or equal to

We begin by considering the less than or equal to inequality variation of the linear constraint, which we will refer to as `lin_le` . Harvey and Stuckey (1998) present a propagator which guarantees bounds consistency for linear constraints which runs in linear time. To simplify

the material slightly we follow Harvey and Stuckey's lead in assuming that each coefficient a_i is a positive integer[3].

Let $\overline{x_i}$ and $\underline{x_i}$ denote the upper and lower bounds of the domain of the variable x_i respectively. The propagator given by Harvey and Stuckey for this constraint relies on the following. If we define F to be:

$$F = b - \sum_{i=1}^{n} a_i \underline{x_i} \tag{2.1}$$

then the upper bound for each x_i can be calculated via

$$x_i \leq \frac{F}{a_i} + \underline{x_i} \tag{2.2}$$

Further, if $F < 0$ then the constraint is unsatisfiable. The propagator requires only two passes over the summands to update the bounds of each x_i; one pass to calculate F and another to update the bounds of each x_i. In follow up work (Harvey and Schimpf, 2002) some improvements are added to this algorithm allowing bounds consistency to be achieved in less than two passes over the summands, but these only effect the speed of the propagator, not which values are deleted from variable domains by propagation.

$$\forall_{0 \leq i \leq 8}.\ x_i \in \{0, \ldots, 5\}$$

$$\sum_{i=0}^{8} x_i \leq 13$$

$$\text{maximise} \sum_{i=0}^{8} i x_i$$

Figure 2.7: An example problem involving a `lin_le` constraint.

We motivate our algorithm for weakening the `lin_le` constraint with an example problem. Consider the problem modelled in Figure 2.7 which includes a single linear constraint $\sum_{i=0}^{8} x_i \leq 13$ over integer variables each with domains $\{0, \ldots, 5\}$. Figure 2.8a and Figure 2.8b show two partial solutions α and β, in which the variables x_1,x_2,x_3 and x_4 are assigned values. In α the upper bounds of the unassigned variables (x_5,x_6,x_7 and x_8) are not impacted by applying Equations (2.1) and (2.2) to calculate bounds. However, in β the upper bound of each unassigned variable can be tightened to 1. When we merge these partial

[3]This allows us to avoid having to write out cases for when the coefficients are positive and negative. For example if $a_i < 0$ in Equation 2.1, we would have to consider the value $\overline{x_i}$ rather than $\underline{x_i}$.

solutions using Algorithm 2.3 to obtain the partial solution γ shown in Figure 2.8c it is clear that the new domains of the unassigned variables will be impacted if Equations (2.1) and (2.2) were applied. To overcome this we weaken the constraint by loosening the bound b.

$x_0 = 0$	$x_0 = 1$	$x_0 = 1$
$x_1 = 1$	$x_1 = 4$	$x_1 = 4$
$x_2 = 2$	$x_2 = 5$	$x_2 = 5$
$x_3 = 2$	$x_3 = 1$	$x_3 = 1$
$x_4 = 2$	$x_4 = 1$	$x_4 = 1$
$\forall_{5 \leq i \leq 8}.\ x_i \in \{0..5\}$	$\forall_{5 \leq i \leq 8}.\ x_i \in \{0, 1\}$	$\forall_{5 \leq i \leq 8}.\ x_i \in \{0..5\}$
(a) A partial solution α	**(b)** A partial solution β	**(c)** A merged partial solution γ

Figure 2.8: Two partial solutions and the partial solution resulting from merging them together.

Recall that when weakening constraints our goal is to ensure that some completion of γ is at least as large (with respect to the cost function) as the optimal completion of both α and β. Since we weaken constraints independently of the objective, we want to ensure that all completions of α and β are still found in the set of completions of γ. Note that in general we do not expect to explore a search space which is as large as the search space of both of the

$$
\begin{gathered}
x_0 = 1 \\
x_1 = 4 \\
x_2 = 5 \\
x_3 = 1 \\
x_4 = 1 \\
\forall_{5 \leq i \leq 8}.\ a_i \in \{0..5\} \\
\sum_{i=0}^{8} x_i \leq 19 \\
\text{maximise} \sum_{i=0}^{8} i x_i
\end{gathered}
$$

Figure 2.9: A completion problem including a weakened `lin_le` constraint.

completion problems of α and β, as we expect to bound the completion problem of γ again later in search. A naïve approach would be to weaken the `lin_le` constraint by setting the bound b to be very large, thus allowing all the unassigned variables to take any value in their domain. This is very inefficient (in this case leading to there being 1296 completions of γ,

with the optimal completion of the relaxed problem having a cost of 141) and is equivalent to deleting the constraint from the model altogether. Instead we seek to modify the bound such that the size of the set of completions is as small as possible while supporting all values in the domain of γ. We do so by increasing the bound by the difference between the sum of the assigned variables in α and the bound. This approach is described in Algorithm 2.5. The weakened constraint is given in Figure 2.9 which shows the relaxed completion problem for γ.

In checking that our approach to relaxing the subproblem by way of weakening the constraint is successful, we find that the largest cost completion of α, γ and β have costs 66, 41 and 68. Note that weakening the constraint, rather than relaxing the problem by removing it, leads to a much tighter upper bound on the cost of the optimal solution ($+2$ rather than $+75$).

Algorithm 2.5 shows our approach to weakening the `lin_le` constraint. In general our weakening algorithms take as input search nodes, at which we record both variables (with their domains) and constraints. In the case of the linear constraint we expect that the constraint is represented by a data type which includes as a member each summand x_i, each coefficient a_i and the bound b. In Algorithm 2.5 we determine the sum of the variables at u (the lower cost input node to the merge operation), s_u and modify the bound of the constraint recorded at m by adding to it the difference between s_u and the bound recorded at u. Modifying the bound in this way ensures that all combinations of values which had support in the input nodes are able to be assigned in the relaxed node.

Algorithm 2.5: An algorithm for weakening the `lin_le` constraint

```
1  weakenLinLe (Node u , Node m)
2  begin
3      s_u, s_v ← 0, 0
4      b_u ← the bound from the lin_le constraint at u
5      b_m ← the bound from the lin_le constraint at m
6      foreach summand x_i involved with lin_le at l_u do
7          if x_i is assigned in l_u then
8              x ← the value assigned to x_i
9              a ← the coefficent of x_i
10             s_u ← s_u + xa
11     t ← b_u − s_u
12     b_m ← b_m + t
13     update the bound of the lin_le constraint with b_m
```

Proposition 2.2. Relaxing the less than or equal to variant of the linear constraint using Algorithm 2.5 allows for the use of the linear constraint propagator as described and DDBB search.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$, and a search node r which records a completion problem of $\mathcal{P}$, which we will call P'. Let $\mathcal{C}$ include linear summation constraints, and assume that f satisfies the limitations set out in Section 2.4.4. We show that relaxed width truncated search using Algorithm 2.2 returns an upper bound on the cost of the completion problem P'. There are two cases to consider. These are when the path through the search tree from r to the optimal solution of the completion problem is included in the relaxed width truncated search tree (regardless of whether or not the width of the search tree has been bounded), and when it is not.

If the path to the optimal solution is included in the relaxed width truncated search tree then Algorithm 2.2 returns a value at least as large as the cost of the optimal solution.

If the path to the optimal solution does not appear in the relaxed width truncated search tree, we know that there exists some path containing nodes which are the result of a merge operation on two search nodes using Algorithm 2.3. We assume that v has high cost under the objective function f than u, without loss of generality. We now show that Algorithm 2.5 ensures that no values in m are without support due to the linear summation constraint, and therefore admits a locally optimal solution which is at least as large as the locally optimal solutions under u and v.

When we merge nodes using Algorithm 2.3 all values which had support in v are still supported in m by construction. We must show that any value introduced from u is given support via Algorithm 2.5. By including support for these values in u we go not guarantee that a "solution" found by Algorithm 2.2 will be a solution to P', and the cost of the solutions found by Algorithm 2.2 may be greater than the optimal solution to P'. Therefore Algorithm 2.2 returns an upper bound for the cost of the optimal solution of P', as required.

Consider a value k introduced to the domain of a variable x_i from u, which does not have support in m after merging v and u via Algorithm 2.3. We know that k had support at u, otherwise the `linear_le` propagator would have removed k from the domain of x_i at u. If b_u is the bound of the `linear_le` constraint at u, and s_u is the sum of the variables which have been assigned values at u, then the inequality $a_i x_i|_k \leq b_u - s_u$ holds. Therefore, by adding $b_u - s_u$ to the bound of the `linear_le` constraint as recorded at m, we ensure that each value k introduced to m from u has support.

□

The proofs that follow for all of our other weakening algorithms are similar to the proof of Proposition 2.2, until the step where we show that the weakening algorithm provides support for values which have none in m. For these proofs we start at the stage of proving that each value introduced from u to m has support via the weakening algorithm being considered, to

avoid repetition.

2.6.2 Weakening sum greater than or equal to

Our approach to weakening the greater than or equal to variant of the linear summation constraint (which we will denote `lin_ge` to ease presentation) is similar to the approach taken for the less than or equal version. Harvey and Stuckey (1998) also provide a propagator for this variant, which relies on the following equation

$$E = \sum_{i=1}^{n} a_i \overline{x_i} - b \tag{2.3}$$

from which the lower bound for each x_i can be calculated via

$$x_i \geq \overline{x_i} - \frac{E}{a_i} \tag{2.4}$$

Similarly to the `lin_le` constraint, the `lin_ge` constraint not satisfiable if $E < 0$. Again the lower bound of each of the summands can be updated in only two passes; one which calculates E and another which sets the bound of each x_i.

Our approach to weakening the `lin_ge` constraint is given in Algorithm 2.6. We again move the bound to weaken the constraint, but in this case we decrease the value of b.

Algorithm 2.6: An algorithm for weakening the `lin_ge` constraint

```
1  weakenLinGe (Node u , Node m)
2  begin
3      s_u, s_v ← 0, 0
4      b_u ← the bound from the lin_ge constraint at u
5      b_m ← the bound from the lin_ge constraint at m
6      foreach summand x_i involved with lin_ge at l_u do
7          if x_i is assigned in l_u then
8              x ← the value assigned to x_i
9              a ← the coefficent of x_i
10             s_u ← s_u + x × a
11     t ← b_u + s_u
12     b_m ← b_m − t
13     update the bound of the lin_ge constraint with b_m
```

Proposition 2.3. Weakening the greater than or equal to variant of the linear constraint using Algorithm 2.6 allows for the valid simultaneous use of the linear constraint propagator as described and DDBB search.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes linear summation constraints, and assume that f satisfies the limitations set out in Section 2.4.4. The proof of Proposition 2.3 is similar to the proof of Proposition 2.2 and differs only when considering how we ensure that all values added to m from u are have support.

Any value recorded at the node u has support at u because for all unassigned x_i and for any value in their domain x, $a_i x \geq b_u + s_u$, where b_u is the bound of the `linear_le` constraint and s_u is the sum calculated over the assigned variables, both as recorded at u. By subtracting $b_u + s_u$ from the bound of the `linear_ge` constraint as recorded at m, b_m, we ensure that all of the values introduced to m from u have support. □

2.6.3 Weakening equal to

The equal to variant of the linear summation constraint (which we will denote `lin_eq` to ease presentation) can be propagated by simultaneously updating the lower and upper bounds of the summands by the inequalities given in Equation 2.3 and Equation 2.4.

Our approach to weakening the `lin_eq` constraint is to impose both a `lin_le` constraint and a `lin_ge` constraint simultaneously over the same summands with the same bound. Weakening these constraints is then equivalent to weakening the corresponding `lin_eq` constraint by allowing a ball of acceptable values around b.

2.7 Weakening max and min constraints

In this section we will provide weakening algorithms for both the max and min constraints. While we can weaken the constraint to get limited use out of it in some cases, for both of these constraints the best approach is sometimes to remove the constraint from the problem altogether. This is to avoid calling the propagator when we know that it will never prune any values from unassigned variables.

2.7.1 Weakening the max constraint

The $\texttt{max}(X, x_{max})$ constraint requires that the variable x_{max} takes the maximum value assigned to a set of integer variables $X = \{x_0, ..., x_n\}$. Beldiceanu (2001), provides propagators for the `max` and `min` constraints that achieve bounds consistency in the following way:

- If x_{max} is not assigned this constraint can be propagated by iterating over the set of variables

X, and setting the lower bound of x_{max} to be the largest value assigned to any variable in X. Similarly `min` can be propagated by iterating through X and setting the upper bound of x_{min} to be the smallest value in X.

- On the other hand, if x_{max} is assigned then the `max` constraint can be propagated by iterating over the set of variables X and setting the upper bound of all unassigned variables to the value of x_{max}. Similarly for `min` if x_{min} is assigned then the constraint can be propagated by iterating over the set of variables X and setting the lower bound of all unassigned variables to be the value of x_{min}

$$\forall_{0 \leq i \leq 5}.\ x_i \in \{1, \ldots, 9\}$$

$$x_{max} \in \{1, \ldots, 9\}$$

$$\texttt{max}(X, x_{max})$$

$$\text{maximise} \sum_{i=0}^{5} i x_i - x_{max}$$

Figure 2.10: An example problem including a max constraint.

We motivate the need to weaken the max constraint by example. Figure 2.10 shows an example problem which includes a single max constraint and Figure 2.11 includes two partial solutions to this problem, α and β, and the partial solution, γ, obtained from merging them together using Algorithm 2.3. Since x_{max} was unassigned in both nodes the domain of x_{max} in γ is the union of the domains of x_{max} in α and β. However, there is no support for x_{max} to take the values 4, 5 and 6 in γ. This is because x_0 is already assigned the value 7 and x_{max} is

		$x_0 = 7$
$x_0 = 4$	$x_0 = 7$	$x_1 = 3$
$x_1 = 2$	$x_1 = 3$	$x_2 = 1$
$x_2 = 3$	$x_2 = 1$	$x_3 = 4$
$x_3 = 1$	$x_3 = 4$	$x_{4,5} \in \{1..9\}$
$x_{4,5} \in \{1..9\}$	$x_{4,5} \in \{1..9\}$	$x_{max} \in \{4..9\}$
$x_{max} \in \{4..9\}$	$x_{max} \in \{7..9\}$	x_{max} is masked with 9
(a) A partial solution α	**(b)** A partial solution β	**(c)** A partial solution γ

Figure 2.11: Two partial solutions of the problem shown in Figure 2.10 where x_{max} is *unassigned* and the completion problem resulting from merging them together.

		$x_0 = 4$
$x_0 = 6$	$x_0 = 4$	$x_1 = 3$
$x_1 = 2$	$x_1 = 3$	$x_2 = 1$
$x_2 = 3$	$x_2 = 1$	$x_3 = 4$
$x_3 = 1$	$x_3 = 4$	$x_{4,5} \in \{1..6\}$
$x_{4,5} \in \{1..6\}$	$x_{4,5} \in \{1..4\}$	$x_{max} = 4$
$x_{max} = 6$	$x_{max} = 4$	x_{max} is masked with 6
(a) A partial solution α	**(b)** A partial solution β	**(c)** A partial solution γ

Figure 2.12: Two partial solutions of the problem shown in Figure 2.10 where m is *assigned* and the completion problem resulting from merging them together.

required to be assigned the maximum value assigned to any x_i. In this case, where the lower bound of x_{max} is not equal in α and β, we mask x_{max} with the largest value in its domain.

Next we consider the case where x_{max} was assigned a value in both partial solutions α and β. Figure 2.12 gives an example of merging two such partial solutions. The need for a weakening algorithm is made apparent by Figure 2.12c, where the variables x_4 and x_5 do not have support for the values 5 and 6, since the maximum allowed value is 4. We overcome this issue by masking x_{max} with the largest value assigned to it across both input nodes α and β.

Algorithm 2.7 shows our approach to weakening the `max` constraint. We assume that the constraint is represented by a data type which includes as members each variable x_i and the variable x_{max}. First we check which of the cases, described above, is relevant (on line 3), and if x_{max} is unassigned in either α or β and their lower bounds are distinct we remove the constraint from the model (line 5). In the case where x_{max} is assigned in both α and β we mask x_{max} with the smallest value assigned to x_{max} across both partial solutions (lines 7 and 8).

We conclude our treatment of the `max` constraint with the following proposition:

Algorithm 2.7: An algorithm which weakens the max constraint.

```
1 maskMax (Node u , Node v, Node m)
2 begin
3     u_max ← x_max at node u
4     v_max ← x_max at node v
5     if the lower bounds of u_max and v_max are not equal then
6         ub ← the largest value in the domain of u_max and v_max
7         mask x_max at node m with the value ub
```

Proposition 2.4. Weakening the `max` constraint using Algorithm 2.7 allows for the valid simultaneous use of the `max` together with DDBB search.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes `max` constraints, and assume that f satisfies the limitations set out in Section 2.4.4. Again, the proof of Proposition 2.4 is similar to the proof of Proposition 2.2, until we have to show that values introduced to m from u are ensured support by Algorithm 2.7, after nodes u and v are merged to create a new node m.

Let x_i be a variable involved in the `max` constraint whose domain, D_{x_i}, includes a value k which is does not have support at the node m after the merge has occurred. All values introduced to m from v have support by construction via Algorithm 2.3, so k has been introduced to m from u and k is greater than the maximum value in the domain of x_{max}. By masking x_{max} at node m with the value of the upper bound of x_{max} at node u, we ensure that k has support when `max` is propagated.

□

2.7.2 Weakening the min constraint

The approach to weakening the min constraint is very similar to the approach to weakening the max constraint, but where we previously considered the lower bounds of x_{max} we now modify the lower bounds of x_{min}. Algorithm 2.8 details these changes.

Algorithm 2.8: An algorithm which weakens the min constraint.

```
1 maskMin (Node u , Node v, Node m)
2 begin
3     u_min ← x_min at node u
4     v_min ← x_min at node v
5     if the upper bounds of u_min and v_min are not equal then
6         lb ← the smallest value in the domain of u_min and v_min
7         mask x_min at node m with the value lb
```

2.8 (Not) weakening the element constraint

In this section we discuss the `element` constraint and show that when merging nodes using Algorithm 2.3 there there is no need to provide an algorithm to weaken the constraint.

The `element` constraint requires that, for an array of integers a and integer variables i and e, $a[i] = e$.

An example `element` constraint is shown in Figure 2.13, where, as an example, setting $i = 1$ forces $e = 1$. This constraint can be propagated by removing values from e which do not correspond to a value in the array indexed by some value in the domain of i, or by removing values v from i if $a[v \in D_i]$ does not appear in the domain of e.

$$a = [1, 0, 0, 1, 1]$$
$$i \in \{1, \ldots, 5\}$$
$$e \in \{0, 1\}$$

Figure 2.13: An example `element` constraint with an array of integers a and two integer variables i and e.

Proposition 2.5. The `element` constraint can be used with the branch and bound algorithm given in Algorithm 2.4 without needing to be weakened during construction of relaxed width truncated search trees.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes `element` constraints, and assume that f satisfies the limitations set out in Section 2.4.4. Again, the proof of Proposition 2.5 is similar to the proof of Proposition 2.2, until we have to show that all values introduced to m from u and v are ensured support by construction, after nodes u and v are merged to create a new node m.

By propagating the constraint as described above, either both the variables i and e will be assigned a value or both i and e are unassigned at each of the nodes u and v. Consider a value k introduced to the domain of i at m from u. If the constraint is propagated as described above then there will be a value l in the domain of e at m introduced from u which supports k.

□

2.9 (Not) weakening the absolute value constraint

In this section we discuss the absolute value constraint and show that it too does not require a weakening algorithm in order for it to be used during search using DDBB as given in Algorithm 2.4.

The absolute value constraint requires that a variable x be assigned a value which is the absolute value of the value assigned to y, that is $x = |y|$. This constraint can be propagated

by iterating through the values in the domain of x, $v \in D_x$, and removing v from D_x if the domain of y does not contain v or $-v$. Similarly, iterating through the values in the domain of y, $w \in D_y$, allows the removal of any w from D_y such that the domain of x does not contain $|w|$.

Proposition 2.6. The absolute value constraint may be used to model problems which are solved using with Algorithm 2.4 without the need for the constraint to be weakened after the creation of relaxed nodes using Algorithm 2.3.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes absolute value constraints, and assume that f satisfies the limitations set out in Section 2.4.4. Again, the proof of Proposition 2.6 is similar to the proof of Proposition 2.2, until we have to show that all values introduced to m from u and v are ensured support, after nodes u and v are merged to create a new node m.

When u and v are merged together to create m, all values introduced from v have support at m by construction. Consider a value k in the domain of x introduced at node m from node u. If the absolute value constraint is propagated as described above then another value l in the domain of y is introduced at node m from node u which supports k at m. This value l is in turn also supported by the inclusion of k. □

2.10 Weakening reified constraints

Constraint reification involves assigning a boolean variable a value dependant on whither or not a constraint is satisfied.

We weaken a constraint reification by weakening the constraint being considered. The boolean variable is set to true if the weakened constraint is satisfied.

2.11 Experimental results

In this section we show empirically that using decision diagram branch and bound search as the search scheme in a constraint solver, together with our constraint weakening algorithms, at times outperforms a forward checking approach. Recall that forward checking search proceeds in a depth first manner and at each search node constraints are propagated such that inconsistent values are removed from the domains of unassigned variables. To compare these two approaches to search we have implemented forward checking in our own solver,

in order to isolate the change of search scheme as the only difference. Adding in our search scheme in an existing constraint solver could lead to us comparing our experimental code against optimised industrial codes. Neither our DDBB or forward checking search schemes compares favourably to industrial quality constraint solvers, which on many problem instances outperform our code by orders of magnitude.

2.11.1 Car factory sequencing problem

The constraints for which we have proposed weakening algorithms for in this chapter allow us to tackle the car factory sequencing problem. In this problem there are n cars which must be produced by the factory, and these cars are not identical since customers can pay more for extra options to be added to a base model. The factory consists of an assembly line of stations at which workers install the various optional parts (e.g. air-con, spoilers, rear diffusers). These stations have varying throughput, with some stations taking longer to complete the installation of their parts than others. Each station's throughput is recorded as the fraction of the total cars passing along the assembly line which it can handle, for example 1 in every 2 or 2 in every 5. The sequence of cars passing along the assembly line must be built so that these constraints are satisfied.

This problem has been studied widely. Dincbas, Simonis, and Hentenryck (1988) show that combining logic programming with constraint programming techniques leads to a tool which can solve instances of the car factory sequencing problem much more efficiently than the state of the art techniques of the time. Regin and Puget (1997) introduce a new filtering algorithm for propagating sequence constraints (i.e. that 3 in every 5 cars passing along the assembly line must not stop at a particular station). They use the car factory sequencing problem as an illustrative example of the competitiveness of their approach and offer a difficult set of benchmark problems. Both of these previous works stress that the car factory sequencing algorithm is difficult and the problem was proven to be NP-complete by Gent (1998). The problem was the focus of the 2005 ROADEF challenge and was proposed to the challenge by the French car manufacturer Renault. Solnon et al. (2008) review the exact and approximation approaches taken by submissions to the challenge. Siala, Hebrard, and Huguet (2014) give an algorithm for propagating the kinds of sequence constraints found in the car factory scheduling problem in linear time and follow this work with a study of heuristics for the problem (2015). There has also been interest in using SAT solvers for the problem (Mayer-Eichberger and Walsh, 2013).

The model for the car factory sequencing problem is given as a Minizinc code listing in Appendix A. It makes use of all three types of linear summation constraint, `lin-le`, `lin-ge` and `lin-eq` as well as the `element` constraint and the reified `lin-eq` constraint. Fig-

							0	1	0	1	1	0	0	1	0	1	1	0
10	5	6					1	0	0	0	1	0	2	0	1	0	0	1
1	2	1	2	1			5	1	1	0	0	0	5	1	1	0	0	0
2	3	3	5	5			2	0	1	0	0	1	1	0	0	0	1	0
0	1	1	0	1	1	0	4	1	0	1	0	0	4	1	0	1	0	0
1	1	0	0	0	1	0	3	0	1	0	1	0	3	0	1	0	1	0
2	2	0	1	0	0	1	3	0	1	0	1	0	2	0	1	0	0	1
3	2	0	1	0	1	0	4	1	0	1	0	0	4	1	0	1	0	0
4	2	1	0	1	0	0	2	0	1	0	0	1	3	0	1	0	1	0
5	2	1	1	0	0	0	5	1	1	0	0	0	5	1	1	0	0	0

(a) The problem instance **(b)** A solution **(c)** The optimal solution

Figure 2.14: An instance of the car factory sequencing problem, a solution and the optimal solution. The standard form of such instances is that the first line records the number of cars to be made, the number of options available and the number of classes of car. The following lines then give each class with its number proceeded by which options are chosen for the class. This particular instance instance is given as a motivating example in (Dincbas, Simonis, and Hentenryck, 1988) and the optimal solution is found by our solver in less than a tenth of second.

ure 2.14 shows an example instance of the car factory scheduling problem and its solution.

2.11.2 Evaluating the effects of heuristic choices

In this section we study the effect that altering some search heuristics has on the time it takes our solver to find optimal solutions and then terminate having proved that they are optimal (that no better solution can be found).

Search heuristics are commonly employed in order to improve execution times of solvers. In particular, altering the order in which variables are chosen to branch on during search (variable ordering heuristics) and altering the order in which these variables are assigned values (value ordering heuristics) can have a huge effect both the number of nodes visited during search and the execution time of a solver. Ordering heuristics can be static, where the order is fixed before search, or dynamic, where the order is chosen during search based on how search proceeds. Common variable ordering heuristics include static orderings such as input order and occurrence (where we choose branch variables in descending order of the number of constraints they are involved in). An example of a dynamic variable ordering is first fail, where the variable with the smallest domain is chosen as the next branch variable.

We will compare our approach of using DDBB as the search scheme in a constraint solver against forward checking for multiple different variable-value ordering combinations, but we begin by examining a heuristic choice introduced in DDBB. When using decision diagram

branch and bound as the search scheme in our solver is the choice of what value the maximum width of width restricted truncated search trees are set to. The effects of altering the permitted width of truncated decision diagrams in optimisation settings are well known. Larger permitted widths allow for stronger bounds, with the trade off being that it takes more time to find these bounds due to the larger number of nodes being explored. A wider restricted search might include more leaf nodes, and this improves the chance of finding a good incumbent solution and quickly improving bounds during search. However, doing more work by exploring a wider search tree does not guarantee that a better solution will be obtained. Therefore there is a balance to be found by doing enough work to improve the quality of solutions found without needlessly adding to the number of search nodes required to find them.

A further complication is that in Algorithm 2.4 we make use of restricted width truncated search trees to decide which nodes to branch on during search. This leads to the situation where there are multiple "good" choices of width, as larger widths will improve the quality of solution found but while branching deeper in the search tree. At times we find that having more branching is instead beneficial.

Figure 2.15 shows a plot of the cumulative number of CSP instances solvable in a given time. The very worst performing width is the smallest chosen and the next worst is the maximum chosen. We would expect this behaviour, as when the width is too large the restricted search trees rarely have their width truncated, and very many restricted searches for larger widths are then essentially just breadth first search. Similarly, when the maximum permitted width is too small search tends towards a depth first approach, only in this instance we do not backtrack to branch but branch at the layer of search where the maximum permitted width is first exceeded. Neither of these situations are ideal. The best possible choice of width for each individual instance varies, which is why we see that when counting the number of problems solvable cumulatively in a given under time very many choices of width perform similarly.

Another heuristic choice we introduce is the maximum permitted width of relaxed width truncated search. We should expect that varying the width of relaxed search has a similar effect to varying the width of restricted search. Wider search trees more closely approximate the exact tree rooted at the same subproblem, and this can lead to stronger bounds as less merge operations occur. However this is again not guaranteed to be the case and in an ideal case we would not want to visit more search nodes than necessary to get good bounds.

We again show cumulative results for the CSP, showing how many instance can be solved in a given time in Figure 2.16 when we vary the maximum permitted width of relaxed search. While we do see an some limited improvement when using greater widths it is mostly the smaller choices which perform best.

The last heuristic choice we introduced in Algorithm 2.4 is the order in which nodes are

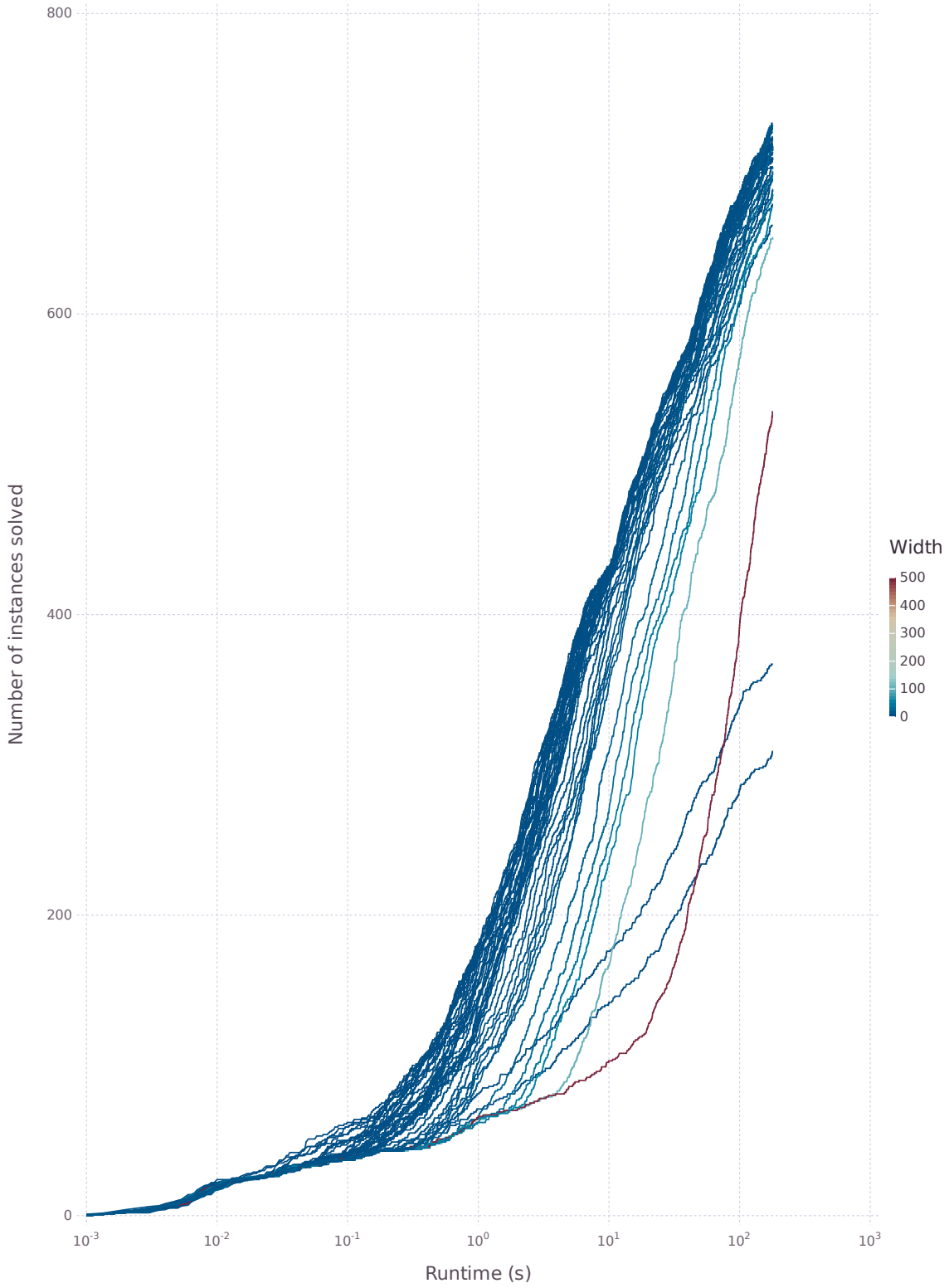

Figure 2.15: Cumulative number of CSP instances solved in a given time as the width of restricted search is varied.

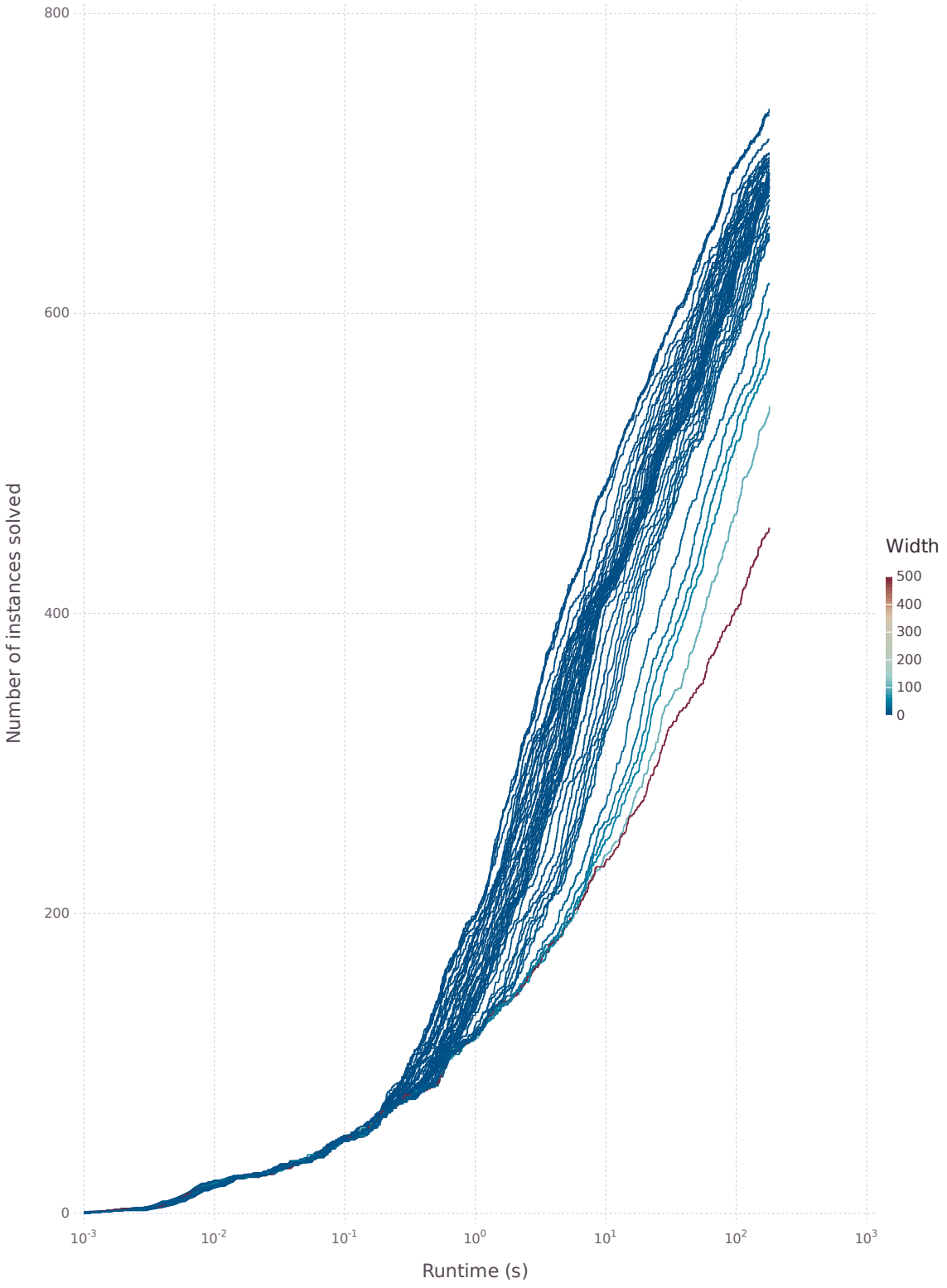

Figure 2.16: Cumulative number of CSP instances solved in a given time as the width of relaxed search is varied.

to be merged during construction of relaxed width truncated search trees constructed via Algorithm 2.2. We consider two options for selecting nodes to merge. The first is to merge nodes to the right of the layer being reduced. The reasoning behind this choice is that the variable and value ordering heuristics, if chosen properly, should lead to lower cost nodes being towards the right of layers. The second choice is that we search through each layer and pick the two lowest cost nodes every time a merge operation occurs. This would lead to the best possible reduction of the layer (the merged nodes are kept as low cost as possible) but the extra effort of finding these nodes may not be worth it. Figure 2.17 shows that the effect this choice has on execution times is often minimal when solving the car factory scheduling problem. Even for instances where there is a larger difference in execution times, which approach is faster can depend on the maximum permitted width chosen for relaxed search. When we compare DDBB against forward checking, we take the approach of merging rightmost nodes first to ensure that the merge ordering compliments variable and value ordering heuristics.

It is worth considering that our heuristic choices are not independent of each other. For example changing the maximum permitted width of restricted search might change the level on which branching occurs in the search tree, but equally changing the variable ordering could also change the level at which a layer becomes too wide. This lack of independent parameters makes it very difficult to find a single best choice of heuristics and parameters for a given family of instances or problems. As we continue to compare DDBB against forward checking we will do so against "good" choices of maximum permitted widths, but these will be fixed for all instances.

2.11.3 Comparison with forward checking

To evaluate our approach we chart the execution time and number of search nodes used by our DDBB search approach against the execution time and number of search nodes used by forward checking. We do this for both an input order and first fail variable orderings and minimum and maximum value orderings. We choose these in particular because they are the suggested ordering heuristics for solvers which interface with Flatzinc, and in this case all of them actually impact the order in which variables and values are chosen. When using the forward checking approach an input order variable ordering branches on variables in the static order in which they appear in the model being solved. A first fail variable ordering branches on variables in order of which unassigned variable has the smallest domain, which changes dynamically as search proceeds. In the context of DDBB we can use these variable orderings to

These figures also include a chart showing the relative speed with which these approaches

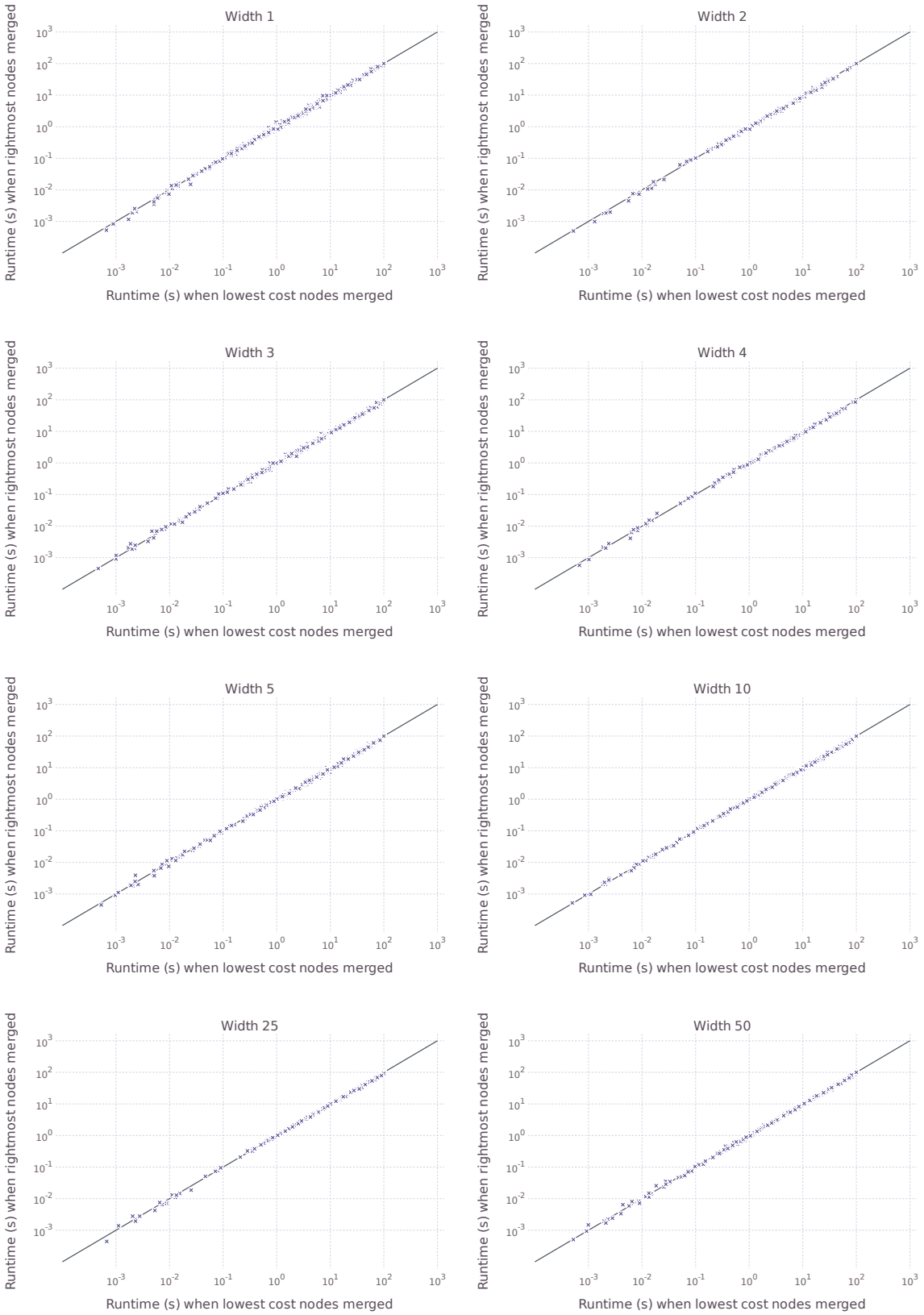

Figure 2.17: Comparing merging low cost nodes versus rightmost nodes when conducting relaxed search on car factory scheduling problem.

explore the search space, confirming that we are not just comparing our DDBB approach against a hobbled forward checking approach and that both approaches manage to evaluate a similar number of nodes per second. choose which variable to assign values to when creating a new layer of search.

Figure 2.18, Figure 2.19, Figure 2.20, and Figure 2.21 show the relative execution times and search nodes used when using an input order and minimum value order, input order and maximum value order, first fail and minimum value order and first fail and maximum value order. For each variable-value ordering combination the DDBB approach outperforms forward checking in around half of the instances we solve, and sometimes by orders of magnitude. While both of these methods of search use the same variable and value ordering heuristics, the different sequence they visit each node in the search tree means that a direct comparison between DDBB search and forward checking for a particular choice of variable-value ordering only tells part of the story.

For each variable-value ordering pair we include charts which show the relative size of the search space when using DDBB versus forward checking, where finding a better incumbent solution more quickly allows for greater pruning of the search space based on the cost of the incumbent. However this benefit of pruning the search space does not always come via relaxed width truncated search. Figure 2.22 shows that for the overwhelming majority of instances, using relaxed width truncated search trees to bound the problem leads to an increase in execution times of our solver. While there is often no benefit in using relaxed width truncated search trees, we can still assert that we are able to prune large parts of the search space due to more quickly finding better solutions than forward checking.

When DDBB performs well, and why

To investigate why DDBB has faster execution times than forward checking for some instances of the car factory scheduling problem, we can focus on how the cost of the incumbent best solution changes over time. Figure 2.23 shows the cost of the best known solution yet found during search against time, for both DDBB and forward checking for a single instance of the CFSP where DDBB completes its execution before forward checking. From this figure we can notice two things, DDBB finds fewer improvements than forward checking, but these improvements are more significant than those found by forward checking. For this instance, forward checking finds a good solution, then backtracks to find improving solutions that are closely related to the initial good solution. DDBB instead continues search by branching at some node at a (comparitively) low depth in the search tree and continues to find under this node a new solution in a different region of the search space from the first. Finding these new solutions requires some luck, but when we do find large improvements on the current

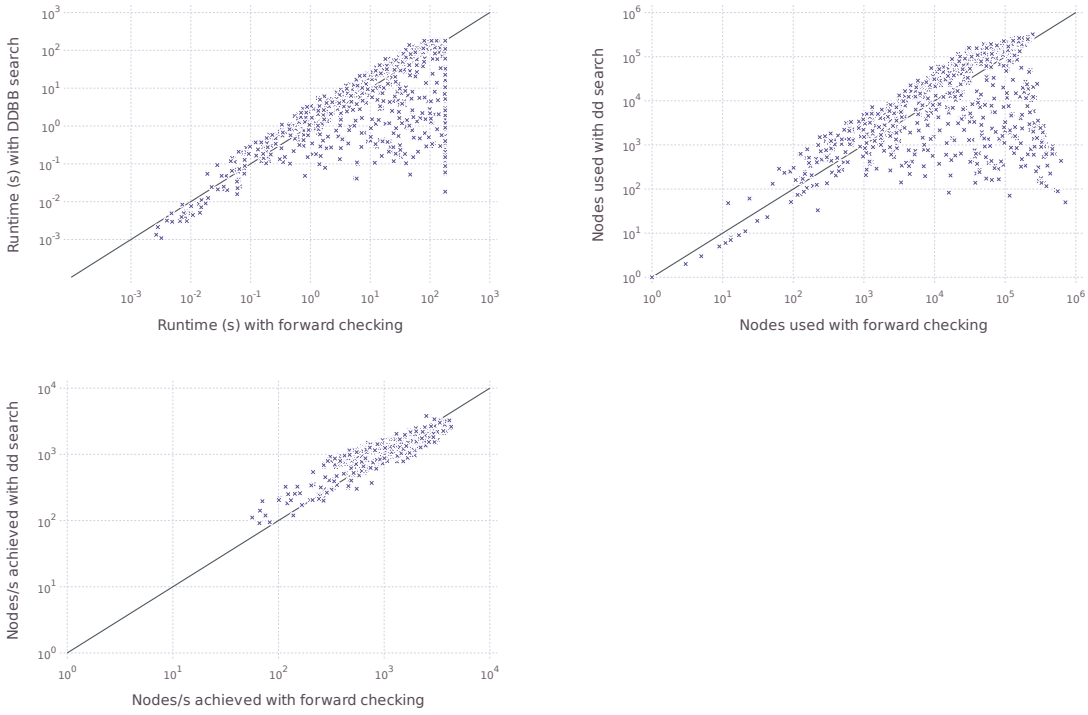

Figure 2.18: DDBB search versus forward checking on the CFSP, where variables are chosen in input order and the minimum value is chosen first from variable domains when branching.

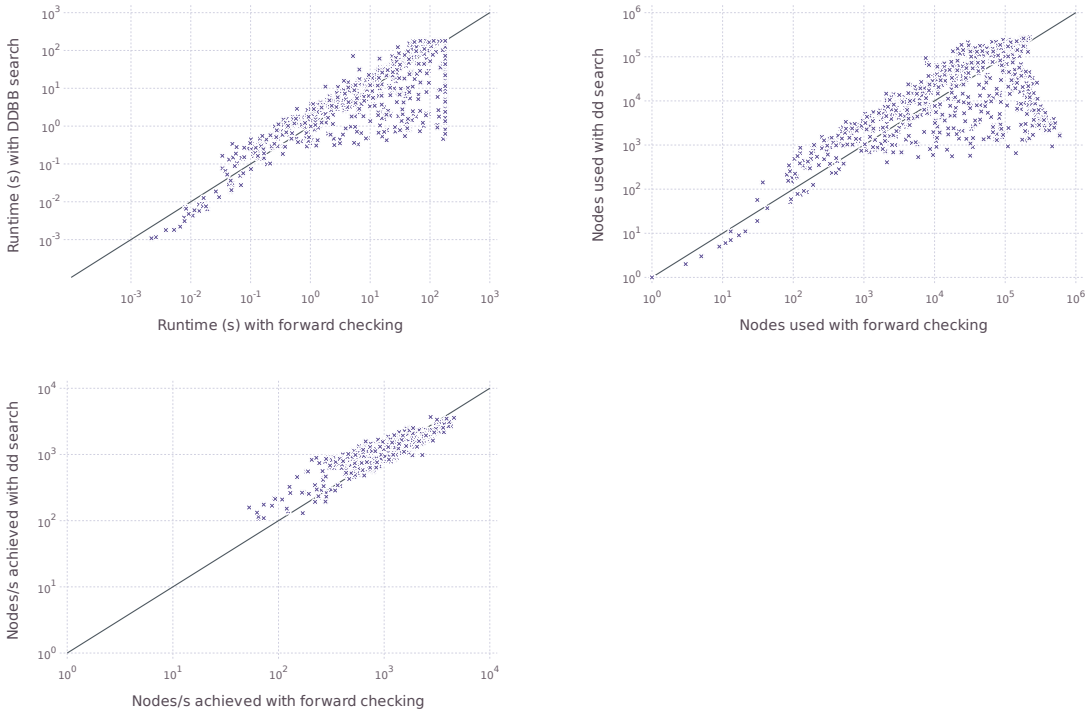

Figure 2.19: DDBB search versus forward checking on the CFSP, where variables are chosen in input order and the maximum value is chosen first from variable domains when branching.

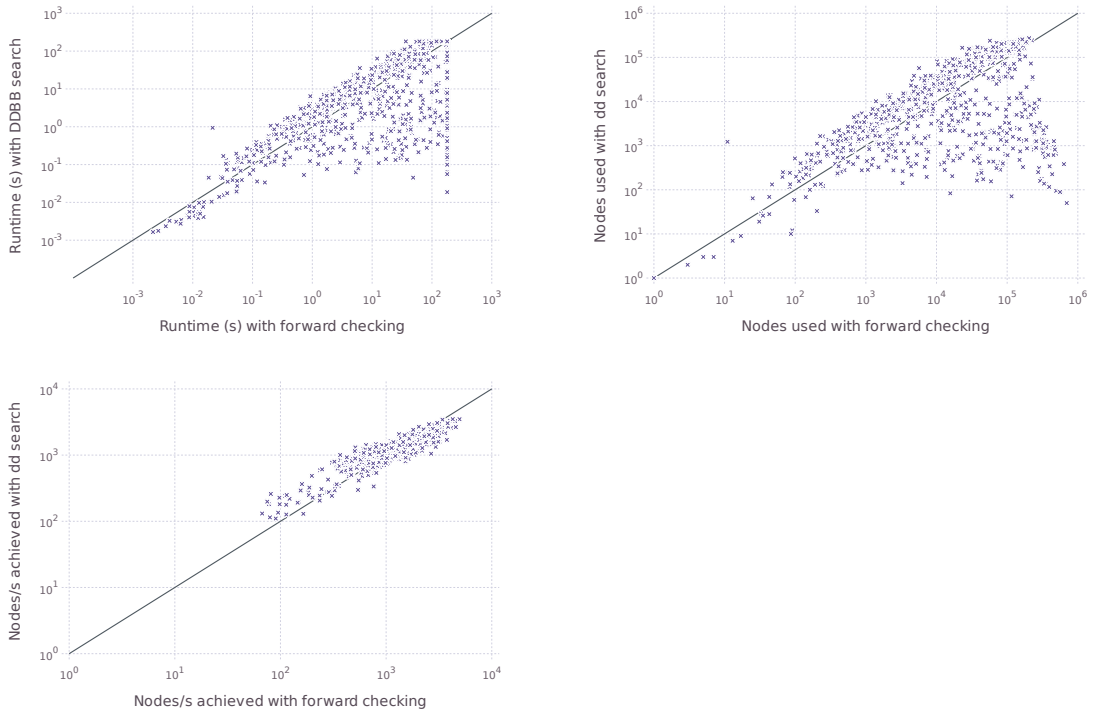

Figure 2.20: DDBB search versus forward checking on the CFSP, where variables are chosen in order of domain size and the minimum value is chosen first from variable domains when branching.

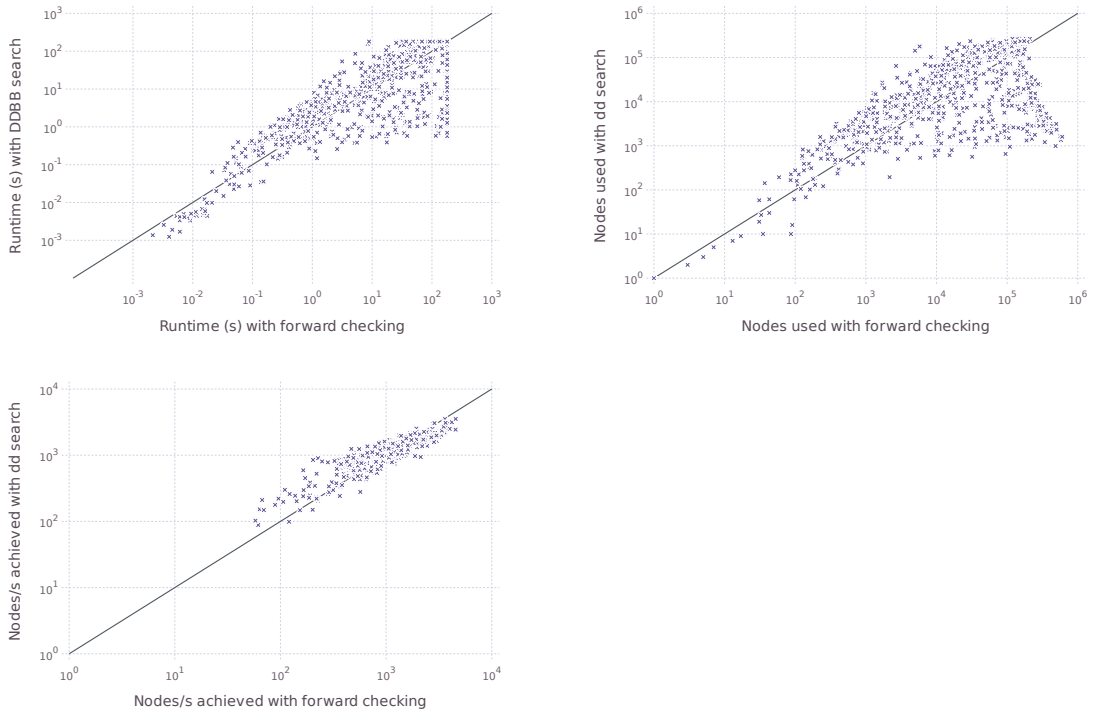

Figure 2.21: DDBB search versus forward checking on the CFSP, where variables are chosen in order of domain size and the minimum value is chosen first from variable domains when branching.

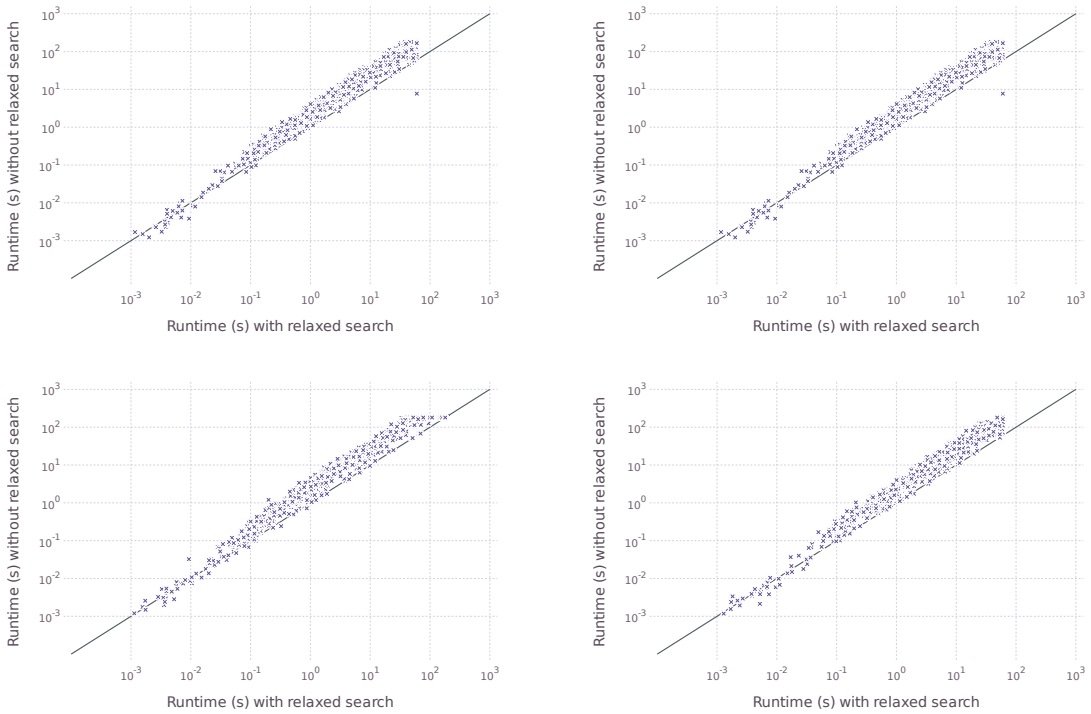

Figure 2.22: DDBB search with and without relaxed search trees on the CFSP, where (from top left, clockwise) variables and values are chosen in order of domain size and the minimum value, domain size and maximum value, in a static order and the maximum value, in a static order and the minimum value.

solution the gains can be significant. On some instances it leads to orders of magnitude faster performance for DDBB. In this instance too the proof is much shorter. This may be due to relaxed width truncated search trees determining that a number of the remaining nodes to be branched on define completion problems whose optimal cost cannot possibly be larger than the cost of the incumbent solution.

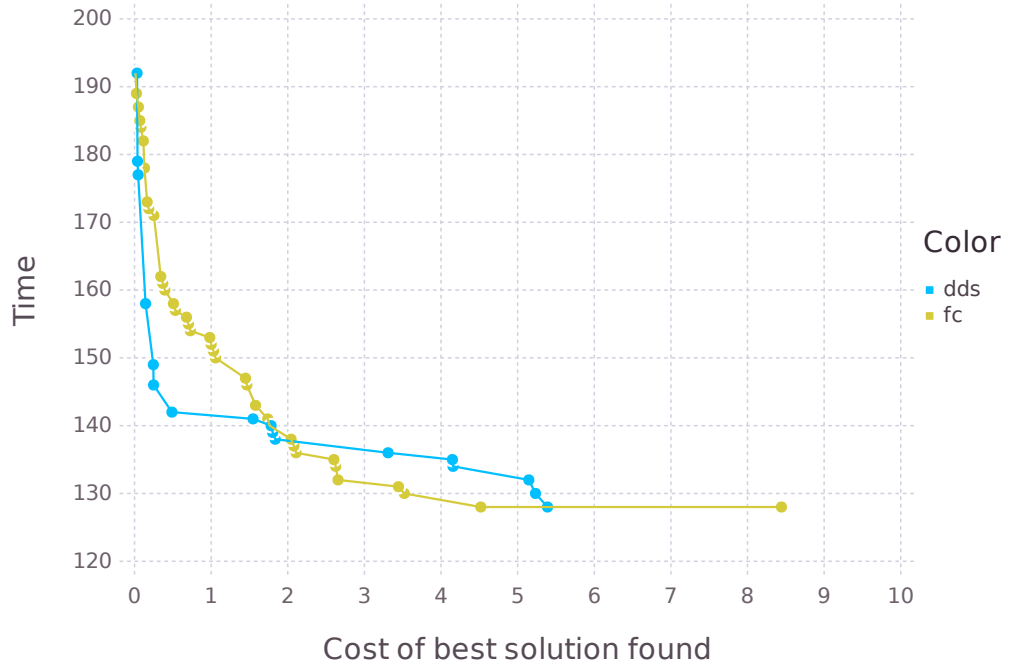

Figure 2.23: Comparing the cost of the best solution found by search against time for both DDBB and FC for an instance of the CFSP.

Maximum independent set problem

Recall that the MISP problem is to find the largest weight set of independent vertices in a vertex weighted graph. The MISP problem includes makes use of `lin_le` constraints, and a model for the problem is given as a Minizinc code listing in Appendix A.

To evaluate the performance of our solver we generate instances of the MISP by randomly assigning weights to vertices in the subgraph isomorphism instances collected from the Stanford Graph Database by Larrosa and Valiente (2002). We choose these benchmark instances as they are have a range of properties: some are connected, some planar, some bipartite. For each of these benchmark instances we create multiple MISP instances by varying both the seed to our generation procedure and the distribution of weights. We consider the clique problem on the compliment graph (where every vertex has weight 1), an even distribution of small weights (1 to a tenth order of the graph), and even distribution of large

weights (1 to half the order of the graph) and an uneven distribution where only a few vertices are given large weights (most weight 1, but can weigh up to half the order of the graph).

Evaluating heuristic choices

Again we evaluate the impact of varying the maximum permitted widths of restricted and relaxed width truncated search trees. Figure 2.24 and Figure 2.25 show the expected behaviour (just as with the CFSP): that the best choice of width for each search is neither too large nor too small.

Comparison with forward checking

Again we evaluate our DDBB based approach against forward checking for different value ordering heuristics (since each decision variable in this problem has $\{0, 1\}$ as its domain). Figure 2.26 and Figure 2.27 compare our approach to forward checking for a minimum value ordering and maximum value ordering respectively. For this problem we do not see then positive results that were seen for the car factory sequencing problem.

Only a handful of instances using the maximum value ordering do we see an improvement in execution time when using DDBB search compared with forward checking. The instances where our approach compares favourable with forward checking are all in the uneven weights class, and through inspecting individual problem instances we see that in these cases we do more quickly find better incumbent solutions than forward checking, but not to the same extent as was the case with the car factory scheduling problem. We again include in each of these figures a chart confirming that both DDBB and forward checking visit a comparable number of nodes per second, to confirm that we are not just comparing against a poorly performing forward checking implementation.

In Figure 2.28 we again compare using relaxed search trees against not using these. The results here are more promising than for the CFSP, with the execution time of a number of instances benefiting from the use of relaxed search trees, but still in many cases using these search trees to bound search only leads to an increase in execution times.

When DDBB performs poorly, and why

Our approach of using DDBB as the search scheme in a constraint programming solver does not perform nearly as well for the MISP as it did for the CFSP. Only for a handful of instances do we see any improvement in execution times using DDBB. Figure 2.29 shows

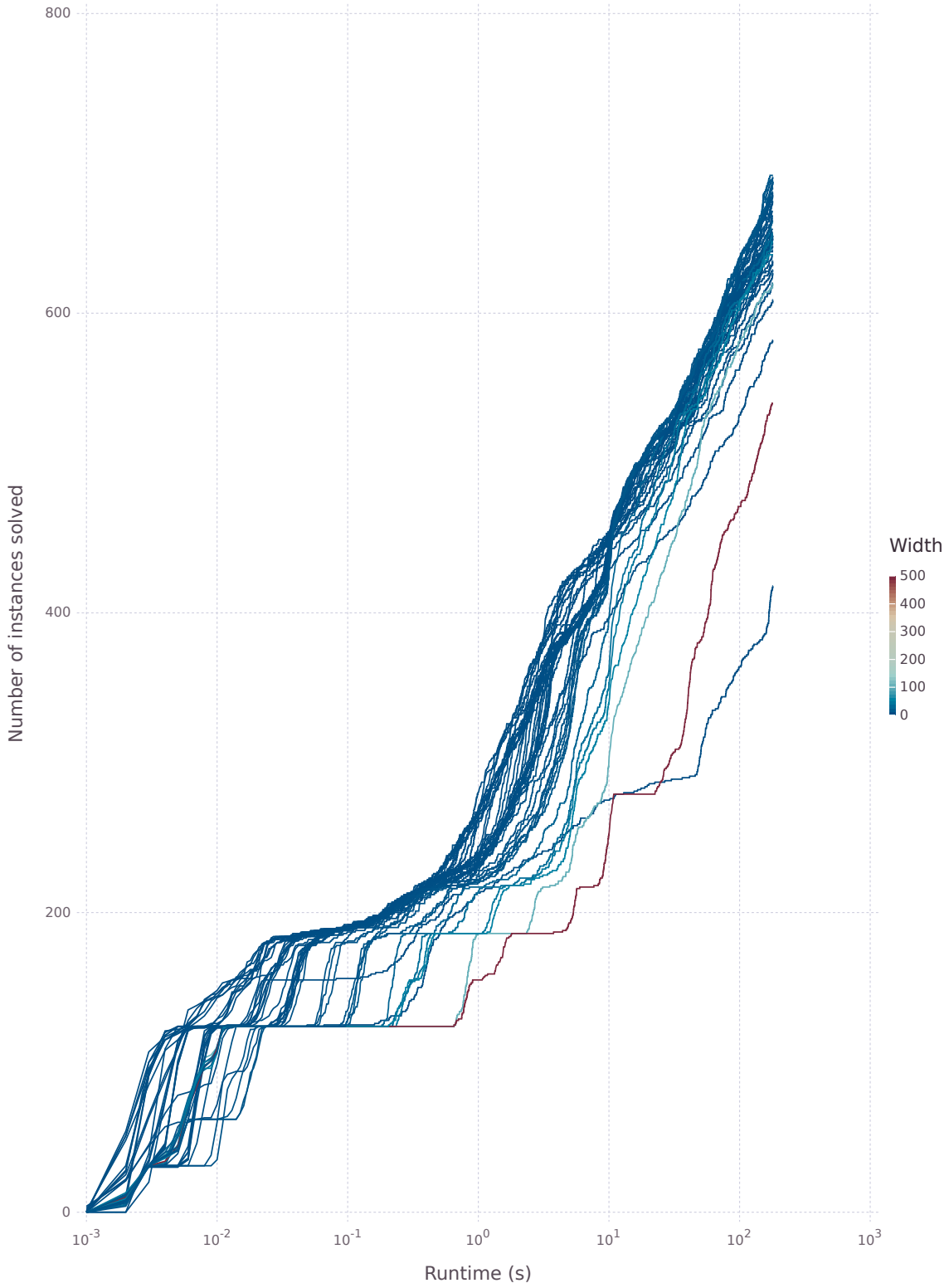

Figure 2.24: Cumulative number of MISP instances solved in a given time as the width of restricted search is varied.

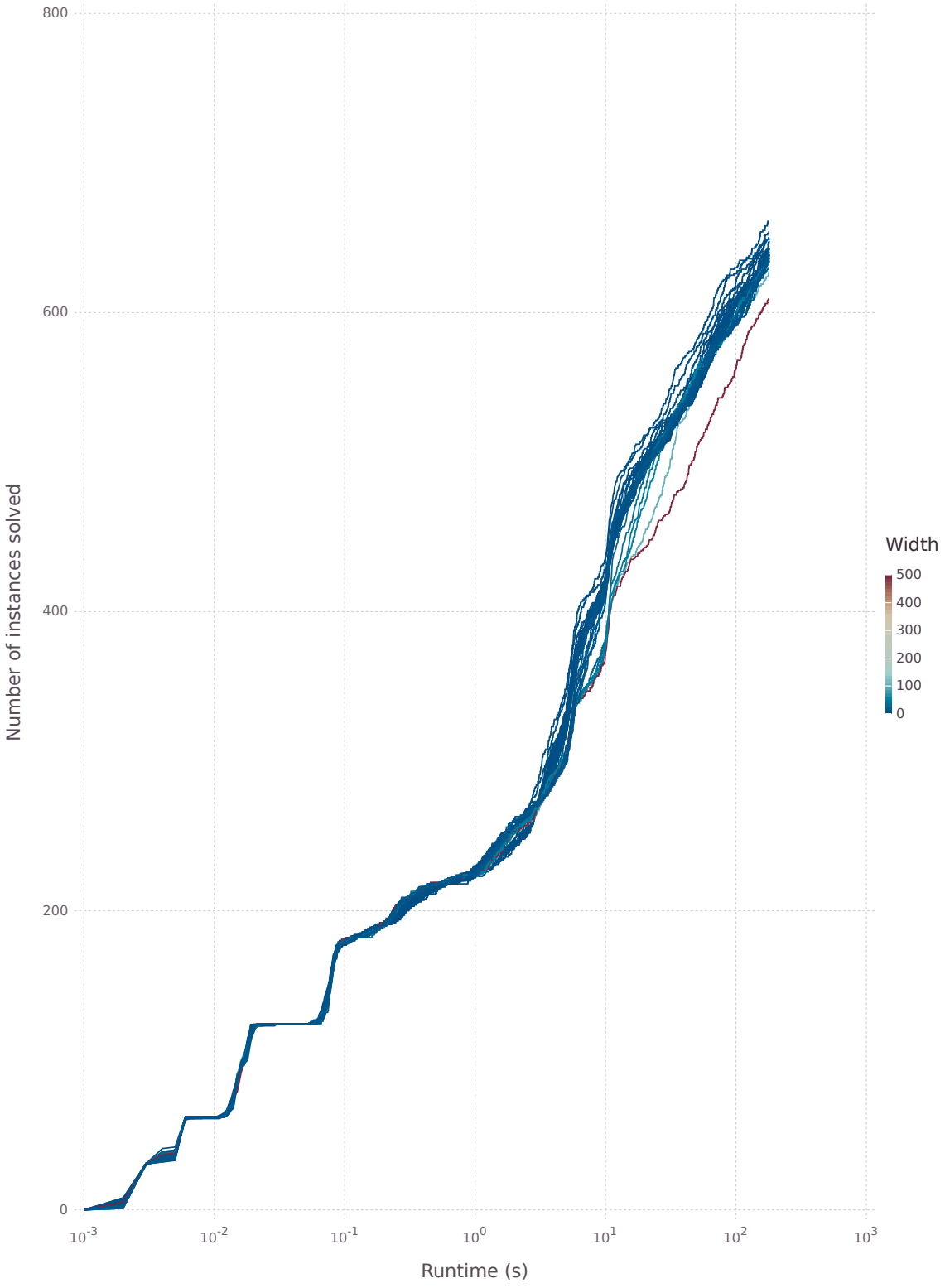

Figure 2.25: Cumulative number of MISP instances solved in a given time as the width of relaxed search is varied.

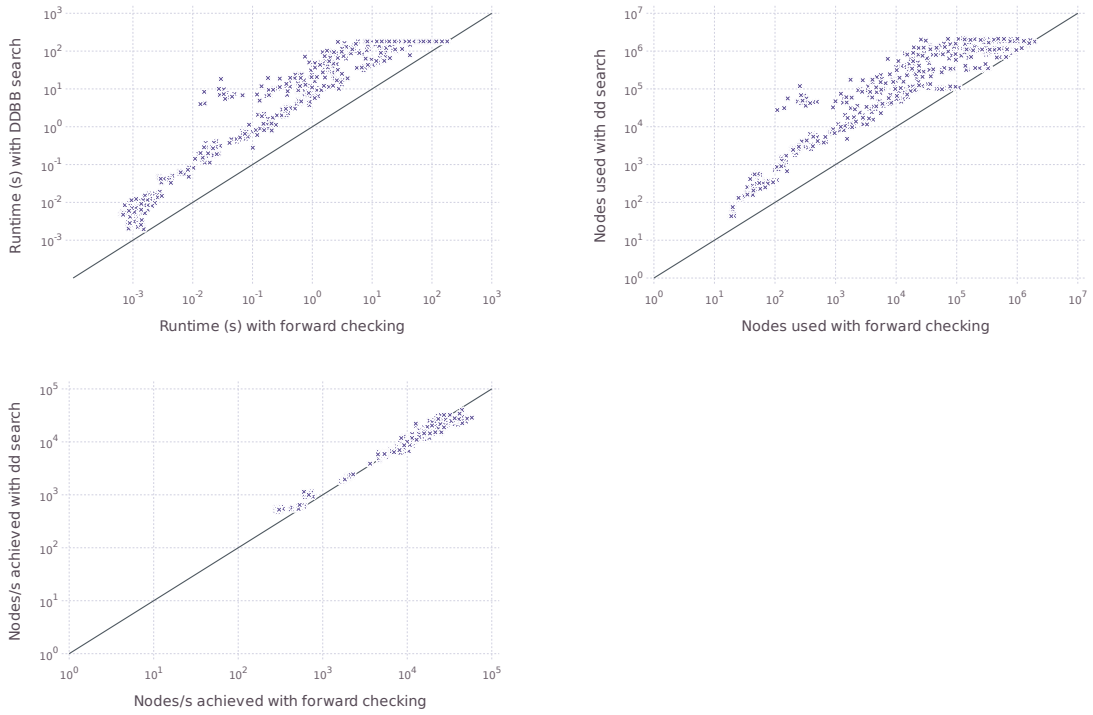

Figure 2.26: DDBB search versus forward checking on the MISP, where variables are chosen in order or domain size and the minimum value is chosen first from variable domains when branching.

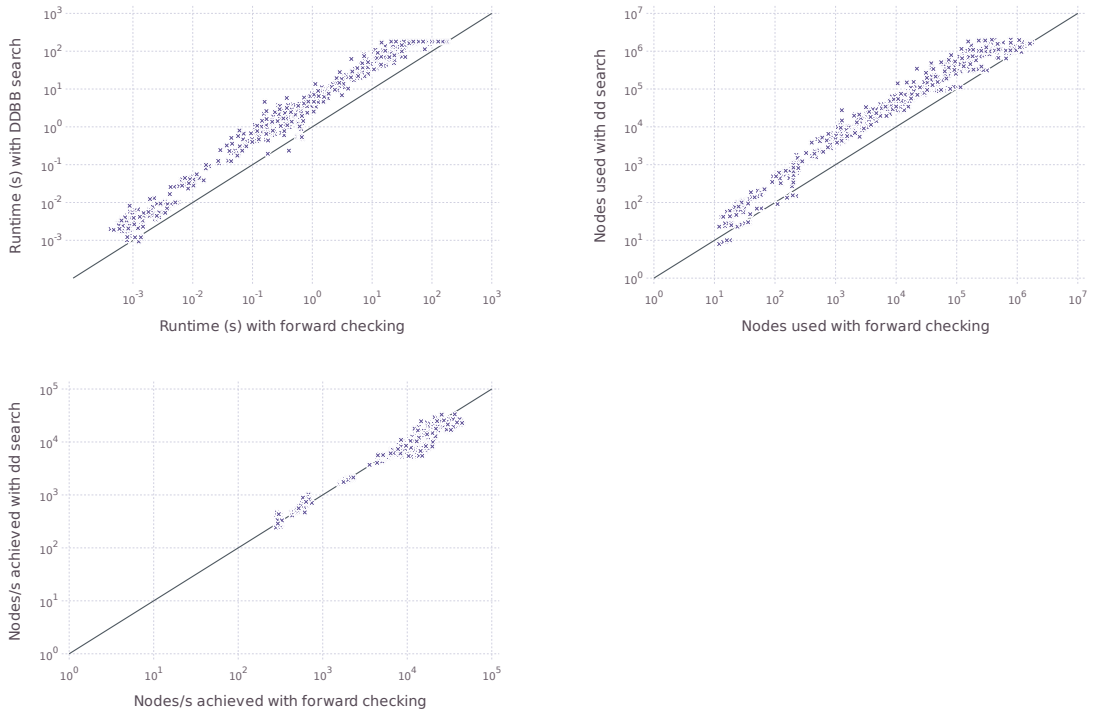

Figure 2.27: DDBB search versus forward checking on the MISP, where variables are chosen in order or domain size and the maximum value is chosen first from variable domains when branching.

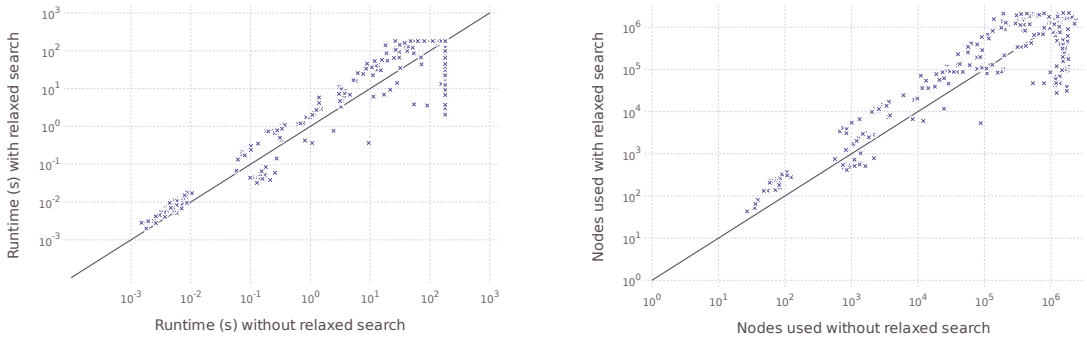

Figure 2.28: DDBB with relaxed diagrams or without.

the cost of the best known solution yet found during search against time, for both DDBB and forward checking for a single instance of the MISP where forward checking completes its execution before DDBB. Here both algorithms use a static variable ordering and minimum value ordering. From Figure 2.29 we can see that while forward checking follows the leftmost branch of search straight to the solution that includes no vertices in milliseconds, DDBB takes seconds to find the same solution, as DDBB does much more work before exploring any leaf nodes. For the instances of the MISP investigated we seldom see the behaviour where DDBB happens upon good solutions earlier than forward checking due to the different order in which it explores the search space. What is shown in Figure 2.29 is typical across many of these instances. These quick incremental improvements made by forward checking mean that it more quickly reduces the search space based on the cost of the current best solution than DDBB can.

The instances which DDBB performs well on are never from the clique complement instances and often from the instances with an uneven distribution of vertex weights, where the same behaviour as outlined for the CFSP is exhibited and DDBB can find larger improvements than forward checking can.

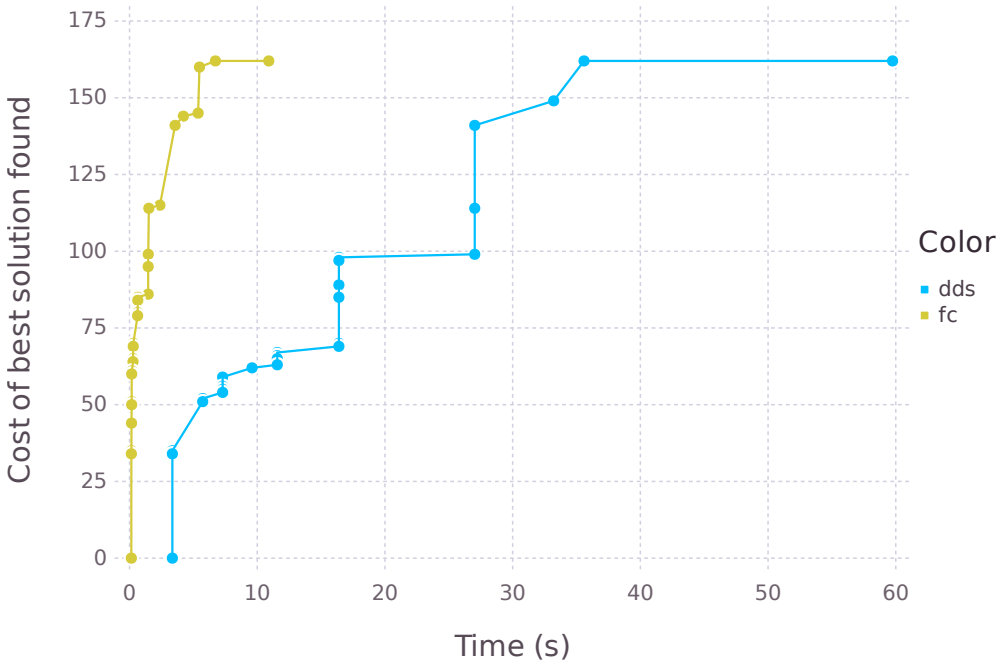

Figure 2.29: Comparing the cost of the best solution found by search against time for both DDBB and FC for an instance of the MISP.

Chapter 3

Weakened all different constraints

3.1 Introduction

In this chapter we consider the all different constraint and other constraints in the "all different" family. We provide weakening algorithms so that these constraints can be used in our decision diagram influenced branch and bound search scheme inside a constraint optimisation solver.

3.2 Background

The all different constraint (which we will refer to as `alldiff`) requires that a set of variables all take distinct values. It is widely used when modelling problems for solution via a constraint solver, appearing in models for quasigroup completion (Pesant, Quimper, and Zanarini, 2014) and balance quasigroup with holes (Kautz et al., 2001), and sports scheduling (Schaerf, 1999). In this section we outline how the `alldiff` constraint is propagated, so that we can understand how our approach to weakening the `alldiff` constraint works together with a propagator for the constraint.

Perhaps the most simple way of propagating the `alldiff` constraint is via its pairwise not-equals decomposition (Figure 1.4 shows such a decomposition of the problem modelled in Figure 1.3). This decomposition works by imposing a `not-equal` constraint for every pair of variables involved in the `alldiff` constraint. In Chapter 1 we used this decomposition of the `alldiff` constraint to show the effect that inference can have when solving problems. It was clear that the decomposition lead to very little inference, failing to determine that 2 values could not be shared across 4 variables which were required to take distinct value assignments.

The amount of time spent at each node inferring which values can be deleted from the domains

of unassigned variables by propagation can have a profound effect on the execution times of solvers across different problems and problem instances. Stergiou and Walsh (1999) show that using the correct strength of inference is important to solve problems efficiently, comparing different levels of consistency. In his survey van Hoeve (2001), also explores the effect of various strengths of inference, pairwise decomposition, bound and range consistency and generalised arc consistency.

The standard algorithm for maintaining generalised arc consistency for the `alldiff` constraint was given by Régin (1994). Régin's algorithm is not simple, relying on several results from graph theory. We will explain the algorithm here at a high level, but in enough detail to ensure that it is clear how our weakening algorithm interacts with this `alldiff` propagator.

Gent, Miguel and Nightingale (2008), survey various optimisations proposed for improving the performance of algorithms which maintain general arc consistency for the alldifferent constraint. In particular they consider domain counting (introduced by Quimper and Walsh (2005) and further developed by Lagerkvist and Schulte (2007) as an optimisation. They also provide in depth implementation details regarding Régin's algorithm.

3.2.1 Regin's alldifferent propagator

For a problem with n variables whose domains share values $\{1, \ldots, d\}$, Régin (1994)'s algorithm begins by constructing a bipartite variable-value graph, B. A graph is *bipartite* if its vertices can be arranged into two sets such that no two vertices in either set are connected by an edge. This graph has a vertex for each variable and each value, giving a set of vertices $V = \{x_1, \ldots, x_n, 1, \ldots, d\}$. An edge is added to the graph between a vertex corresponding to a variable and a vertex corresponding to a value if a variable's domain includes the value. It follows that this graph is bipartite as there is never an edge between two vertices which both represent either variables or values. After the graph B is constructed a maximum cardinality matching of B is then sought. A *matching* of a bipartite graph is a set of edges in the graph such that not more than 1 edge is incident on any vertex. A *maximum cardinality matching* M is a matching with the greatest number of edges. M is said to have size $|M|$ (where $|M|$ is the number of edges in the matching M). When $|M| < n$ not all variables can be given distinct values and so the constraint is not satisfiable. Figure 3.2 shows such a matching for the problem modelled in Figure 3.1. Figure 3.4 gives an example of problem instance where the size of the matching tells us that an `alldiff` constraint is not satisfiable.

To find a maximal matching M of the bipartite variable value graph B the problem is transformed into a corresponding maximal flow problem. The graph B is augmented with source and terminal vertex, with the source vertex being connected to each variable vertex,

$$
\begin{aligned}
x_{1,2} &\in \{0, 1\} \\
x_3 &\in \{0, 1, 2, 3\} \\
x_4 &\in \{2, 3\} \\
\texttt{alldiff}(x_1, x_2, x_3, x_4)
\end{aligned}
$$

Figure 3.1: An example problem involving a single `alldiff` constraint.

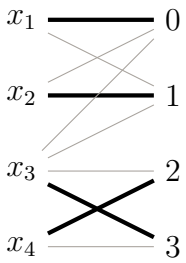

Figure 3.2: A maximum cardinality matching on the variable value graph obtained from the problem modelled in Figure 3.1. The bold edges represent the edges chosen to be in the matching M.

$$
\begin{aligned}
x_{1,2,3} &\in \{0, 1\} \\
x_4 &\in \{1, 2\} \\
\texttt{alldiff}(x_1, x_2, x_3, x_4)
\end{aligned}
$$

Figure 3.3: An example of an unsatisfiable problem involving a single `alldiff` constraint.

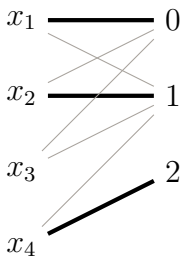

Figure 3.4: A maximum cardinality matching on the variable value graph constructed from the problem modelled in Figure 3.3, where $|M| < n$. From the size of the matching we can ascertain that the `alldiff` constraint is not satisfiable.

similarly the terminal vertex is connected to each value vertex. A digraph is produced from this augmented version of B where each variable-value edge in B is directed from the variable vertex to the value vertex. Edges are from s are directed towards the variable vertices and edges from the value vertices are directed towards t. The maximum flow through this augmented graph can be determined using existing algorithms, for instance the Ford-Fulkerson method (Ford and Fulkerson, 2009). Figure 3.5 shows a graph which ensures the maximum flow through the augmented graph.

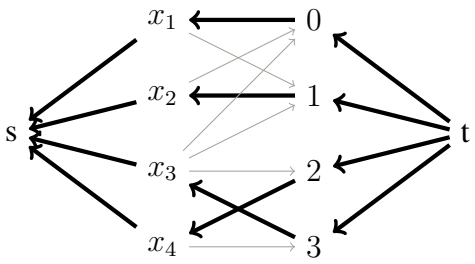

Figure 3.5: A flow computed through the augmented variable value graph obtained from the problem modelled in Figure 3.1.

The matching found in the flow graph is partitioned into strongly connected components. A strongly connected component of a graph is a subgraph in which each vertex is connected to each other vertex via a cycle. Tarjan's algorithm (Tarjan, 1972) can be used to find all strongly connected components in linear time. Any variable-value edge in the flow graph which crosses two strongly connected components and is not part of the matching corresponds to an value which can be removed from the domain of a variable. Figure 3.6 highlights the strongly connected components of the graph in Figure 3.5. The edges $x_3 \rightarrow 0$ and $x_3 \rightarrow 1$ cross both of these strongly connected components, which proves that the values 0 and 1 can be removed from the domain of x_3.

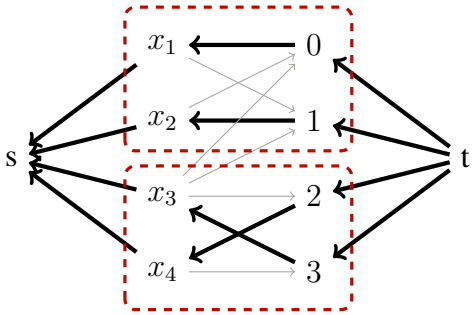

Figure 3.6: The flow graph from Figure 3.5 with the strongly connected components $x_1 \rightarrow 1 \rightarrow x_2 \rightarrow 0 \rightarrow x_1$ and $x_3 \rightarrow 2 \rightarrow x_4 \rightarrow 3 \rightarrow x_3$ highlighted in the rust coloured boxes. The edges $x_3 \rightarrow 0$ and $x_3 \rightarrow 1$ cross both these strongly connected component and so 0 and 1 can be removed from the domain of x_3.

In summary, Régin's algorithm follows the following steps:

1. Construct a bipartite variable-value graph;
2. Find a maximal cardinality matching in the augmented variable-value graph, return false if the size matching is smaller then the number of variables;
3. Find all strongly connected components in the augmented variable-value graph;
4. Remove values from variable domains which correspond to edges which; cut across strongly connected components, and are not included in the matching.

3.3 Weakening all different

In this section we give an algorithm for weakening the `alldiff` constraint. Section 3.2 gave an overview of how the `alldiff` constraint constraint is typically propagated in a constraint solver. As before we begin by providing a motivating example for why an algorithm which weakens the constraint is necessary before considering the algorithm itself.

$$\forall_{0 \leq i \leq 7}.\ x_i \in \{1, \ldots, 9\}$$

$$\texttt{alldiff}(x_i)$$

$$\text{maximize} \sum_{i=0}^{3} x_{2i} - x_{2i+1}$$

Figure 3.7: An example problem involving an `alldiff` constraint.

Consider the problem shown in Figure 3.7 which imposes a single `alldiff` constraint across eight integer variables taking values from 1 to 9. The cost function, f, we want to maximise in this example takes the sum of variables with even index minus the variables with odd index.

Figure 3.8 shows two partial assignments (with costs -1 and 5) and the resulting merged partial assignment. In this case rather than altering the constraint, we mask variables which have values that will impact the domains of the unassigned variables in the new partial assignment. In Figure 3.8c we can see that assigning x_3 to 7 would result in the removal of 7 from the domains of the unassigned variables were the `alldiff` constraint to be propagated to GAC. For the `alldiff` constraint we mask variables with a value which appears in the domain of any variable, or has already been used to mask another variable. To achieve this we mask variables using wildcard values, starting with a wildcard value one larger than the largest value involved in the `alldiff` constraint. In this way we ensure that the new completion problem returns a value at least as large as any completion of the input partial assignments.

$x_0 = 1$	$x_0 = 1$	$x_0 = 1$
$x_1 = 5$	$x_1 = 3$	$x_1 = 3$
$x_2 = 2$	$x_2 = 5$	$x_2 = 5$
$x_3 = 3$	$x_3 = 7$	$x_3 = 7$ masked with 10
$x_4 = 4$	$x_4 = 9$	$x_4 = 9$ masked with 11
$\forall_{5 \leq i \leq 7}.\ x_i \in \{6, 7, 8, 9\}$	$\forall_{5 \leq i \leq 8}.\ x_i \in \{2, 4, 6, 8\}$	$\forall_{5 \leq i \leq 8}.\ a_i \in \{2, 4, 6, 7, 8, 9\}$
(a) A partial solution α	**(b)** A partial solution β	**(c)** A partial solution γ

Figure 3.8: Two partial solutions and the partial solution resulting from merging them together, with variables masked to weaken the `alldiff` constraint.

In this case the optimal completion of α (with $x_5 = 6$, $x_6 = 9$ and $x_7 = 7$) has cost -5; the optimal completion of β (with $x_5 = 2$, $x_6 = 8$ and $x_7 = 4$) has cost 7; and the optimal completion of γ (with $x_5 = 2$, $x_6 = 9$ and $x_7 = 4$) has cost 8. This approach is superior to removing the constraint, as we still require that all unassigned variables are different from each other. If the constraint were to be removed from the completion problem of γ then its optimal solution would have cost 10 (with x_5 and x_7 both assigned the value 2).

Our algorithm for masking variables to weaken the `alldiff` constraint is given in Algorithm 3.1. In this case we assume that the constraint is represented by a data structure which includes each variable which is required to take distinct values, a boolean flag to denote if each variable is masked and an integer for each variable which records the value of that variable's masked value. For each assigned variable which is not already masked we check if its value appears in the domain of any unassigned variable. If it is then we mask the variable with the next available wildcard value (on line 10) and increment the wildcard value. Once variables are masked using Algorithm 3.1 we expect that algorithms that propagate `alldiff` treat masked variables as if their value is the wildcard value with which the variable is masked.

Proposition 3.1. Weakening the `alldiff` constraint by masking variables using Algorithm 3.1 allows for the use of the `alldiff` constraint with DDBB.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes `allDiff` constraints, and assume that f satisfies the limitations set out in Section 2.4.4. Again, the proof of Proposition 3.2 is similar to the proof of Proposition 2.2, until we have to show that all values introduced to m from u and v are ensured support, after nodes u and v are merged to create a new node m.

Consider a value k introduced the domain of a variable $x \in \mathcal{X}$ at node m from node u, and let k be without support at m. Since all values at u and v had support before merging, and

Algorithm 3.1: An algorithm for masking variables when weakening the `alldiff` constraint.

```
1  maskAllDifferent (Node m)
2  begin
3      D ← the set of variable domains recorded at m which are involved in the constraint
4      w ← 1 + largest value across all domains in D
5      foreach domain of variable x, D_x ∈ D do
6          foreach domain of variable x', D_x' ∈ D \ D_x do
7              if x is assigned a value and x is not masked then
8                  a ← the value assigned to x
9                  if a ∈ D_x' then
10                     mask x with the value w
11                     w ← w + 1
```

all values inherited in m from v have support by construction, the value k must already be assigned to some other variable at m. If this variable is masked with a value w which is not yet included in the domain of any variable at m, k is given support. When the bipartite graph B is constructed a new edge from the variable to w is introduced, allowing the propagation of the relaxed subproblem at m.

□

3.4 Weakening the Alldifferent except 0 constraint

In this Section we discuss the "AllDifferent except zero" (which we will refer to as `alldiff_0`) constraint and provide an algorithm to weaken it so that it can be used in our constraint solver.

The `alldiff_0` constraint is similar to `alldiff`, but now any number of variables under the constraint are allowed to take the value 0 while all others are required to take distinct values. It is unsurprising that our method for weakening the `alldiff_0` constraint is similar to the method for weakening `alldiff`, but for `alldiff_0` all variables which need to have their value masked are now masked with the value 0. Algorithm 3.2 shows our approach to masking `alldiff_0` with the main differences being only masking variables if their assignment is nonzero (line 6), and masking variables with the single allowed wildcard value (line 9).

Proposition 3.2. Weakening the `alldiff_0` constraint by masking variables using Algorithm 3.2 allows for the use of the `alldiff` constraint with DDBB.

Algorithm 3.2: An algorithm for masking variables when weakening the `alldiff_0` constraint.

```
1 maskAllDifferentExcept0 (Node m)
2 begin
3     D ← the set of variable domains recorded at m which are involved in the constraint
4     foreach domain of variable x, D_x ∈ D do
5         foreach domain of variable x', D_x' ∈ D \ D_x do
6             if x is assigned a value and x is neither masked nor assigned the value 0 then
7                 a ← the value assigned to x
8                 if a ∈ D_x' then
9                     mask x with the value 0
```

Proof. The proof of Proposition 3.2 is similar to the proof of Proposition 3.1, but rather than causing a new edge to be added to the bipartite variable-value graph when we mask variables with a unique wildcard value, masking variables using the value 0 allows them to be ignored by the propagator. □

3.4.1 Weakening all different with a single wildcard mask

We could use this approach of using a single wildcard to weaken the `alldiff` constraint. We could weaken the standard `alldiff` constraint by masking variables which have value assignments which appear in the domains of some unassigned variables with a single wildcard value, and allow the variables to take all different values except the wildcard value. This single wildcard value would have to be chosen so that it is not among the initial values in the domain of any variable involved in the `alldiff` constraint. Masking variables with this one wildcard value instead of one value per masked variable does not make our masking algorithm any more efficient, but it would allow for more efficient propagation of the weakened constraint. This is because the single wildcard approach allows us to disregard masked variables, rather than simply adding enough edges to the initial variable-value bipartite graph to support all of the unassigned values in the weakened constraint.

3.5 Weakening the at most n values constraint

In this section we provide an algorithm for weakening the "at most n values" constraint (which we will call `AtMost`).

The `AtMost`(X, n) constraint requires that an array of variables X take at most n distinct

values. We will again demonstrate that there is a need for our algorithm for weakening the constraint by an example. Consider the problem modelled in Figure 3.9 which imposes that a set of 10 integer variables taking values ranging from 1 to 7 should contain at most 3 distinct values. In Figure 3.10 we have two partial solutions which are merged to obtain a partial solution γ. In the case of `AtMost` we need to consider the assignments which have been made in each of the partial solutions α and β contained at nodes u and v.

$$\forall_{0 \leq i \leq 9}.\ x_i \in \{1, \ldots, 7\}$$

$$\texttt{AtMost}(X, 3)$$

$$\text{maximize} \sum_{i=0}^{9} x_i$$

Figure 3.9: An example problem involving an `AtMost` constraint.

Let $c_X^\alpha(v)$ be the number of values in the set of variables X which are assigned the value v in the partial solution α. Let d_X^α be the number of distinct values assigned to variables in X in the partial solution α, and let $s_X^{\alpha,\beta}$ denote the number of values which appear in both α and β. Using Iversen bracket notation[1] we can write this as:

$$c_X^\alpha(v) = \sum_{i=0}^{n-1} [x_i = v] \tag{3.1}$$

$$d_X^\alpha = \sum_{v} [c_X^\alpha(v) \neq 0] \tag{3.2}$$

$$s_X^{\alpha,\beta} = \sum_{v} [c_X^\alpha(v) \neq 0 \wedge c_X^\beta(v) \neq 0] \tag{3.3}$$

In order to ensure that the completion problem of γ admits a suitably large solution (that is, a solution at least as large as α and β), we change the number of distinct values allowed by the constraint,n, to be the value

$$o = d_X^\alpha + d_X^\beta - s_X^{\alpha,\beta} + \texttt{max}(n - d_X^\alpha, n - d_X^\beta) \tag{3.4}$$

The first three terms of this expression ensure that all of the values which appear in assignments across both partial solutions have support in the domains of the unassigned variables in γ. The final term ensures that as many further values can be added to X as might have been in either α or β. In the case presented in Figure 3.10 we weaken `AtMost` by changing the number of distinct values allowed by the constraint from 3 to 5 by assigning n the value 5.

[1] [P] = 1 if P is true, [P] = 0 if P is false

$x_0 = 1$	$x_0 = 5$	$x_0 = 5$
$x_1 = 2$	$x_1 = 5$	$x_1 = 5$
$x_2 = 1$	$x_2 = 6$	$x_2 = 6$
$x_3 = 1$	$x_3 = 5$	$x_3 = 5$
$x_4 = 2$	$x_4 = 6$	$x_4 = 6$
$\forall_{5 \leq i \leq 7}.\ x_i \in \{1, \ldots, 7\}$	$\forall_{5 \leq i \leq 8}.\ x_i \in \{1, \ldots, 7\}$	$\forall_{5 \leq i \leq 9}.\ x_i \in \{1, \ldots, 7\}$
$\texttt{AtMost}(X, 3)$	$\texttt{AtMost}(X, 3)$	$\texttt{AtMost}(X, 5)$
(a) A partial solution α	**(b)** A partial solution β	**(c)** A completion problem γ

Figure 3.10: Two partial solutions to the problem modelled in Figure 3.9 and the partial solution resulting from their merger, with n masked to weaken the `AtMost` constraint shown in Figure 3.9.

This accounts for there being 4 distinct value assignments across both partial solutions and each partial solution having support for 1 further value. This process is given in Algorithm 3.3

Algorithm 3.3: An algorithm to weaken the `AtMost` constraint by increasing the number of distinct values allowed by the constraint.

```
1 maskAtMost (Node u, Node v, Node m)
2 begin
3     p ← calculate o using Equation (3.4) over the partial solutions at u, v and m
4     n ← the number of distinct values allowed by the AtMost constraint at m
5     if p > n then
6         set the maximum number of distinct values allowed at m to p
```

Proposition 3.3. Weakening the `atMost` constraint by masking variables using Algorithm 3.3 allows for the use of the `atMost` constraint in problems solved using Algorithm 2.4.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes `atMost` constraints, and assume that f satisfies the limitations set out in Section 2.4.4. Again, the proof of Proposition 3.3 is similar to the proof of Proposition 2.2, until we have to show that all values introduced to m from u and v are ensured support, after nodes u and v are merged to create a new node m.

Consider that after merging there are some unassigned variables $\mathcal{X}' \subset \mathcal{X}$ at node m. To ensure that the completion problem recorded at m is at least as large as the completion problem at either u and v, we have to ensure that all variable assignments to $\mathcal{X}'$ at u and v are allowed at m. Assume that some completion of u is not allowed at m after merging. Let k be the number of unique values assigned to variables across u and v, and let l be the largest number of values

yet to be used in either u and v. Setting the maximum number of unique values allowed at m to $k + l$ allows any completion of u to be included as completions at m.

□

3.6 Weakening the at least n values constraint

In this section we present an approach to weaken the at least n values constraint (which we will refer to as the `AtLeast` constraint) after merging nodes during search.

The $\texttt{AtLeast}(X, n)$ constraint requires that each variable in an array of variables X takes values at least as large as an integer n. The process of weakening the `AtLeast` constraint is similar to weakening `AtMost`, in so far as we concentrate on assigned variables rather than the domains of unassigned variables. In this case though our approach is to weaken the constraint by removing it entirely from the problem. The problem shown in Figure 3.11 is similar to that shown in Figure 3.9, with the difference being that we now require a smaller set of variables X to take at least 3 distinct values in any solution. Figure 3.12 shows a partial solution α being merged with a partial solution β. The set of possible completions of α includes the assignment $x_4 = 5$ and $x_5 = 5$, but no completion of γ could include both these assignments (before we weaken the constraint). Our approach is then to weaken the constraint by setting n to the number of distinct values in γ. This allows all the unassigned variables to take any value in their domain.

$$\forall_{0 \leq i \leq 5}.\ x_i \in \{1, \ldots, 7\}$$

$$\texttt{AtLeast}(X, 3)$$

$$\text{maximize} \sum_{i=0}^{5} x_i$$

Figure 3.11: An example problem including an `AtLeast` constraint.

Algorithm 3.4: An algorithm to weaken the `atLeast` constraint

```
1 maskAtLeast (Node m)
2 begin
3     p ← the number of distinct values assigned to variables involved in AtLeast at m
4     n ← the number of distinct values required by the AtLeast constraint at m
5     if p < n then
6         set the minimum number of distinct values allowed at m to p
```

		$x_0 = 6$
$x_0 = 1$	$x_0 = 6$	$x_1 = 6$
$x_1 = 2$	$x_1 = 6$	$x_2 = 5$
$x_2 = 3$	$x_2 = 5$	$x_3 = 6$
$x_3 = 4$	$x_3 = 6$	$x_{4,5} \in 1, \ldots, 7$
$x_{4,5} \in \{1, \ldots, 7\}$	$x_{4,5} \in \{1, \ldots, 7\}$	$\texttt{AtLeast}(X, 2)$
(a) A partial solution α	**(b)** A partial solution β	**(c)** A completion problem γ

Figure 3.12: Two partial solutions to the problem modelled in Figure 3.11 and the partial solution resulting from their merger, with n set to 2 to weaken the AtLeast constraint shown in Figure 3.9.

Proposition 3.4. Weakening the `atLeast` constraint by masking variables using Algorithm 3.4 allows for the use of the `atLeast` constraint in problems solved using Algorithm 2.4.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes `atLeast` constraints, and assume that f satisfies the limitations set out in Section 2.4.4. Again, the proof of Proposition 3.4 is similar to the proof of Proposition 2.2, until we have to show that all values introduced to m from u and v are ensured support, after nodes u and v are merged to create a new node m.

Consider that after merging there are some unassigned variables $\mathcal{X}' \subset \mathcal{X}$ at node m. To ensure that the completion problem recorded at m is at least as large as the completion problem at either u and v, we have to ensure that all variable assignments to $\mathcal{X}'$ at u and v are allowed at m. Assume that some completion of u is not allowed at m after merging. Setting the maximum number of unique values allowed at m to the current number of unique values assigned to variables at m allows any completion of u to be included as completions at m.

□

3.7 (Not) weakening the allEqual constraint

In this section we describe the `allEqual` constraint and show that it does not require a weakening algorithm if partial solutions are merged using Algorithm 2.3. In the case of the all equal constraint, we do not need to do any work when merging nodes to weaken the constraint. Here we outline why this is the case.

The `allEqual` constraint requires that a set of variables X all take the same value. It can be propagated removing all values in the symmetric difference of the domains of each variable in X from each variable's domain.

Proposition 3.5. The `allEqual` constraint may be used to model problems which are solved using with Algorithm 2.4 without the need for the constraint to be weakened after the creation of relaxed nodes using Algorithm 2.3.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes `allEqual` constraints, and assume that f satisfies the limitations set out in Section 2.4.4. Again, the proof of Proposition 3.5 is similar to the proof of Proposition 2.2, until we have to show that all values introduced to m from u and v are ensured support, after nodes u and v are merged to create a new node m.

When u and v are merged together to create m, all values introduced from v have support at m by construction. Consider a value k in the domain of x introduced at node m from node u. If the `allEqual` constraint is propagated as described above then another value l in the domain of y is introduced at node m from node u which supports k at m. This value l is in turn also supported by the inclusion of k. □

3.8 Experimental results

In this section we present empirical results from a solver which implements the search scheme given in Algorithm 2.4 and the weakening algorithm for the `AllDiff` constraint. We test our solver using the problem of finding optimal Golomb rulers as well as an allocation problem.

3.8.1 Optimal Golomb rulers

Recall that a Golomb ruler is an ordered sequence of integers a_i which represent *marks* on an imaginary ruler. The distance between any pair of marks on the ruler must be distinct from the distance between any other pair. The number of marks n on the ruler is its *order* and the *length* of the ruler is the largest distance between any pair of marks. A Golomb ruler is *optimal* if no Golomb ruler of smaller length with the same order exists.

Golomb rulers are named after the mathematician Solomon Golomb. They are not only interesting mathematical objects, but have various real world applications. They can be used in radio communications to place channels throughout a radio spectrum to reduce distortion between channels (Babcock, 1953); in x-ray crystallography to resolve instances when two

different lattices produce the same diffraction pattern (Bloom and Golomb, 1977); in fibre optic communications (Gagliardi, Robbins, and Taylor, 1987); and in radio astronomy to position arrays of sensors (Biraud, Blum, and Ribes, 1974).

The search for optimal Golomb rulers of particular order remains an active area of research. Dollas, Rankin, and McCracken (1998) find optimal rulers up to order 19 using an exact search algorithm. While optimal Golomb rulers are known to order 27, the later orders are found by a massively distributed search organised by distributed.net, which is a project which allows the public to donate their computer's idle time towards large searches (Hayes, 1998). Currently, as of October 2019, distributed.net are searching for an optimal Golomb ruler of order 28. Galinier (2001) consider the use of constraint programming to search for optimal Golomb rulers. Kocuk and van Hoeve (2019) compare linear integer programming, constraint programming and quadratic integer programming approaches for the Golomb ruler problem and find that while quadratic integer programming performs best for proving optimality at low orders, constraint programming is favourable when the order of the ruler grows. This is due to the ability to more easily parallelise the constraint programming approach.

A model for finding optimal Golomb rulers is given as a Minizinc code listing in Appendix A. This model relies only on the `lin-le` and `lin-eq` linear summation constraints as well as an `alldiff` constraint to ensure that no distance is measured more than once.

In Tables 3.1 and 3.2 we evaluate the effect of changing the maximum permitted width of restricted and relaxed search trees for the problem of finding optimal Golomb rules. As expected (and explained in Section 2.11.2 using either a width width is too small or too large results in a much larger search space being explored than is necessary. However, for this problem however we do not see an instance by instance difference in the best width, with each instance visiting the fewest number of nodes with the same maximum permitted widths. We also see that as the width increases there is not a monotonic increase in the number of nodes used, while for some widths the number of search nodes used is the same. This first point is likely due to different amounts of branching occurring for differing widths, and the second being due to some choices of maximum permitted width will resulting in the same layers in search being bounded. The large variability in execution time for smaller widths is due to noise. For

We also compare our approach to using DDBB as the search scheme in a constraint optimisation solver against a forward checking approach, the results of which are shown in Table 3.3, where the execution times are reported with a timeout of ten minutes. For the problem of finding optimal Golomb rulers we find that our approach is beaten by forward checking. This is true for all of the choices of variable and value ordering heuristics considered. By reviewing the order in which optimal solutions are found by search it is clear that the DDBB approach

Width	Order 3		Order 4		Order 5		Order 6		Order 7	
	Nodes	Runtime	Nodes	Runtime	Nodes	Runtime	Nodes	Runtime	Nodes	Runtime
0.1	20	0.002985	286	0.046508	3276	0.924470	42964	29.775549	344777	418.706665
0.5	21	0.002095	154	0.022211	686	0.260199	2426	3.710149	10762	36.540497
1	22	0.002071	160	0.021127	696	0.302193	2445	3.629534	10590	36.740383
2	21	0.000763	229	0.026542	716	0.298703	2345	3.684064	7923	31.884909
3	21	0.000686	232	0.025871	1665	0.461398	8052	8.763066	9534	32.315067
4	21	0.000681	235	0.025604	1676	0.549792	8075	8.492616	28886	81.824463
5	21	0.000747	238	0.026288	1672	0.540431	8098	8.616967	28934	82.305412
6	21	0.000677	241	0.030577	1680	0.549969	8857	8.550164	28982	83.530128
7	21	0.000691	244	0.029855	1688	0.558217	8876	8.823674	36378	88.529007
8	21	0.000732	247	0.024942	1696	0.557772	8895	8.916242	36420	88.299889
9	21	0.000739	250	0.027646	1704	0.564427	8914	8.704473	36461	86.931641
10	21	0.000767	253	0.024884	1712	0.530516	8933	9.269555	36501	86.552330
15	21	0.001072	227	0.022841	1752	0.569316	9028	8.988946	36491	89.047020
20	21	0.000683	227	0.020382	1792	0.545889	9123	9.373609	36661	87.417046
25	21	0.000759	227	0.016766	1832	0.551636	9218	8.943694	36831	87.425377
30	21	0.001074	227	0.020117	3120	0.826003	9300	9.714022	37001	92.780319
35	21	0.001044	227	0.026461	3140	0.732748	9395	9.292076	37171	92.798294
40	21	0.000667	227	0.024857	3160	0.790835	9490	9.933311	37341	93.651543
45	21	0.001163	227	0.025243	3180	0.964897	9585	9.648463	37511	89.069046
50	21	0.001062	227	0.016395	3200	0.763184	28953	18.966980	37681	79.698898
100	21	0.000731	227	0.016405	3400	0.816030	29453	19.083916	178798	251.891632

Table 3.1: Comparison of execution times of our solver when finding optimal Golomb rulers for differing widths of restricted search.

Width	Order 3		Order 4		Order 5		Order 6		Order 7	
	Nodes	Runtime	Nodes	Runtime	Nodes	Runtime	Nodes	Runtime	Nodes	Runtime
0.1	22	0.001940	160	0.018493	696	0.246356	2445	2.947325	10590	28.250847
0.5	22	0.002330	160	0.026524	901	0.339994	3099	3.895968	13482	45.018528
1	22	0.002019	201	0.027217	1238	0.384766	4837	6.068108	17919	64.961304
2	22	0.001942	224	0.023845	1676	0.446129	7037	8.269117	24861	86.597992
3	22	0.001859	235	0.032501	1942	0.531748	8519	9.672113	29470	105.233551
4	22	0.002066	236	0.019952	2071	0.532100	9623	9.824576	33002	109.017548
5	22	0.002171	236	0.028671	2169	0.549625	10591	10.186145	36074	116.133514
6	22	0.002302	236	0.027799	2232	0.593757	11396	11.264503	39438	122.611206
7	22	0.002244	236	0.028740	2300	0.600656	12097	11.101996	42394	131.901337
8	22	0.002346	236	0.020678	2365	0.607488	12825	11.941392	44976	136.044769
9	22	0.001754	236	0.024749	2425	0.653021	13491	12.202990	47416	139.618851
10	22	0.002385	236	0.032573	2479	0.648015	14199	12.580430	50176	140.983368
15	22	0.002314	236	0.026828	2697	0.611372	17214	14.343936	63403	160.305588
20	22	0.002038	236	0.031104	2870	0.685898	19750	14.855579	75516	178.294510
25	22	0.002184	236	0.028591	2983	0.658672	21852	16.781200	86418	192.665878
30	22	0.002039	236	0.020304	3055	0.540537	23709	13.336346	96321	185.245026
35	22	0.001620	236	0.020089	3085	0.547713	25199	14.234142	105557	199.135483
40	22	0.002112	236	0.020197	3085	0.555695	26455	14.414518	114282	209.404922
45	22	0.001947	236	0.020478	3085	0.540736	27417	14.968359	122489	219.352493
50	22	0.001750	236	0.020523	3085	0.546124	28213	15.140100	130126	228.230103
100	22	0.001983	236	0.020429	3085	0.524751	32690	16.063148	182399	288.428619

Table 3.2: Comparison of execution times of our solver when finding optimal Golomb rulers for differing widths of relaxed search.

		Order					
		3	4	5	6	7	8
Input order/min value	Runtime DDBB	0.002093	0.021789	0.259210	2.982404	29.984241	280.420898
	Space DDBB	22	160	696	2445	10590	48610
	Runtime FC	0.000306	0.002789	0.029246	0.375646	5.807307	89.925438
	Space FC	9	5	18	94	857	5568
Input order/max value	Runtime DDBB	0.001326	0.016747	0.192924	2.165610	40.541050	–
	Space DDBB	19	69	224	1118	9380	67536
	Runtime FC	0.000421	0.003900	0.043045	0.649819	10.288877	212.142807
	Space FC	11	19	50	292	1931	17836
First fail/min value	Runtime DDBB	0.001838	0.018601	0.244221	2.955035	28.915529	276.317169
	Space DDBB	22	69	698	2432	10282	46742
	Runtime FC	0.000341	0.002801	0.029858	0.362931	5.870193	89.835159
	Space FC	9	5	18	94	857	5568
First fail/max value	Runtime DDBB	0.001311	0.020695	0.202024	2.259621	40.443085	–
	Space DDBB	19	69	224	1116	9140	66918
	Runtime FC	0.000402	0.003773	0.042179	0.701993	11.129051	224.506592
	Space FC	11	19	50	312	2022	19114

Table 3.3: Comparison of execution times and size of the search space for our DDBB approach compared with forward checking when solving instances of the optimal Golomb rulers problem. All runtimes are in seconds.

explores the search tree in a suboptimal manner when compared with forward checking. This allows forward checking find good solutions more quickly and in turn more effectively prune the search space based on the size of the incumbent best solution found by search. In Chapter 5 we will return to the problem of finding optimal Golomb rulers to explore the effect of varying additional solver parameters.

3.8.2 Cell block assignment

The goal of the cell block assignment problem is ensure that a number of prisoners are interned in a grid of cells such that female and male prisoners are interned only in their allocated half of the grid and that no cells adjacent to "dangerous" prisoners occupied. Interring someone in a cell has an associated cost and the overall cost of interring all prisoners is to be minimised. This problem appears in the Basic Modelling for Discrete Optimisation course run by The University of Melbourne on the Coursera platform (Stuckey, 2016). The cell block assignment problem makes use of `lin_le`, `lin_ge` and `allDiff` constraints and a model for the problem is included in Appendex A.

When varying the width of restricted and relaxed width truncated search trees, we expect that widths that are too small or too large will lead to poor performance when compared to widths inbetween. Figure 3.13 and Figure 3.14 confirm that for the cell block assignment problem we see this expected behaviour.

We again test our approach of using DDBB as the search scheme in a constraint optimisation solver against the use of forward checking. Figure 3.15, Figure 3.16, Figure 3.17 and

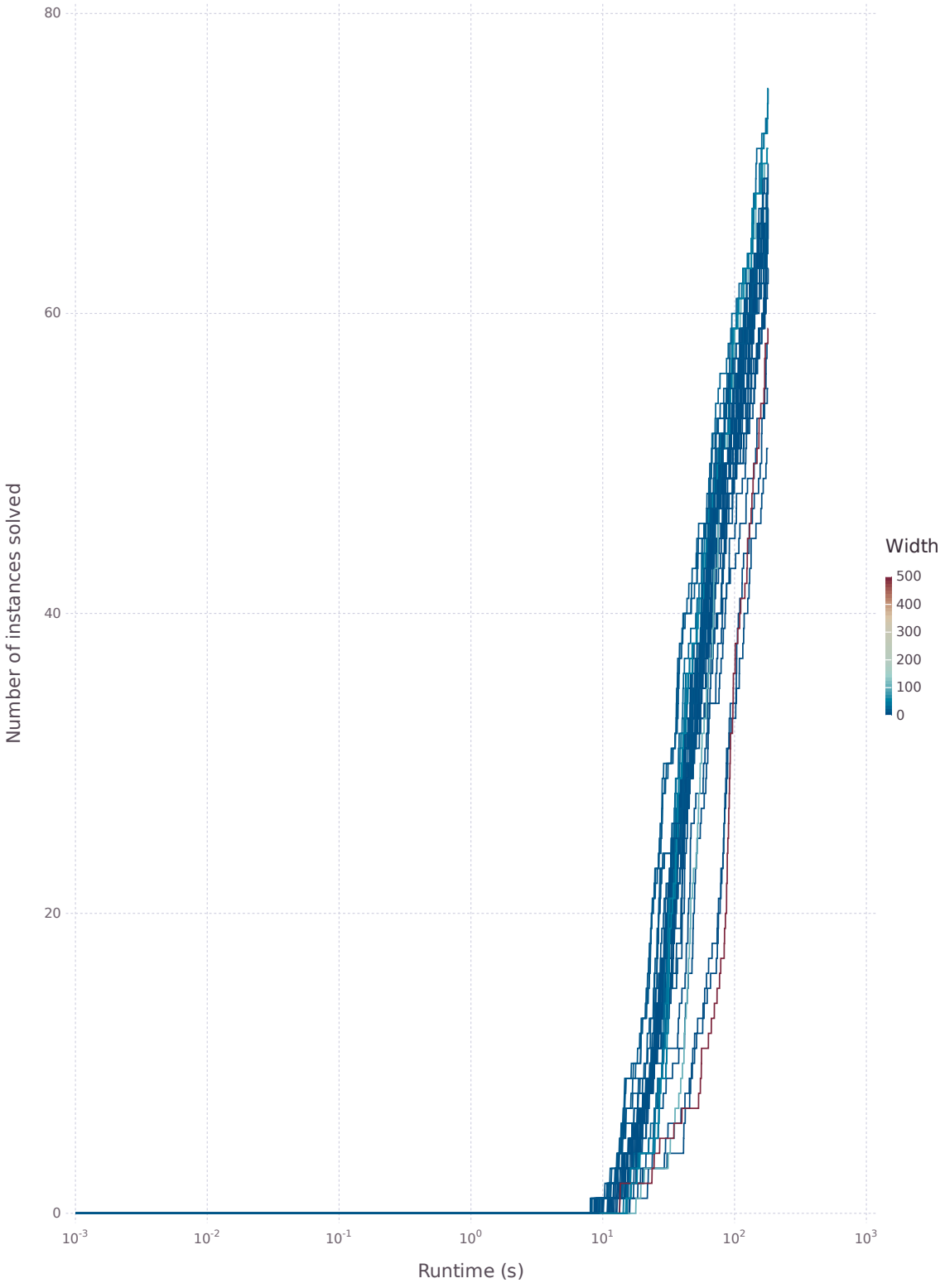

Figure 3.13: Cumulative number of cell block assignment instances solved in a given time as the width of restricted search is varied.

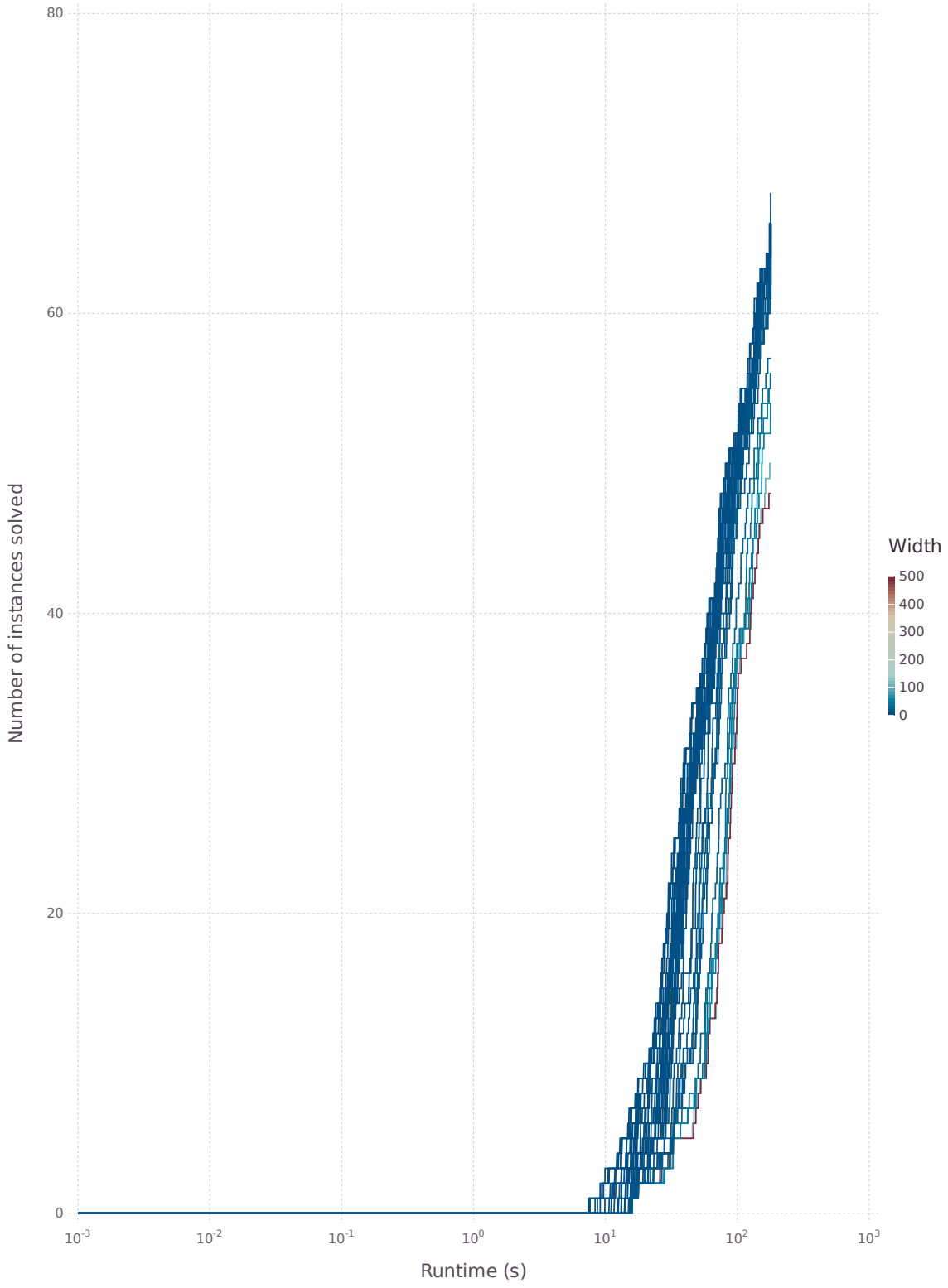

Figure 3.14: Cumulative number of cell block assignment instances solved in a given time as the width of relaxed search is varied.

Figure 3.18 compare the execution times, size of the search space and number of nodes visited per second for various combination of variable and value ordering heuristics. Although in many instances, across all combinations of heuristic choices, we see a number of instances for which DDBB outperforms forward checking, the effect is not as pronounced as it was for the car factory scheduling problem as shown in Chapter 2. On some instances we do see that DDBB explores a smaller search space than forward checking, however for many instances this is not the case. The charts showing the relative speed of each approach give an insight into why exploring a larger search space does not hamper DDBB search as it often visits more nodes per second than forward checking for the cell block allocation problem. We also check if using relaxed search makes a positive impact on execution times or not. In Figure 3.19 once again find that there is a mixed outlook, with some instances benefiting from the use of relaxed search trees while many more do not.

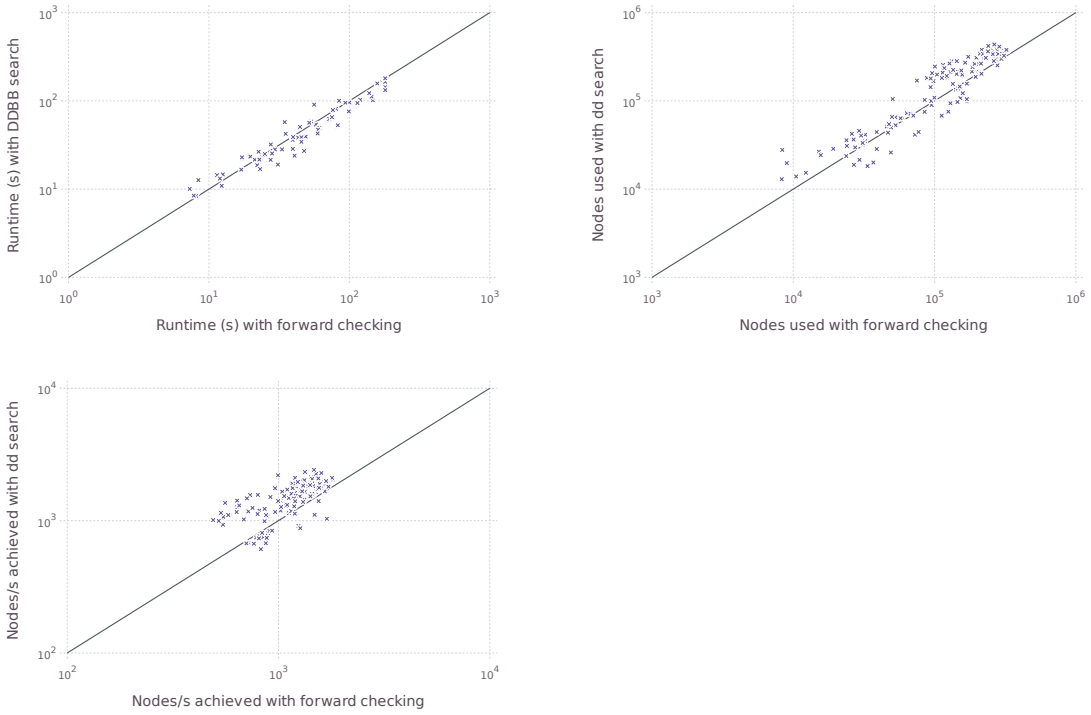

Figure 3.15: DDBB search versus forward checking on the cell block allocation problem, where variables are chosen in input order and the minimum value is chosen first from variable domains when branching.

Why DDBB performs well, and why

To investigate why DDBB has faster execution times than forward checking for some instances of the cell block allocation problem, we again focus on how the cost of the incumbent best solution changes over time. Figure 3.20 shows the cost of the best known solution yet found during search against time, for both DDBB and forward checking for a single instance of

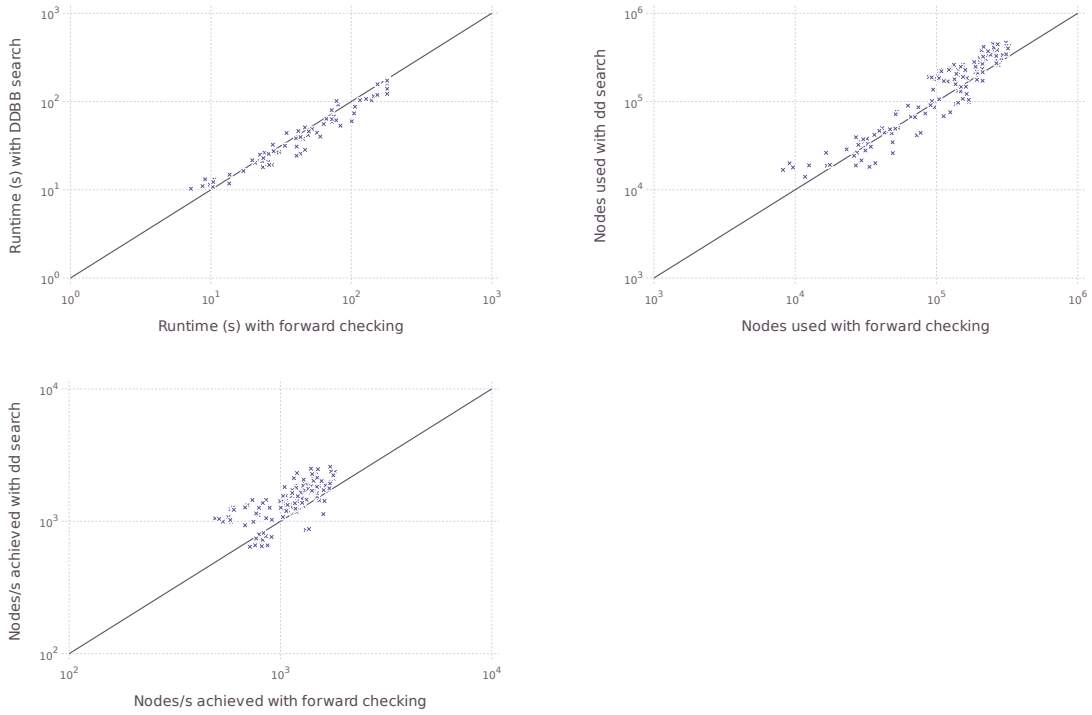

Figure 3.16: DDBB search versus forward checking on the cell block allocation problem, where variables are chosen in input order and the maximum value is chosen first from variable domains when branching.

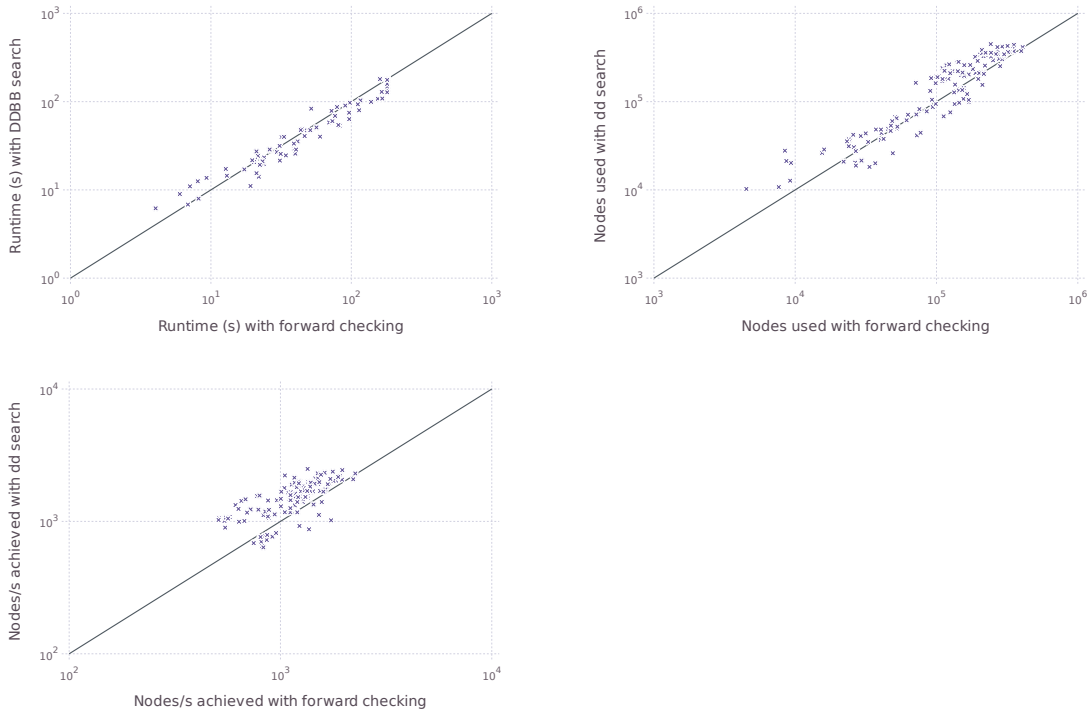

Figure 3.17: DDBB search versus forward checking on the cell block allocation problem, where variables are chosen in order of smallest domain size and the minimum value is chosen first from variable domains when branching.

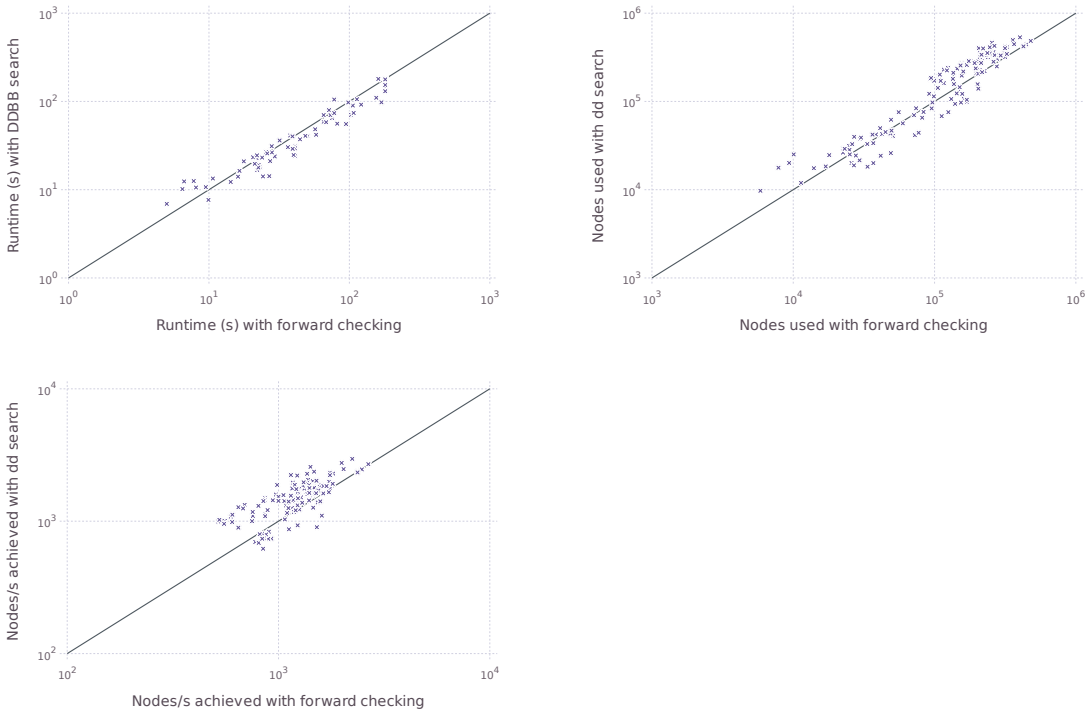

Figure 3.18: DDBB search versus forward checking on the cell block allocation problem, where variables are chosen in order of smallest domain size and the maximum value is chosen first from variable domains when branching.

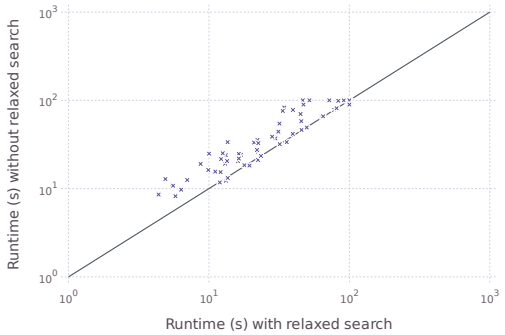

Figure 3.19: Using relaxed width truncated search trees versus not on the cell block allocation problem, where variables are chosen in order of smallest domain size and the minimum value is chosen first from variable domains when branching.

CBA where DDBB completes its execution before forward checking. We can again notice that DDBB and forward checking do not find the same solutions, and the improvements DDBB makes are better than those made by forward checking. Again this is due to forward checking finding a good solution, then backtracking to find improving solutions that are more closely related to this initial good solution. DDBB instead continues search in a new area of the search space. We again note that finding such candidate solutions involves some luck, that DDBB can "skip" intermediate incumbent solutions found by forward checking is not inherently guaranteed by the approach. For the problem studied in Figure 3.20 we do not see any advantage in when proving that the solution is optimal, therefore we can determine that relaxed decision diagrams do not always help then pruning the search space while proving that a solution is optimal.

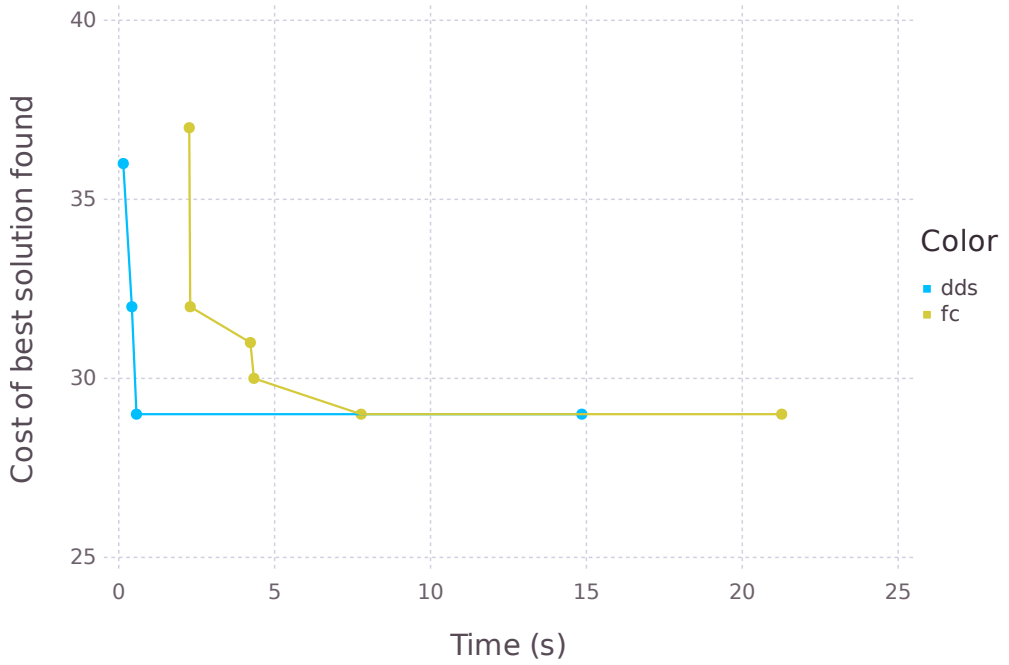

Figure 3.20: Comparing the cost of the best solution found by search against time for both DDBB and FC for an instance of the cell block allocation problem.

Chapter 4

Weakening symmetry reduction constraints

4.1 Introduction

In this chapter we focus on providing relaxations to a number of symmetry reduction constraints. Lexicographic and ordering constraints are commonly used as symmetry breaking constraints (Gent, Petrie and Puget, 2006). In problems where multiple solutions are symmetrically equivalent (that is they are equivalent up to renaming variables or values), these constraints help to reduce the size of the search space and often lead to vastly improved execution times. One particular area in which symmetry reduction is particularly useful is in graph search problems. Some of the symmetry breaking constraints that we introduce in this chapter we will use in Chapter 6 when solving extremal graph problems.

There are many lexicographic and ordering constraints but we choose to focus on `lexle` and `valueProceedsChain`.

4.2 Background

When we search to find a solution to a problem, it can be the case that due to symmetries inherent in the problem that multiple solutions are equivalent to one another. If multiple subspaces of the search space lead to equivalent solutions then not taking symmetries into account leads to needless repetition of work during search, which ultimately slows down the execution times of constraint solvers.

In the Handbook of Constraint Programming (Gent, Petrie, and Puget, 2006) (which provides

a survey of the field) it is claimed that the problem of reducing symmetries in Constraint Programming is solved in theory, meaning that the methods by which symmetries can be reduced are now well known. Solutions to reducing symmetries in combinatorial problems involve either:

Remodelling problems to eliminate symmetries.

Adding symmetry breaking constraints to models of problems to eliminate symmetries.

Symmetry breaking during search by building up information about the symmetry group of a problem instance while a solver searches for a solution to the problem.

However research continues because of the difficulty in applying these solutions in practical algorithms. The reduction of execution times of solvers through the use of symmetry reduction is also a lure for researchers. There are two main branches of symmetry reduction techniques: symmetry breaking during search (SBDS) and symmetry breaking before search (SBBS). SBDS makes use of results from group theory to build up the symmetry group of a problem while search runs, and excludes subspaces of search which lead to equivalent solutions by exploiting knowledge of the symmetry group. SBBS takes the alternative approach of applying constraints to the model of a problem and then solving the problem using standard approaches to search.

SBBS is by far the most common technique in practice. Constraints which are added to a model with the intention to break some symmetry in the problem are known as symmetry breaking constraints. As an example consider the problem of finding optimal Golomb rulers. In this problem permutation symmetries can also be broken, requiring that each mark in the ruler is placed at a smaller value than the succeeding marks.

While this type of ad-hoc approach to symmetry reduction using symmetry breaking constraints is common, there do exist some general techniques for reducing symmetry when modelling problems using constraint programming. Following standard techniques mitigates the chance of over constraining a model when adding symmetry breaking constraints. In the context of enumeration problems this would lead to lost solutions which are not equivalent to any solutions found during search. In the context of optimisation problems incorrectly throwing out areas of the search space might lead to the optimal solution not being found. For optimisation problems this type of bug would only be apparent from checking against the output of another solver, very rarely be apparent, making detection of this type of bug quite difficult.

4.2.1 Lex-leader

One particular standard technique for adding symmetry breaking constraints to a model is the lex-leader method. To apply the lex-leader method the equivalence classes of solutions under the symmetries of the problem are identified and then a single representative from each class is defined as the canonical solution. The symmetries of the problem are then broken by applying extra constraints to the model such that only canonical solutions satisfy these constraints. In practice it can be useful to apply constraints that break most but not all of the symmetries of a problem, and then determine which canonical solutions are actually equivalent after search.

The development of this technique grew from work by Puget (1993) who proved that if a CSP has symmetry then it is possible to add constraints to a model to reduce the symmetries. Puget showed that reduced models could be solved more quickly than the models they were based on. Crawford et al. (1996) built on Puget's work with a technique for generating symmetry constraints for breaking variable symmetries. They also coined the term "lex-leader". These early works broke symmetries in SAT problems, but the results apply to constraint programming and other solution technologies.

Gent, Harvey, and Kelsey (2002) note that caution must be taken when using the lex-leader method together with search heuristics, noting that using the wrong heuristic can lead to an increase in the execution times of solvers when symmetry breaking constraints are imposed.

A particular example of the lex-leader constraint is in problems which admit models which are defined on matrices of variables (Flener et al., 2002). The rows and columns of the matrix of variables can be lexicographically ordered. Examples of these kinds of lexicographic ordering constraints of variables include extremal graph search. Codish et al. (2019) introduce special lexicographic ordering constraints for problems where the matrix of variables represents the adjacency matrix of a graph.

4.2.2 Examples of symmetry breaking

The Handbook of Constraint Programming lists several successful applications of symmetry breaking to solving combinatorial problems. Here we pick a few examples.

The maximum density still life problem is based on Conway's Game of Life (Gardner, 1970). A still life is a stable pattern which is not impacted by the iteration of the game. The problem is then to find the most dense possible stable pattern, the pattern with the most live cells, which fits in an $n \times n$ game board. Two stable patterns might be equivalent up to rotating the game board, or flipping it about the horizontal, vertical or diagonal axis, or any combination of rotations and flips producing a total of 8 symmetries. The maximum density still life problem

has seen applications of remodelling and symmetry breaking constraints (B. M. Smith, 2002, Bosch and Trick, 2004).

The social golfers problem is to find a schedule of games which allows for a number of golfers to play in groups of a given size over a number of weeks such that no golfer plays in the same group twice. The symmetries of this problem are much more complicated than in Conway's Game of Life with vastly more symmetries, as we can permute the golfers, the groups and the order the games are played in to move between solutions. One state of the art algorithm for the social golfers problem makes use of symmetry breaking constraints (Harvey and Winterer, 2005).

The peaceable coexisting armies of queens problem problem is to place two armies of queens on a chessboard, such that no queen attacks another queen in the opposing army. The goal is to find the maximum size of the armies for an $n \times n$ board. The symmetries in this problem are similar to those in Conway's game of life, as we can rotate the board or flip it about an axis to move between solutions. Smith, Petrie and Gent (2004) use symmetry breaking during search to help solve this problem.

4.3 Weakening lexicographic ordering constraints

In this section we provide algorithms for weakening lexicographic ordering constraints. In Section 4.2 we explained how constraints of this type are often used to enforce that only a canonical member of a class of symmetrically equivalent solutions is found during search.

4.3.1 Weakening lex less than or equal to

In this section we provide an algorithm which weakens the lexicographically less than or equal to constraint (which we call `lexle`).

The constraint

$$\texttt{lexle}(A, B)$$

enforces that the array A be less than or equal to array B lexicographically when considered as strings of integers. Frisch, Hnich, et al. (2002) give an algorithm which propagates `lexle` to maintain generalised arc-consistency. This constraint occurs in models over binary variables more often than integer variables. Our algorithm for weaking the constraint works in both of these cases, but for ease of understanding our example problem for this constraint will involve binary variables.

First consider a motivating example shown in Figure 4.1, which models a problem using two arrays of boolean variables A and B with A required to be lexicographically less than or equal to B.

$$A = (a_0, a_1, ..., a_7)$$

$$\forall_{0 \leq i \leq 7}.\ a_i \in \{0, 1\}$$

$$B = (b_0, b_1, ..., b_7)$$

$$\forall_{0 \leq i \leq 7}.\ b_i \in \{0, 1\}$$

$$\texttt{lexle}(A, B)$$

$$\text{maximise} \sum_{i=0}^{7} (a_i - b_i)$$

Figure 4.1: An example problem involving a `lexle` constraint.

Figure 4.2 illustrates the need for a weakened version of `lexle`, giving two partial solutions to the problem modelled in Figure 4.1 and the completion problem resulting from them being merged. Notice that we have not assumed that the variables will be assigned values in any logical order (the most sensible being from left to right). This causes the value of b_1 to be unassigned in the partial solution α while it is assigned in the partial solution β due to propagation. In the completion problem the variable b_1 again has domain $\{0, 1\}$ but in order for the `lexle` constraint to be satisfied it can only take the value 1. To overcome this we weaken the constraint by masking the variable a_1 with the smallest value in the domain of b_1. However, we are not finished. In this case, if we were to assign b_1 the value 0, while a_1 is masked with the value 0, there would be no support for the assignment $a_3 = 1$. We resolve this issue by masking b_1 with the largest value in the domain of a_1. We would continue to mask variables in this way, until $b_i > a_i$ for some pair of assigned a_i and b_i.

Algorithm 4.1 shows our strategy for masking variables in merged nodes to weaken `lexleq`. The algorithm steps through the arrays from left to right and compares each a_i and b_i pair. If both are assigned and the value given to a_i is less than that given to b_i then we know that the constraint is satisfied and propagation will have no effect on any unassigned variables to the right of index i, so the algorithm can terminate (line 10). If both variables are assigned and the value give to a_i is greater than the value given to b_i then the constraint is not satisfied and we mask a_i with the value given to b_i (line 12). In the case where a_i is assigned and b_i is not, if the value given to a_i is greater than the smallest value in the domain of b_i, s, we mask a_i with s (lines $13 - 17$). In the case where a_i is not assigned but b_i is, if the value given to b_i is smaller then the largest value in the domain of a_i, t, we mask b_i with t (lines $18 - 22$).

$$\begin{array}{lcccccccc} A: & 0 & 0 & 0 & a_3 & 0 & 0 & 0 & 0 \\ B: & 0 & b_1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$
$$a_3 \in \{0,1\} \;\; b_1 \in \{0,1\}$$

(a) A partial solution α

$$\begin{array}{lcccccccc} A: & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ B: & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \end{array}$$
$$a_3 \in \{0,1\}$$

(b) A partial solution β

$$\begin{array}{lcccccccc} A: & 1 & \underline{1} & 0 & a_3 & 1 & 0 & 1 & 0 \\ B: & 1 & b_1 & 0 & \underline{0} & 1 & 1 & 0 & 1 \end{array}$$
$$a_3 \in \{0,1\} \;\; b_1 \in \{0,1\}$$
a_1 is masked with the value 0
b_1 is masked with the value 1

(c) A relaxed completion problem for partial solution γ

Figure 4.2: Two partial solutions of the problem modelled in Figure 4.1 and the completion problem resulting from merging them together. Masked variables in the completion problem are underlined.

Algorithm 4.1: An algorithm for weakening the `lexle` by masking variables.

```
1  maskLexLe (Node m)
2  begin
3      (a_0, ..., a_{n-1}) ← the array of variables A at node m
4      (b_0, ..., b_{n-1}) ← the array of variables B at node m
5      for i ← 0 to n do
6          if a_i and b_i are assigned and not masked then
7              x ← the value assigned to a_i
8              y ← the value assigned to b_i
9              if x < y then
10                 return
11             if x > y then
12                 mask a_i at node m with the value y
13         else if a_iis assigned and not masked then
14             x ← the value assigned to a_i
15             y ← the lower bound of b_i
16             if x > y then
17                 mask a_i at node m with the value y
18         else if b_iis assigned and not masked then
19             x ← the upper bound of a_i
20             y ← the value assigned to b_i
21             if x < y then
22                 mask b_i at node m with the value x
```

Weakening the `lexle` constraint using Algorithm 4.1 allows for the valid simultaneous use of the `lexle` constraint propagator as described and DDBB search in a constraint solver.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes `lexle` constraints, and assume that f satisfies the limitations set out in Section 2.4.4. Again, the proof of Proposition 4.1 is similar to the proof of Proposition 2.2, until we have to show that all values introduced to m from u and v are ensured support, after nodes u and v are merged to create a new node m.

Consider a value k in the domain of a variable a_i which is introduced to node m from node u, and let k be without support at m. If k doesn't have support then it is greater than the upper bound of the corresponding b_i at node m. Masking b_i with the upper bound of a_i provides support for k. Similarly, consider a value l in the domain of a variable b_i which is introduced to node m from node u, and let l be without support at m. If l doesn't have support then it is less than the upper bound of the corresponding a_i at node m. Masking a_i with the lower bound of b_i provides support for l.

□

Weakening lex greater than or equal to

To weaken the lex greater than or equal to constraint, we only have to construct the lex less than or equal to constraint by swapping the order of the arrays A and B.

4.4 A relaxed value proceeds chain constraint

In this section we provide an algorithm for weakening the value proceeds chain (VPC) constraint. This constraint enforces that an array of integer variables X satisfies $x_i \leq \max(x_0, \ldots, x_{i-1}) + 1$. That is, the value assigned to a variable is not greater than one more than the maximum value assigned to any variable to the left of its position in the array.

Again we lead with an example. Consider the problem modelled in Figure 4.3 which includes a single VPC constraint over an array of 7 integer variables which take values from 0 to 6. Figure 4.4 gives as example of two partial solutions α (cost -3) and β (cost -2) being merged with respect to the cost function to give a partial solution γ. The completion problem of γ requires that the x_1 is assigned either the value 1 or 2, and both these assignments are supported. However the next unassigned variable in the array which is unassigned does not have support for all of the values in its domain. Without masking variables in the partial

$$X = (x_0, x_1, ..., x_6)$$

$$\forall_{0 \leq i \leq 6}.\ x_i \in \{0, 6\}$$

$$\texttt{VPC}(X)$$

$$\text{maximise} \sum_{i=4}^{6} x_i - \sum_{i=0}^{3} x_i$$

Figure 4.3: An example problem

solution γ, x_5 cannot take the value 5 (and as a consequence x_6 will never be assigned 6). Our method for dealing with these cases is to mask x_{i-1} with the value which is one less than the largest value in x_i (this is done on lines 12-15 of Algorithm 4.2. In the case of Figure 4.4c this means masking x_4 with the value 4 so that all the values in the domain of x_5 have support.

$$1 \quad x_1 \quad 2 \quad 3 \quad 4 \quad x_5 \quad x_6$$
$$x_1 \in \{1, 2\}$$
$$x_5 \in \{1, 2, 3, 4, 5\}$$
$$x_6 \in \{1, 2, 3, 4, 5, 6\}$$

(a) A partial solution α

$$1 \quad 1 \quad 2 \quad 1 \quad 3 \quad x_5 \quad x_6$$
$$x_5 \in \{1, 2, 3\}$$
$$x_6 \in \{1, 2, 3, 4\}$$

(b) A partial solution β

$$1 \quad x_1 \quad 2 \quad 1 \quad \underline{3} \quad x_5 \quad x_6$$
$$x_1 \in \{1, 2\}$$
$$x_5 \in \{1, 2, 3, 4, 5\}$$
$$x_6 \in \{1, 2, 3, 4, 5, 6\}$$
$$x_4 \text{ is masked with } 4$$

(c) A relaxed completion problem for partial solution γ

Figure 4.4: Two partial solutions to the problem modelled in Figure 4.3 and the completion problem resulting from merging them together. Masked variables in the completion problem are underlined.

Algorithm 4.2 shows how we mask variables to weaken the VPC constraint. Lines $7 - 10$ keep v updated to be the largest assignment in the array, and p to be the index of the rightmost assigned variable. In lines $11 - 15$, if an unassigned variable has values which are too large to have support, we mask the rightmost assigned variable at index p with a value large enough to provide this support. Finally, v is updated to take the value that x_p was masked with.

Proposition 4.2. Weakening the VPC constraint using Algorithm 4.2 allows for the valid simultaneous use of the VPC constraint and DDBB search in a constraint solver.

Proof. Consider a constraint optimisation problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}, f)$, where $\mathcal{C}$ includes VPC constraints, and assume that f satisfies the limitations set out in Section 2.4.4. Again, the

Algorithm 4.2: An algorithm for masking variables to weaken the value proceeds chain constraint.

```
1  maskValueProceedsChain (Node m)
2  begin
3      (x_0, ..., x_{n-1}) ← the array of variables in VPC at node m
4      v ← 0
5      p ← 0
6      for i ← 0 to n do
7          if x_i is assigned and not masked then
8              a ← the value assigned to x_i
9              if a = v + 1 then v ← a
10             p ← i
11         ub ← the upper bound of x_i
12         if ub > v + 1 + (i − p) then
13             n ← ub − (i − p)
14             v ← n
15             mask x_p at node m with n
```

proof of Proposition 4.2 is similar to the proof of Proposition 2.2, until we have to show that all values introduced to m from u and v are ensured support, after nodes u and v are merged to create a new node m.

Consider a value k in the domain of some x_i at node m which is introduced from node u, and let k be without support at m. Masking x_{i-1} with the value $k-1$ provides support for k in the domain of x_i.

□

4.5 Results

In this section we present results from using our solver to solve the forbidden subgraph problem, a problem which we return to in Chapter 6 to investigate using canonical graph search with DDBB search.

The goal of the forbidden subgraph problem is to find the maximum number of edges which can exist in a graph of a given order subject to some set of constraints on the properties of the graph. We will consider the case where the constraints impose that the graph includes no cycles of length three or four. The forbidden subgraph problem involves `lin_le`, `lin_eq` and `lexle` constraints. A model for the forbidden subgraph problem is given in Figure 6.3, to which we add constraints that each row of the adjacency matrix of the graph should be

		Order						
		6	5	7	8	9	10	11
Min value	Runtime DDBB	0.002116	0.015202	0.123717	0.813164	5.335966	30.736689	140.292831
	Space DDBB	92	322	1363	6315	25695	85180	238312
	Runtime FC	0.001525	0.008419	0.039647	0.223612	1.291028	8.535055	75.985458
	Space FC	31	89	269	936	3747	16412	101523
Max value	Runtime DDBB	0.002048	0.018906	0.175279	0.742474	2.651785	10.060629	94.026543
	Space DDBB	94	359	1640	4653	9136	20775	127247
	Runtime FC	0.001080	0.006202	0.029673	0.177567	0.978722	5.421451	65.995232
	Space FC	16	62	192	681	2745	10035	87430

Table 4.1: Comparison of execution times and size of the search space for our DDBB search based approach to search compared with forward checking.

lexicographically less than the succeeding row.

The results shown in Table 4.1 show that for the FSP DDBB does not beat forward checking an inspecting an single instance of the FSP we see that forward checking and DDBB find the same cost solutions, but forward checking finds these more quickly than DDBB. The nature of the FSP means that DDBB is unlikely to find good solutions early, because, for each order of graph, there are many more solutions which admit n edges than $n + 1$ edges.

Chapter 5

Implementing a solver and evaluating performance

5.1 Introduction

In this chapter we give an overview of the implementation details of a solver which which we have written to empirically evaluate the effectiveness of the approach laid out in the preceding chapters. We focus on the specifics of how we implement this solver such that it can be executed in parallel and make use of multicore CPUs. We give results which motivate the effectiveness of this parallelisation.

Our solver is written in Golang, a programming language developed at Google with contributions from the open source community. The language is also commonly called "Go". For a introduction to Go and a history of its early development see (Meyerson, 2014). Our solver makes use of Algorithm 2.4 as its search scheme, as well as the weakening algorithms given in Chapters 2, 3 and 4 to ensure that constraint propagators can be used together with relaxed width truncated search.

5.2 Why write a constraint solver in Go?

In this section we justify our choice of Go as the language in which we implement our solver. We also discuss some downsides to using Go to write a constraint solver which became apparent throughout our time working with it.

5.2.1 The other options

Before implementing a general purpose constraint solver, our main research interest was in solving graph search problems by using the decision diagram branch and bound algorithm introduced in Chapter 2. In Chapter 6 we will revisit this work to show that the approach is reasonably competitive for solving a particular graph search problem, the forbidden subgraph problem, but here we cover some implementation details from this work.

Our initial efforts to implement a forbidden subgraph problem solver used the Java programming language (Arnold, Gosling, and Holmes, 2000). This solver was written in such a way that it only could be used for graph problems where edges are added to a graph with no edges, until the largest graph that satisfied some constraints was found. After our initial prototyping we wished to implement a parallel version, but this is not easy in Java where the programmer is expected to explicity deal with threading processes. This led us to develop a similar solver in C++, and the goal of changing languages was to make use of Intel's Cilk++ job scheduler. Providing that the programmer implements their code in a sensible way, adding parallelisation using Cilk++ can be as easy as editing a single line of code. This was the case for us. However, we chose to stop working in C++ due to Cilk++'s deprecation during our research.

5.2.2 Benefits to using Go

After our initial work writing a forbidden subgraph problem solver in Java and C++ we knew what we were looking for in a programming language, namely a built in job scheduler for parallelisation. Go's Goroutines make it easy to evaluate functions concurrently, and its channel type allows for communication between concurrently executed functions.

5.2.3 Downsides

The main downside to using Go is that it is garbage collected. In languages which are not garbage collected the onus is on the programmer to ensure that memory is allocated and deallocated properly throughout the execution of a program. In a garbage collected language the burden of deallocating memory is lifted from the programmer and handled by the garbage collector.

Go's garbage collector has two main downsides that hamper the performance of our solver and our ability to evaluate its performance on many problem instances. The first downside is that Go's garbage collector is designed for high uptime, concurrent (probably distributed) microservices and is optimised for latency at the cost of throughput. We would instead prefer

less overall time spent on garbage collection. Go's garbage collector is also concurrent, and we have to take this into account when running experiments. For example, on a 32 core machine we cannot simply run 16 individual problem instances simultaneously as each instance of our solver will attempt to use up to 32 cores when in the garbage collection phase. In this case care must be taken to ensure that each instance of the solver can utilise at maximum 2 cores.

5.3 Parallelising our solver

The DDBB search algorithm that we use as the inspiration for our approach is shown to be easily adapted to run in parallel by processing multiple jobs from the queue concurrently when branching (Bergman, Ciré, Sabharwal, et al., 2014). Given this, and since multicore hardware is now ubiquitous, we feel that it is reasonable to utilise parallelism to see if the benefits seen when parallelising DDBB in the literature carry over to our approach and can reduce the execution times of our solver.

In this section we begin by giving a short review of just some of the research effort which has been put into parallel constraint solvers, an existing parallel decision diagram branch and bound algorithm, as well as parallel combinatorial search in general. We then detail how we make use of Go's built in Goroutines and its primitive channel type to parallelise our solver.

5.3.1 Parallel computing

For decades software developers could expect that the processors on which their code ran would roughly double in performance every one to two years. This phenomenon is famously known as Moore's law, named after Gordon Moore noted the trend in his 1965 article (2006). However this trend has stopped, slowing down in early 2000s as issues such as processor energy consumption and difficulties with manufacturing smaller transistors took effect. Nowadays multicore processors are the norm, which combine multiple processors known as cores on a single chip. To take advantage of this multicore hardware to speed up the execution time of programs the developer can write concurrent code, where multiple computations and subroutines are executed at the same time (usually on distinct processor cores). Writing code which is parallelised in this way is, however, in general more difficult than writing a sequential program and in some cases not possible at all Sutter (2005), Sutter and Larus (2005).

The type of parallelism we are interested in is task-parallel programs. Task-parallel programs are written as a number of multiple tasks, where each task is a subroutine. Tasks might

communicate with other tasks, have ordering constraints (task A runs before task B) and tasks might create new tasks. There exist a number of off the shelf task-parallel frameworks which assist software developers when writing parallel code by handling task scheduling. Some examples include Intel's Cilk++ (Leiserson, 2009) and Thread Building Blocks (Reinders, 2007) , HPX (Kaiser et al., 2014), OpenMP (Dagum and Menon, 1998) and the Golang runtime task scheduler (Deshpande, Sponsler, and Weiss, 2012). Since these frameworks take care of task scheduling the software developer is freed up to concentrate on the structure of each task and the program as a whole.

5.3.2 Parallel code in Go

In this section we review what tools Go includes for running concurrent code in parallel on modern multicore processors. Our parallel implementation of our solver makes use of each of the following tools.

Mutexes When multiple workers require access to the same variable care must be taken to ensure that no conflicts arise. If worker A reads a variable it might be the case that that variable's value is updated by one or more other workers while worker A continues to make use of the old value of the variable in subsequent calculations. Restricting the variables to be used by one worker at a time is a software design pattern known as mutual exclusion. The data structures which enforce this restriction are often called mutexes. Go's standard library includes the `sync.Mutex` data structure which has member functions `lock` and `unlock`. A call to `lock` allows a single single worker exclusive access to a block of code which lasts until a call to `unlock`. We make use of mutexes in our parallel implementation of our solver when ensuring that the node representing the incumbent best solution can be communicated between all workers.

Goroutines Goroutines are threads whose execution is managed by the Go runtime. Any function in Go can be run on its own thread by prepending the `go` keyword to a call to the function. All of the Goroutines in a program share the same address space, and so care must be taken to ensure that there are no conflicts arising from two goroutines accessing shared memory. While mutexes are a useful way to achieve this, Go also provides another method, namely channels.

Channels Typically the memory issues where multiple workers simultaneously access the same variable are not handled in Go programs through the use of mutexes. Instead channels

are used. Channels are primitive types in Go which are used as pipelines for communicating data between Goroutines. Goroutines can push data to a channel or pull data from it. These push and pull operations are blocking. If a worker is to pull data from a channel it must wait till some other worker to push data to that channel. Similarly if a worker is to push data to a channel then there must be some other worker ready to pull that data. Channels may be buffered, which allows channels to store data pushed to them so that push operations are no longer blocking, until the buffer is full. We make use of channels when communicating nodes to branch on between worker processes and the supervisor process which keeps track of the queue of jobs.

Go's select statement Go includes a special type of control flow statement to allow a Goroutine to listen to multiple channels at the same time. The select statement is similar to a more typical switch statement but its cases can include pushing data to a channel or pulling data from a channel. The body of each case is only executed if the channel operation can be done. This gets around the blocking nature of channels when a single worker is to listen to more than one simultaneously.

5.3.3 Writing efficient parallel code in Go

As parallel programs become more common, and frameworks for parallelising code (such as those mentioned in Section 5.3.1) gain traction, the practice of writing parallel code is no longer solely left to experts. Nanz, West, and da Silveira (2013) give code written by non-expert users of several of these parallelisation frameworks to expert users and the expert users then made comment on this code and provide their own implementations for comparison. In the case of code written using Go, the trend for non-expert implementations was to write recursive divide and conquer implementations which start one Goroutine per recursion. Meanwhile the expert implementations followed a distribute-work-synchronise design pattern with one worker Goroutine per processor core. Although the Go runtime can cope with scheduling many thousands of Goroutines, in general a more efficient approach is to write code which uses as many worker Goroutines as there are processor cores available. Nanz, West, and da Silveira do note, however, that the more efficient expert implementations required more effort on the part of the author.

5.4 Parallel combinatorial search

Parallelising combinatorial search so that solvers can make use of multiple processor cores concurrently can lead to speed ups in the execution time of such solvers. This has a real impact on all real world applications which are time sensitive. In the setting of the dynamic delivery problems mentioned in Section 1.1.1 proving delivery drivers with their optimal jobs every hour makes little practical sense, in practice we would want to be able to update the jobs available to drivers much more quickly than this. Another motivating factor for parallelisation might be the distributed nature of a problem across several localities (Prosser, Conway, and Muller, 1992).

In this section we cover at a high level what the existing methods of parallelising search are, what solvers have implemented which strategies, and cover the parallel version of the decision diagram branch and bound algorithm and finally present a parallel version of our decision diagram branch and bound constraint solver. While we focus on reviewing the literature necessary to understand the position of parallel decision diagram branch and bound search within the wider field of parallel combinatorial search, Archibald's (2018) thesis provides a more in depth review of combinatorial search in general.

5.4.1 Approaches parallelising combinatorial search

In general there are three main approaches to parallelising combinatorial search, which are outlined by Gendron and Crainic (1994) in their survey paper. Here we use the same terminology as Gendron and Crainic. The following approaches need not be used in isolation; parallel node processing can be used in a solver which also uses space-splitting and portfolio approaches might uses solvers that use either, or both, parallel node processing and space-splitting. For the purposes of parallelising our own solver we will use a space splitting approach.

Parallel node processing

The parallel node processing approach to parallelising combinatorial search works by parallelising the inference algorithms which run at each node. In constraint programming this means exploiting parallelism for either individual constraint propagators or by propagating constraints concurrently. Nguyen and Deville (1998) take the latter approach and introduce a distributed algorithm for maintaining arc-consistency. In a wider sense this approach also includes the bounding algorithms in a branch and bound setting. The goal of this approach is to minimise the time it takes to process each node in the search space, without altering how

the space is explored.

Space-Splitting

The space-splitting approach to parallelising combinatorial search takes the search space and splits it up. Searching each subspace forms a task which is explored by a worker and these workers may or may not communicate information, such as the current best known solution when solving optimisation problems.

Portfolio

The portfolio approach to parallelising combinatorial search concurrently runs (typically, but not always, sequential) core solvers (where a core solver is a solver which is not a portfolio solver). The aim of the portfolio approach is to make use of multicore hardware by speculatively running many different core solvers, or many instances of the same solver but with different search heuristics, simultaniously. For this approach each solver individual search must differ in some way, otherwise there is a doubling up of tasks between workers (Balyo, Sanders, and Sinz, 2015). The portfolio approach can be quite successful, so much so that solver competitions have banned solvers which just repackage other core solvers in a portfolio. The SAT competition enforces such a ban, with the portfolio solver SATzilla being a previous winner in 2007 and 2009 (Xu et al., 2011). SATzilla packages multiple core solvers and uses machine learning to determine which solvers from its portfolio to give as tasks to workers. The feeling within some parts of the SAT community is that portfolio solvers stifle research progress. Weidenbach (2017) explores this and concludes that portfolio solvers which use reasoning to determine which individual solvers to use, with which input parameters, as tasks or includes communication between solvers should continue to be an area of focus for research efforts. The constraint solver Choco takes the portfolio approach in the parallel tracks of the Minizinc Challenge (achieving a speed up of around 3 compared to their sequential implementation).

5.4.2 Parallel constraint programming

Constraint programming solvers are, compared to other solver technologies, often suitable for parallelisation. The commercial mixed integer programming solver Gurobi often struggles to achieve further speedups when using three or more threads. In its distributed version it manages a speedup of two to three times on eight machines (Gurobi promotional material, 2019). Parallel SAT solvers also struggle to scale well, achieving speedups of around three

when using 32 cores (Järvisalo et al., 2012). In constraint programming various strategies have been used to parallelise search to better effect. These include recursively applying search goals (Perron, 1999), work stealing (Chu, Schulte, and Stuckey, 2009), embarrassingly parallel search (Régin, Rezgui, and Malapert (2014), Yasuhara et al. (2015), Malapert, Régin, and Rezgui (2016)) and parallel limited discrepancy search (Moisan, Gaudreault, and Quimper, 2013).

5.4.3 Parallel decision diagram branch and bound

The particular parallel search that we are interested in is a parallel version of the decision diagram branch and bound algorithm on which our work is based. Bergman, Ciré, Sabharwal, et al. (2014)[1] propose a centralised dynamic space-splitting strategy based on DDBB. Their approach involves a supervisor worker which records a pool of nodes which are to be processed. The supervisor worker distributes these nodes to workers starting with the most promising nodes which construct relaxed and restricted BDDs based at the node. The workers reply to the supervisor with a set of nodes to explore or communicate the to supervisor and all other workers an improvement to the lower bound of the objective. The workers also send the upper bound found from constructing relaxed diagrams to the supervisor, so it might prune nodes from the pool.

5.5 Parallelising our solver

To parallelise our solver based on Algorithm 2.4 we follow the centralised work balancing approach taken in (Bergman, Ciré, Sabharwal, et al., 2014) and described in Section 5.4.3. A benefit of following this approach is that it allows us to write code which fits the one Goroutine per processor core approach advocated for in (Nanz, West, and da Silveira, 2013) which we described in Section 5.3.3. Note though that in our implementation the supervisor thread only distributes nodes to workers in a breadth first order, it does not attempt to rank nodes to pass out the most promising nodes to workers first.

In our parallel implementation we have a single supervisor thread which organises tasks and a single type of worker thread. The task completed by the worker threads is given in Algorithm 5.1 and is very similar to the body of the while loop in Algorithm 2.4. The key differences are that line 4 is now blocking, as the worker waits to receive a node u from the supervisor thread. The worker thread then explores relaxed width truncated search to provide

[1] (Bergman, Ciré, Sabharwal, et al., 2014) parallelises the approach of the later paper (Bergman, Ciré, van Hoeve, et al., 2016)

an upper bound to the optimal solution of the subproblem defined at the node u, if search has already found an incumbent solution. If this upper bound is greater than the cost of the current incumbent solution then the worker continues by exploring restricted width truncated search. In the case where the width of the search tree is restricted, the nodes in L_c are communicated to the supervisor thread so that these can be added to the pool of jobs. If when exploring the restricted width truncated search tree the worker finds a solution which is has a higher cost than the cost of the incumbent it updates the value of the incumbent, which is kept as a global variable to which all workers and the supervisor thread have access. A mutex is used to avoid race conditions when workers interact with this shared variable.

Updates to the incumbent are kept thread safe by use of a mutex, and in our implementation we make use of channels to keep track of which workers threads are currently doing work as well as for passing nodes between the supervisor and worker threads. The supervisor thread keeps a buffered channel of length equal to the number of worker threads, and "punches in" each worker by pushing a value to this channel before passing it a node to work on. When the worker is finished exploring relaxed and restricted width truncated search trees for the node it "punches out" by pulling from the channel. The purpose of this is so that we know when search is complete. It is no longer enough to stop search then the pool of jobs is empty, as we might have worker threads still exploring search. However when there are no more jobs left in the supervisor thread's pool and there are no values in the buffered channel we know that search is complete.

Algorithm 5.1: The task of a worker in parallelised DDBB search.

```
2  begin
3      while true do
4          u ← a node u sent from the supervisor thread
5          ub ← −∞
6          if search has found an incumbent best solution then
7              ub ← solveRelaxed(u)
8          z_opt ← the cost of the incumbent best solution
9          if ub > z_opt then
10             L_c, lb, candidate ← solveRestricted(u)
11             communicate each node in L_C to the supervisor thread
12             if lb > z_opt then
13                 z_opt ← lb
14                 incumbent ← update the shared incumbent with candidate
```

5.5.1 Results

We again return to the problems discussed earlier in this dissertation, on this occasion to show effect that parallelising search has on the time it takes our solver to find optimal solutions. We show that the method of splitting the search space up between different worker threads when branching in DDBB search leads to a constraint optimisation solver which scales well as workers are added. This agrees with the findings in (Bergman, Ciré, Sabharwal, et al., 2014) showing that DDBB scales well when more worker threads are added (this is not necessarily the case for all approaches to search). We also verify that these speed ups are consistent and reproducible (that is, each time the solver is run the deviation in execution times is small). We want to do check this reproducibility to show that we can avoid race conditions which lead to an unfavourable branching order. If we assume that we would rather branch towards the left of search trees (due to variable and value ordering heuristics trying to place good solutions at this side of the search tree) then we could see that workers branching on nodes further right in search returning their solutions early disturbing the ordering in which the whole search tree is explored.

Note that we do not compare our results to a parallel approach to forward checking. This is due to the choices this introduces when deciding how to parallelise forward checking (do we branch on some level of search or run a portfolio approach using multiple variable and value ordering heuristics concurrently?) and how to tune the extra parameters that parallelising forward checking would introduce.

Figure 5.1 shows the effect that doubling the number of workers has on the execution times of the solver when solving the car scheduling problem for numbers of workers ranging from 1 to 32. Although the charts shown in Figure 5.1 do show improvements in execution time for many instances we do see some instances for which performance degrades as more workers are added to the problem. This is due to the addition of workers creating race conditions when workers report back nodes to branch on to the supervisor thread. The order in which workers report back nodes to branch on is affecting the order in which the search space is explored.

Even though we do see slow downs when doubling the number of workers Figure 5.2 shows that for instances which take more than 1 second to complete we achieve up to an order of magnitude speed up when using 16 workers compared with 1 worker. For these instances with shorter executions times it is either simply not worth the effort to bring up multiple workers, or another instance of our solver reporting variable times for instances which are solved very quickly. Figure 5.2 also includes a chart showing that multiple runs using 16 worker threads give runtimes which do not show much deviation.

Our scalable and reproducable parallel speed ups are not only found when solving the car

factory scheduling problem. Figure 5.4 shows the effect of adding workers when solving the maximum independent subgraph problem. The results for the MISP are similar to the car factory scheduling problem, again showing that we may sometimes see a slow down when doubling the number of workers. However in Figure 5.3 we see that, for this problem, our approach scales well for all the instances solved when comparing the execution times achieved by 1 and 16 workers. Figure 5.3 also confirms that our parallel solver produces reproducable execution times and that scaling to 16 worker threads allows us to solve more instances faster than forward checking than we had previously managed with 1 worker.

Parallelising search has similar results for the optimal Golomb rulers problem. Table 5.2 shows that for multiple choices of variable and value heuristics we achieve good scaling for up to 16 worker threads.

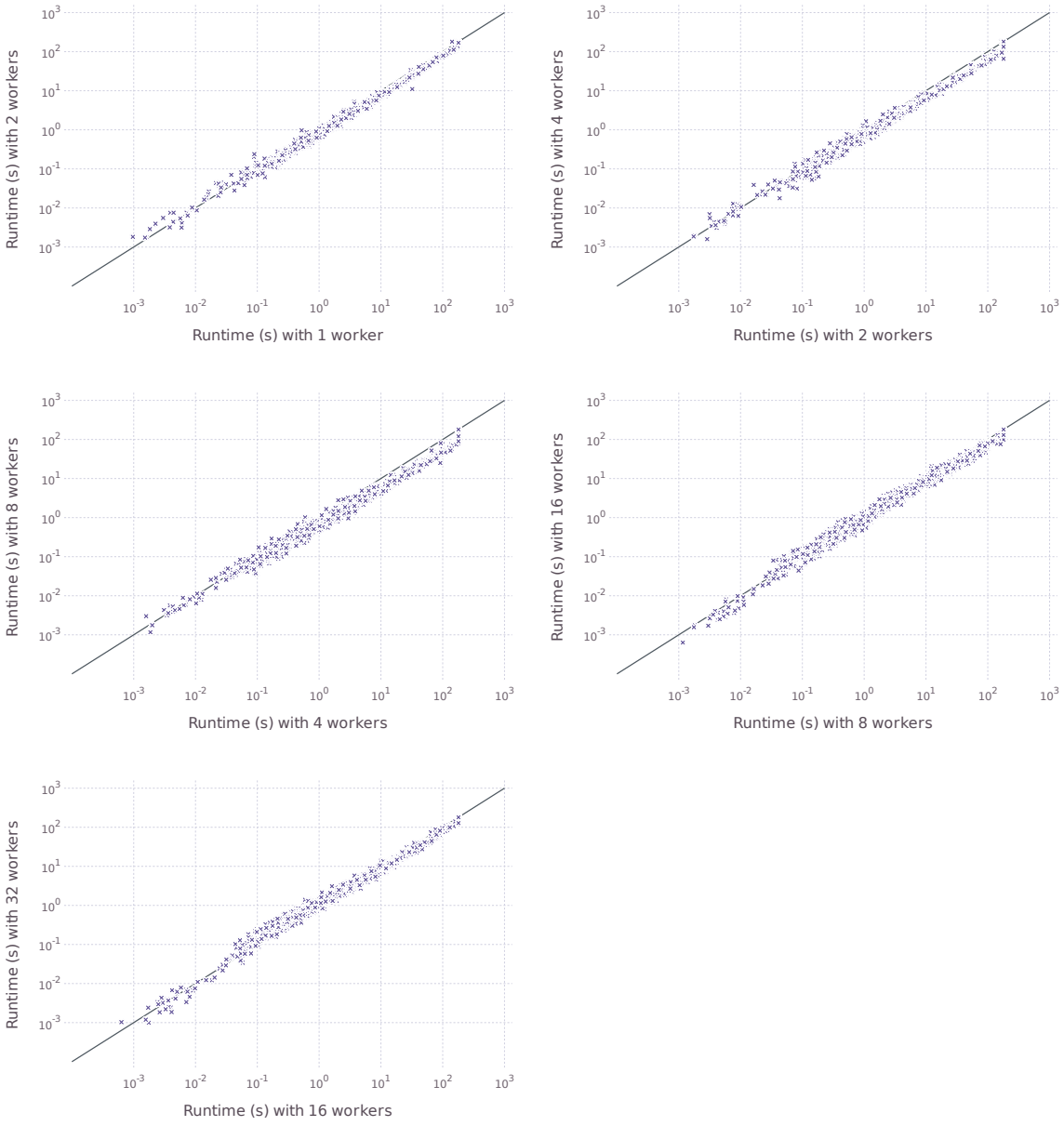

Figure 5.1: The effect of adding workers on execution times for the car factory scheduling problem.

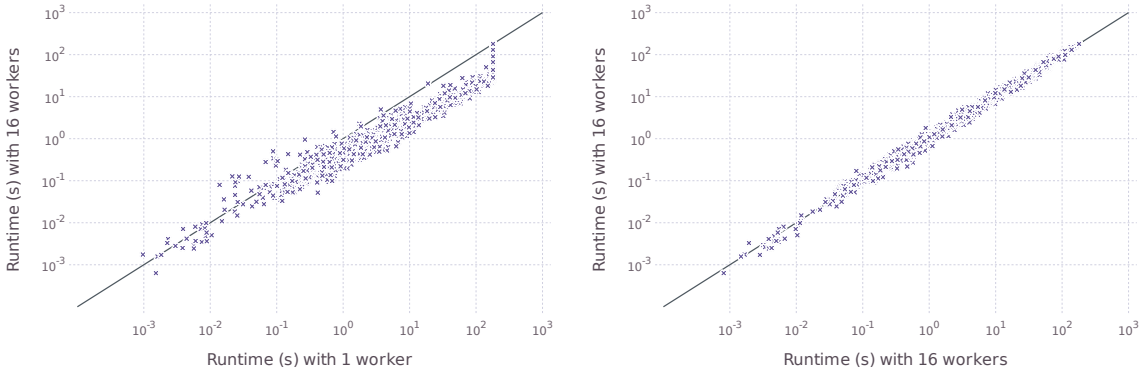

Figure 5.2: The effect of using 16 workers versus 1 worker for the car factory scheduling problem and the reproducibility of execution times when using 16 workers.

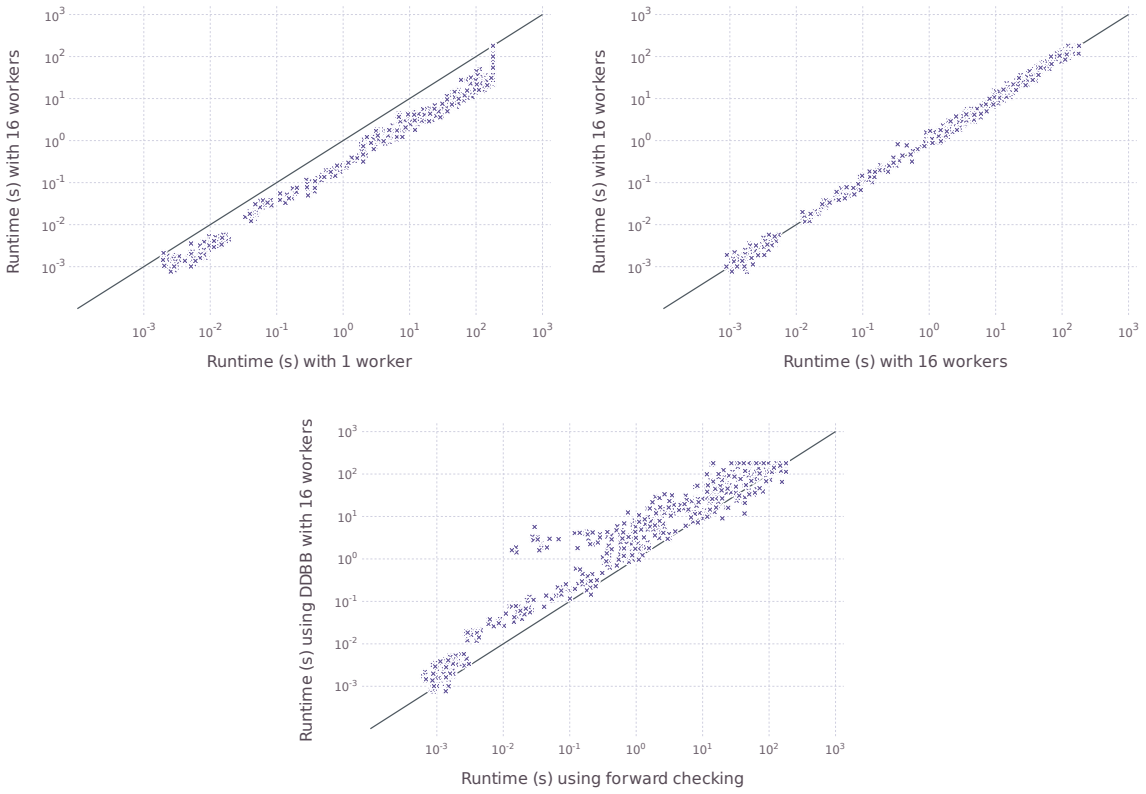

Figure 5.3: The effect of using 16 workers vs 1 worker; a reproducibility result; and improved performance against forward checking using 16 workers for the MISP problem.

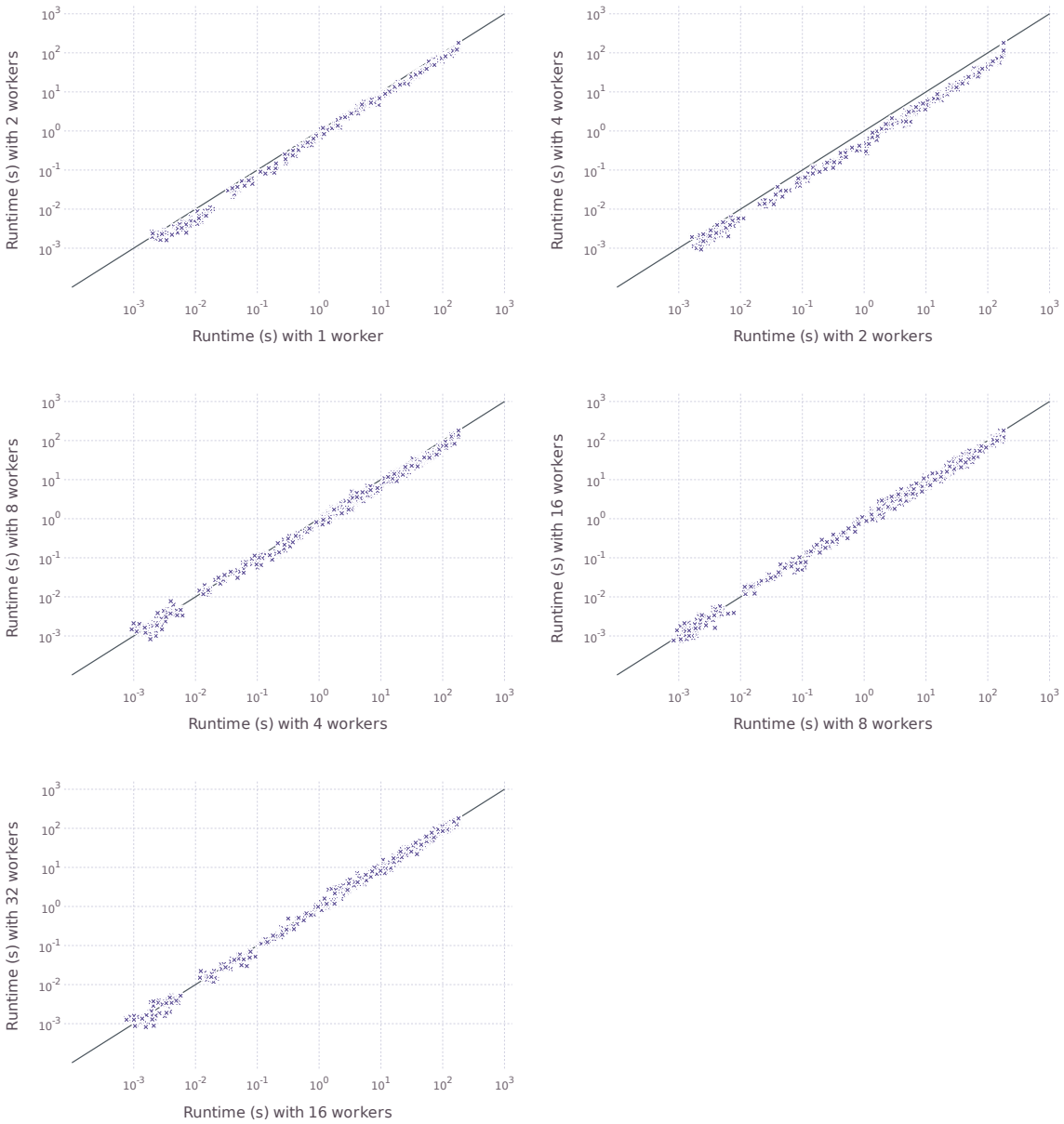

Figure 5.4: The effect of adding workers on execution times for the MISP.

Heuristics	Workers	Order 3		Order 4		Order 5		Order 6		Order 7		Order 8	
		Nodes	Runtime	Nodes	Runtime	Nodes	Runtime	Nodes	Runtime	Nodes	Runtime	Nodes	Runtime
input order/min value	1	22	0.002093	160	0.021789	696	0.259210	2445	2.982404	10590	29.984241	48610	280.420898
	2	22	0.001452	193	0.012121	872	0.171398	3335	2.048705	12882	18.926025	53259	162.697754
	4	19	0.000845	226	0.008805	1286	0.130143	4911	1.661415	17507	13.311588	67635	105.184341
	8	22	0.000927	191	0.007176	1209	0.093732	6480	1.522509	23793	10.492380	87888	71.958679
	16	22	0.001001	217	0.007632	1147	0.103240	5548	1.159400	24858	7.610966	113291	55.715733
input order/max value	1	19	0.001326	69	0.016747	224	0.192924	1118	2.165610	9380	40.541050	67536	600.000244
	2	17	0.000999	75	0.010499	224	0.108833	1143	1.248151	9757	24.096241	91786	373.246887
	4	19	0.000990	89	0.008275	224	0.077967	1143	0.726613	9867	13.537811	91943	192.773804
	8	17	0.001028	76	0.007033	219	0.064100	1519	0.768975	9778	8.416217	92047	103.240891
	16	17	0.000967	79	0.006854	255	0.059278	1458	1.000439	10064	6.579718	92614	62.525742
first fail/min value	1	22	0.001838	160	0.018601	698	0.244221	2432	2.955035	10282	28.915529	46742	276.317169
	2	22	0.001394	197	0.013188	959	0.179684	3334	2.047255	12735	19.072641	50837	155.896469
	4	22	0.000810	230	0.008930	1309	0.132485	4849	1.642257	17446	13.662830	59187	95.555603
	8	21	0.000945	198	0.007147	1223	0.092567	6560	1.546953	23573	10.484475	89676	72.993217
	16	20	0.001106	215	0.008175	1107	0.106821	5473	1.144000	24863	7.578467	110178	54.746403
first fail/max value	1	19	0.001311	69	0.020695	224	0.202024	1116	2.259621	9140	40.443085	66918	600.000244
	2	17	0.000940	70	0.013226	223	0.130812	1142	1.249129	9512	23.826376	90454	374.381500
	4	23	0.001019	76	0.008347	230	0.085210	1156	0.732458	9653	13.508552	91260	194.294296
	8	17	0.000988	75	0.007092	223	0.064028	1547	0.758330	9693	8.126902	91604	104.247383
	16	19	0.001032	82	0.006997	269	0.057001	1454	1.005092	9874	6.128321	92065	63.453548

Table 5.1: Comparison of execution times of our solver when finding optimal Golomb rulers for differing numbers of worker threads.

Chapter 6

Graph search problems

In this chapter we return to using our approach of weakening constraints to use DDBB to solve the forbidden subgraph problem. The forbidden subgraph problem is a is a graph search problem and we pair the approach given in earlier chapters with a common technique for graph search known as canonical graph search. We make use of an existing tool `Nauty` to ensure that we do not introduce equivalent graphs during search.

6.1 Introduction

A graph search problem is a decision problem where the goal is to find if there exists a graph which satisfies some set of constraints (for example maximum degree, minimum girth, maximum circumference). A common extension of these problems is to find the maximum or minimum solution with respect to some cost function. It is these optimisation problems that this chapter focuses on. A closely related problem is the enumeration variant which counts the number of graphs which satisfy the desired constraints. In this chapter we use the Forbidden Subgraph Problem (where, for a given order of graph, we try to maximise the number of edges present while not including one or more specified subgraphs) as a running example throughout.

Graph search problems are difficult in part due to the high number of potential symmetries. A naïve search will find multiple copies of the same graph up to isomorphism, which can lead to a vast amount of repeated work during search. One way to overcome this issue is to use an approach which only stores one canonical graph for each isomorphism class, using a tool like `Nauty` (McKay and Piperno, 2014). However, although such graph isomorphism tools often perform well in practice, repeated calls to them can cause the approach of including them in search to be uncompetitive. Another option is to use incomplete symmetry

breaking constraints or predicates as discussed in Chapter 4. Codish et al. (2019) use such an approach when they proposed an enhanced lexicographical ordering constraint for the forbidden subgraph problem. The downside to this approach is that these constraints often admit multiple solutions which represent graphs in each isomorphism class. However the additional computational effort at each node is small, making the approach attractive, even though it does not ensure that no symmetrically equivalent solutions are found.

While testing new ideas to tackle graph search problems researchers will often write simple dynamic programs. However, these do not scale well and are often only used for limited validation of results. The DDBB search algorithm (Bergman, Ciré, van Hoeve, et al., 2016) that we have based our own work on solves dynamic programming models indirectly, and without exploring the whole search space of the dynamic program.

This chapter proposes a method for solving graph optimisation problems using the decision diagram branch and bound approach to search in a constraint solver proposed in Chapter 2. We make use of the work in earlier chapters which weakens constraints. In this chapter we make use of a C++ implementation of Algorithm 2.4 so that we may include `Nauty` in our solver, which we also parallelise by using Intel's Cilk++ C++ language extension. We explore the relationship between branch and bound search and canonicalised search, which are two techniques which conflict with each other.

6.2 Graphs, Graph Isomorphism and the FSP

Recall that a graph $G = (V, E)$ is a set of vertices V (which we will sometimes write as $V(G)$ for clarity) together with set of adjacent pairs of vertices, called edges, E (and if two vertices v and w are adjacent we write $v \sim w$; that a path in a graph is a sequence of vertices such that each consecutive pair are connected by an edge; and that a cycle is a path that begins and ends at the same vertex. The *girth* of G is the length of the smallest possible cycle contained in G.

A *graph isomorphism* between graphs G and H is a bijective mapping $\phi : V(G) \to V(H)$ such that $v \sim_G w$ if and only if $\phi(v) \sim_H \phi(w)$. If there is an isomorphism between G and H, we say that G and H are isomorphic. In this chapter we make use of Nauty to check if two graphs are isomorphic. For each class of isomorphic graphs Nauty can return a unique canonical representation. Figure 6.1 shows why considering isomorphisms in search is important. Even in the simple problem of enumerating all graphs of order 3, there are multiple symmetrically equivalent graphs found.

A *(non-induced) subgraph isomorphism* from a graph P (typically called the *pattern*) to a graph T (the *target*) is an injective mapping $\mathrm{V}(P) \rightarrowtail \mathrm{V}(T)$ which maps adjacent vertices

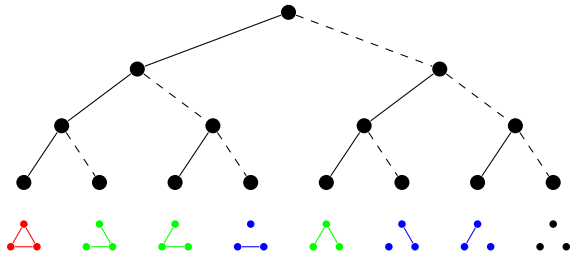

Figure 6.1: The search tree for generating all graphs of order three. Each edge is treated as a variable, and the graphs are presented at the leaf nodes. Each graph which is the same up to isomorphism is rendered in the same colour.

in P to adjacent vertices in T. An *extremal graph* is a graph which, for some given number of vertices and some set of constraints, $\mathcal{C}$, contains the maximum number of edges possible while satisfying $\mathcal{C}$. The *Forbidden Subgraph Problem*, (FSP), is to find an extremal graph which does not admit as a non-induced subgraph any member of some set of graphs $\mathcal{P}$. The size of the largest graph of order v with minimum girth $n+1$ is denoted by $f_n(v)$. Figure 6.2 shows an example of an extremal graph on 6 vertices with no 3- or 4-cycle.

The maximum number of edges permitted in graphs of order v for this instance of the FSP have been determined by Garnick, Kwong, and Lazebnik (1993) up to $v = 24$, using mathematical proof. They also use a local search approach to provide bounds for $v \leq 200$. Codish et al. (2019) use this variant of the FSP as a motivating example to show the impact of new symmetry breaking constraints and use a SAT solver to provide empirical results for the minimum girth 5 variant of the FSP for orders of graph from $11 - 24$, 26, 31 and 32. Their approach is to view the problem as a sequence of decision problems and they report runtimes for both the decision problem for the optimal solution and the proof that it is optimal. Note that the execution times for our approach are not competitive with the work of Codish et al. **annealing** (**annealing**) also make use of local search techniques, using simulated annealing to provide bounds for both the minimum girth 5 and minimum girth 6 variations of the FSP.

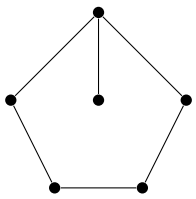

Figure 6.2: The largest graph with 6 vertices which does not include a 3-cycle or 4-cycle.

$$\forall_{1\leq i<j\leq v}.\ (A[i,j] = A[j,i] \quad \text{and} \quad A[i,i] = 0) \tag{6.1}$$

$$\forall_{i,j,k}.\ A[i,j] + A[j,k] + A[k,i] < 3 \tag{6.2}$$

$$\forall_{i,j,k,l}.\ A[i,j] + A[j,k] + A[k,l] + A[l,i] < 4 \tag{6.3}$$

$$\forall_{i,j}.\ \text{maximise} \sum_i \sum_j A[i,j] \tag{6.4}$$

Figure 6.3: Basic constraint model for extremal graph problems (no cycles of length 4 or less) with v vertices and e edges

6.3 Modelling the Forbidden Subgraph Problem

In this section we progressively build upon a simple model for finding extremal graphs of minimum girth 5. We show how we can include optimisations from the literature in our implementation and that we can make use of `Nauty` to only keep a single graph from each isomorphism class when building layers of our search trees. We include an overview of how well DDBB parallelises for this problem. We also show how parallelisation and canonical search interact when both are used together.

6.3.1 A First Model

Following the approach set out in Chapter 2 means that we can start with the same constraint model shown in (Codish et al., 2019). We model the problem using a binary variable for each edge, representing G by an $n \times n$ adjacency matrix A where $A[i,j] = A[j,i]$ and $A[i,j] = 1$ if $i \sim_G j$, and $A[i,j] = 0$ otherwise. Figure 6.3 shows the model for this minimum girth 5 instance of the FSP as given in (Codish et al., 2019). We also require a cost function, and this is added to the model as a fourth condition. Constraint 6.1 ensures that A represents simple undirected graphs. Constraints 6.2 and 6.3 enforce that A cannot admit three- or four-cycles.

6.3.2 Symmetry Breaking

To make use of `Nauty` during search, we enhance each node by also recording the canonical representative of the partial solution recorded at the node. We pass to `Nauty` an adjacency matrix where each assigned variable takes its assignment and each unassigned variable has the value 0. When adding a newly created node to a layer, we first check that no node currently in the layer is labelled with the same canonical representative.

We also make use of a degree sequence based symmetry breaking constraint and add it to the model as an example of the use of a partial symmetry breaking constraint. This additional constraint requires that:

$$\forall_{1 \leq i \leq v-1} \sum_{i} A[i] \geq \sum_{i} A[i+1] \tag{6.5}$$

where $A[i]$ is the i^{th} row in the adjacency matrix. Figure 6.4 shows the relative execution times when we include this constraint, canonicalised search using `Nauty` and both approaches together in search for a range of graph orders. Similarly, Figure 6.5 shows the effect of each approach and their combination on the size of the search space. From these figures it is apparent that while adding `Nauty` to search gives the greatest single improvement it is advantageous to use partial symmetry breaking constraints together with canonicalised search. We also make sure that including ether symmetry reduction technique speeds up execution times and reduces the size of the search space.

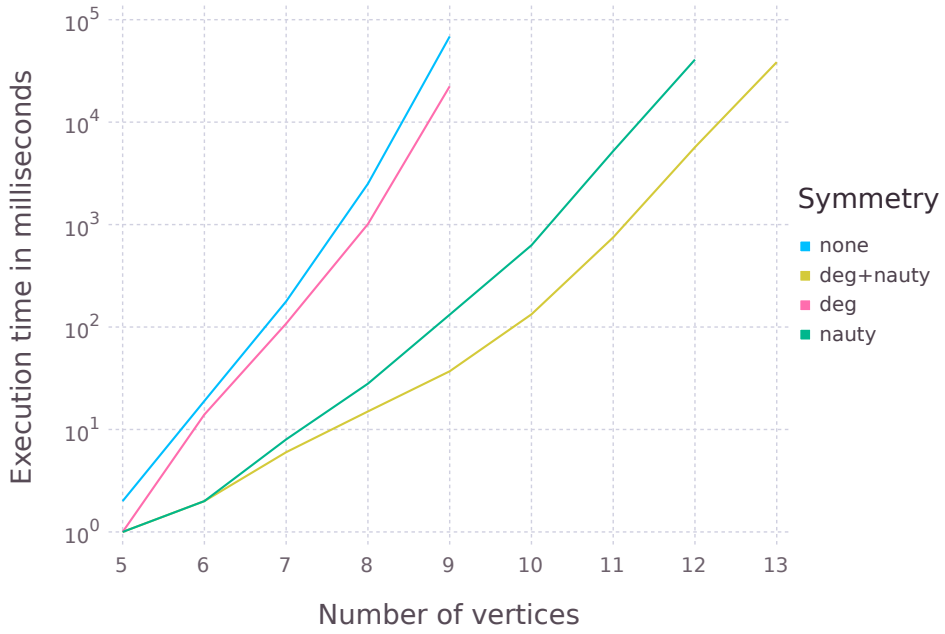

Figure 6.4: Comparison of execution times (in milliseconds) when including different symmetry reduction approaches.

As a sanity check to ensure that we are not just slowing down a canonical search, we ran the algorithm without using relaxed search trees to provide bounds during search. Without bounds we perform a parallelised breadth first search. Figure 6.6 and Figure 6.7 show that we do gain benefit from using relaxed search trees to provide bounds during search. These relaxed search trees allow us to prune search such that a smaller number of search nodes are explored overall

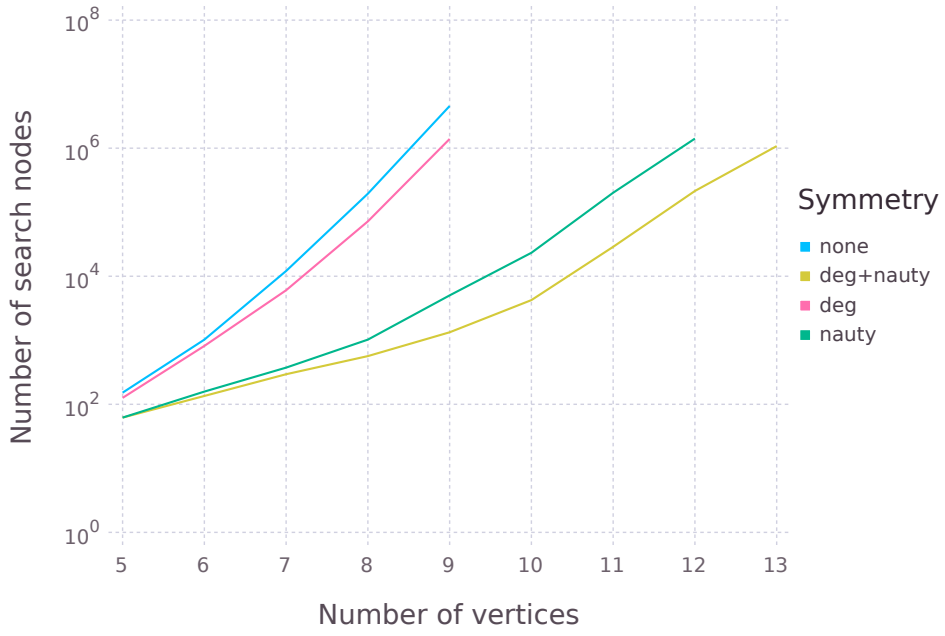

Figure 6.5: Comparison of search space size when including different symmetry reduction approaches

(including those in bounding search trees) which in turn leads to reduced execution times. The results for graphs of order 8 appear to be the opposite of the other orders with respect to execution times, but this is due to noise inherent in trying to measure experiments which last only a very short time. When considering the number of nodes used, we see that then benefit given by including relaxed nodes is apparent for all orders of graph chosen.

6.3.3 Parallelism

As stated previously, our implementation of DDBB is easily parallelised. For each node in a layer that is not pruned when branching, we start a new job that proceeds with search from the node. This happens recursively, with each job also branching in the same way if its initial exact search becomes too wide. The only communication that occurs is when jobs compare solutions to update the incumbent solution. Using Intel's Cilk++ to schedule and execute jobs in parallel and this requires only that we use a `cilk_for` loop to iterate over the nodes that are to be branched on. However, the use of Cilk Plus does not come without a cost. While in general we do get good speedups as shown in Figure 6.8, using Cilk Plus leads to variable execution times. Intermittently for some instances the execution time for 32 workers would approach the single threaded execution time. Similar behaviour when using Cilk Plus is reported in (McCreesh and Prosser, 2015). We can also see that it takes several graph orders for the expected sequence of the lines in Figure 6.8 to emerge. For these lower

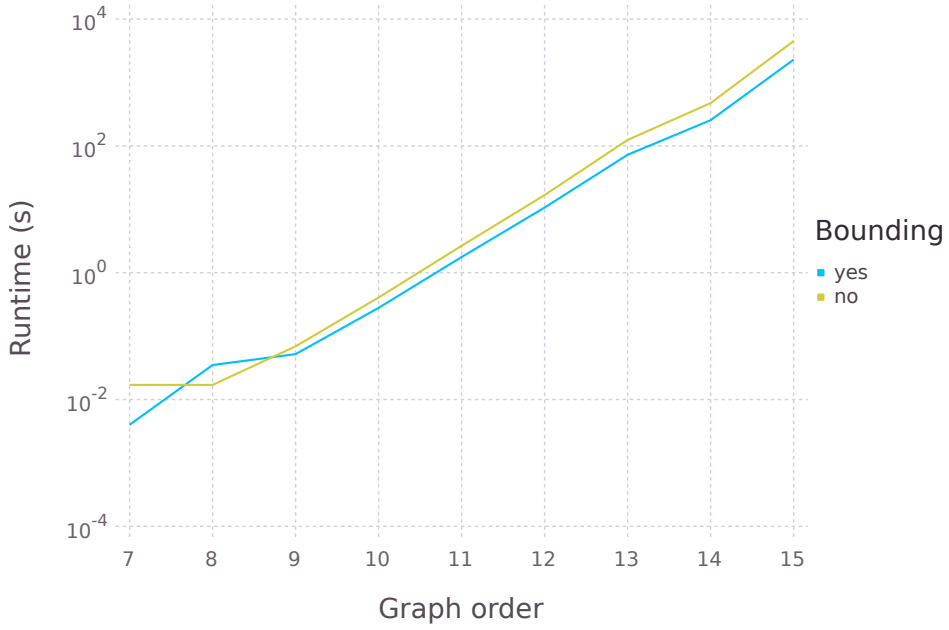

Figure 6.6: Execution times when relaxed search trees are and are not in use.

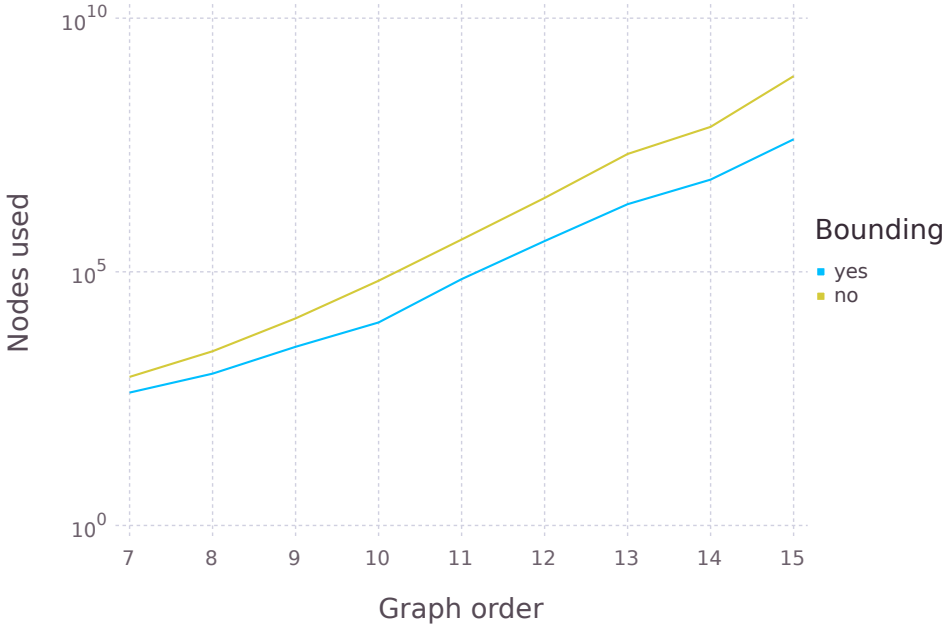

Figure 6.7: The size of search space when relaxed search trees are and are not in use.

orders the benefit from running more workers does not outweigh the cost of starting them up and communicating via shared data.

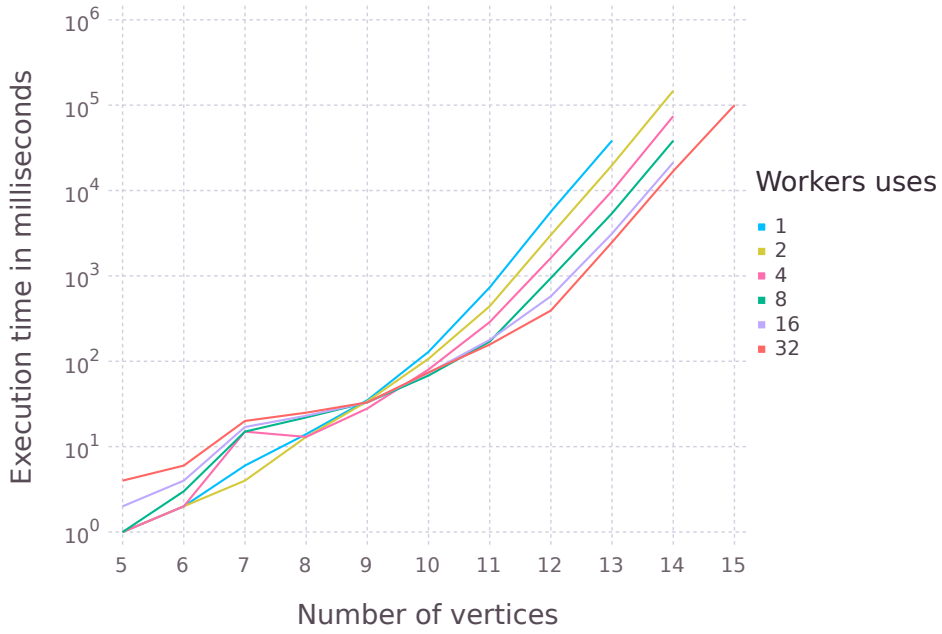

Figure 6.8: Parallel speedups, with runtimes in milliseconds

6.3.4 Branch and bound vs Canonicalised Search

There is an inherent conflict between using canonicalised search as a symmetry reduction technique and branch and bound search. Through splitting up the search space when branching we expect that we dilute the effectiveness of this approach to symmetry reduction. When we determine that branching should occur at a layer in a restricted search tree, subsequent layers are split between multiple workers and we do not communicate what canonical solutions have been found across multiple workers. Given that layers of search are split between multiple workers we can only reduce symmetries with `Nauty` locally on the search space being explored by each individual job given to a worker. In this way we localise the effectiveness of the canonical search approach to symmetry breaking to each individual subproblem being tackled in a single job. Note that incomplete symmetry breaking constraints are not adversely affected by splitting the search space up in this way.

However, Figure 6.9 shows that diluting the effectiveness of symmetry reduction using `Nauty` has only a limited effect on the size of the search space. The results produced throughout this chapter have been obtained by branching when a layer is as wide as the expected maximum height of the restricted search tree(the number of unassigned variables). In Figure 6.9 we

instead branch when the width of a layer is some multiple w of the height, and plot w against the size of the search space for a range of sizes of graph. We would expect that branching at a lower layer during search leads to fewer jobs being created and stronger pruning from `Nauty` as more isomorphism solutions are found in larger layers, however Figure 6.9 paints a more complicated picture. While it is clear that larger maximum permitted search widths perform better, there is once again not a monotonic relationship between varying search width and performance of our solvers. This shows that making better choices about which nodes to branch on can overcome the lack of search space reduction via isomorphisms. For many instances the smallest possible search space is found when a balance is struck between the amount of branching done (the width of restricted search) and having wide enough layers such that `Nauty` is impactful.

6.3.5 The Effect of Bounding Search Widths

As well as in the search width, there are two further locations in DDBB search where there is a heuristic choice to be made regarding the width of the search tree (in both restricted and relaxed search). As we have shown, changing each of these widths has an effect on both execution times and the size of the search space. In Sections 4.2 and 4.3 we have fixed the maximum width permitted to be equal to the height of the search tree (that is, the number of variables still to be assigned when the subproblem is first constructed). Here we consider how altering the maximum width of relaxed and restricted search trees affects execution times.

In the relaxed search trees which provide bounds during search, increasing the maximum width increases the strength of the bounds. A tighter bound will be found by a wider relaxed search that includes fewer relaxed nodes, and therefore more closely resembles the search tree rooted at the same node. However, there is a trade-off to be made regarding the strength of the bounds and execution times. A narrower search three might give weaker bounds, but these bounds are computed much more quickly. Bounding search using a relaxed search that closely resembles search means that we are approximately doubling our workload. Figure 6.10 shows the impact on performance that comes from doing too much or too little work in relaxed search. As expected there is a sweet spot between not having wide enough relaxed search trees to get good bounds and This is why we see this wing like behaviour in the lines in Figure 6.10.

6.3.6 Further optimisations

Throughout this chapter we have built on a simple model with symmetry reduction techniques that we introduce as novel improvements to DDBB search in the case where the problem

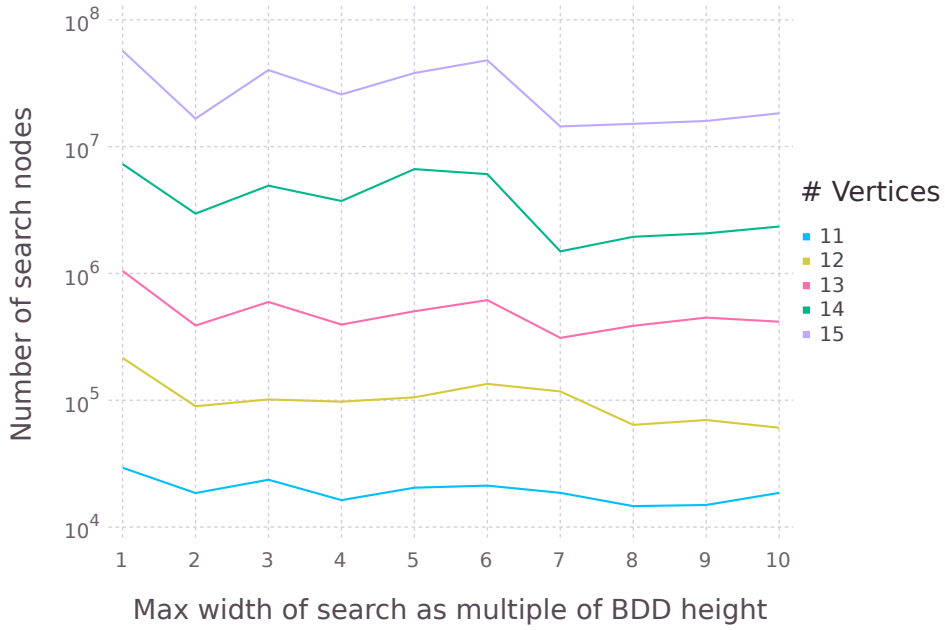

Figure 6.9: The effect of the maximum permitted width of a layer during restricted search on the search space when nodes are merged using `Nauty`.

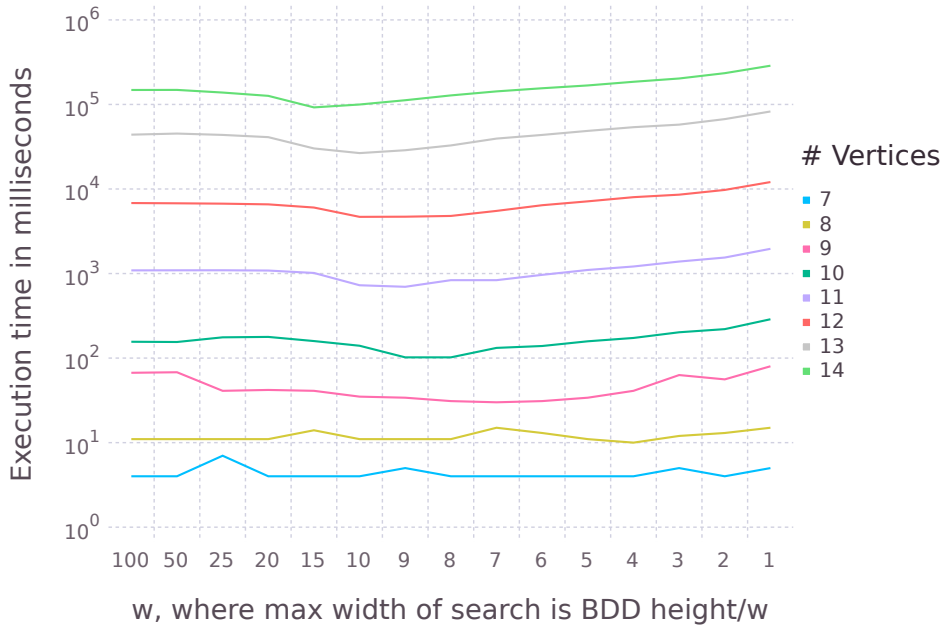

Figure 6.10: Effect of maximum permitted width of relaxed search trees on execution time of our algorithm

$$v \leq 1 + \delta\Delta \leq 1 + \delta^2$$

$$\delta \leq e - f_4(v-1)$$

$$\Delta \leq \frac{2 \times e}{v}$$

Figure 6.11: Bounding conditions from Propositions presented in (Garnick, Kwong, and Lazebnik, 1993).

being modelled is a graph search problem.

For the variant of the FSP that we have used to illustrate our approach we have not exploited some useful propositions presented in (Garnick, Kwong, and Lazebnik, 1993). These propositions bound the maximum and minimum degree of a vertex belonging to a graph of order v if the value of the optimal solution for $v-1$ and e, the expected number of edges for a graph of order v, are known. Figure 6.11 shows these bounding conditions. These additional constraints are of use when solving the forbidden subgraph decision problem (i.e. does there exist a graph with v vertices and e edges which does not include some family of forbidden subgraphs?). In order to make use of these additional constraints (Codish et al., 2019) addresses the optimisation variant of the FSP as a sequence of decision problems and reports the execution times for both e equals the optimal value value and e equals the optimal value plus one (which is an unsatisfiable instance of the decision problem, and the proof of the optimal size of graph).

We cannot make full use of these results when solving the optimisation problem with our approach as we do not know a value for e at the top of search. However, we can use these propositions to provide an upper bound for bound of the size of the optimal solution and Δ. This allows us to seed our algorithm with the upper bound for the optimal solution. We can also can set that the maximum degree of the first row in A is less than or equal to Δ. We can also terminate computation of relaxed search trees if a partial solution of size $e+1$ is found and terminate the entire algorithm if search finds a candidate solution of size e. For some instances where the bound we calculate for e is tight this greatly improves execution times. The problem for $v = 15$ is solved by our solver in less than 1 second.

	Order						
	6	5	7	8	9	10	11
Runtime DDBB	0.002116	0.015202	0.123717	0.813164	5.335966	30.736689	140.292831
Space DDBB	92	322	1363	6315	25695	85180	238312
Runtime DDBB+Nauty	0.002153	0.014643	0.120640	0.802695	5.382084	28.446503	105.735497
Space DDBB+Nauty	92	322	1363	6020	22839	67807	152032
Runtime FC	0.001525	0.008419	0.039647	0.223612	1.291028	8.535055	75.985458
Space FC	31	89	269	936	3747	16412	101523

Table 6.1: Comparison of execution times of DDBB search with and without Nauty compared with forward checking

6.4 Adding Nauty to our Go implementation

By using CGo, Go's library that allows Go code to make use of C libraries, we can see if adding Nauty to our solver makes our code any more competitive when compared with forward checking. The results given in Table 6.1 were recorded using a model which includes the lexicographic ordering constraints from Chapter 4. We can see that with this stronger partial symmetry break (than the degree based symmetry used elsewhere in this chapter) that using `Nauty` has no effect on reducing the size of the search space for smaller orders. However, even when using `Nauty` does begin to reduce the size of the search space it still does not beat a forward checking approach. However, when considering the execution times for including `Nauty` in our solver written in Go we have to consider that calls to CGo are slow. Even though forward checking still uses less search nodes than our approach combined with `Nauty` this is still an interesting result that the use of `Nauty` reduces the size of the search space when used with DDBB search. It might be that for some other graph search problems our approach is more performant.

Chapter 7

Conclusion

In this chapter we summarise the work presented in this thesis. We then suggest the direction that future work building on this thesis could take.

7.1 Summary

Chapter 2

In Chapter 2 we introduce decision diagram branch and bound search, after first reviewing relevant literature regarding the use of decision diagrams in the field of optimisation. We detail how we adapt decision diagram branch and bound search to be used in a constraint solver in Algorithm 2.1, Algorithm 2.2, Algorithm 2.3 and Algorithm 2.4. To use these algorithms as as the search scheme in a solver we accommodate the problem relaxations introduced during by using Algorithm 2.3 by weakening individual constraints. Our goal when weakening constraints is to ensure that propagating constraints after a node is created by merging to search nodes does not cause the deletion of any values from the domains of the unassigned variables in the newly created node. In general we achieve this by masking variables involved in the constraint with a value which differs from their assignment, such that all values in the domains of variables at the newly created node have support.

However, this method of weakening constraints is not a "one size fits all" approach and we show how to weaken a number of constraints. In some instances we instead show that there is no need to weaken a particular constraint.

We end Chapter 2 with an empirical evaluation which shows that for some instances of the car factory sequencing problem our approach to using DDBB search as the search scheme

in a constraint solver performs well when compared with forward checking. We zoom in on one instance to show the reasons why DDBB outperforms forward checking on some instances, namely that better solutions can be found more quickly and the search space can be agressively pruned using relaxed width truncated search. We also evaluated the performance of our solver using the maximum independent subgraph problem, for which our approached is outperformed by forward checking for all but a few instances due to the nature of the problem for most distributions of vertex weights we investigate.

Chapter 3

In Chapter 3 we continue to provide weakening algorithms for constraints, in this instance weakening a number of global constraints in the all different class of constraints. We begin by considering the `alldiff` constraint and we explain, in high level terms, how Regin's `alldiff` propagator works to determine if an `alldiff` constraint is unsatisfiable and, if it is not, to prune values from the domains of unassigned variables which cannot appear in any solutions. Our approach to weakening the `alldiff` constraint is to mask any variable assignment which is in conflict with a value in the domain of any unassigned variable with a wildcard value. We give an algorithm for this masking procedure which weakens the `alldiff` constraint in a way which requires no modifications to the propagation algorithm.

We provide yet more weakening algorithms for global constraints in the remainder of this chapter and again end with a short empirical evaluation of our solver, using the optimal Golomb rulers problem and the cell block allocation problem. For Golomb rulers our solver is outperformed by forward checking, but outperforms forward checking on a number of instances of the cell block allocation problem.

Chapter 4

In Chapter 4 we turn our attention to proving weakening algorithms for symmetry breaking constraints. These are of particular interest to us, given that our first attempts to use decision diagram branch and bound as the search scheme for a constraint solver came in our implementation of a solver focused on extremal graph problems (we return to this problem in Chapter 6). We begin by giving a short review of symmetry breaking constraints, and cover the popular lex-leader method for introducing symmetry reduction to the model of a problem.

We provide weakening algorithms for lexicographic ordering constraints and the `valueProcedesChain` constraint and evaluate our the performance of our solver on the forbidden subgraph problem, for which forward checking outperforms DDBB search.

Chapter 5

In Chapter 5 we conduct the bulk of our experimental work. We also give implementation details about our solver. We provide a study into how altering the maximum width of the restricted and relaxed width truncated search trees used in Algorithm 2.3 effect both the execution times and size of the search space used by DDBB search.

We parallelise our solver in an attempt to improve its performance by exploiting modern multicore processors. We give a brief overview of parallel computing and approaches to parallel search. When parallelising our solver we make use of Goroutines and channels to schedule tasks and communicate between workers which conduct search. We provide empirical evidence to support that our approach both scales well (more workers leads to faster execution times of our solver) and gives reproducable execution times.

Chapter 6

In Chapter 6 use DDBB search together with canonical graph search by implementing a solver in C++ which makes use of the graph isomorphism library Nauty. Though using `Nauty` we can remove all nodes from a search tree which represent a graph which has already appeared at some node we have already considered. We also parallelise this solver and explore the conflicting relationship between canonical search and branch and bound.

7.2 Future work

In this section we give our suggestions for possible areas of focus for work which builds on this thesis.

7.2.1 Hybrid approach to search

It might be that using width truncated search trees to provide bounds during search is useful, but that it is better paired with more traditional search schemes. For example in the context of embarrassingly parallel search it might be useful to use bounding search to prune jobs before parallel search begins. Decision diagram branch and bound may also pair well with clause learning and restarts.

7.2.2 Provide weakening algorithms for more constraints

To truly say that we have an approach which is appropriate for a general purpose constraint solver we should provide weakening algorithms for more constraints. An goal could be to provide a weakening algorithm for each constraint in the Minizinc language by working on the benchmark problems provided by the language developers. In particular the implementation of implications constraints would open up a number of new problems on which we could benchmark our solver.

7.2.3 A more efficient implementation

Although we report that our approach of using DDBB search as the search scheme in a constraint solver performs well against forward checking, it still lags behind industrial quality solvers. Although the comparative assessment against forward checking is useful as it limits the only difference between the approach to the way in which the search tree is explored, to truly ascertain the competitiveness of our approach we should implement a solver with performance in mind. For example, our current approach struggles to allow the garbage collector in Golang to work efficiently at times for both DDBB search or forward checking. We also do not do any preprocessing of the input models, which is a standard approach for industrial solvers.

7.2.4 Tuning solver parameters

We saw that for some problems there was a sweet spot where our choice widths improved the execution time of our solver the most. However for other problems the best choice of width changes on an instance by instance basis. When this is the case it is very difficult to tune these parameters by hand, but we could make use of machine learning to tune parameters, perhaps making use of a tool such as SMAC (Hutter, Hoos, and Leyton-Brown, 2011).

7.2.5 Better ordering of jobs (in parallel search)

Throughout our experiments we have explored jobs when branching in a depth first manner, in an attempt to find better incumbent solutions more quickly than forward checking through the limited amount of search explored by restricted width truncated search trees. We also saw that adding more workers in parallel could change the order in which nodes were visited, impacting the execution time of our solver adversely for some instances of multiple problems

(although when using 16 workers when compared to 1 worker this behaviour was all but eliminated). We should follow the approach of Bergman, Ciré, Sabharwal, et al. (2014) in allowing the supervisor thread to order jobs in a best first manner, which would mitigate the impact of these race conditions.

References

Akers, Sheldon (June 1978). “Binary Decision Diagrams”. In: *IEEE Transactions on Computers* 27.06, pp. 509–516. ISSN: 0018-9340. DOI: 10.1109/TC.1978.1675141 (cit. on p. 22).

Andersen, Henrik R., Tarik Hadzic, John N. Hooker, and Peter Tiedemann (2007). “A Constraint Store Based on Multivalued Decision Diagrams”. In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pp. 118–132. DOI: 10.1007/978-3-540-74970-7_11 (cit. on p. 24).

Archibald, Blair (2018). “Algorithmic Skeletons for Exact Combinatorial Search at Scale”. PhD thesis. University of Glasgow (cit. on p. 106).

Arnold, Ken, James Gosling, and David Holmes (2000). *The Java Programming Language*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201704331 (cit. on p. 102).

Babcock, Wallace C (1953). “Intermodulation interference in radio systems frequency of occurrence and control by channel selection”. In: *The Bell System Technical Journal* 32.1, pp. 63–73 (cit. on p. 81).

Balyo, Tomás, Peter Sanders, and Carsten Sinz (2015). “HordeSat: A Massively Parallel Portfolio SAT Solver”. In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, pp. 156–172. DOI: 10.1007/978-3-319-24318-4_12 (cit. on p. 107).

Beldiceanu, Nicolas (2001). “Pruning for the Minimum Constraint Family and for the Number of Distinct Values Constraint Family”. In: *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, pp. 211–224. DOI: 10.1007/3-540-45578-7_15 (cit. on p. 45).

Bergman, David, André A. Ciré, Ashish Sabharwal, Horst Samulowitz, Vijay A. Saraswat, and Willem Jan van Hoeve (2014). “Parallel Combinatorial Optimization with Decision Diagrams”. In: *Integration of AI and OR Techniques in Constraint Programming - 11th*

International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings, pp. 351–367. DOI: 10.1007/978-3-319-07046-9_25 (cit. on pp. 103, 108, 110, 133).

Bergman, David, André A. Ciré, Willem Jan van Hoeve, and John N. Hooker (2016). "Discrete Optimization with Decision Diagrams". In: *INFORMS Journal on Computing* 28.1, pp. 47–66. DOI: 10.1287/ijoc.2015.0648 (cit. on pp. 1, 17, 24, 25, 36, 108, 118).

Bergman, David, Willem Jan van Hoeve, and John N. Hooker (2011). "Manipulating MDD Relaxations for Combinatorial Optimization". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings*, pp. 20–35 (cit. on p. 24).

Biraud, F, E Blum, and J Ribes (1974). "On optimum synthetic linear arrays with application to radioastronomy". In: *IEEE Transactions on Antennas and Propagation* 22.1, pp. 108–109 (cit. on p. 82).

Bloom, Gary S. and Solomon W. Golomb (1977). "Applications of numbered undirected graphs". In: *Proceedings of the IEEE* 65.4, pp. 562–570 (cit. on p. 82).

Bosch, Robert and Michael A. Trick (2004). "Constraint Programming and Hybrid Formulations for Three Life Designs". In: *Annals OR* 130.1-4, pp. 41–56. DOI: 10.1023/B:ANOR.0000032569.86938.2f (cit. on p. 94).

Bryant, Randal E. (1985). "Symbolic manipulation of Boolean functions using a graphical representation". In: *Proceedings of the 22nd ACM/IEEE conference on Design automation, DAC 1985, Las Vegas, Nevada, USA, 1985.* Pp. 688–694. DOI: 10.1145/317825.317964 (cit. on pp. 12, 22).

Bryant, Randal E. (1992). "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams". In: *ACM Comput. Surv.* 24.3, pp. 293–318. DOI: 10.1145/136035.136043 (cit. on p. 22).

Chu, Geoffrey, Christian Schulte, and Peter J. Stuckey (2009). "Confidence-Based Work Stealing in Parallel Constraint Programming". In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, pp. 226–241. DOI: 10.1007/978-3-642-04244-7_20 (cit. on p. 108).

Codish, Michael, Alice Miller, Patrick Prosser, and Peter J. Stuckey (2019). "Constraints for symmetry breaking in graph representation". In: *Constraints* 24.1, pp. 1–24 (cit. on pp. 93, 118–120, 127).

Crawford, James M., Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy (1996). "Symmetry-Breaking Predicates for Search Problems". In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996.* Pp. 148–159 (cit. on p. 93).

Cruz-Filipe, Luís, João Marques-Silva, and Peter Schneider-Kamp (2017). "Efficient Certified Resolution Proof Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science, pp. 118–135. DOI: 10.1007/978-3-662-54577-5_7 (cit. on p. 10).

Dagum, Leonardo and Ramesh Menon (January 1998). "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1, pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313 (cit. on p. 104).

Deshpande, Neil, Erica Sponsler, and Nathaniel Weiss (2012). *Analysis of the Go runtime scheduler* (cit. on p. 104).

De Uña, Diego, Graeme Gange, Peter Schachte, and Peter J. Stuckey (2019). "Compiling CP subproblems to MDDs and d-DNNFs". In: *Constraints* 24.1, pp. 56–93. DOI: 10.1007/s10601-018-9297-2 (cit. on p. 23).

Dincbas, Mehmet, Helmut Simonis, and Pascal Van Hentenryck (1988). "Solving the Car-Sequencing Problem in Constraint Logic Programming". In: *8th European Conference on Artificial Intelligence, ECAI 1988, Munich, Germany, August 1-5, 1988, Proceedings*. Pp. 290–295 (cit. on pp. 51, 52).

Dollas, Apostolos, William T Rankin, and David McCracken (1998). "A new algorithm for Golomb ruler derivation and proof of the 19 mark ruler". In: *IEEE Transactions on Information Theory* 44.1, pp. 379–382 (cit. on p. 82).

Downey, Rodney G. and Michael R. Fellows (1995). "Fixed-Parameter Tractability and Completeness I: Basic Results". In: *SIAM J. Comput.* 24.4, pp. 873–921. DOI: 10.1137/S0097539792228228 (cit. on p. 11).

Dunning, Iain, Joey Huchette, and Miles Lubin (2017). "JuMP: A Modeling Language for Mathematical Optimization". In: *SIAM Review* 59.2, pp. 295–320. DOI: 10.1137/15M1020575 (cit. on p. 4).

Flener, Pierre, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh (2002). "Breaking Row and Column Symmetries in Matrix Models". In: *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, pp. 462–476. DOI: 10.1007/3-540-46135-3_31 (cit. on p. 93).

Ford, Lester R. and Delbert R. Fulkerson (2009). "Maximal flow through a network". In: *Classic papers in combinatorics*. Springer, pp. 243–248 (cit. on p. 72).

Frisch, Alan M., Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel (2007). "The Design of ESSENCE: A Constraint Language for Specifying

Combinatorial Problems". In: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pp. 80–87 (cit. on pp. 4, 13).

Frisch, Alan M., Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh (2002). "Global Constraints for Lexicographic Orderings". In: *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, pp. 93–108. DOI: 10.1007/3-540-46135-3_7 (cit. on p. 94).

Gagliardi, R, J Robbins, and Herbert Taylor (1987). "Acquisition sequences in ppm communications (corresp.)" In: *IEEE Transactions on Information Theory* 33.5, pp. 738–744 (cit. on p. 82).

Galinier, Philippe (2001). "A constraint-based appproach to the Golomb ruler problem". In: *Third International Workshop on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)* (cit. on p. 82).

Gardner, Martin (1970). "Mathematical games". In: *Scientific American* 222.6, pp. 132–140 (cit. on p. 93).

Garey, M. R. and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman. ISBN: 0-7167-1044-7 (cit. on p. 9).

Garey, M. R. and David S. Johnson (1981). "Approximation Algorithms for Bin Packing Problems: A Survey". In: *Analysis and Design of Algorithms in Combinatorial Optimization*. Ed. by G. Ausiello and M. Lucertini. Vienna: Springer Vienna, pp. 147–172. ISBN: 978-3-7091-2748-3. DOI: 10.1007/978-3-7091-2748-3_8 (cit. on p. 11).

Garnick, David K, YH Kwong, and Felix Lazebnik (1993). "Extremal graphs without three-cycles or four-cycles". In: *Journal of Graph Theory* 17.5, pp. 633–645 (cit. on pp. 119, 127).

Gendron, Bernard and Teodor Gabriel Crainic (1994). "Parallel Branch-and-Branch Algorithms: Survey and Synthesis". In: *Operations Research* 42.6, pp. 1042–1066. DOI: 10.1287/opre.42.6.1042 (cit. on p. 106).

Gent, Ian P. (1998). *Two results on car-sequencing problems* (cit. on p. 51).

Gent, Ian P., Warwick Harvey, and Tom Kelsey (2002). "Groups and Constraints: Symmetry Breaking during Search". In: *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, pp. 415–430. DOI: 10.1007/3-540-46135-3_28 (cit. on p. 93).

Gent, Ian P., Ian Miguel, and Peter Nightingale (2008). "Generalised arc consistency for the AllDifferent constraint: An empirical survey". In: *Artif. Intell.* 172.18, pp. 1973–2000. DOI: 10.1016/j.artint.2008.10.006 (cit. on p. 70).

Gent, Ian P., Karen E. Petrie, and Jean-Francois Puget (2006). "Symmetry in Constraint Programming". In: *Handbook of Constraint Programming*, pp. 329–376. DOI: 10.1016/S1574-6526(06)80014-3 (cit. on p. 91).

Giles, Katherine and Willem Jan van Hoeve (2016). "Solving a Supply-Delivery Scheduling Problem with Constraint Programming". In: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*. Ed. by Michel Rueher. Vol. 9892. Lecture Notes in Computer Science. Springer, pp. 602–617. ISBN: 978-3-319-44952-4. DOI: 10.1007/978-3-319-44953-1_38 (cit. on p. 2).

Glover, Fred W. (1990). "Tabu Search - Part II". In: *INFORMS J. Comput.* 2.1, pp. 4–32. DOI: 10.1287/ijoc.2.1.4 (cit. on p. 11).

Gocht, Stephan, Ciaran McCreesh, and Jakob Nordström (2020). "Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. Ed. by Christian Bessiere. ijcai.org, pp. 1134–1140. DOI: 10.24963/ijcai.2020/158 (cit. on p. 10).

Goldberg, Evguenii I. and Yakov Novikov (2003). "Verification of Proofs of Unsatisfiability for CNF Formulas". In: *2003 Design, Automation and Test in Europe Conference and Exposition, 3-7 March 2003, Munich, Germany*. IEEE Computer Society, pp. 10886–10891. DOI: 10.1109/DATE.2003.10008 (cit. on p. 10).

Goldreich, Oded (2010). *P, NP, and NP-Completeness: The basics of computational complexity*. Cambridge University Press (cit. on p. 9).

Gottlieb, Jens, Markus Puchta, and Christine Solnon (2003). "A Study of Greedy, Local Search, and Ant Colony Optimization Approaches for Car Sequencing Problems". In: *Applications of Evolutionary Computing, EvoWorkshop 2003: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, and EvoSTIM, Essex, UK, April 14-16, 2003, Proceedings*, pp. 246–257. DOI: 10.1007/3-540-36605-9_23 (cit. on p. 25).

Hadzic, Tarik and John N. Hooker (2006). *Postoptimality analysis for integer programming using binary decision diagrms*. Tech. rep. Carnegie Mellon University (cit. on p. 23).

Hadzic, Tarik and John N. Hooker (2007). "Cost-Bounded Binary Decision Diagrams for 0-1 Programming". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 4th International Conference, CPAIOR 2007, Brussels, Belgium, May 23-26, 2007, Proceedings*, pp. 84–98. DOI: 10.1007/978-3-540-72397-4_7 (cit. on p. 23).

Hadzic, Tarik, John N. Hooker, Barry O'Sullivan, and Peter Tiedemann (2008). "Approximate Compilation of Constraints into Multivalued Decision Diagrams". In: *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney,*

Australia, September 14-18, 2008. Proceedings, pp. 448–462. DOI: 10.1007/978-3-540-85958-1_30 (cit. on p. 24).

Haralick, Robert M. and Gordon L. Elliott (1980). "Increasing Tree Search Efficiency for Constraint Satisfaction Problems". In: *Artif. Intell.* 14.3, pp. 263–313. DOI: 10.1016/0004-3702(80)90051-X (cit. on pp. 6, 10).

Harvey, Warwick and Joachim Schimpf (2002). *Bounds Consistency Techniques For Long Linear Constraints* (cit. on p. 40).

Harvey, Warwick and Peter J. Stuckey (1998). "Constraint Representation for Propagation". In: *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, pp. 235–249. DOI: 10.1007/3-540-49481-2_18 (cit. on pp. 39, 44).

Harvey, Warwick and Thorsten Jan Winterer (2005). "Solving the MOLR and Social Golfers Problems". In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*. Ed. by Peter van Beek. Vol. 3709. Lecture Notes in Computer Science. Springer, pp. 286–300. ISBN: 3-540-29238-1. DOI: 10.1007/11564751_23 (cit. on p. 94).

Hayes, Brian (1998). "Computing science: Collective wisdom". In: *American Scientist* 86.2, pp. 118–122 (cit. on p. 82).

Heule, Marijn, Warren A. Jr. Hunt, and Nathan Wetzler (2013). "Verifying Refutations with Extended Resolution". In: *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. Ed. by Maria Paola Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer, pp. 345–359. DOI: 10.1007/978-3-642-38574-2_24 (cit. on p. 10).

Hoda, Samid, Willem Jan van Hoeve, and John N. Hooker (2010). "A Systematic Approach to MDD-Based Constraint Programming". In: *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, pp. 266–280. DOI: 10.1007/978-3-642-15396-9_23 (cit. on p. 24).

Hooker, John N. (2013). "Decision Diagrams and Dynamic Programming". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, pp. 94–110. DOI: 10.1007/978-3-642-38171-3_7 (cit. on p. 24).

Hurley, Barry and Barry O'Sullivan (2016). "Introduction to Combinatorial Optimisation in Numberjack". In: *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, pp. 3–24. DOI: 10.1007/978-3-319-50137-6_1 (cit. on p. 4).

Hutter, Frank, Holger H. Hoos, and Kevin Leyton-Brown (2011). "Sequential Model-Based Optimization for General Algorithm Configuration". In: *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, pp. 507–523. DOI: 10.1007/978-3-642-25566-3_40 (cit. on p. 132).

Järvisalo, Matti, Daniel Le Berre, Olivier Roussel, and Laurent Simon (2012). "The International SAT Solver Competitions". In: *AI Magazine* 33.1 (cit. on p. 108).

Jefferson, Christopher, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent, eds. (1999). *CSPLib: A problem library for constraints*. http://www.csplib.org (cit. on p. 13).

Kaiser, Hartmut, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey (2014). "HPX: A Task Based Programming Model in a Global Address Space". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, Eugene, OR, USA, October 6-10, 2014*, 6:1–6:11. DOI: 10.1145/2676870.2676883 (cit. on p. 104).

Kautz, Henry A., Yongshao Ruan, Dimitris Achlioptas, Carla P. Gomes, Bart Selman, and Mark E. Stickel (2001). "Balance and Filtering in Structured Satisfiable Problems". In: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pp. 351–358 (cit. on p. 69).

Kocuk, Burak and Willem Jan van Hoeve (2019). "A Computational Comparison of Optimization Methods for the Golomb Ruler Problem". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, pp. 409–425. DOI: 10.1007/978-3-030-19212-9_27 (cit. on p. 82).

Kozen, Dexter C. (1992). *Design and Analysis of Algorithms*. Texts and Monographs in Computer Science. Springer. ISBN: 978-3-540-97687-5. DOI: 10.1007/978-1-4612-4400-4 (cit. on p. 6).

Lagerkvist, Mikael Z. and Christian Schulte (2007). "Advisors for Incremental Propagation". In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pp. 409–422. DOI: 10.1007/978-3-540-74970-7_30 (cit. on p. 70).

Lai, Yung-Te, Massoud Pedram, and Sarma B. K. Vrudhula (1994). "EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition". In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 13.8, pp. 959–975. DOI: 10.1109/43.298033 (cit. on p. 23).

Larrosa, Javier and Gabriel Valiente (2002). "Constraint satisfaction algorithms for graph pattern matching". In: *Mathematical. Structures in Comp. Sci.* 12.4, pp. 403–422. ISSN: 0960-1295. DOI: http://dx.doi.org/10.1017/S0960129501003577 (cit. on p. 62).

Lee, C. Y. (July 1959). “Representation of switching circuits by binary-decision programs”. In: *The Bell System Technical Journal* 38.4, pp. 985–999. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1959.tb01585.x. (Cit. on p. 21).

Leiserson, Charles E. (2009). “The Cilk++ concurrency platform”. In: *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, pp. 522–527. DOI: 10.1145/1629911.1630048 (cit. on p. 104).

Malapert, Arnaud, Jean-Charles Régin, and Mohamed Rezgui (2016). “Embarrassingly Parallel Search in Constraint Programming”. In: *J. Artif. Intell. Res.* 57, pp. 421–464. DOI: 10.1613/jair.5247 (cit. on p. 108).

Mayer-Eichberger, Valentin and Toby Walsh (2013). “SAT Encodings for the Car Sequencing Problem”. In: *POS-13. Fourth Pragmatics of SAT workshop, a workshop of the SAT 2013 conference, July 7, 2013, Helsinki, Finland*, pp. 15–27 (cit. on p. 51).

McCreesh, Ciaran (2017). “Solving Hard Subgraph Problems in Parallel”. PhD thesis. University of Glasgow (cit. on p. 13).

McCreesh, Ciaran and Patrick Prosser (2015). “The Shape of the Search Tree for the Maximum Clique Problem and the Implications for Parallel Branch and Bound”. In: *TOPC* 2.1, 8:1–8:27. DOI: 10.1145/2742359 (cit. on p. 122).

McKay, Brendan D. and Adolfo Piperno (2014). “Practical graph isomorphism, II”. In: *J. Symb. Comput.* 60, pp. 94–112 (cit. on p. 117).

Meyerson, J. (September 2014). “The Go Programming Language”. In: *IEEE Software* 31.5, pp. 104–104. DOI: 10.1109/MS.2014.127 (cit. on p. 101).

Moisan, Thierry, Jonathan Gaudreault, and Claude-Guy Quimper (2013). “Parallel Discrepancy-Based Search”. In: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, pp. 30–46. DOI: 10.1007/978-3-642-40627-0_6 (cit. on p. 108).

Moore, Edward F. (1959). “The shortest path through a maze”. In: *Proc. Int. Symp. Switching Theory, 1959*, pp. 285–292 (cit. on p. 6).

Moore, Gordon E. (September 2006). “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” In: *IEEE Solid-State Circuits Society Newsletter* 11.3, pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860 (cit. on p. 103).

Nakahata, Yu, Jun Kawahara, and Shoji Kasahara (2018). “Enumerating Graph Partitions Without Too Small Connected Components Using Zero-suppressed Binary and Ternary Decision Diagrams”. In: *17th International Symposium on Experimental Algorithms, SEA 2018, June 27-29, 2018, L’Aquila, Italy*, 21:1–21:13. DOI: 10.4230/LIPIcs.SEA.2018.21 (cit. on p. 23).

Nanz, Sebastian, Scott West, and Kaue Soares da Silveira (2013). "Examining the Expert Gap in Parallel Programming". In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pp. 434–445. DOI: 10.1007/978-3-642-40047-6_45 (cit. on pp. 105, 108).

Nethercote, Nicholas, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack (2007a). "MiniZinc: Towards a Standard CP Modelling Language". In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pp. 529–543. DOI: 10.1007/978-3-540-74970-7_38 (cit. on p. 4).

Nethercote, Nicholas, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack (2007b). "MiniZinc: Towards a Standard CP Modelling Language". In: *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*. CP'07. Providence, RI, USA: Springer-Verlag, pp. 529–543. ISBN: 978-3-540-74969-1 (cit. on p. 13).

Nguyen, T. and Yves Deville (1998). "A Distributed Arc-Consistency Algorithm". In: *Sci. Comput. Program.* 30.1-2, pp. 227–250. DOI: 10.1016/S0167-6423(97)00012-9 (cit. on p. 106).

O'Neil, Ryan J. and Karla Hoffman (2018). *Exact Methods for Solving Traveling Salesman Problems with Pickup and Delivery in Real Time* (cit. on p. 2).

Perron, Laurent (1999). "Search Procedures and Parallelism in Constraint Programming". In: *Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, pp. 346–360. DOI: 10.1007/978-3-540-48085-3_25 (cit. on p. 108).

Pesant, Gilles, Claude-Guy Quimper, and Alessandro Zanarini (2014). "Counting-Based Search: Branching Heuristics for Constraint Satisfaction Problems". In: *CoRR* abs/1401.4601. arXiv: 1401.4601 (cit. on p. 69).

Promotional material, Gurobi (2019). *Ditributed Optimisations: Use multiple machines for maximum performance*. Tech. rep. (cit. on p. 107).

Prosser, Patrick (1995). *MAC-CBJ: maintaining arc consistency with conflict-directed backjumping*. Tech. rep. Department of Computer Science, University of Strathclyde (cit. on p. 7).

Prosser, Patrick, Chris Conway, and Claude Muller (1992). "A constraint maintenance system for the distributed resource allocation problem". In: *Intelligent Systems Engineering* 1.1, pp. 76–83 (cit. on p. 106).

Prud'homme, Charles, Jean-Guillaume Fages, and Xavier Lorca (2017). *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (cit. on p. 4).

Puget, Jean-Francois (1993). “On the Satisfiability of Symmetrical Constrained Satisfaction Problems”. In: *Methodologies for Intelligent Systems, 7th International Symposium, ISMIS '93, Trondheim, Norway, June 15-18, 1993, Proceedings*, pp. 350–361. DOI: 10.1007/3-540-56804-2_33 (cit. on p. 93).

Quimper, Claude-Guy and Toby Walsh (2005). “The All Different and Global Cardinality Constraints on Set, Multiset and Tuple Variables”. In: *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2005, Uppsala, Sweden, June 20-22, 2005, Revised Selected and Invited Papers*, pp. 1–13. DOI: 10.1007/11754602_1 (cit. on p. 70).

Régin, Jean-Charles (1994). “A Filtering Algorithm for Constraints of Difference in CSPs”. In: *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*. Pp. 362–367 (cit. on pp. 70, 72).

Régin, Jean-Charles and Jean-Francois Puget (1997). “A Filtering Algorithm for Global Sequencing Constraints”. In: *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, pp. 32–46. DOI: 10.1007/BFb0017428 (cit. on p. 51).

Régin, Jean-Charles, Mohamed Rezgui, and Arnaud Malapert (2014). “Improvement of the Embarrassingly Parallel Search for Data Centers”. In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. Ed. by Barry O'Sullivan. Vol. 8656. Lecture Notes in Computer Science. Springer, pp. 622–635. ISBN: 978-3-319-10427-0. DOI: 10.1007/978-3-319-10428-7_45 (cit. on p. 108).

Reinders, James (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc. (cit. on p. 104).

Schaerf, Andrea (1999). “Scheduling Sport Tournaments using Constraint Logic Programming”. In: *Constraints* 4.1, pp. 43–65. DOI: 10.1023/A:1009845710839 (cit. on p. 69).

Schröer, Olaf and Ingo Wegener (1998). “The Theory of Zero-Suppressed BDDs and the Number of Knight's Tours”. In: *Formal Methods in System Design* 13.3, pp. 235–253. DOI: 10.1023/A:1008681625346 (cit. on p. 23).

Selman, Bart, Hector J. Levesque, and David G. Mitchell (1992). “A New Method for Solving Hard Satisfiability Problems”. In: *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992*. Ed. by William R. Swartout. AAAI Press / The MIT Press, pp. 440–446 (cit. on p. 11).

Shannon, Claude Elwood (1940). “A symbolic analysis or relay and switching circuits.” PhD thesis. Massachusetts Institute of Technology, Dept. of Electrical Engineering (cit. on p. 21).

Siala, Mohamed, Emmanuel Hebrard, and Marie-José Huguet (2014). "An optimal arc consistency algorithm for a particular case of sequence constraint". In: *Constraints* 19.1, pp. 30–56. DOI: 10.1007/s10601-013-9150-6 (cit. on p. 51).

Siala, Mohamed, Emmanuel Hebrard, and Marie-José Huguet (2015). "A study of constraint programming heuristics for the car-sequencing problem". In: *Eng. Appl. of AI* 38, pp. 34–44. DOI: 10.1016/j.engappai.2014.10.009 (cit. on p. 51).

Smith, Barbara (n.d.). *CSPLib Problem 001: Car Sequencing*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. http://www.csplib.org/Problems/prob001 (cit. on p. 147).

Smith, Barbara M. (2002). "A Dual Graph Translation of a Problem in 'Life'". In: *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, pp. 402–414. DOI: 10.1007/3-540-46135-3_27 (cit. on p. 94).

Smith, Barbara M., Karen E. Petrie, and Ian P. Gent (2004). "Models and Symmetry Breaking for 'Peaceable Armies of Queens'". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, pp. 271–286. DOI: 10.1007/978-3-540-24664-0_19 (cit. on p. 94).

Solnon, Christine, Van-Dat Cung, Alain Nguyen, and Christian Artigues (2008). "The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem". In: *European Journal of Operational Research* 191.3, pp. 912–927. DOI: 10.1016/j.ejor.2007.04.033 (cit. on p. 51).

Stergiou, Kostas and Toby Walsh (1999). "The Difference All-Difference Makes". In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pp. 414–419 (cit. on p. 70).

Stuckey, Peter J. (2016). *Cell block*. URL: https://github.com/MiniZinc/minizinc-examples/tree/master/cell_block (visited on October 21, 2020) (cit. on pp. 84, 147).

Stuckey, Peter J., Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer (2014). "The MiniZinc Challenge 2008-2013". In: *AI Magazine* 35.2, pp. 55–60 (cit. on p. 4).

Sutter, Herb (2005). "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3 (cit. on p. 103).

Sutter, Herb and James R. Larus (2005). "Software and the concurrency revolution". In: *ACM Queue* 3.7, pp. 54–62. DOI: 10.1145/1095408.1095421 (cit. on p. 103).

Tarjan, Robert Endre (1972). "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2, pp. 146–160. DOI: 10.1137/0201010 (cit. on p. 72).

Van Beek, Peter (n.d.). *CSPLib Problem 006: Golomb rulers*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. http://www.csplib.org/Problems/prob006 (cit. on p. 147).

Van Hoeve, Willem Jan (2001). "The alldifferent Constraint: A Survey". In: *CoRR* cs.PL/0105015 (cit. on p. 70).

Van Laarhoven, Peter J. M. and Emile H. L. Aarts (1987). *Simulated Annealing: Theory and Applications*. Vol. 37. Mathematics and Its Applications. Springer. ISBN: 978-90-481-8438-5. DOI: 10.1007/978-94-015-7744-1 (cit. on p. 11).

Wegener, Ingo (2000). *Branching programs and binary decision diagrams: theory and applications*. Vol. 4. SIAM (cit. on p. 22).

Weidenbach, Christoph (2017). "Do Portfolio Solvers Harm?" In: *ARCADE 2017, 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements, Gothenburg, Sweden, 6th August 2017*, pp. 76–81 (cit. on p. 107).

Xu, Lin, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown (2011). "SATzilla: Portfolio-based Algorithm Selection for SAT". In: *CoRR* abs/1111.2249. arXiv: 1111.2249 (cit. on p. 107).

Yasuhara, M., Toshiyuki Miyamoto, K. Mori, S. Kitamura, and Y. Izui (2015). "Multi-objective embarrassingly parallel search for constraint programming". In: *2015 IEEE International Conference on Industrial Engineering and Engineering Management, IEEM 2015, Singapore, December 6-9, 2015*, pp. 853–857. DOI: 10.1109/IEEM.2015.7385769 (cit. on p. 108).

Appendix A

Problem Models

This appendix includes the models used for the problems solved in this thesis given as listings of Minizinc code. The code given in Listing A.1 (B. Smith, n.d.) and Listing A.2 (van Beek, n.d.) can be found in the CSPlib repository of problems. The code given in Listing A.3 can be found in the Minizinc Examples Github repository (Stuckey, 2016). The code given in Listing A.4 is my own.

Listing A.1: Car factory sequencing problem model

```
%
% Car sequencing in MiniZinc.
%
% This is based on the OPL3 model car.mod.
%
% Compare with the Comet model
% http://www.hakank.org/comet/car.co
%
%
% This MiniZinc model was created by Hakan Kjellerstrand, hakank@bonetmail.com
% See also my MiniZinc page: http://www.hakank.org/minizinc
%
% Model modified to match CSPLib data format by Chris Mears.
%
% Licenced under CC-BY-4.0 : http://creativecommons.org/licenses/by/4.0/
%
% include "globals.mzn";
int: numclasses;
int: numoptions;
int: numcars;
set of int: Classes = 1..numclasses;
set of int: Options = 1..numoptions;
set of int: Slots = 1..numcars;

array[Classes] of int: numberPerClass;

array[Classes,Options] of int: optionsRequired;
array[Options] of int: windowSize;
array[Options] of int: optMax;

array[Options] of int: optionNumberPerClass = [sum(j in Classes) (numberPerClass[j] * optionsRequired[j,i]) | i in Options];

% decision variables

array[Slots] of var Classes: slot;
array[Options, Slots] of var 0..1: setup;


var int: z = sum(s in Classes) (s*slot[s]);
```

```
solve minimize z;

output [
 "z: " ++ show(z) ++ "\n" ++
 "slot: " ++ show(slot) ++ "\n"
] ++
[
  if j = 1 then "\n" else " " endif ++
    show(setup[i,j])
  | i in Options, j in Slots
];
```

Listing A.2: Optimal Golomb rulers

```
%-----------------------------------------------------------------------------%
% Golomb rulers
% prob006 in csplib
%-----------------------------------------------------------------------------%
% From csplib:
% A Golomb ruler may be defined as a set of m integers 0 = a_1 < a_2 <
% ... < a_m such that the m(m-1)/2 differences a_j - a_i, 1 <= i < j
% <= m are distinct. Such a ruler is said to contain m marks and is of
% length a_m. The objective is to find optimal (minimum length) or
% near optimal rulers.
%
% This is the "ternary constraints and an alldifferent" model
%-----------------------------------------------------------------------------%

include "globals.mzn";

int: m;
int: n = m*m;

array[1..m] of var 0..n: mark;

array[1..(m*(m-1)) div 2] of var 0..n: differences =
    [ mark[j] - mark[i] | i in 1..m, j in i+1..m];

constraint mark[1] = 0;

constraint forall ( i in 1..m-1 ) ( mark[i] < mark[i+1] );

constraint alldifferent(differences);

    % Symmetry breaking
constraint differences[1] < differences[(m*(m-1)) div 2];

solve :: int_search(mark, input_order, indomain, complete)
    minimize mark[m];

output [show(mark)];

%-----------------------------------------------------------------------------%
%-----------------------------------------------------------------------------%
```

Listing A.3: Cell Block Allocation

```
int: k;
set of int: PRISONER = 1..k;
int: n;
set of int: ROW = 1..n;
int: m;
set of int: COL = 1..m;
set of PRISONER: danger;
set of PRISONER: female;
set of PRISONER: male = PRISONER diff female;
array[ROW,COL] of int: cost;


array[PRISONER] of var ROW: r;
array[PRISONER] of var COL: c;

%constraint forall(p1, p2 in PRISONER where p1 < p2)
%                (abs(r[p1] - r[p2]) + abs(c[p1] - c[p2]) > 0);
include "alldifferent.mzn";
```

```
constraint alldifferent([r[p] * m + c[p] | p in PRISONER]);

constraint forall(p in PRISONER, d in danger where p != d)
                ( abs(r[p] - r[d]) + abs(c[p] - c[d]) > 1 );

constraint forall(p in female)(r[p] <= (n + 1) div 2);
constraint forall(p in male)(r[p] >= n div 2 + 1);

var int: totalcost = sum(p in PRISONER)(cost[r[p],c[p]]);
solve minimize totalcost;

output ["cost = \(totalcost)\n"] ++
       [ "Prisoner \(p) " ++ if p in female then "(F)" else "(M)" endif ++
         " in [\(r[p]),\(c[p])].\n"
       | p in PRISONER ];
```

Listing A.4: Maximum Independent Set Problem

```
%
% Maximum Independent Set
%

int: n;
array[1..n,1..n] of 0..1: A; % adjacency
array[1..n] of int: cost;
array[1..n] of var 0..1: v; % v[i] = 1 <-> ith vertex is in max independent set

% declare constraints
constraint forall(i,j in 1..n where i<j)(if A[i,j] = 1 then v[i] + v[j] < 2 else true endif);

var int: setCost; % number of vertices in the max independent set
constraint setCost = sum (i in 1..n) (cost[i] * v[i]);

solve maximize setCost;
```