Partial Compilation of Constraint Problems

Andrea Balogh
120227786

Thesis submitted for the degree of Doctor of Philosophy

NATIONAL UNIVERSITY OF IRELAND, CORK

College of Science, Engineering and Food Science

School of Computer Science & Information Technology
Insight SFI Research Centre for Data Analytics

December 2024

Head of School: Professor Dirk Pesch

Supervisors: Professor Barry O'Sullivan
Dr. Guillaume Escamocher

Contents

List of Figures

List of Tables

I, Andrea Balogh, certify that this thesis is my own work and has not been submitted for another degree at University College Cork or elsewhere.

Andrea Balogh

Acknowledgements

The journey of producing this dissertation has been a special one. I learned a lot of new concepts, tools and techniques about Constraint Programming, Boolean Satisfiability and Knowledge Compilation, but also about myself, how to prioritise, manage time, expectations and patience.

First of all, huge thanks to my main supervisor Professor Barry O'Sullivan for supporting me through all these. Every time we talked, I was inspired, motivated and reassured. I am immensely grateful I got to learn so much from him about technical topics, but also about leadership. In my second year, I started working with my second supervisor, Dr. Guillaume Escamocher. I am grateful to Barry for setting this up. Thanks, Guillaume for all your feedback, patience and always being available for long technical discussions. I am also grateful to my examiners, Professor Ken Brown and Professor Ian Miguel, for their insightful comments and suggestions.

I am grateful for all the people I met along this journey. Thanks Padraigh and Azzam for your friendship. Thanks Begüm for involving me with Women in AI, with all your enthusiasm and support, you always reminded me of my self-confidence when I forgot about it. I feel fortunate to have met my fellow PhDs in the lab, sharing our experiences, knowledge, worries and lunchtime made this journey easier and more enjoyable. Thanks especially for this, Buvana, Vedat, Jheisson, Marlon, Sharmi and many more. Thanks Helmut, for all your insights during the Roadef challenge, learned a lot just by listening to your ideas. Thanks Chrys, for all the interesting conversations and for always speaking of me so highly. Also, thanks to the administrative staff Caitriona, Linda and Eleanor for their support and also to Peter for his technical support.

In my opinion, finding work-life balance is very important, and I would like to thank all my friends and family for reminding me of this while still supporting me when I had a deadline and just needed to prioritise work a bit. I am grateful for the support of my family, my mother Erika, my father Zoltán, my brother Tomi and my sister-in-law Liis. Especially thanks to my brother Tomi for showing interest in my research, even if zoning out very fast, your different perspective and laid-back attitude helped me when spiralling. Thanks to all my friends, especially Eme, our chats while going to work always put me in a good mood, giving me more energy for the day. Climbing has been one of my hobbies that always helped me unwind and recharge, thanks to all my fellow climbers and my instructor, Andy.

Last but not least, a big thanks to John, my partner, for everything he has done and endured for me. Your cheerfulness, support and our holidays helped me recharge when my batteries were running low.

This dissertation would not have been possible without the financial support of the Science Foundation Ireland, Grants 12/RC/2289-P2 and 16/RC/3918 which are co-funded under the European Regional Development Fund.

Abstract

Constraint Programming (CP) is a powerful paradigm for solving combinatorial problems, but due to a large search space, solving can be very time consuming. Diagnosis, planning, product configuration are example use-cases [XBZ+21]. These problems are often used in an online setting, where a user expresses queries with additional constraints that are, together with the same problem instance, used many times to answer queries related to model counting, satisfiability, equivalence, etc. Most such queries are NP-hard problems. Knowledge Compilation (KC) methods were developed to deal with the complexity of solving combinatorial problems online by creating a representation offline that is able to answer queries in time that is polynomial in the size of the representation [DM02].

Finding compact representations is often the bottleneck of compilation methods. Due to time or memory limits, compilation may timeout leaving the user without any compiled representation. One approach that is sometimes used is *partial compilation* whereby a subset of the solutions to the initial constraints are compiled, e.g. those that are considered most important or most likely to be useful. There are many use-cases where some solution loss is acceptable. For example, one might wish to compile a large subset of solutions to an embedded device where space is at a premium [OP06], or in a diagnosis setting, where storing the most likely solutions (diagnoses) might be sufficient.

In the CP and SAT (Boolean Satisfiability) communities, various methods have been implemented to reduce search time of solvers. Inspired by this, we investigate whether knowledge compilation benefits from these methods. Namely, we look at the effects of symmetry breaking and backbone relaxation. Symmetry breaking constraints can be added to a combinatorial problem to eliminate symmetries, in the expectation that this will reduce the number of states to be explored, both solutions and non-solutions, thereby speeding-up search. We explore whether breaking symmetries leads to a more compact representation, and if these can be compiled within less time and memory. The utility of symmetry breaking is that symmetric solutions can be reconstructed. We consider four compilers and four highly symmetrical problems to show the effect of symmetry breaking.

Another approach we explore is relaxing the definition of backbones, denoted as *selective backbone*. The selective backbone is a set of assignments that is part of a subset of solutions that optimises a scoring function. Such a scoring function could be maximising the model count of the compilation after assigning the

selective backbone, or the ratio between the model count and the compiled representation size. Large selective backbones allow for smaller representations but also eliminate more solutions. We compare different methods of computing the selective backbone that offers the best compromise. We propose four iterative heuristic types and a variety of scoring functions. Some depend on statistical information about variables, some approximate the model count and some calculate the true model count at each iteration. In order to deem our approach successful two criteria must be satisfied: that there is a significant reduction in size and that the important subset of solutions is kept. Since many subsets of solutions exist, representing the most preferred, most likely, most compact, or most important ones for the user is important. For this reason, we look at different ways to define the importance of a variable. In case of unweighted instances, all variables have the same importance, whereas for weighted instances weights represent a preference or probability. This shows that the most important subset is selected, not just the largest one. We conduct experiments on a variety of instances from the domains of circuit design, planning and a generated dataset difficult for model counting.

The iterative heuristics we propose to obtain selective backbones are monotonically decreasing the number of solutions represented. Because of this, identifying the number of iterations where the representation is compact enough is important. We propose a method that infers, from a set of small instances, the percentage of variables that need to be assigned for the representation to be compact This is especially important in case the initial problem does not compile, since instead of trying to compile at every iteration we can first assign all proposed variables and then compile.

Chapter 1

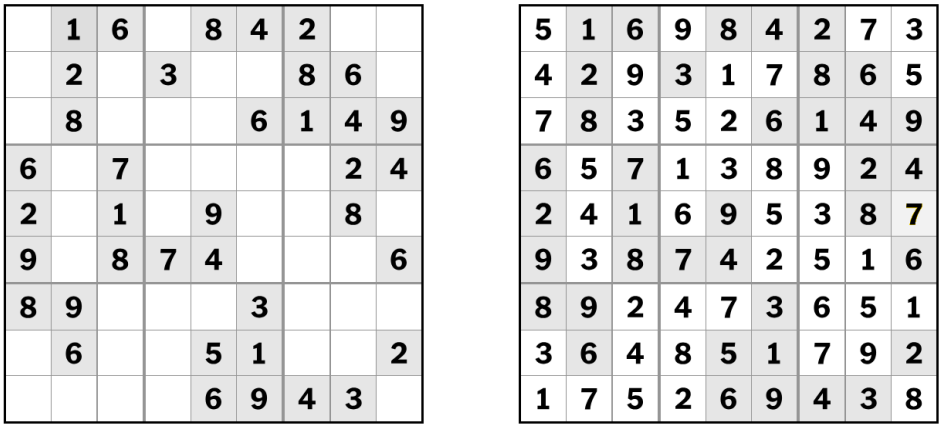
Introduction

1.1 Motivation

The field of Artificial Intelligence (AI) has rapidly evolved over the past few decades with both solving a large variety of problems and developing various techniques [RN20]. Artificial Intelligence is a big umbrella term for methods solving a variety of problems [RN20]. Most recent developments focus on data-driven approaches where models are learned from observing the real world. These are called machine learning approaches [GBC16]. The aim of them is to generalise and learn from the past. However, this relies on data being collected fairly and representing everything one needs to know about a particular problem. Also through the act of generalising these models act like black boxes, without the capability of explaining why certain decisions were made. Recent progress has been made in the field of explainable AI to deal with this. Other approaches focus on search-based methods [RN20], where the search space representing the solutions to the problem is explored. From here we focus on declarative methods where the user knows the rules and constraints that have to be satisfied in order to solve the problem at hand. These approaches are inherently easier to explain but can be very difficult to model as both domain knowledge and technical knowledge are needed for a well performing model. Another benefit of such approaches is that they are more transparent and explainable inherently [Dar23]. Since these systems are built on rules, understanding them is easier so recognising bias and faulty assumptions is easier. This is a very crude categorization of AI technologies, there is a lot of overlap between the two and other approaches that might not completely fit in either. Nevertheless from now on we focus on search-based solving, more particularly combinatorial problems.

The way we live life, make choices and interact with systems and the human artefacts around us all have an element of combinatorics. The abundance of materials, processes, methods and choices provides a combinatorial flavour to most problems we try to solve with AI [RvBW06]. For example, many services and products are highly customisable, and a lot of their components can be changed. Manufacturing a car, scheduling timetables, working shifts, and loading trucks with containers, all are examples of such problems. Since not all combinations of possible components are acceptable a decision model is formulated, defining what combinations are acceptable. This sets the scene perfectly for AI solutions dedicated to combinatorial problems. Such methods are Constraint Programming [RvBW06], Boolean Satisfiability [BHvMW09], Mixed Integer Programming [BT06], etc. The user models the problem at hand by expressing it in terms of variables with a set of domain values and constraints on these. An easy to digest example is the Sudoku puzzle (Figure 1.1). One can model Sudoku in terms of variables and constraints: each cell is a variable with possible values of digits 1-9 and constraints are the rules that each row, column and 3×3 square contains unique digits. One way to solve the Sudoku is by making assumptions and deductions on values of certain cells and looking at the implications of those. We do this until we find a solution or realise a previous decision was not correct, at which point we undo part of the solution and continue deducing the rest of the problem. For example, if we decide to put 3 at the empty cell in the top left corner, it is a valid decision, it satisfies all constraints. As a consequence of this decision, we can eliminate 3 from the possible digits for the cells that get affected by this. This is called propagation. We quickly notice that with this decision the top right 3×3 box has no valid cell for number 3. This means we have to revert this decision and all of its effects and make a different one. Similar algorithms have been explored in the field of constraint programming to efficiently search the space of all possible solutions to identify the ones that satisfy the given constraints.

Other real-world examples that use CP and SAT are diagnosis [Aze07, RG07], planning [LL22, BB19], scheduling [BLPN06, GSMT98], bioinformatics [Gas01], time tabling [VH03], product configuration [XBZ+21] and nurse rostering [RAL17]. Using these methods two important factors will determine how fast and scalable a model is: the solver used and the way the problem is modelled. Throughout the years a significant effort has been made in the area of well-performing solvers and advanced modelling languages and techniques [Stu10, NSB+07, RvBW06]. Various solvers have been implemented and depending on the problem different



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

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

Figure 1.1: Example of a Sudoku and its solution from `https://www.nytimes.com/puzzles/sudoku/easy` on 21 Oct 2024

solvers might work better. For example, the support of specific global constraints or specific domain types or the flexibility to include custom propagators might influence the choice of the solver [RvBW06]. Since 2002 SAT competitions [1] have been organized to compare different SAT solvers and since 2008 the MiniZinc challenge focuses on CP solvers [2]. Modelling a real word problem efficiently is a form of art, finding the best way to define variables and the relations between their domain values can have many formulations. Creativity and deep understanding of the problem as well as modelling languages are needed for a good model [RvBW06].

For example, in case of the N-queens problem, a different approach to what a variable represents can speed up solving [GPP06]. The N-queens problem requires the placement of N queens on an $n \times n$ chess board such that no queen is under attack. We can model this such that each variable represents a cell on the chessboard with values "queen"/"no queen", and will yield in $n \times n$ variables and constraints between them. Alternatively, if we denote each row on the chessboard by a variable where the value assigned is from the set $\{1, \ldots, n\}$ then the assigned value represents the position of the queen to be placed on that row. This brings the number of variables from the previous $n \times n$ to only n, with also fewer constraints. From the problem description one can deduce that there has to be one and only one queen on each row and each column. This observation leads to the decision that instead of having $n \times n$ variables and n constraints for each row forbidding more than one variable on a row to have the value "queen", we can

[1] `https://satcompetition.github.io/`
[2] `https://www.minizinc.org/challenge`

have n variables with domain values $\{1, \ldots, n\}$ that basically replace the previous constraints. This is just a small simple example of how a deep understanding of the problem and creativity of assigning meaning to variables hugely impact the model size and with that potentially the solving time too.

Knowledge Compilation (KC) methods were developed to deal with the complexity of solving combinatorial problems online by creating a representation offline that is able to answer queries online in time that is polynomial in the size of the representation [DM02]. These problems are often used in an online setting, where the same problem instance with added constraints by the user is used many times to answer queries related to model counting, satisfiability, equivalence, etc. Knowledge compilation has been used for compiling feature models to representations supporting polytime queries [SKH+24]. These are models used to describe the feasible combinations of features describing a product. Configuration problems can also be expressed as feature models. A product is composed of a set of features, where not all combinations are feasible. A feature model is built from a set of features and a set of constraints limiting the valid configurations. Previously they were solved by translating them to SAT problems, but in [SKH+24] they explored various possible representations to use.

An example query used within knowledge compilation is model counting, which in the domain of product configuration would be counting the number of all possible configurations of a product or counting the number of configurations consistent with a user's set of choices. Most such queries are NP-hard problems [DM02]. The field of knowledge compilation explores various representations, analysing their size and time complexity with respect to different queries and transformations.

A major advance in the field occurred with the introduction of richer, nested representations, as they perform better than the flat ones such as CNF and DNF [DM02]. The representations are based on Negational Normal Forms (NNF), a directed acyclic graph with OR and AND nodes. Paths in the graph denote solutions to the problem. One of the first and most widely used representation is the Binary Decision Diagram (BDD) [Bry92], where nodes represent the decision for each variable and edges the chosen assignment. An extension of this is Sentential Decision Diagrams (SDD) [Dar11].

The usefulness of such representation has been shown across various fields such as model-based diagnosis [HD05, MDM14], explainable machine learning [SDC19, DH23], probabilistic inference [SCD19] and planning [PBDG05, Hua06].

The choice of which representation to compile to is similar to the choice of the data structure used in a software system: the operations one will perform the most on that structure will drive the selection. Some of the most important queries and transformations of compiled representations are [DM02]: consistency/validity check, clausal entailment check, implicant check, equivalence check, sentential entailment check, model counting, conditioning, forgetting, conjunction, disjunction and negation. Users typically interact with these models by defining extra constraints that are used to query the compiled model.

In the context of the car configuration problem, we define the possible cars by the set of constraints P. For example, model counting can be used to define how many different types of cars can be manufactured given P. Consistency checking is used to check if there is at least one car configuration that satisfies all constraints in P. Clausal entailment is used to check if a clause is implied by the constraints of P. For example, if P describes a problem where if the car has a sunroof, it has the luxury package and the luxury package includes leather seats, then P entails the clause that the customer has the luxury package or has no sunroof. An example transformation is conditioning, where, for example, the user defines that they only want cars with an automatic transmission. This will restrict P to show only configurations with automatic transmission. Forgetting is the transformation that simplifies the formula in case the user does not particularly care about the value of a feature. Say if a user is indifferent to having leather seats, the occurrence of this is removed from the constraints while still keeping the rest of the problem unchanged.

In Section 2.3 we introduce in detail some of the compiled representations and mention which queries they support in time polynomial in their size. Finding compact representations is often the bottleneck of compilation methods [LM21]. There has been a lot of effort in speeding up the compilation methods [LM21]. In many applications, such as on-board diagnosis or reasoning on embedded processors, large problems might still fail to compile leaving the user without any practical representation of the problem. One approach that is sometimes used is *approximate* or *partial* compilation whereby a subset of the solutions of a set of constraints are compiled, e.g. those that are considered most important or most likely to be useful [OP06]. Others have considered partial compiled representations that are generated from a search-tree over which a variable ordering is defined, and removing domain values that would lead to dead-ends, thereby also removing solutions [BCFR08]. This is referred to as backtrack-free representation. There are many use-cases where some solution loss is acceptable. For

example, in case of embedded devices where space is limited, it might be sufficient to store only a subset of the solutions [OP06]. Similarly in a diagnosis setting, storing the most likely solutions (diagnoses) might be sufficient, since in most cases these would be most useful ones. Since the size of the compilation impacts performance it motivates the need for the partial compilations to have a small size. Another crucial criteria is the usefulness of the representation, that is, what information it retains from the initial problem.

Queries and transformations are executed in time polynomial in the representation size. Thus, the assumption is that if partial compilation represents a smaller solution set, then the compilation would be smaller too, leading to faster answers to queries. The answers to queries and transformations change in the context of partial compilation according to the partial information kept. In the dissertation, we focus on partial compilation in a way that restricts the initial solution set. That means answers using the partial compilations will also be a subset of the initial compilation answers. For example, answers to model counting given by partial compilation will be an underestimate of the real count. Answers to queries with partial compilation remain sound but not complete. Let us consider a scenario where a device with limited space is using a partial compilation for fault diagnostics. Ideally, the partial compilation would contain the most common diagnosis. A diagnosis found by a partial compilation of an issue to a specific fault would also be valid with respect to the full compilation. If no diagnosis is kept as a valid solution in the partial compilation, it will not give a misdiagnosis, but might fail to find one. In this case, the user could turn to a higher performing system for further investigation.

In this dissertation, we explore various ways to obtain partial compilations. Namely, we look at whether models where we break symmetries have a more compact representation, and if these can be compiled within less time and memory. Another approach we explore is relaxing the definition of backbones, denoted as *selective backbone*. Both of these approaches change the problem by adding additional constraints. These methods are compiler agnostic but the behaviour of different compilers might still react differently as their representations are also different. Representations are canonical with respect to the variable ordering used, and this can have a huge impact on the size [NW07]. In the following chapters we explore the use of different compilers, not for the reason of comparing them, but to show that our approach can be applied more broadly. The choice of the tool and what representation to compile to should be based on what queries and transformations the use cases require.

We define two criteria to measure the success of a partial compilation. Firstly, that there is a significant reduction in size and secondly that the important subset of solutions is kept. Both are equally important for the partial compilation to be useful. Since many subsets of solutions exist, representing the right one is important. For different use cases different criteria might be most important such as the most preferred, or most likely, or most compact, or most important subset. For this reason, in this dissertation we explore various techniques for selecting the subset of solutions to be compiled. We looked at different ways to define the importance of a variable. In case of unweighted instances, all variables have the same importance, whereas for weighted instances weights of a variable represent a preference or probability of a value assignment. In case of unweighted instances we do not make a difference between solutions, thus we look for the largest subset of solutions. However, in case of weighted instances, each solution can have a different weight, so a partial compilation should represent the most important solutions not just the largest subset. In [Dar20a] the authors point out the connection between compilation methods and representations and how these can be used in the context of probabilistic reasoning.

1.2 Thesis Statement

The thesis defended in this dissertation is:

> "It is possible to develop techniques that can selectively compile combinatorial problems in a way that ensures a more compact representation can be achieved while still maintaining a large subset of the most important solutions."

The approaches we propose to defend the thesis statement are changing the model to be compiled, not the tools or the representations they produce.

Symmetry breaking is a common approach used in constraint programming to eliminate some solutions/non-solutions to speed up search. Due to the modelling choices or to the nature of the problems we are solving, symmetrical solutions/non-solutions exist. In case of the N-queens problem, where one has to place N queens on a chessboard without them attacking each other, if one rotates the board 90 degrees, a symmetrical solution is obtained. Therefore eliminating symmetrical solutions and keeping only one solution per symmetry group, leads to a smaller solution set without much loss of information. Following this concept, we showed that we can achieve more compact representation for four

different compilers by eliminating symmetrical solutions.

However, not all problems have symmetries and formulating symmetry-breaking constraints by hand for problems with symmetries can be very difficult, so we proposed an approach that relies on another technique used to simplify constraint solving: backbone relaxation. Backbones are variable assignments that are part of all solutions. Assigning backbones and propagating their effect will not change the set of solutions, but will simplify the problem. We relaxed the definition of backbone, to a selective backbone, a set of assignments that are present in a subset of solutions that optimises a scoring function. We propose a few iterative heuristics that extend the selective backbone at each iteration given a criterion they optimize. We empirically demonstrate that the selective backbone obtains compact compilations with most of the solutions still present, on a large set of CNF instances.

The first heuristics we proposed used expensive scoring functions to decide on the next assignment to extend the selective backbone. This was a bottleneck for scalability, so we improved on this by proposing faster scoring functions. In order to show the power of the selective backbones, we also evaluate it on a set of weighted constraint satisfaction problems. In this problem each literal has a weight, the weight of a solution is the product of the literal weights and the weighted model count is the sum of the weights of the solutions. By integrating the literal weights in the scoring function we show that the selective backbone not only selects the largest subset of solutions to represent but the most important one, the one with the highest weighted model count.

The heuristic functions we describe are iterative, in each iteration they restrict the problem more by adding a variable assignment. Because of this, it is a difficult task to identify how many and which variables need to be fixed in order to obtain the best compromise between the solutions represented and the size of the compilation. To overcome this we look at a method to identify the size of the selective backbone by the percentage of variables assigned given a smaller dataset.

1.3 Contributions

This dissertation presents a set of approaches to obtain compact partial compilations, each of them supported by a publication.

The first approach is presented in the paper:

> *Andrea Balogh and Barry O'Sullivan: Breaking symmetry for knowledge compilation* in Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, 2023.

In this paper we investigate if breaking symmetries always lead to a smaller compiled representation. The utility of symmetry breaking is that, in most cases, symmetric solutions can be reconstructed. Symmetry breaking constraints can be added to a combinatorial problem to eliminate symmetries, in the expectation that this will reduce the number of states to be explored, both solutions and non-solutions, thereby speeding up search. One approach that is sometimes used is *partial compilation* whereby a subset of the solutions of a set of constraints are compiled, e.g. those that are considered most important or most likely to be useful. As use cases, we consider some simple CSPs such as a clique of not-equal constraints, N-queens, Balanced Incomplete Block Designs, and Graceful Graph Labelling. We introduced symmetry breaking constraints to these models and compiled them using four different tools, two top-down compilers and two bottom-up compilers. To the best of our knowledge, no such study was conducted before. There has been some research in the area of combining symmetry breaking and compilation but with a different focus. In general, the top-down compilers reduce the compilation size more when breaking symmetry and they also compile faster. However, comparing the tools is not the main point, rather we focus on compilations with and without symmetry breaking. We considered 40 instances in total and 4 compilers. In general, the partial compilations are smaller than the initial ones, and in some cases, when initial compilation fails, compilation after removing symmetries succeeds. The first tool managed to obtain partial compilation for 3 out of 6 instances when initial compilation failed. Similarly, the second for 1 out of 18, the third 4 out of 5 and the last compiled all 6 times the initial compilation failed. In these highly symmetrical problems compilation size almost always reduces, but this might not be the case for problems with less symmetry. For the clique of not-equals compilation size reduction is proportional to solution reduction, whereas for N-queens more varied gaps exist for the different compilers.

The second supporting publication explores another approach, expanding the definition of backbones:

> *Andrea Balogh, Guillaume Escamocher and Barry O'Sullivan: Partial compilation of SAT using selective backbones*, in Proceedings of European Conference on Artificial Intelligence, ECAI 2023.

The goal of this paper is to significantly decrease the compiled representation size of a given Boolean instance with a large representation, while preserving as much information about the instance as possible. We achieve this by assigning values to a subset of variables in such a way that the resulting instance has a much smaller representation than the original one, and its number of solutions is almost as high as the starting one. We call the set of variable assignments that we make the selective backbone of the solutions that we keep. Large selective backbones allow for smaller representations, but also eliminate more solutions. We compare different methods of computing the selective backbone that offers the best compromise. In this paper we focus on Binary Decision Diagrams (BDDs) as the compiled representation and a set of generated instances, the ISCAS suite and some planning benchmarks for experiments [3]. We empirically demonstrate that for many Boolean instances, belonging to various problem types, it is possible to obtain partial compilations that are far more space-efficient than the full one in terms of the number of solutions that are preserved. For 78.57% of the instances we looked at, we obtained at least one compilation that is more compact than the initial one.

The third work supporting this thesis expands the idea of the *selective backbone* to weighted constraint satisfaction problems as well as proposes new heuristics to obtain the selective backbone and a way to identify the best selective backbone size:

> *Andrea Balogh, Guillaume Escamocher and Barry O'Sullivan: Determining the Most Promising Selective Backbone Size for Partial Knowledge Compilation, CPAIOR 2025*

While the previous approach assumes that all models are equally useful to the user we now consider a more general setting, one where the user has preferences on the models. Furthermore, the intermediate steps that the previous method takes to create a small compiled instance require making a number of computations that are expensive in both time and space, including completely compiling the full initial instance, which might not be achievable for large instances. Even with a more efficient metric used for the intermediate steps when it is a good idea to stop the process of identifying the selective backbone remains a question. The work in this paper consists in addressing these three issues. First, the previous work is extended to take into account the weight preferences on the models. This means that when given the choice between two sets of models, and assuming that

[3] `https://www.cril.univ-artois.fr/KC/benchmarks.html`

they have the same effect on the representation size, we prefer choosing the set with the higher weighted model count, regardless of its cardinality. Then we look at how statistical properties of literals can be used as heuristics to pick which models to remove, and show that some of these offer a way to find a decent compromise between the quantity of knowledge lost and the amount of space saved, without the need for any compilation other than the compact final one. We show this on the same dataset as before with uniformly generated weights and the use of the D4 compiler. We have compared eight methods to select models to eliminate from a weighted CNF instance such that the representation size of the resulting instance is as small as possible, while the weighted model count remains as high as possible. Two of these methods are based on the approaches from the previous contribution about the unweighted version of the problem and are computationally expensive because they require a large number of instance compilations. One method is a version of random variable selection. The other five methods are novel to this paper and only rely on instance properties that can be computed quickly, so they can return a set of models to remove in a much briefer time than compilation-based heuristics.

Two findings stand out in particular within this work. First, it turns out that the selective backbones that characterise the sets of models to keep will lead to drastically different results depending on whether they are applied to the weighted or unweighted instances. Indeed, we have shown that the same selective backbone can result in a weighted model count close to the original one, while discarding most of the models. This shows that our approaches successfully take into account the fact that some models are worth more than others. The second main discovery that we have made is that some approaches that are simple in nature can still achieve decent results. Two heuristics manage to improve the targeted ratio (weighted model count divided by representation size) five to almost twenty-fold, depending on the type of instances considered. This shows that a compact representation of a weighted CNF instance can be obtained with minimal information loss without any kind of expensive computation. Lastly, we look at identifying the best selective backbone size given a smaller dataset. We explore the quality of the selective backbones for all percentages of variable assignments. We use this information to identify the best size of selective backbone and evaluate the performance on a larger dataset. The results show that 60% of the compilations are compact on a larger dataset at the indicated iteration.

1.4 Outline

The remainder of this dissertation is organised as follows. Chapter 2 presents the necessary background information needed. The main purpose of this chapter is to define the concepts and terminology used in this dissertation and give examples to familiarize the reader more with the theoretical aspects. Chapter 3 is the first novel technical chapter. Here we present the idea of introducing symmetry breaking as a way to restrict the represented solution set and obtain a partial compilation. In Chapter 4 we explore another approach to restrict the solution set. We propose a set of heuristic functions to identify parts of the problem to fix and to compile the rest to a partial compilation. We call this selective backbone. In Chapter 5 we build on the idea of selective backbone and apply it to weighted problems as well as define improved heuristics for them. Since the heuristics are iterative an important factor is the number of iterations one should perform in order to obtain a good partial compilation. We propose a simple approach to predict this and evaluate it. At last, Chapter 6 gives some concluding remarks and proposes some future directions.

Chapter 2

Background

Abstract. *Before diving into the technical details a short introduction to the related fields and topics is given. The two main topics to know in order to understand the rest of this dissertation are combinatorial problems and knowledge compilation. We introduce the notation and terms used for Constraint Satisfaction and Boolean Satisfiability. Within these fields, we highlight symmetry breaking and backbones. We also describe model counting approaches as this is used as a measure for the evaluation of a partial compilation. We give a short introduction to Knowledge Compilation and list some of the representations we use later on, including examples of them.*

In this chapter, we introduce notation and definitions for various concepts used later in the dissertation. In this dissertation, we focus on combinatorial problems and more specifically on scenarios where these systems are used in an online fashion and when the problems get too large. In Section 2.1 we introduce methods used to solve combinatorial problems. We discuss declarative methods, where problems are expressed in terms of variables and relationships between the values they can simultaneously have. Dedicated languages are used to model these problems. Dedicated solvers have been developed to solve such combinatorial problems [GPP06]. We look at Constraint Satisfaction Problems in Section 2.1.1 and at Boolean Satisfiability in Section 2.1.2. In Section 2.1.4 and Section 2.1.5 we discuss two important topics within the field of Constraint Programming (CP) that we used to obtain compact partial compilations. We then discuss an important use case of combinatorial problems, the act of counting how many solutions exist to a problem, named model counting. In Section 2.2 we explore the different

methods used to perform model counting, exact and approximate methods as well as based on search and on knowledge compilation techniques. In Section 2.3 we give an introduction to the field of Knowledge Compilation (KC) and describe some of the compiled representations we use later on in this dissertation.

2.1 Combinatorial Problem Solving

2.1.1 Constraint Satisfaction

Constraint Programming (CP) is a declarative paradigm, mostly used to solve combinatorial problems [RvBW06]. The user states the problem in terms of variables, possible values for them and constraints. A dedicated solver created to efficiently search the space of all possible solutions is used to solve the problem. We formally define a Constraint Satisfaction Problem (CSP) in Definition 2.1.

Definition 2.1 (Constraint Satisfaction Problem (CSP))**.** *A CSP is defined as a triple* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *, where* $\mathcal{X}$ *is a set of variables,* $\{X_1, \ldots, X_n\}$ *,* $\mathcal{D}$ *is a set of domains,* $\{D_1, \ldots, D_n\}$*, one for each variable, denoting the possible values they can take and* $\mathcal{C}$ *is a set of constraints that specify allowable combinations of values.*

Example 2.1. *We define the CSP illustrated in Figure 2.1 on* $\mathcal{X} = \{X_1, X_2, X_3, X_4, X_5\}$*,* $\mathcal{D} = \{D_1 = D_2 = \{a, b, c\}, D_3 = D_4 = D_5 = \{a, b\}\}$ *and the constraints* $X_1 > X_3$*,* $X_2 > X_3$*,* $X_4 > X_3$*,* $X_5 > X_3$ *and* $X_4 > X_5$*.*

The term problem class describes the general structure of the problem, whereas problem instances represent a problem in more detail, an actual instantiation of the problem. For example, a classic CSP class is the Sudoku problem, where each cell is denoted by a variable, with domains being numbers 1-9. The constraints of this problem are the rules of the puzzle, that is that every row, column, and 3×3 square must have different values. The instantiation of this problem is defined by the initial cells that are fixed.

Definition 2.2 (Variable Assignment [GPP06])**.** *A variable assignment is a pair* (X_i, a) *where* $X_i \in \mathcal{X}$ *is a variable and* $a \in D_i$ *is a value assigned to the variable. We also denote this by* $X_i = a$*.*

Definition 2.3 (Solution [GPP06])**.** *Let* $\mathcal{A}$ *be a set of variable assignments* $\mathcal{A} = \{(X_1, d_j), \ldots, (X_i, d_k)\}$*. In case* i *is equal to the number of variables in* $\mathcal{X}$ *and all the constraints are satisfied by* $\mathcal{A}$*, then the set of assignments* $\mathcal{A}$ *is a solution to the CSP. If* $i > 0$ *but* $i < |\mathcal{X}|$ *the set of assignments* $\mathcal{A}$ *is a* ***partial solution****.*

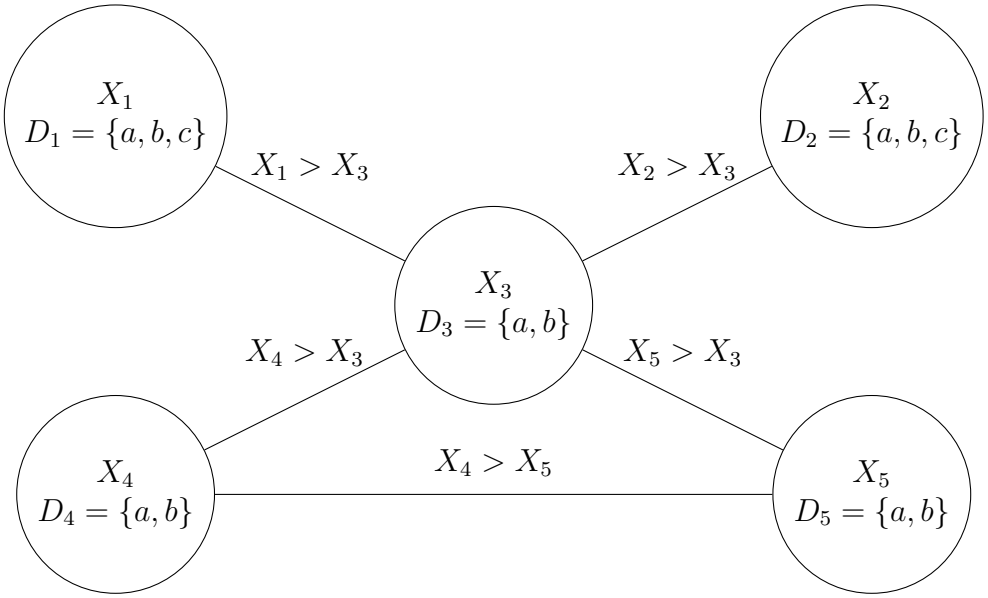


Figure 2.1: Constraint network of Example 2.1 as defined in [Mac77]

Constraints can be extensionally or intensionally defined. Intensional constraints are usually expressed in a mathematical formula, such as in Example 2.1. Extensional constraints are expressed as a table, with columns denoting variables and each row expresses a valid assignment. The number of variables participating in a constraint defines the arity of the constraint.

The *constraint network* is a graphical representation of a CSP. Nodes represent variables and arcs denote constraints. In case of non-binary constraints, the arcs are represented with a hyper-arc. The constraint network of Example 2.1 is shown in Figure 2.1.

In order to solve a CSP dedicated solvers have been developed such as OR-Tools [PF] Choco [PF22] etc. The main techniques implemented in solvers are inference and search. Throughout solving a search tree is created that is used to track the solving process. Most CP solvers offer complete search, it is guaranteed to find all solutions to the problem or prove the problem is infeasible. A CSP is split into smaller CSPs by assigning a value to a variable. Each of these steps is represented in the search tree, the root represents the initial CSP and the subsequent nodes represent the intermediate steps. If the domains are finite sets, the solutions satisfying the constraints could be found by enumerating the combination of all the values in the domains. But even for small instances, this is a huge search space to explore.

Inference

Inference techniques, such as constraint propagation and domain filtering achieve a defined level of consistency that reduces the possible variable-value combinations needed to try [BLPN06]. *Domain filtering* is the process that given a constraint C the domain values of the variables of C that are not consistent with C are removed. For example given a constraint $C : X \geq 3$ and the variable X with domain $D_X = \{0, 1, 2, 3, 4, 5, 6\}$ the values $\{0, 1, 2\}$ would be removed during domain filtering since they do not satisfy the constraint C. This change in the domains then will be *propagated* to other constraints, making them consistent too.

There are different levels of consistency. Node consistency is satisfied when all domain values satisfy unary constraints. Arc consistency is defined on binary constraints. There are two arcs defined between variable X_i and X_j using constraint C, (X_i, X_j) and (X_j, X_i). The arc (X_i, X_j) is consistent if for every domain value for variable X_i there is a corresponding value in D_j satisfying C. Generalised arc consistency extends this to non-binary constraints. Path consistency ensures that given two variables X_i, X_j, any tuple from $D_i \times D_j$ that satisfies binary constraints can be extended to a third variable.

We show the different types of consistencies using Example 2.1 introduced in [Mac77]. For example, arc consistency is violated at arc (X_2, X_3), since value $a \in D_2$ has no corresponding value in D_3 such that $X_2 > X_3$. This inconsistency is the root cause of several subtrees failing rooted at: $\{X_1 = a, X_2 = a\}$, $\{X_1 = b, X_2 = a\}$, and $\{X_1 = c, X_2 = a\}$. If we would backtrack when such an inconsistency is found the search tree would be much smaller. The example of path consistency is visible on the path (X_3, X_4, X_5). The values $a \in D_3$ and $b \in D_4$ are consistent with unary and the binary constraints since $\{X_3 = a, X_5 = b\}$ is valid as well as $\{X_4 = b, X_5 = a\}$ but there is no value in D_5 that would satisfy both constraints simultaneously.

Search

Search is usually implemented as *backtracking* search. In general, backtracking consists of sequentially instantiating variables with values from their domains. When an inconsistent state is achieved the last assignment is reverted and the next available is tried. The variable ordering heuristic will dictate the order of the variables to assign, and value ordering heuristics indicate the order of the values in the domain of each variable. This type of exploration will construct a

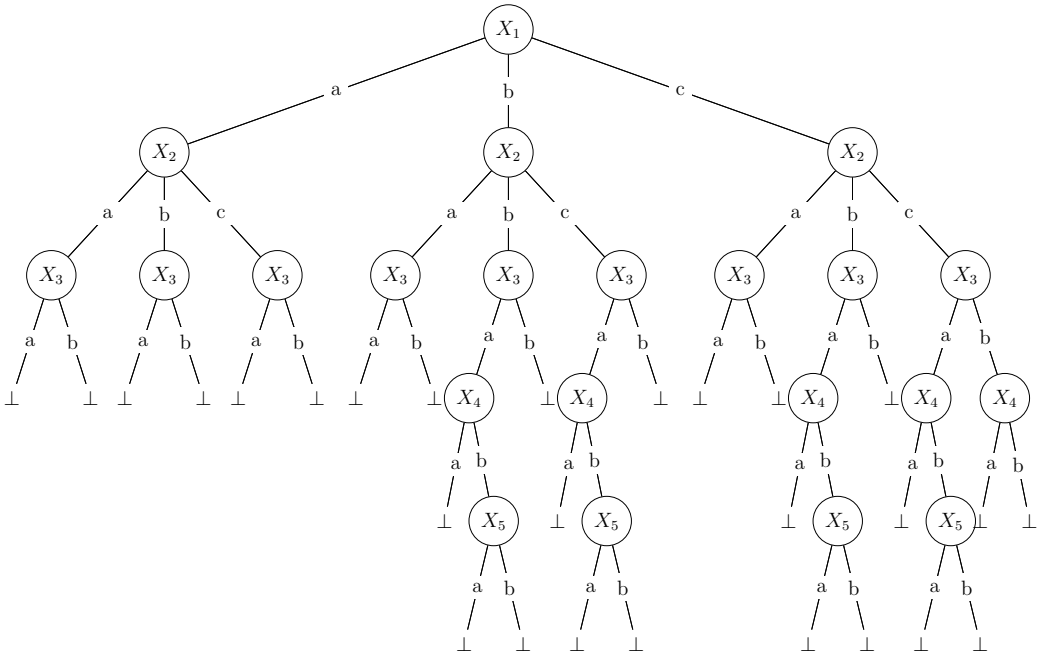


Figure 2.2: Search tree for Example 2.1

search tree. Figure 2.2 shows the search tree of Example 2.1. Here a branch is stopped when the assignment of the last variable does not satisfy some constraint. Using the concept of *consistency*, branches could be stopped earlier. When each variable has been assigned a solution is reached, and the search backtracks to explore all solutions.

2.1.2 Boolean Satisfiability

A propositional formula is defined over a finite set of boolean variables $\mathcal{X}$. The Boolean Satisfiability Problem can be regarded as a special case of CSP, such that all variables can have only values: *true* or *false*. The Boolean Satisfiability problem was the first to be proven NP-Complete [Coo71], and has since been one of the most popular constraint problems to be studied. An expression on Boolean variables and Boolean operators (AND/conjunction, OR/disjunction, XOR, etc.) is called a propositional logic formula or Boolean expression. A *clause* is an expression that consists of only disjunctions of *literals*. A literal l is either the Boolean variable l or its negation $\neg l$. Boolean Satisfiability consists of deciding whether a propositional logic formula can be satisfied given suitable value assignments to the variables of the formula.

Most SAT solvers require the problem to be defined in Conjunctive Normal Form (CNF), that is conjunctions of clauses each of which is a disjunction of literals or

their negation.

Definition 2.4 (CNF Representation)**.** *A SAT instance in CNF is composed of:*

- *A set of n Boolean variables denoted by $\mathcal{X}$.*
- *A conjunction of m clauses. Each clause is in turn composed of a disjunction of literals, where a literal can either be a variable or its negation.*

A clause with no literals, the empty clause is inconsistent. A clause with a single literal is a unit clause.

Example 2.2. *Below we present a formula, consisting of five clauses and on the right of this an example solution. Highlighting in bold we show which literal makes each clause satisfied, thus making the whole formula satisfied, providing a solution to the formula.*

A formula:	*A solution:* $\neg a, b, \neg c, \neg d$
$(a \vee \neg d)$	$(a \vee \neg \mathbf{d})$
$(\neg b \vee \neg c \vee \neg d)$	$(\neg b \vee \neg c \vee \neg \mathbf{d})$
$(a \vee b \vee d)$	$(a \vee \mathbf{b} \vee d)$
$(\neg a \vee \neg b)$	$(\neg \mathbf{a} \vee \neg b)$
$(\neg b \vee \neg c \vee d)$	$(\neg b \vee \neg \mathbf{c} \vee d)$

An *assignment* is a mapping from Boolean variables to $\{true, false\}$.

Definition 2.5 (Partial Assignment)**.** *Let I be a SAT instance and let B be a set of p variables of I. A set of p assignments, one to each variable of B, is a* partial assignment*. If P is a partial assignment on a set B of variables, then we say that B is the* support *of P.*

Definition 2.6 (Solution)**.** *If C is a clause, then we say that a partial assignment P* satisfies *C if a variable that is assigned 1 in P is present in C, or if the negation of a variable that is assigned 0 in P is present in C. If I is an instance, then a partial assignment to all variables of I that satisfies all clauses of I is a* solution*, or* model*, for I.*

Solving

Most modern SAT solvers systematically search the space of truth assignments using inference method to speed up the search. A popular SAT solver technique that performs systematic search is the Davis Putnam Logemann Loveland (DPLL) algorithm [DLL62]. Practical DPLL-based SAT solvers started to ap-

pear from the mid 1990s as a result of enhancements such as clause learning, non-chronological backtracking, branching heuristics, restart strategies, and lazy data structures [SS96, ZMMM01, GSK98, MMZ+01]. The DPLL algorithm is based on unit propagation, pure literal elimination and branching [BHvMW09]. A search tree is constructed in order to keep track of the search. As described earlier propagation is the process where the rest of the problem is restricted given a constraint.

During **unit propagation** clauses that contain a single literal are propagated. Since the CNF is a conjunction of clauses, each clause has to be satisfiable on its own. Thus a single literal clause, unit clause has to be satisfied. Propagating the unit clause l will consist of checking each clause in the CNF. In case literal l is present, the clause gets eliminated since it is already satisfied this way. If the negation of l is present, this is eliminated from the cause because we know it will never have a true value.

A **pure literal** is a literal that occurs in the CNF only with one polarity, either all occurrences are l or $\neg l$. A pure literal can be assigned as it will only satisfy clauses, since no opposite of it exists it cannot lead to unsatisfiability. After unit propagation and pure literal elimination, the solver chooses a variable to assign to *true* or *false*, according to its search strategy. This assignment is propagated similarly to unit clause propagation. Then the next iteration starts again with unit clause propagation and pure literal elimination. At any stage, if a clause becomes empty the respective partial solution cannot be extended further and backtracking is needed. If the CNF becomes empty, all clauses have been satisfied and a satisfying assignment has been found. It is important to note that when the search finishes not all variables might be assigned. In case a variable is unconstrained or becomes unconstrained during solving that variable can have either *true* or *false* assigned.

DPLL is based on chronological backtracking [BHvMW09]. Backtracking is the process of moving from the current level to a previous one. If backtracking to a previous level is done only after having tried both $true$ and $false$ assignments at the current level, then it is called *chronological backtracking*. This type of search will ensure complete exploration of the search space. One of the drawbacks of this type of search is that it does not take into account the contradictions that trigger backtracking. Non-chronological backtracking overcomes this by allowing the search to get away from parts of the search space that will not lead to a solution [SS96]. In SAT this can be achieved by identifying the *conflict set* [Dec90]

and backtracking to the most recent variable from this set that was not flipped yet. The conflict set contains all assignments that contribute to the derivation of the empty clause. This type of backtracking partially prevents the search from reaching the same conflicts. To further prevent the search from reaching the same conflict, conflict-driven clauses can be added to the CNF. These are clauses identified during unit resolution. The process of adding a conflict-driven clause to the CNF is called conflict learning. This is the basis of Conflict Driven Clause Learning SAT solvers such as Chaff [MMZ+01], Satz [LA97], etc.

2.1.3 Encoding CSPs into SAT

A CSP can be transformed to a SAT problem by a variety of encodings: direct, log, support, etc. [BHW03]. In this work we only look at the direct encoding, which assigns a literal to each variable-value pair and expresses constraints in terms of these. We chose this representation as it is the simplest form to express a CSP in SAT, and easy to follow the meaning of a literal. Other encodings might behave differently with respect to partial compilation. For each set of literals that connect to the same CSP variable, additional constraints are added to ensure exactly one value is assigned to the variable. These are the at-least-one (ALO) and at-most-one (AMO) constraints, as their name indicates they ensure at least one of the literals for the variable is true, and that only one such literal is chosen. Conflicts are expressed by conflict clauses, disjunction between negations of the literals.

Table 2.1: Variable value pairs denoted by literals

variable	X_1			X_2			...	X_n		
value	d_1	...	d_m	d_1	...	d_m	...	d_1	...	d_m
literal	$l_{1,1}$	...	$l_{1,m}$	$l_{2,1}$	...	$l_{2,m}$	...	$l_{n,1}$	...	$l_{n,m}$

Definition 2.7 (Direct Encoding)**.** *Let us define a CSP with n variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ each having a domain of m values $D_i = \{d_1, \ldots, d_m\}$ and k constraints $\mathcal{C} = \{C_i, \ldots, C_k\}$. We assume each variable has the same domain but that is not a restriction. The direct encoding associates a literal to each variable-value pair, as denoted in Table 2.1. The CNF would consist of 3 types of clauses:*

- *ALO (at-least-one) of the form $l_{i,1} \vee l_{i,2} \vee \ldots \vee l_{i,m}$ where $l_{i,m}$ are literals denoting all value assignments for variable X_i. Each variable would have a separate ALO clause.*
- *AMO (at-most-one): pairwise negations of each literal associated with a*

variable. In this case for variable X_i, it would have the form: $\neg l_{i,1} \vee \neg l_{i,2}$, $\neg l_{i,1} \vee \neg l_{i,3}$, ... , $\neg l_{i,(m-1)} \vee \neg l_{i,m}$. We define such clauses for all variables.

- *conflict clauses: each inconsistent combination of literals is forbidden by a clause containing the disjunction of the negation of those literals. This means at least one of the variables has to have a different value than stated in these clauses. As an example let us suppose the inconsistent tuples for constraint C_1 on variables X_1, X_2, X_3 are: $\{(d_1, d_2, d_3), (d_2, d_3, d_4), \ldots, (d_{m-2}, d_{m-1}, d_m)\}$. This constraint will be represented by the clauses:*

$$\neg l_{1,1} \vee \neg l_{2,2} \vee \neg l_{3,3},$$

$$\neg l_{1,2} \vee \neg l_{2,3} \vee \neg l_{3,4},$$

$$\ldots ,$$

$$\neg l_{1,m-2} \vee \neg l_{2,m-1} \vee \neg l_{3,m}.$$

Example 2.3. *The CSP of instance E is defined by*

- *variables:* $\mathcal{X} = \{X_1, X_2, X_3\}$
- *domains:* $\mathcal{D} = \{D_1, D_2, D_3\}$ *such that* $D_1 = \{1, 2, 3\}$, $D_2 = D_3 = \{2, 3, 4\}$
- *constraints:*

$$X_1 \neq X_3$$

$$X_1 \leq X_2$$

$$X_2 \leq X_3$$

As previously we used the direct encoding to represent the CSP in CNF. The SAT instance is expressed on the set of Boolean variables $\{A, B, C, D, E, F, G, H, I\}$ as shown in Table 2.2. The ALO and AMO clauses are defined on the set of variables $\{A, B, C\}$, $\{D, E, F\}$ and $\{G, H, I\}$. The rest of the constraints are expressed with conflict clauses:

$X_1 \neq X_3 : C_1 = (\neg B, \neg G)$, $C_2 = (\neg C, \neg H)$

$X_1 \leq X_2$: $C_3 = (\neg C, \neg D)$

$X_2 \leq X_3 : C_4 = (\neg E, \neg G)$, $C_5 = (\neg F, \neg G)$, $C_6 = (\neg F, \neg H)$

Table 2.2: Literal association for the direct encoding

Variable	X_1			X_2			X_3		
Value	=1	=2	=3	=2	=3	=4	=2	=3	=4
Literal	A	B	C	D	E	F	G	H	I

2.1.4 Symmetry Breaking

Symmetry breaking in CP has been widely recognised as having a large impact, but research around it is still sparse compared to other aspects in the field [GPP06]. A good overview of the different definitions and types of symmetry can be found in [CJJ+06]. Symmetry is defined in a variety of ways. Generally, symmetry is defined as a bijective function mapping solutions to solutions, non-solutions to non-solutions. It can be inherent in the problem, a property of the solutions or introduced while modelling. In the literature there are many definitions and types of symmetries, but the most common are:

- variable symmetry defined as a permutation over variables $X \in \mathcal{X}$ of the CSP,
- value symmetry defined as a permutation among domain values in D_i of each variable X_i individually,
- variable-value symmetry defined on variable value pairs.

Each of these permutations are defined such that they preserve the set of solutions. We give an example for these symmetries with the help of the N-queens problem in Section 3.5(page 52). More details of these categorisations can be found in [CJJ+06]. Next, we show an example of symmetry in Example 2.4.

Example 2.4 (Variable symmetry)**.** *Let a CSP* $(\mathcal{X}, \mathcal{C}, \mathcal{D})$ *be defined such that* $\mathcal{X} = \{X_1, X_2, X_3, X_4\}$, $\mathcal{D} = \{D_1, D_2, D_3, D_4\}$ *with* $D_1 = D_2 = D_3 = \{1, 3\}$ *and* $D_4 = \{7, 9\}$ *and the only constraint in* $\mathcal{C}$ *is* $X_1 + X_2 + X_3 = X_4$.

Solutions to Example 2.4 are:

- $S_1 : 1 + 3 + 3 = 7$
- $S_2 : 3 + 1 + 3 = 7$
- $S_3 : 3 + 3 + 1 = 7$
- $S_4 : 3 + 3 + 3 = 9$

Since summation is a commutative operation the CSP in Example 2.4 will contain symmetry: given a valid sum any order of its values will be equivalent to it.

Another closely related concept is *value interchangeability*, where all values of a variable are equivalent, any permutation of them is still a solution/non-solution respectively. [Fre91, MdlBW09, CJJ+06].

A set of symmetrical solutions belong to an algebraic group [GPP06]. Breaking the symmetries means that through additional constraints we eliminate solutions such that at most one representative of the symmetry group remains a solution. Similarly, these constraints will eliminate symmetrical non-solutions. This means domains, typically, get more restricted which offers the potential for more constraint propagation. On the one hand symmetry breaking reduces the number of valid solutions with respect to the symmetry functions, thus avoiding branches that would lead to a symmetrical solution. On the other hand the same is valid for non-solutions, branches that would lead to symmetrical non-solutions get pruned earlier.

Generally, there are three ways to deal with symmetries: reformulate the problem or dynamic or static symmetry breaking [GPP06]. Reformulating the problem depends on the problem at hand and can be very difficult requiring deep domain knowledge. Both static and dynamic symmetry breaking will seek to eliminate solutions such that preferably only one of each equivalence class is kept. In a dynamic setting, this is implemented by a solver that adapts its search procedure, for example during branching. We do not elaborate much on dynamic symmetry breaking, more information can be found in [GPP06]. Static symmetry breaking consists of adding constraints to the initial model. These constraints are called Symmetry Breaking Predicates (SBP). This approach was first introduced for SAT problems [CGLR96] and later applied to other similar fields. The *lex-leader* method is the most commonly used static symmetry breaking [GPP06]. It consists of imposing an ordering on the set of variables and use it to construct a lexicographical ordering on the set of assignments.

Depending on the problem at hand, recognising symmetry and formulating constraints to break all symmetries can be challenging, and often domain knowledge is required. Some tools have been implemented to deal with this at various levels. At a constraint microstructure level, `saucy` [DLSM04] or `nauty` [MP14] can be used to recognise automorphisms in graphs. A graph is created such that vertices represent SAT literals and edges possible combinations of these. Shatter [AMS03] and BreakID [DBBD16] were implemented using `saucy` to identify symmetries, and then generate predicates to eliminate them.

Savile Row is a modelling assistant for CP which also has an option for sym-

metry breaking. It takes as input a constraint model defined in a high-level language, ESSENCE' [NR16], and transforms this into various CP and SAT models [NAG+17]. It uses a graph automorphism solver to detect symmetries and breaks them using the lex-leader method [FFH+02].

Symmetry breaking is not always beneficial, for example, local search performs better when we do not break symmetries [PR05]. Also for some problems, with many symmetries the overhead of extra constraints to break all symmetries might not be beneficial enough. Also when not looking for all solutions some of the easier-found solutions might be eliminated by symmetry breaking [Pre03]. The implications of combining different techniques developed to speed up solving are not always clear. In [Pre03] they show that symmetry breaking can have a negative impact on solving, but it depends on which SAT solver and what problem is being considered. SAT solvers are implemented following the same base technique but they will have implementation details and different added techniques that will make them behave differently on the same problem.

2.1.5 Backbones

The concept of backbones was introduced in 1999 to characterise the hardness of decision problems, in a study about phase transition in SAT [MZK+99]. The backbone is a measure of the amount of shared structure among the set of all solutions to a given problem instance [AGKS00]. Informally it is the set of assignments common for all solutions of the problem. It was formally defined in [WGS03] as a set of variables that has a unique assignment that satisfies all solutions.

Definition 2.8 (Backbone)**.** *Let I be a satisfiable constraint instance. A* backbone *B is a set of assignments within instance I that is part of all solutions of I. The backbone of an instance is the set of all assignments that are part of all solutions.*

To be consistent with the notions we later introduce we refer to a backbone as a set of assignments rather than a set of variables.

Example 2.5 (Backbone)**.** *Let us define a CNF formula using the clauses: $(\neg b \vee c)$, $(b \vee d \vee e)$, $(c \vee \neg e)$, $(\neg a \vee e)$, (a). The solutions to this formula are listed below in Table 2.3.*

Looking at the CNF in Example 2.5 it is obvious that $a = 1$ is a backbone, otherwise the last clause cannot be satisfied. This is called a trivial backbone, defined by a unit clause. Both $c = 1$ and $e = 1$ are also backbones since the same

Table 2.3: Solutions to Example 2.5

	a	b	c	d	e
S_1	1	0	1	0	1
S_2	1	0	1	1	1
S_3	1	1	1	0	1
S_4	1	1	1	1	1

value appears in all solutions in Table 2.3, but this is not that obvious from the CNF.

Backbones have been used in optimisation problems and Maximum Satisfiability too [KSW05, SW01, CG01]. If a problem has a large backbone it means there are many opportunities to assign variables incorrectly [KSTW05]. If the opposite of the backbone literal is assigned it will not lead to a solution. Such problems are hard for systematic search solvers. Problems with large backbones are also hard for local search methods since the solutions are clustered.

Backbones can be used for example in the context of a configuration problem to tell the user what component can only have a single value. This can make the user experience more efficiency using such a system. However, recognising backbones is not always trivial.

Finding backbones is intractable in general, and it is both NP-hard and NP-easy [KSTW05]. The work [MJL10] describes two ways to recognise backbones. One is model enumeration, where solutions are enumerated and the intersection of them is computed. Another method is iterative SAT testing, evaluating both l and $\neg l$ if one is SAT and the other is UNSAT means we have found a backbone literal since there are only solutions with one polarity of the literal.

2.2 Model Counting

Combinatorial problems are used for a variety of purposes: proving that a problem is satisfiable, finding all solutions, finding the best solution, etc. We have seen examples of this in the previous sections. Another usage of these formulations is model counting, the problem that counts the number of solutions to a given formula F [BHvMW09]. We use the terms model and solution interchangeably. SAT looks at the satisfiability of propositional formulas, model counting also denoted as $\#SAT$ looks at computing the number of models satisfying a propositional formula. It is denoted as $\#F$ and is $\#P$-complete problem. Various probabilistic inference problems, such as Bayesian network reasoning, can be translated into

model counting problems [SBK05, BHvMW09].

Definition 2.9 (Model Counting (MC/$\#SAT$))**.** *Model counting is the problem of computing the number of models for a propositional formula.*

Example 2.6 (Model Count)**.** *Given the CNF:* $(\neg b \vee c)$, $(b \vee d \vee e)$, $(c \vee \neg e)$, $(\neg a \vee e)$, (a) *and Table 2.3 listing all solutions the model count of the CNF is 4, since there are 4 solutions satisfying the formula.*

In essence, SAT looks for one solution, whereas $\#SAT$ looks for the number of all solutions. DPLL can be extended to find all solutions but is not going to be very efficient. The heuristics used by SAT to narrow down the solution space to one solution will not help when we want to explore the whole solution space. Since the scalability of $\#SAT$ is still an issue methods to approximate the model count have also been explored. Two categories of model counters have been explored: *exact counters* and *approximate counters* [BHvMW09]. Exact model counters can be further divided in two categories: DPLL-style exhaustive search and Knowledge Compilation based. Approximate model counting can be divided into techniques that quickly estimate the count without any guarantee, and techniques that provide upper and lower bounds with a correctness guarantee.

In [LMY21] a tool is presented that combines search- and compilation-based model counting. Taking advantage of literal equivalence they proposed an efficient model counting technique whose trace corresponds to a generalisation of Decision-DNNF. Since the early 2000s variable-elimination-based model counters started to appear In [LMY21] they categorize model counters in three:

- **search-based:** predominantly rely on component caching and conflict driven clause learning. Some of the most popular model counters in this category are Cachet [SBB+04], sharpSAT [Thu06] and Ganak [SRSM19].
- **compilation-based:** the formula is compiled to a representation that is able to perform model counting in polynomial time. We give examples of such tools(d4, PySDD, MiniC2D, etc.) and their representation in Section 2.3.
- **variable-elimination-based:** this is a method that was introduced to answer probabilistic queries with respect to Bayesian networks [ZP96]. This is based on dynamic programming frameworks that recursively eliminate variables while counting the models of the problem. Some tools that incorporate variable elimination are ADDMC [DPV20] and nestHDB [HTW20].

2.2.1 Exact Model Counting

One of the first exact model counters, CDP [BL99], is an algorithm based on the David-Putnam procedure. It works by recursively splitting a formula into two branches based on a variable and its negation. At each step, clauses that get satisfied are removed from the formula, and the model count is returned when all clauses have been satisfied. Some variables might be eliminated before getting assigned if they only occur in clauses that have been satisfied by previous variable assignments. In case f variables have been assigned out of n variables, this satisfied branch contains 2^{n-f} solutions as the partial assignment can be extended by any assignment for the rest of the $n-f$ variables. This type of model counting is an incremental approach that is able to provide a lower bound based on the explored search space in case the search times out. Some of the most used exact model counters based on DPLL-style SAT solving are sharpSAT [Thu06], Cachet [SBB+04], etc.

Another approach used for exact model counting is through compiling a CNF formula into another logical form that supports model counting in polynomial time with respect to the size of the representation. The advantage of using compilation techniques is that the time consuming operation of compiling is done once after which the obtained representation can be used to answer several queries. Extending the representation is also possible without recompiling the whole formula. Section 2.3 gives an introduction to knowledge compilation, the field that deals with the various compilation techniques and representations.

2.2.2 Approximate Model Counting

Exact model counters often suffer with scalability issues so for this reason approximate methods have been proposed. Model counting is not necessarily hard for problems with large numbers of variables or solutions but more so for problems where solutions are spread out through the search space. Approximate counters are judged based on their quality of estimate and confidence of correctness.

Three main categories of approximate model counters are discussed in the literature [LMY23]. These are based on the guarantee on the estimated model count. The first category is parametrised by (ϵ, δ). Given a formula θ with Z models the estimated model count will be in given in the interval $[\ (1+\epsilon)^{-1}Z,\ (1+\epsilon)Z\ ]$ with a guarantee of $1-\delta$. We say that ϵ is the tolerance of the count, and $1-\delta$ is its confidence. ApproxMC is part of this class [CMV13].

The second category of counters are called lower (or upper) bounding counters. These counters offer a confidence of at least $1 - \delta$. The large majority of approximate counters used in practice are bounding counters. Examples include SampleCount [GHSS07], BPCount [KSS08], MBound (and Hybrid-MBound) [GSS06], and MiniCount [KSS08].

The final category of counters offers weaker guarantees but can offer relatively close approximations. Examples of this category are SearchTreeSampler [EGS12], ApproxCount [WS05], PartialKC [LMY23] and SampleSearch [GD11].

Approximate counters often explore methods of uniform or near uniform sampling. Sampling approaches are based on the idea that a discrete distribution D is defined on all the possible truth assignments σ , such that:

$$D(\sigma) = \begin{cases} 1/Z, & \text{if } \sigma \text{ is a solution, with } Z \text{ being the number of distinct solutions} \\ 0, & \text{otherwise} \end{cases}$$

This distribution is then sampled. Sampling methods produce assignments according to this distribution. Importance sampling has also been used for approximate model counting [GD07].

2.2.3 Weighted Model Counting

Weighted Model Counting (WMC) is mostly used in the context of probabilistic inference. Bayesian networks are encoded in CNF, and weights are added to each variable based on the network probabilities [SBK05]. Two types of variables are defined: chance and state. Chance variable weights are defined, such that when evaluated to *true* $w(l) \in [0, 1]$ and when evaluated to *false* $w(\neg l) = 1 - w(l)$. In case of state variables, both $w(l)$ and $w(\neg l)$ are 1, meaning both instantiations are equivalent in terms of weight, there is no difference between them from a perspective of importance.

Definition 2.10 (Weighted Model Count [CFMV15])**.** *The weight of a model/ solution is equal to the product of the weights of the literals that satisfy the input formula. The weighted model count (WMC) of a formula F is the sum of the weights of each model/solution of F.*

Example 2.7 (Weighted Model Count)**.** *Given the CNF seen before: $(\neg b \vee c)$, $(b \vee d \vee e)$, $(c \vee \neg e)$, $(\neg a \vee e)$, (a) we define a set of weights for each literal as shown in Table 2.4. Table 2.5 lists all solutions with values in the table denoting*

the weight rather than 0/1 values. Each row shows a solution/model and the weights for each variable, whose product will be the weight of that solution, as shown in the last column.

Table 2.4: Weights for Example 2.7

a	$\neg a$	b	$\neg b$	c	$\neg c$	d	$\neg d$	e	$\neg e$
0.9	0.1	0.7	0.3	0.6	0.4	0.75	0.25	0.85	0.15

Table 2.5: Solutions and their weights for Example 2.7

	a	b	c	d	e	Model Weight
S1	0.9	0.3	0.6	0.25	0.85	0.034
S2	0.9	0.3	0.6	0.75	0.85	0.103
S3	0.9	0.7	0.6	0.25	0.85	0.080
S4	0.9	0.7	0.6	0.75	0.85	0.241

We can regard model counting as a special case of weighted model counting where all weights are 1. In Example 2.6 each solution has the weight of 1 whereas for Example 2.7 each has a different value.

Just like in case of model counting, weighted model counting can be performed through the solving process or through compilation. The tools for weighted model counting are more limited. In [SBK05] they extended a DPLL-style algorithm to perform weighted model counting on Bayesian Networks. Compilation tools supporting weighted model counting include Dsharp [MMBH12], MiniC2D [OD15], d4 [LM17] etc. Approximate weighted model counting is explored in [CFM$^+$14].

2.3 Knowledge Compilation

Decision Diagrams have been around for a long time, originally introduced for circuit design and formal verification [Lee59, Bry86]. Knowledge Compilation (KC) methods were developed to use decision diagrams to deal with the complexity of solving combinatorial problems online by creating a representation offline that can answer queries in time that is polynomial in the size of the representation. [DM02].

Selecting the appropriate representation to use can have a huge impact. A representation language is somewhat human readable, such as CNF, Horn clauses, etc. whereas a target compilation language might not be human readable but should be tractable enough with respect to some queries/transformations. In [DM02] they define a target compilation language one that supports polytime clausal entailment. The Knowledge Compilation Map [DM02] tells us the more queries

become more tractable the representation can become exponentially larger. The KC Map was created to define a structure that describes the relationship between some representations to help make an informed decision which is suitable for a use case. One should first define the queries and transformation its application needs and then choose the most succinct representation that supports these. Often generating the smallest representation is the bottleneck of compilation methods[DM02]. Similar maps have since been developed looking at multi-valued decision diagrams [AFNP14] or real-values decision diagrams [FMNS14].

Research in the area has focused on different representations and tools such as probabilistic reasoning, as well as applying knowledge compilation to different use cases [Dar20a, SCD18a, CLM21]. Prime implicates have also been explored as a compact representation [DM02, SdV01].

Negational Normal Form (NNF) is defined as the base of the compilation map. It is a rooted Directed Acyclic Graph (DAG) that represents a propositional formula. The NNF contains AND, OR, and negation gates, where negation gates are only wired by input variables [BF21]. This is not a tractable representation but many of its subsets are, due to the additional properties they satisfy. Next, we give a formal definition of the NNF and describe the properties that lead to other representations being tractable.

Definition 2.11 (Negational Normal Form [DM02])**.** *A sentence in Negational Normal Form is a rooted, directed acyclic graph, where each leaf node is either true, false, the literal l or its negation, $\neg l$. Each internal node represents either a conjunction (AND node) or disjunction (OR node) with arbitrarily many children.*

NNF does not qualify as a target compilation language as it does not permit queries in polynomial time [DM02]. Subsets of NNF are defined by decomposability, determinism, smoothness, decision, and ordering properties. These properties as defined in [DM02] contribute to making subsets of the NNF tractable:

- Decomposability: this property holds if for each conjunction ($\wedge$) in the NNF the children of it do not share variables. For example, in Figure 2.3 the red circled AND ($\wedge$) node has children that are defined and separate subsets of variables $\{A, B\}$ and $\{C, D\}$. Similarly, all AND nodes satisfy this criterion.
- Determinism: this property holds if for each disjunction ($\vee$), the children pairwise are logically contradicting. For example, in Figure 2.3 the red circled OR ($\vee$) node has the children $\{\neg A \wedge B\}$ and $\{A \wedge \neg B\}$ which

logically contradict. Similarly, all OR nodes satisfy this criterion.

- Smoothness: this property holds if for each disjunction, the children of the OR nodes are defined on the same variables. As shown in Figure 2.3 the red circled OR nodes children are both defined on the same variables. This is valid for the rest of the OR nodes in the NNF.
- Decision: a decision node N is either *true*, *false* or an OR node of the form $(\neg X \wedge \alpha) \vee (X \wedge \beta)$, meaning if X then β else α, where X is a variable and α, β are decision nodes. Every NNF that satisfies the decision property is also deterministic.
- Ordering: Decision variables appear in the same order on any path in the NNF.

To efficiently use an NNF for model counting it must satisfy decomposability and determinism [BHvMW09]. Decomposability allows model counting algorithms to calculate the count at an AND node by multiplying the counts of the children nodes. That is if N is an AND node $\#N = \#C_1 \times \#C_2 \times \cdots \times \#C_n$ where $C_1, C_2, \ldots, C_n$ are children nodes of N. Since variables are not shared

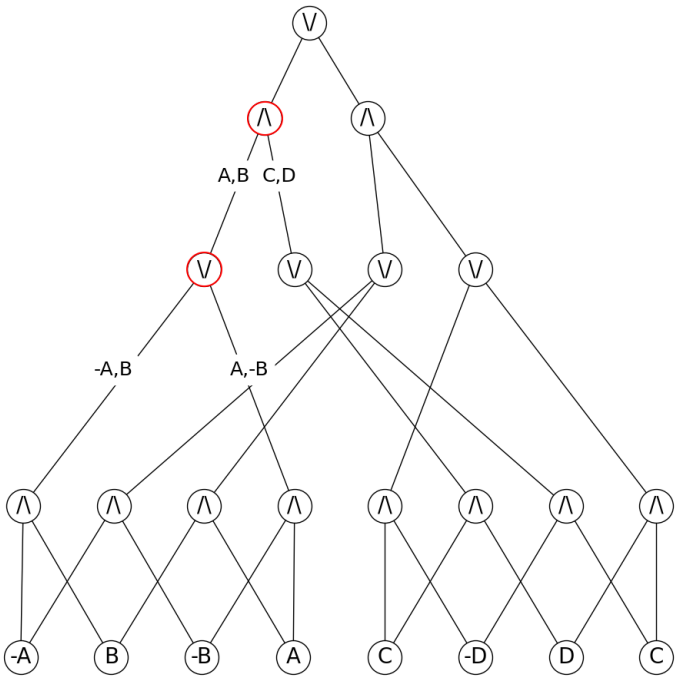


Figure 2.3: Decomposability, determinism and smoothness of an NFF as shown in [DM02].

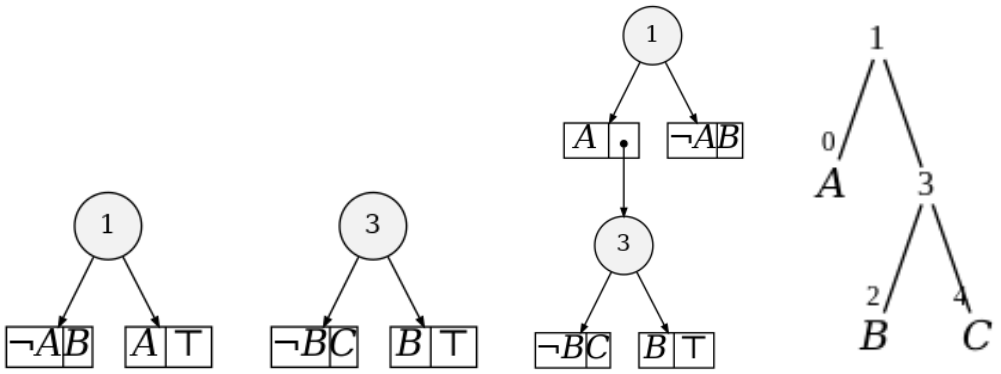
between child nodes there is no possibility of counting the same solution more than once. Determinism allows model counters to calculate the count an OR node by summing the counts of the child nodes. That is if N is an OR node $\#N = \#C_1 + \#C_2 + \cdots + \#C_n$ where $C_1, C_2, \ldots, C_n$ are children nodes of N. This ensures that we are covering all possible combinations of solutions and including their model counts. These properties allow model counting algorithms to calculate $\#P$ of a formula P by assigning 1 to each leaf node, traversing the representation satisfying the above properties and summing the model counts of children of OR nodes and multiplying the model counts of children of AND nodes. The model count at the root node is equal to $\#P$ [BHvMW09].

Decomposable Negational Normal Form (DNNF) was introduced to characterize consistency-based diagnoses [Dar01]. It extends the NFF by the decomposability property which makes it a tractable representation [OD17].

Compilation methods

A knowledge base expressed in CNF can be compiled using top-down [LM17] or bottom-up [CD13a] compilation approach. The top-down approach recursively compiles the clauses using conditioning, for example the trace of DPLL using a SAT solver could create one [OD15]. Bottom-up compilation takes each CNF clause and compiles it after which using the *apply* [Bry86] operation joins the representations together efficiently. Top-down compilers work only on CNF inputs, whereas the bottom-up compilers *apply* operation is able to deal with other input types as well. We illustrate the *apply* operation in Figure 2.4. The SDD of the clause $A \vee B$ and $B \vee C$ is represented in Figure 2.4a and Figure 2.4b respectively. The *apply* operation joins these to form the SDD in Figure 2.4c. In here, node 1 corresponds to node 1 from subfigure (a), extending the case when $\neg B$ is true to make sure C is true, so that it represents the correct formula. When B is true the formula is satisfied no matter what C evaluates to, so this branch needs no extension. Top-down compilation has been compared to bottom-up compilation for OBDDs [HD04] and for SDDs [OD15], in both cases top-down compilation being faster.

The size of the representation depends on a variable ordering. This can be expressed as a list or in case of some representations as a tree structure called **variable tree (vtree)** [PD08]. This is a dissection of a variable ordering such that each leaf node in a vtree represents a variable and internal nodes map to nodes in the compiled representation. An example vtree and an SDD can be seen



(a) $A \vee B$ (b) $B \vee C$ (c) $(A \vee B) \wedge (B \vee C)$ (d) vtree used

Figure 2.4: Example of the apply operation on SDD

in Example 2.4. The variable ordering used for the OBDD significantly impacts its size but it is not feasible to find the best ordering as this is an NP-hard problem [NW07]. This is generalised to other representations as well, a representation is canonical with respect to the variable ordering. Usually, some limited search is performed to find a good ordering but usually not the optimal one.

Table 2.6: Tools and their compiled representations

Tool:	CUDD [1]	PySDD [2]	ZSDD [3]	D4 [4]	MiniC2D [5]
Compiled Representation:	OBDD	SDD	ZSDD	Decision-DNNF	Decision-SDD

In the next sections, we present some of the existing representations along with an example of them. In Table 2.6 we consider the five compilers we used and the representations they generate. For each representation, we compile the same CNF, which is the direct encoding of a CSP of a sum constraint. The running example is defined on four variables (X_1, X_2, X_3, X_4) each with two domain values. The constraint of the running example is: $X_1 + X_2 + X_3 = X_4$. The mapping of the variable assignment and the literals that appear in the compilations is in Table 2.7.

[1] https://web.archive.org/web/20150215010018/http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[2] https://github.com/wannesm/PySDD

[3] https://github. com/nsnmsak/zsdd/

[4] https://www.cril.univ-artois.fr/KC/d4.html

[5] http://reasoning.cs.ucla.edu/minic2d/

Table 2.7: Direct encoding literals

Problem variables	X_1		X_2		X_3		X_4	
Domain values	1	3	1	3	1	3	7	9
Associated literals	A	B	C	D	E	F	G	H

2.3.1 Ordered Binary Decision Diagram (OBDD)

A Binary Decision Diagram is a directed acyclic graph that represents a set of solutions. Edges are labelled by variable assignments and paths from the root node to the terminal *true* node represents the solutions [Bry92].

The earliest and most widely used compilation language is the Ordered Binary Decision Diagram (OBDD), which has been used in circuit analysis and synthesis [Bry92, Bry86]. It is an extension of BDD by imposing a fixed variable order-

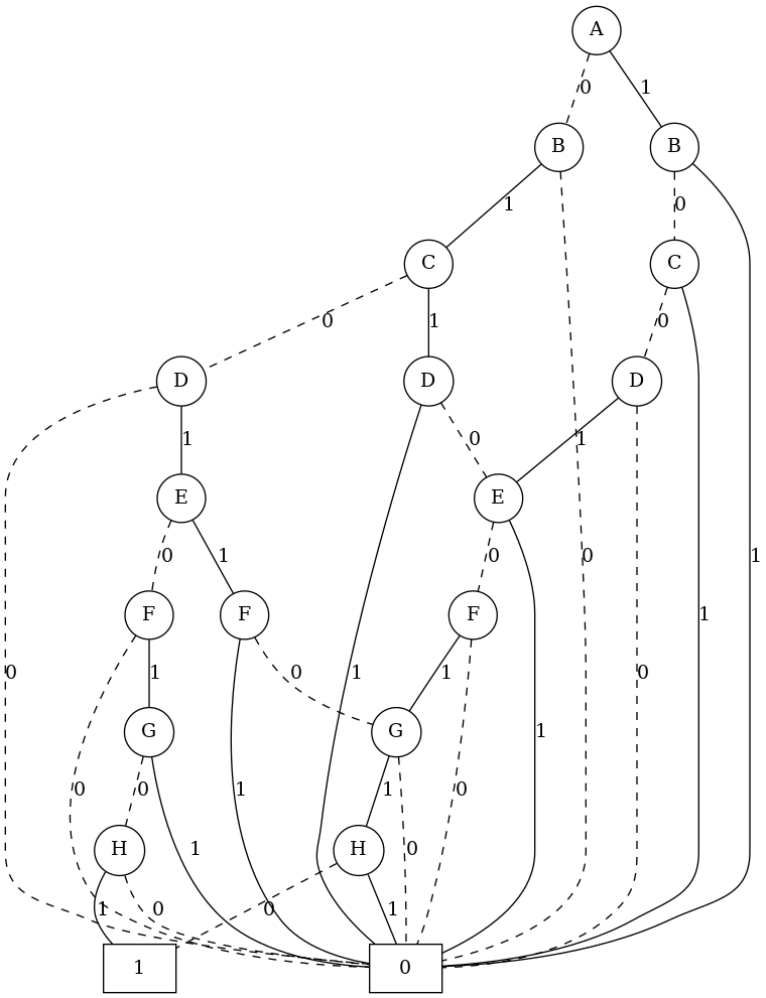


Figure 2.5: OBDD of direct encoded CSP: $X_1 + X_2 + X_3 = X_4$, with domains $\{1, 3\}$ for X_1, X_2, X_3 and $\{7, 9\}$ for X_4. Literal A denotes the assignment $X_1 = 1$, B denotes $X_1 = 3$ etc.

ing. OBDDs are canonical with respect to the ordering. SAT models are often compiled to OBDDs, so that these representations can be queried instead of solving the SAT model again. One of the most popular OBDD compilers is CUDD. Queries that OBDD can answer in polynomial time are consistency checking, validity checking, clausal entailment, implicant check, equivalence check, model counting and model enumeration [DM02].

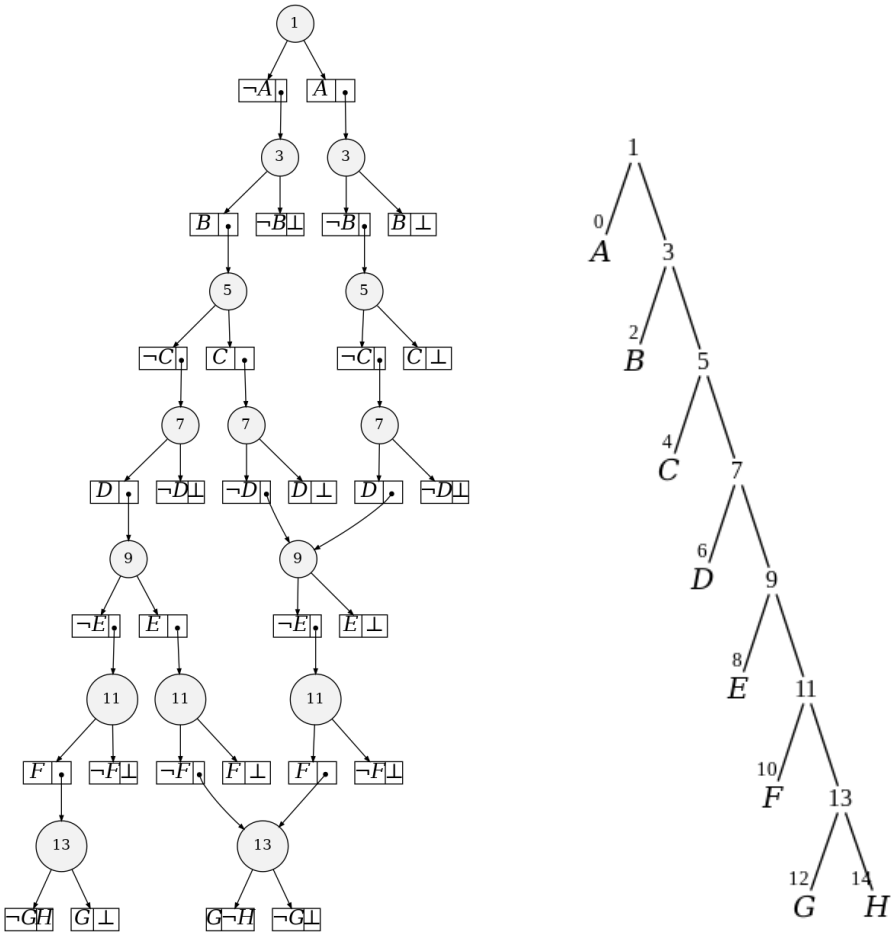
Figure 2.5 shows an OBDD of a sum constraint. We use the direct encoding as shown previously, thus literals A-H each represents a variable-value assignment. Each level in the OBDD represents a literal, edges labelled with 1 denote *true* and edges labelled 0 denote the assignment of *false*. A path in the OBDD that finishes in the box labelled 1 is a solution to the formula. Paths that finish in the box labelled 0 are non-solutions. When a path does not assign value to each variable it means regardless of the rest of the variables the partial assignment is a solution/non-solution. For example, the partial assignment $\{A = 1, B = 1\}$ is not a solution no matter what values the rest of the variables have since this already violated the AMO constraint.

2.3.2 Sentential Decision Diagrams

The deterministic Decomposable Negation Normal Form (d-DNNF) has been successfully used in various applications such as probabilistic reasoning, planning, and diagnosis but one of its drawbacks is that it does not support the *apply* operation in polytime [PD08]. For this reason, the concept of structured decomposability was formulated, which created a new representation called structured d-DNNF. Structured decomposability entails that a vtree is used.

Sentential Decision Diagram (SDD) is an extension of NNF with structured decomposability and strong determinism [Dar11]. These properties make it a strict subset of d-DNNF and a strict superset of OBDDs. OBDDs have been extended to Sentential Decision Diagrams (SDDs) such that instead of making decisions on literals they decide on sentences. While OBDD decision nodes correspond to a variable SDD decision nodes correspond to a node in a tree representation that orders the variables, the vtree. As we saw in Figure 2.5 at each level of the OBDD a decision is made on one variable given by a variable ordering. In the case of an SDD at each level, variables are partitioned according to the vtree. Different types of vtrees exist: right-linear, left-linear, balanced and random. In case of a right-linear vtree, as shown in Figure 2.6 the SDD is equivalent to an OBDD. SDDs appeared with multiple variations of it: ZSSD, Decision-SDD,

Tagged SDD, Variable Shift SDD, and various forms to support probabilistic and weighted model counting [FFW+19, NDN20, KdBCD14].



(a) compiled SDD (b) vtree used

Figure 2.6: SDD and the used vtree for the direct encoded CSP: $X_1 + X_2 + X_3 = X_4$, with domains $\{1, 3\}$ for X_1, X_2, X_3 and $\{7, 9\}$ for X_4. Literal A denotes the assignment $X_1 = 1$, B denotes $X_1 = 3$ etc.

An SDD is represented by a DAG (e.g. Figure 2.6a), where decision nodes (circles) represent (X,Y)-decompositions, i.e. an OR operation on sentences. These correspond to the nodes in the vtree associated with the SDD (e.g. Figure 2.6b). A vtree is a binary tree whose leaves correspond to literals. The rectangles in the SDD represent the AND operation between primes (left box) and subs (right box). The number of nodes is equal to the number of OR-nodes and the size of the SDD is equal to the number of AND-nodes, 15 and 30 respectively in Figure 2.6a.

PySDD [Dar11] is a Python wrapper for the open-source C Sentential Decision Diagram (SDD) package that compiles CNF/DNF to SDD. PySDD is a bottom-up compiler, compiling clauses to SDDs, then using the `apply` operation to combine these as well as using vtree search along the way to find a vtree that minimizes

the size of the representation. The size of the SDDs are defined by the vtree that is used to create them. OBDDs are canonical with respect to variable ordering, SDDs are with respect to vtrees.

2.3.3 Zero-Suppressed Sentential Decision Diagrams

Zero-Suppressed Sentential Decision Diagrams (ZSDDs) are an extension of SDDs such that while terminal symbols of SDDs are the literal X or $\neg X$, ZSDDs have terminals ϵ and $\perp$, where ϵ represents Boolean functions that return 1 if all variables are 0 and $\perp$ represents *false* [NYMN16]. ZSDDs tend to be a better choice for sparse problems, where the number of solutions is few and only a few variables have values equal to 1. Solution counting runs in time linear in the size of ZSDD. In a ZSDD a decision node is removed if the corresponding function returns true only when all variables from either left or right subtrees are assigned 0. The compiler used for this is also called ZSDD. It is a bottom-up compiler, and the representation can be used as a more compact alternative to SDDs [NYMN16].

Zero-Suppressed Binary Decision Diagram (ZDD) [Min93] is an extension of BDDs. In BDDs a node is removed if both edges of a node point to the same

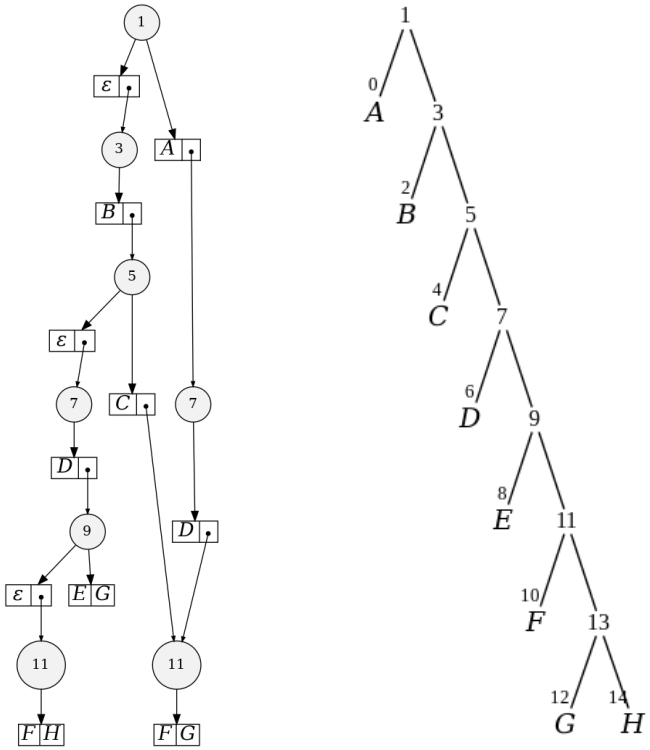


Figure 2.7: ZSDD representing the CSP $X_1 + X_2 + X_3 = X_4$, with domains $\{1, 3\}$ for X_1, X_2, X_3 and $\{7, 9\}$ for X_4. Literal A denotes the assignment $X_1 = 1$, B denotes $X_1 = 3$ etc.

node, whereas in ZDDs it is removed if the positive edge points to terminal node 0. ZSDD is a superset of ZDD just as SDD is for BDD [NYMN16].

Figure 2.7 visualizes a ZSDD for the same CSP as Figure 2.6. Similarly, circle nodes represent the OR nodes and rectangles the AND nodes. Although both use the same vtree the ZSDD is smaller, containing 8 nodes and its size is 11, compared to 15 nodes and size 30 of the SDD. This ordering is not always the case.

2.3.4 Decision-DNNF

D4 is a top-down compiler with the target representation of Decision-DNNF [LM17]. Decision-DNNF is a strict subset of Deterministic Decomposable Negational Normal Form (d-DNNF). The C2D [6] and Dsharp [7] are tools that can be used to compile a CNF to d-DNNF [Dar04, MMBH12]. The d-DNNF is a subset of DNNF with the added property of determinism and DNNF extends NNF with decomposability.

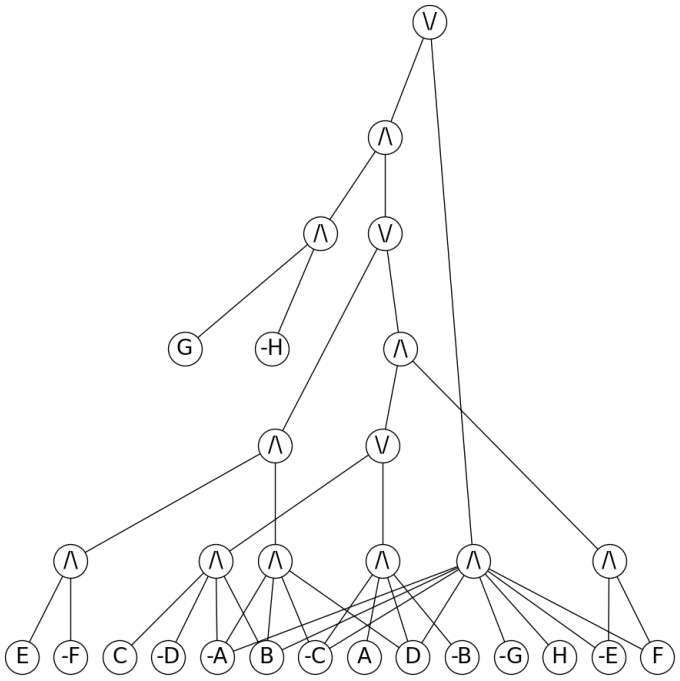


Figure 2.8: Decision-DNNF representing the CSP $X_1 + X_2 + X_3 = X_4$, with domains $\{1, 3\}$ for X_1, X_2, X_3 and $\{7, 9\}$ for X_4. Literal A denotes the assignment $X_1 = 1$, B denotes $X_1 = 3$ etc.

[6] http://reasoning.cs.ucla.edu/c2d/
[7] https://github.com/QuMuLab/dsharp

Determinism is satisfied if children of OR-nodes are pairwise contradictory. Decision-DNNF is a strict subset of d-DNNF, such that each OR node is also a decision node. A decision node N on decision variable X has the form $(\neg X \wedge Y) \vee (X \wedge Z)$, meaning if X then Z else Y. The size of the representation is defined by the number of edges of the graph. Example 2.8 illustrates a compiled representation of a Decision-DNNF of the constraint $X_1 + X_2 + X_3 = X_4$, same as before.

2.3.5 Decision-SDD

MiniC2D uses a top-down approach to compile CNF to a Decision-SDD [OD15]. A vtree node whose left child is a leaf is called a Shannon node, otherwise it is a decomposition node. An SDD with any vtree can be a Decision-SDD as long as each decomposition decision node has the form $\{(p, s_1), (\neg p, s_2)\}$ where $s_1 = \top, s_1 = \bot$ or $s_1 = \neg s_2$ [OD15]. A special form of vtree called decision vtree guarantees an SDD to be a Decision-SDD. The compiler takes as an input a CNF and a decision vtree. A clause is compatible with a vtree node if some of its literals are on the right and some on the left side of the node. A vtree for a CNF is a decision vtree if and only if every clause is compatible with only Shannon vtree nodes [OD15]. Thus, for every clause the lowest vtree node is identified, which is a common root to the literals, and the left child of this node has to be a leaf node. A right-linear vtree is equivalent to a vtree where all internal nodes are Shannon nodes, this makes the Decision-SDD equivalent to an OBDD.

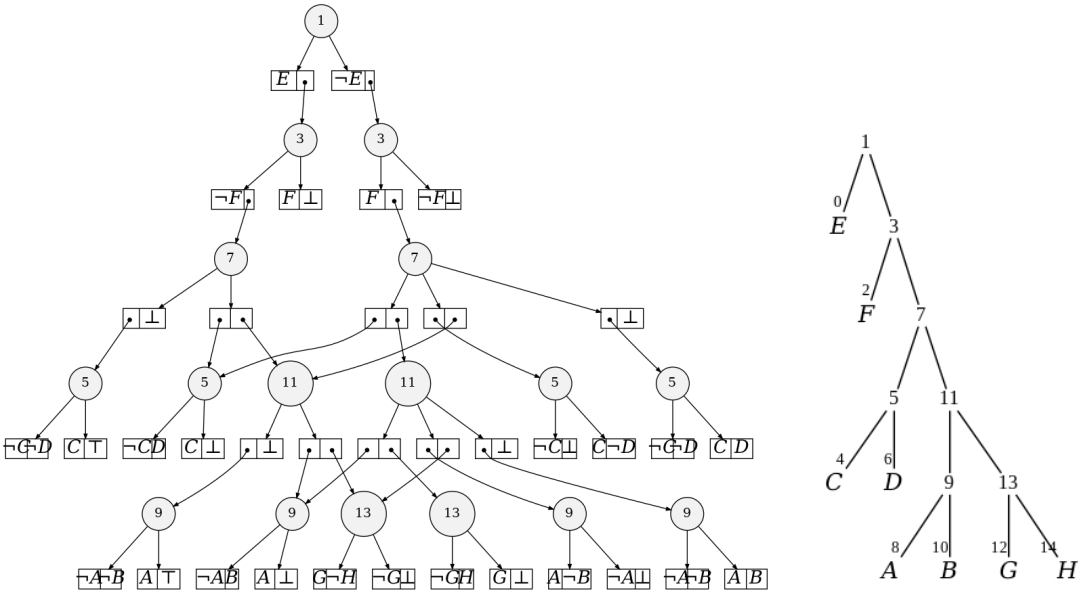


Figure 2.9: Decision-SDD representing the CSP $X_1+X_2+X_3 = X_4$, with domains $\{1, 3\}$ for X_1, X_2, X_3 and $\{7, 9\}$ for X_4. Literal A denotes the assignment $X_1 = 1$, B denotes $X_1 = 3$ etc. and the vtree used

An example representation of a Decision-SDD is found in Figure 2.9. We compiled the same CNF as in previous examples of compiled representations. The same notation is used as in Figure 2.6 which represents an SDD. Both figures compiled the same CNF but a different vtree is used.

Default parameters for compilation use the primal graph as an initial vtree with a random balance factor. Due to this random initialization, compiled representations across multiple runs find different sized SDDs. The output of MiniC2D is an NNF and a vtree that can be used to create a decision-SDD in linear time.

2.4 Summary

In this chapter we introduced the notation and definitions used in the next three technical chapters. We gave an introduction to the history and evolution of various methods. We first defined combinatorial problems, focusing on Constraint Programming and Boolean Satisfiability. We introduced the encoding we used to translate CSPs to SAT instances. In Section 2.1.4 and Section 2.1.5 we described two techniques in CP that we relied on to define partial compilations: symmetry breaking and backbone. Finally, we defined and gave examples of the knowledge compilation representations we used throughout this dissertation.

Chapter 3

Partial Compilation of CSPs through Symmetry Breaking

Abstract. *In this chapter, we explore the effect of symmetry breaking on Knowledge Compilation (KC) of combinatorial problems. We look at four different compilation tools and their respective compiled representations. We experiment on four highly symmetrical problems: a clique of not-equal constraints, N-queens, Graceful Graph Labelling and Balanced Incomplete Block Designs. We describe the CSPs as well as the symmetry breaking constraints added. We evaluate the effects of symmetry breaking by looking at the runtime and size of the resulting representations. This chapter is based on our previous publication in ACM/SAC 2023: Breaking Symmetry for Knowledge Compilation [BO23].*

3.1 Introduction

The aim of this chapter is to explore one of the ways to obtain a partial knowledge compilation of a combinatorial problem. As mentioned before, knowledge compilation is used when a problem that rarely changes is required to answer several types of queries multiple times.The computational overhead is moved offline by compiling all solutions to an efficient graph representation that is able to answer relevant queries in polynomial time with respect to its size. The expensive computation of compilation is done once, offline, then the representation is used in an online setting. Partial compilations can be useful for a few different reasons, such as space and time saving technique or a modelling choice to represent only

a specific subset of solutions.

One criterion for a partial compilation to be helpful is to represent a useful subset of solutions. The set of solutions to be eliminated must be carefully chosen. In this chapter we discuss an approach to eliminate symmetrical solutions based on a technique from the Constraint Programming (CP) and SAT (Boolean Satisfiability) community. Modelling Constraint Satisfaction Problems (CSPs) can be a challenging task. One needs to understand the domain of the problem as well as modelling techniques [RvBW06]. These are important to ensure that the CSP models are useful, scalable and fast to solve. In the CP and SAT communities various methods have been implemented to reduce search time of solvers and reduce the search space explored. One technique that can help speed up solving these problems is symmetry breaking. Inspired by this, we investigated if knowledge compilation also benefits from this method. Namely, we looked at whether models where we break symmetries have a more compact representation, and if these can be compiled within less time and memory.

Next, in Section 3.2 we describe the approach used and related works and in Section 3.3 we describe the high level methodology we used. In the following sections, we discuss each of the four CSP problems separately: describe the problem statement, give a small example and present the results for the four compilers on a set of instances. Section 3.4 looks at the clique of not-equals problem, Section 3.5 looks at the N-queens problem, Section 3.6 looks at the Graceful Graph Labelling problem and Section 3.7 looks at the Balanced Incomplete Block Design (BIBD) problem. We chose these problems because they have been in the focus of research in symmetry in constraint satisfaction [RvBW06]. Also, they are highly symmetrical which leaves a lot of opportunities to observe our hypothesis. Even though they are small academic problems a lot of real world problems can be reduced to them. We conclude with a summary of the chapter.

3.2 Approach

As mentioned before, compiled representations of CSPs represent the set of solutions as a Directed Acyclic Graph (DAG) with properties that support polytime queries and transformations. Our hypothesis is that when we eliminate symmetries, sub-graphs that represent the difference between the two solutions get eliminated and thus paths that represent the two symmetrical solutions can be merged into one. Therefore, if solutions are represented as paths in a graph,

will having fewer solutions to represent always lead to a smaller graph? In other words, if we break symmetries in the CSP, will its compiled representation always be smaller? If not, what is the relationship between these and for what classes of problems and symmetries? Removing symmetries often reduces computational overhead while solving, is this the same for compilation?

To answer these questions we considered some simple CSPs such as a clique of not-equal constraints, N-queens, [1] Graceful Graph Labelling [2] and Balanced Incomplete Block Designs. [3] We introduce symmetry breaking constraints to these models and compile them into four different representations using four different tools. We chose to look at the effect of symmetry breaking across a variety of representations to show that our method is representation and tool agnostic. We defined the representations we explore in Section 2.3 (page 29). We chose the problems mentioned above as they have been heavily researched from the point of view of symmetry breaking. The symmetry breaking constraints we use for each problem have been well defined previously.

The utility of symmetry-breaking from an approximate compilation perspective is that symmetric solutions could be reconstructed later. Since we know the size of a symmetry group, that is how many solutions are symmetrical, when we eliminate the symmetries we know how many solutions have been eliminated. In case more than one symmetry group exists there might be an overlap between solutions, but still we know at most how many solutions got eliminated. With this, we can calculate a good estimate of the number of solutions to the initial problem before eliminating symmetrical solutions. We do not focus on reconstructing the solutions but in most cases it could be done. Often with symmetry breaking, implied constraints are also added to speed up search. In general, adding more constraints to the problem that do not restrict the solution set, such as implied constraints, could negatively or positively affect the size of the compilation. These could lead to different variable orderings, which highly affect the size of the compilation. We did not explore the effects of such extra constraints on compilation.

Example 3.1. $X_1 + X_2 + X_3 = X_4$ *such that* $X_1, X_2, X_3 \in \{1, 3\}$ *and* $X_4 \in \{7, 9\}$

We illustrate our motivation with Example 3.1. Table 3.1 presents all solutions to Example 3.1. The second and third rows denote the possible values for each variable and the literal we associate to that assignment. We can see that S_1,

[1] https://www.csplib.org/Problems/prob054/
[2] https://www.csplib.org/Problems/prob053/
[3] https://www.csplib.org/Problems/prob028/

Table 3.1: All solutions for Example 3.1

	X_1		X_2		X_3		X_4		Solution
	=1	=3	=1	=3	=1	=3	=7	=9	
	A	B	C	D	E	F	G	H	
S1	1	0	0	1	0	1	1	0	1+3+3=7
S2	0	1	1	0	0	1	1	0	3+1+3=7
S3	0	1	0	1	1	0	1	0	3+3+1=7
S4	0	1	0	1	0	1	0	1	3+3+3=9

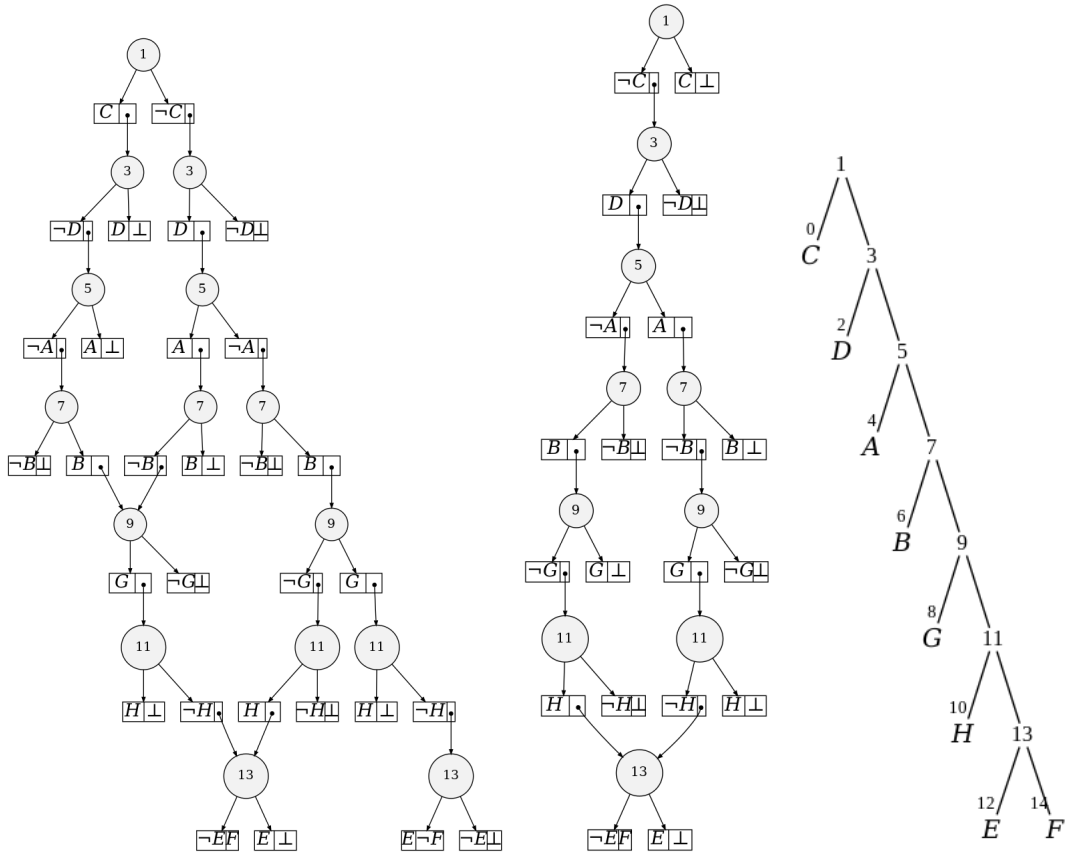
S_2 and S_3 are symmetrical solutions, the same values are used just in different ordering. We define Example 3.2 as an extension to Example 3.1 by adding two symmetry breaking constraints: $X_1 \leq X_2$, $X_2 \leq X_3$. These additional constraints in Example 3.2 break this symmetry and leave only $S1$, one of the three solutions, valid.

Example 3.2. $X_1 + X_2 + X_3 = X_4$, $X_1 \leq X_2$, $X_2 \leq X_3$ *such that* $X_1, X_2, X_3 \in \{1, 3\}$ *and* $X_4 \in \{7, 9\}$

Figure 3.1 shows the two compiled representations for the model with and without symmetry breaking. We compiled the problems to a widely used representation, the Sentential Decision Diagram (SDD). This is described in Section 2.3.2. Figure 3.1 contains two SDDs: Figure 3.1a represents Example 3.1 and Figure 3.1b shows the compilation with added symmetry breaking, Example 3.2. The number of nodes is equal to the number of OR-nodes and the size of the SDD is equal to the number of AND-nodes, 15 and 30 respectively in Figure 3.1a. Using the same vtree, we can easily observe which paths in the SDD get eliminated when adding symmetry breaking, that is, which symmetrical solutions are removed. Visually looking at the compilations, we notice that a large subtree can be eliminated with symmetry breaking. The difference points to the effects of symmetry breaking on the size of a representation. We notice that the size of the representation reduces from 30 to 20, with eliminating two symmetrical solutions. The idea is that this behaviour would repeat on a larger scale as well. The subtrees of the decision diagram that are common between symmetrical solutions, introduce repetitiveness, when we break the symmetry we eliminate this.

Related works

There has been some research in the area of combining symmetry breaking and compilation, but with a different focus, they often change the compiled representation or the tools, whereas in our proposed approach, we only change the input



(a) SDD of Example 3.1, no symmetry breaking

(b) SDD of Example 3.2, containing symmetry breaking

(c) vtree used

Figure 3.1: SDD of a CSP with and without symmetry breaking

formula to compile. Previous approaches consider dynamic symmetry breaking during compilation [BKLM14] or changing the compiled representation to deal with symmetry [NDN20]. Methods to reduce compilation size have been explored through the context of modifying compilation languages to represent symmetries more efficiently. Dynamic symmetry breaking during compilation has been explored in [BKLM14] by extending the languages FBDD and DDG to Sym-FBDD and Sym-DDG respectively. The Variable Shift SDD (VS-SDD) [NDN20] was introduced to exploit variable substitution by merging subtrees which represent the same formula but with different literals. This is obtained by modifying the vtree such that each node is assigned an ID and thus VS-SDD has an offset of k associated to it. Vtree nodes expressing the same Boolean formulas are merged into one and the offset is used to associate the different literals to this. In [FMS22] they introduce an extension to the OMDD representation with the aim to reduce the size of the representation. They extract some information about some of the

variables and store this in a separate formula, these formulas forming a satellite structure around the main formula, the nucleus. The satellites are connected to the nucleus by a unique variable instantiation. The effect of symmetry breaking at CNF level and at domain level with automated tools and manual formulation has been studied on model counting [WUA+20]. Tools used for automatic symmetry breaking in SAT problems create auxiliary variables while breaking symmetries, thus for our setup using these tools is difficult. Our hypothesis, that sub-graphs that occur multiple times due to symmetries would be merged, does not apply in a model where we reformulate the problem with additional literals. The lack of automated static symmetry breaking tools on SAT level, which would recognise symmetries and impose constraints to break them without introducing new variables, makes it difficult to test our hypothesis on any arbitrary SAT benchmark. Also no CNF benchmarks exist with formulations of models with and without symmetry breaking. This leaves us with CSPs where we manually formulate symmetry breaking constraints and express the problems in CNF in order to be compiled and compared. When dealing with symmetry formally, one often uses definitions from group theory [Mic06]. The way we break symmetry is ad-hoc, other options from group theory could have been used as well.

We not only try to obtain smaller compilations by merging symmetrical solutions but by also eliminating them. Since symmetry breaking speeds up solving and some of the compilers are based on the solver's traces, the speed of compiling with the presence of symmetry breaking might also be faster. Detaching dealing with symmetry breaking from the specific compiler implementation allows us to test the technique on multiple compilers. This could also initiate a debate on where it is best to deal with symmetries, during modelling, on the compiled representation definition or the compiler tools themselves. Do all compilers and all the representations behave similarly when symmetry is broken or is it unique to the compiler? If we lift symmetry breaking out of the compiler implementation we can reuse the same problem formulation with different tools and representations, finding the most suitable one. By looking at symmetry breaking as a problem property and not a compiler capability we benefit from the improvements in the regular solvers and make it more general use. This way the choice of how to deal with symmetry if it exists in the problem is moved to the modelling phase rather than at the phase of choosing compilation representation/tool.

3.3 Methodology

We consider four compilers: PySDD, [4] ZSDD, [5] D4, [6] and MiniC2D [7] and their respective representations. The four tools compile to the following representations respectively: Sentential Decision Diagram (SDD) defined in Section 2.3.2, Zero-Suppressed Sentential Decision Diagram (ZSDD) defined in Section 2.3.3, Decision-Disjunctive Negational Normal Form (d-DNNF) defined in Section 2.3.4 and Decision-SDD defined in Section 2.3.5. Keep in mind, when we refer to the different compilers we also refer to the different representations to which they compile. For all experiments, a machine was used with 11th Gen Intel Core i7-1165G7 @ 2.80GHz × 8 running Ubuntu 20.04.3 and the same specification with Ubuntu 22.04. The main comparisons we do are between instances with and without symmetry breaking, for each pair the same machine was used. We used the command line version of each tool, with their default parameters and a 30 minute timeout. For consistent time measurement we used the Linux command `time`.

We considered four problems, each described in the following sections along with the CSP for each of the problems. We used the direct encoding defined at Definition 2.1.3 (page 20) to formulate the problems as CNF. The benchmark is available online.[8] For all problems we have two CNFs, one with symmetry breaking constraints and one without. We compile these separately with each tool and compare the initial compilation with the partial compilation. By initial compilation we mean the model without symmetry breaking and by partial compilation we mean the model with added symmetry breaking. The focus is on comparing the initial compilation with the partial compilation for each representation separately, not necessarily comparing the tools to each other. As each tool compiles to a different representation they will be useful in different scenarios, thus our aim is to provide a method for a partial compilation that works across a range of representations. We run each compilation five times and the results shown are the average results about time and size. PySDD used by default a balanced initial vtree and post-compilation vtree search. MiniC2D is the only compiler that is not deterministic, as its default parameters include a random balance factor for constructing a vtree. In each of the following experiments, we looked at compila-

[4] `https://github.com/wannesm/PySDD`
[5] `https://github. com/nsnmsak/zsdd/`
[6] `https://www.cril.univ-artois.fr/KC/d4.html`
[7] `http://reasoning.cs.ucla.edu/minic2d/`
[8] `https://github.com/baloghAndi/SB_and_KC_CNF_benchmark`

tion time and size, defined by node count. We use the terms size and node count interchangeably, as well as solution and model count.

In the rest of this chapter, we represent each use case separately by describing the problem, defining the CSP model and the symmetry breaking constraints used. We give a small example for each problem and then present the results.

3.4 Cliques of Not-Equals

As a first use case, we reconstructed the alldifferent constraint using a clique of not-equal constraints with and without symmetry breaking. We created 11 CSP instances with 10 variables and pairwise not-equal constraints. The domains of the 10 variables within an instance are the same. Each instance represents a different domain size, ranging between 10-20. For each of the 11 instances, we created an additional instance where symmetry breaking is added. Thus, in total, we looked at 22 instances.

3.4.1 Model

In Problem 3.1 we describe the CSP model and give an example instantiation of it in Example 3.3.

Problem 3.1 (Cliques of Not-Equals)**.** *The problem is defined as a CSP on:*

- *the set of n variables* $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$,
- *with the set domains* $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ *such that* $D_i = \{1, 2, \ldots, m\}$,
- *constraints :* $\forall i \neq j\ ,\ X_i \neq X_j$,
- *added symmetry breaking constraint:* $X_1 < X_2 < X_3 < ... < X_n$.

Example 3.3 (Cliques of Not-Equals)**.** *As an example we instantiate the Cliques of Not-Equals problem on* $n = 3$ *variables such that each domain has* $m = 3$ *values.*

- *variables:* $\mathcal{X} = \{X_1, X_2, X_3\}$,
- *domains:* $\mathcal{D} = \{D_1, D_2, D_n\}$ *such that* $D_1 = D_2 = D_3 = \{1, 2, 3\}$,
- *constraints:*

$$X_1 \neq X_2, \qquad X_1 \neq X_3, \qquad X_2 \neq X_3,$$

- *added symmetry breaking constraint:* $X_1 < X_2 < X_3$.

All solutions without symmetry breaking to Example 3.3 are shown in Table 3.2. The solution S_1 is the only assignment that remains valid after the defined symmetry breaking constraint is added. In case $n = m$ instances with symmetry breaking will have a single solution, that is the solution $S = \{1, 2, \ldots, n\}$.

Table 3.2: All solutions to Example 3.3, solution in bold is the only one valid after breaking the symmetry.

	X1	X2	X3
S1	**1**	**2**	**3**
S2	1	3	2
S3	2	1	3
S4	2	3	1
S5	3	1	2
S6	3	2	1

To express Problem 3.1 in CNF we use the direct encoding (Definition 2.1.3) such that each variable value pair is assigned a Boolean variable where the positive literal expresses the assignment, and constraints are expressed on these. For instances in our dataset the number of literals ranges from 100 to 200, and the number of clauses from 910 to 11360.

3.4.2 Results

We ignore ZSDD, as compilation fails for all instances due to excessive memory use. No tool was able to compile the problem with domain size 18 without symmetry breaking, with only PySDD succeeding for domain sizes 19 and 20. This behavior of PySDD is quite strange, however might be due to a particularly bad variable ordering. All runs start with the default parameters, which might

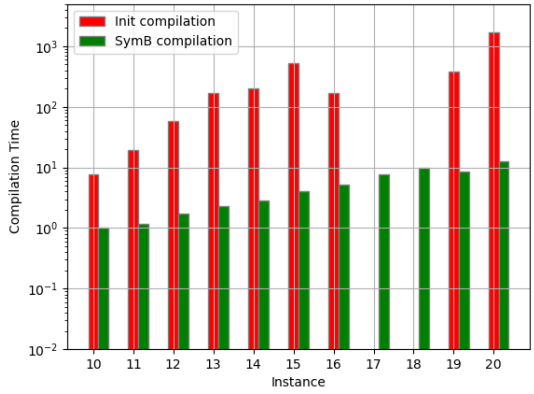
Table 3.3: Time measurements in seconds for compiling clique of not-equals

Domain	PySDD		MiniC2D		D4		Model Count	
size	Init	SymB	Init	SymB	Init	SymB	Init	SymB
10	7.60	1.02	19.03	0.46	19.15	0.05	3628800	1
11	19.35	1.18	16.15	0.49	69.02	0.03	39916800	11
12	57.68	1.74	32.78	0.50	169.28	0.03	239500800	66
13	169.42	2.29	30.49	0.64	369.00	0.04	1037836800	286
14	199.44	2.80	65.95	0.68	652.62	0.05	3632428800	1001
15	531.12	4.10	85.47	0.80	1019.86	0.05	10897286400	3003
16	170.63	5.20	139.00	1.00	1321.55	0.06	29059430400	8008
17	-	7.58	141.09	1.20	-	0.07	70572902400	19448
18	-	9.91	-	1.52	-	0.08	-	43758
19	376.82	8.45	-	1.66	-	0.09	335221286400	92378
20	1685.89	12.60	-	2.19	-	0.11	670442572800	184756

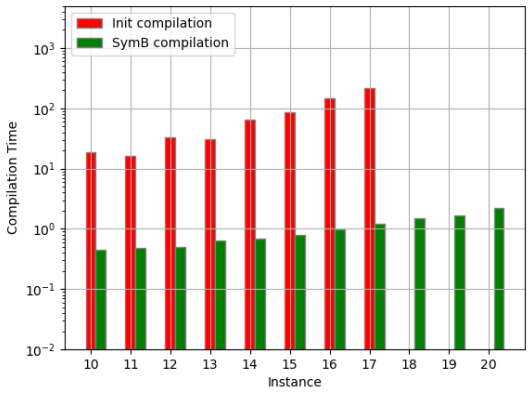
Table 3.4: Size measurements for compiling clique of not-equals

Domain size	PySDD		MiniC2D		D4		Model Count	
	Init	SymB	Init	SymB	Init	SymB	Init	SymB
10	19768	272	668743.4	199	2616751	90	3628800	1
11	41929	441	2238849.8	439.2	9260088	313	39916800	11
12	81518	653	3320898.6	622.6	22268278	498	239500800	66
13	187757	826	7516849.8	1075.4	50075258	746	1037836800	286
14	195015	1030	16547016.4	3632.4	91864534	1020	3632428800	1001
15	899580	1301	26786678.8	16621.8	151085371	1487	10897286400	3003
16	307686	1409	44592981	36275.8	246739107	1945	29059430400	8008
17	-	1656	33779490	47154.8	-	2933	70572902400	19448
18	-	2041	-	74144.8	-	3570	-	43758
19	897798	2366	-	98443.4	-	4193	335221286400	92378
20	5208404	2522	-	257089.6	-	5789	670442572800	184756

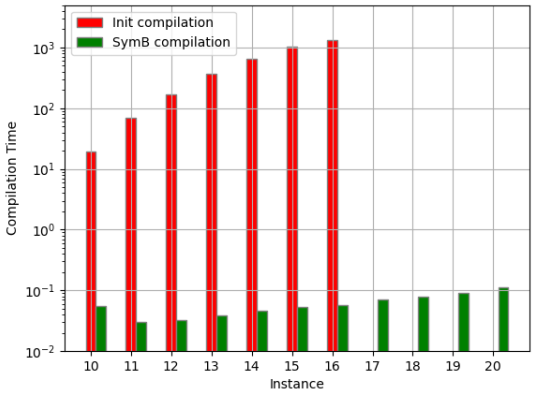
be providing a particularly unsuitable vtree for the two failing instances (17,18), causing them to time out. Even for their respective problems with symmetry breaking, the compilation seems to be slower than expected. There is a larger gap between instances with domain sizes 16 and 17, and compilation of domain size 18 takes even more time than that of 19.



(a) PySDD



(b) MiniC2D



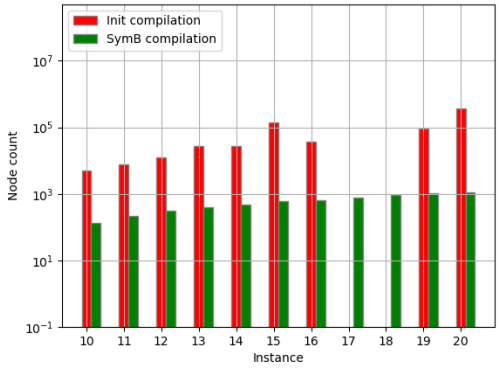
(c) D4

Figure 3.2: Compilation time for each tool for the initial problem and with symmetry breaking for the Clique of not-equals problem

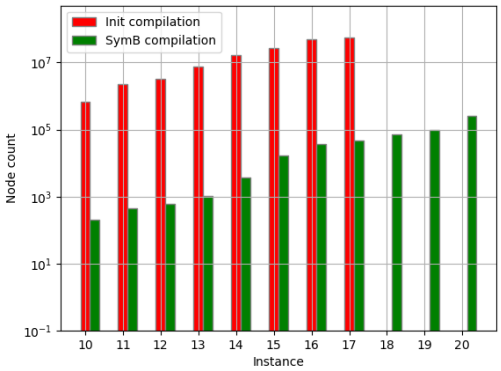
Table 3.3 presents the results with and without symmetry breaking, with time expressed in seconds. Table 3.4 shows the size of the compilations for each tool with and without symmetry breaking. In both tables, the column "Init" refers to the compilation without symmetry breaking and the column "SymB" to the compilation with symmetry breaking.

For all instances there is a significant reduction both in compilation size and time that is also proportional to the reduction in solutions. In larger instances, without symmetry breaking the compilers time out, whereas with symmetry breaking, these models compile within a few seconds.

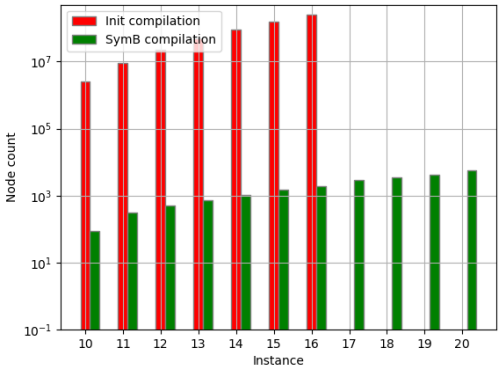
Figure 3.2 shows a bar plot for the compilation time of each instance. Figure 3.3 shows a bar plot for the node count of each instance. In both figures instances are ordered as in Table 3.4 and missing bars denote that the compilation timed out. The difference between the two coloured bars highlights the difference between compilations of instances with and without symmetry breaking.



(a) PySDD



(b) MiniC2D



(c) D4

Figure 3.3: Number of nodes for each tool for the initial problem and with symmetry breaking for the Clique of not-equals problem

3.5 N-queens Problems

We consider the N-queens problem: the problem of placing N queens on an $N \times N$ chessboard such that they do not attack each other. The 8-queens problem was first formulated in 1848 by the chess player Max Bazzel and gained huge attention from the constraint programming community [RvBW06]. This problem is very important when looking at symmetry as it contains multiple different types. We look at N-queens problems with $N = \{4, 5, \ldots, 15\}$, that is 12 instances without symmetry breaking and the respective 12 instances with added symmetry breaking. In total we evaluate the behaviour of 24 instances on the four compilation tools with their respective representations.

3.5.1 Model

The CSP is defined by N variables, each representing a row on a chessboard and a set of not-equal constraints restricting the queens from attacking each other on rows, columns and diagonals. We define symmetry breaking constraints in a way that needs no new variables. We break horizontal and vertical symmetries by imposing the constraints that the first queen has to be on the left half of the board and on the left hand side of the last queen. As previously we define the CSP model in Problem 3.2 and describe a small example in Example 3.4.

Problem 3.2 (N-queens)**.** *We define the problem as a CSP:*

- *on the set of n variables* $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ *each representing a row on the chessboard,*
- *with the set of domains* $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ *such that* $D_i = \{1, 2, \ldots, n\}$*. Each domain is equivalent, representing the index of the columns where a queen could be placed.*
- *The constraints impose that each column has only one queen by adding an alldifferent constraint between all variables. Two additional alldifferent constraints are added on the set of variables* $\{X_i + i, \forall i\}$ *and separately on* $\{X_i - i, \forall i\}$ *to restrict diagonal attacks.*
- *Symmetry breaking constraints restrict the first queen to be:*
  - *on the left hand side of the board:* $X_1 < (N + 1)/2$*,*
  - *on the left hand side of the last queen:* $X_1 < X_n$*,*

– *at least as close to the top left corner than any queen from a chessboard's edge is to any corner:*

* $\forall 1 < i \leq (N+1)/2, (Q_i \neq 1 \vee Q_1 < i)$,
* $\forall (N+1)/2 < i \leq N, (Q_i \neq 1 \vee Q_1 \leq N - i + 1)$,
* $Q_N \leq N - Q_1 + 1$,
* $(Q_1 \neq 1) \vee (\forall 2 < i \leq N, Q_i \neq 2 \vee Q_2 < i)$.

Example 3.4 (N-queens)**.** *As an example, we instantiate the N-queens problem with $N = 4$ variables, that is on a 4×4 chess board 4 queens must be placed such that they do not attack each other.*

- *variables:* $\mathcal{X} = \{Q_1, Q_2, Q_3, Q_4\}$,
- *domains:* $\mathcal{D} = \{D_1, D_2, D_3, D_4\}$ *such that* $D_1 = D_2 = D_3 = D_4 = \{1, 2, 3, 4\}$,
- *constraints:*

  alldifferent$([Q_1, Q_2, Q_3, Q_4])$

  alldifferent$([Q_1 + 1, Q_2 + 2, Q_3 + 3, Q_4 + 4])$

  alldifferent$([Q_1 - 1, Q_2 - 2, Q_3 - 3, Q_4 - 4])$,
- *symmetry breaking constraints:*
  – $Q_1 < 2.5$,
  – $Q_1 < Q_4$,
  – $Q_2 \neq 1 \vee Q_1 < 2$,
  – $Q_3 \neq 1 \vee Q_1 \leq 2$ *and* $Q_4 \neq 1 \vee Q_1 \leq 1$,
  – $Q_4 \leq 4 - Q_1 + 1$,
  – $(Q_1 \neq 1) \vee (Q_3 \neq 2 \vee Q_2 < 3)$ *and* $(Q_1 \neq 1) \vee (Q_4 \neq 2 \vee Q_2 < 4)$.

Figure 3.6 illustrates the two solutions (S1, S2) to the 4-queen problem. The N-queens problem has the symmetries of the square. If we mirror the solution on the right side by the vertical middle axis we obtain the solution on the left. This is the symmetry that can be broken by adding constraints that make sure the queen on the first row is on the left side of the chessboard. The N-queens problem is very important in the field of symmetry because it contains a lot of types of symmetries. Variable symmetry is present through the reflection around

Figure 3.6: All solutions to the 4-queens problem, with the solution on the left $S1 : Q_1 = 2, Q_2 = 4, Q_3 = 1, Q_4 = 3$ and solution on the right $S2 : Q_1 = 3, Q_2 = 1, Q_3 = 4, Q_4 = 2$

the horizontal axis through the centre of the board, similarly value symmetry is present through the vertical axis reflection, and variable-value symmetry maps every $X_i = j$ to $X_j = i$, that is the reflection around the top-left and bottom-right diagonal. In this example, S1 will be the unique solution according to the symmetry breaking constraints we add.

Expressing Example 3.4 in CNF direct encoding is similar to Example 3.3. Each variable value pair is assigned a positive literal. The negation of the literal means that value is not assigned to the variable. At-most-one and At-least-one clauses are used to make sure each variable is assigned a single value. The *alldifferent* constraints from Example 3.4 are expressed as pairwise not equal clauses in the CNF. The symmetry breaking clauses are expressed with the use of conflict clauses. Our instances contain 16 to 144 literals and from 80 to 5689 clauses.

3.5.2 Results

As previously the aim is to compare the initial (without symmetry breaking) and partial (with symmetry breaking) compilations. Table 3.5 shows time and Table 3.6 the node count of each compiler, without (columns "Init") and with symmetry breaking (columns "SymB"). Around $50 - 80\%$ of the solutions get

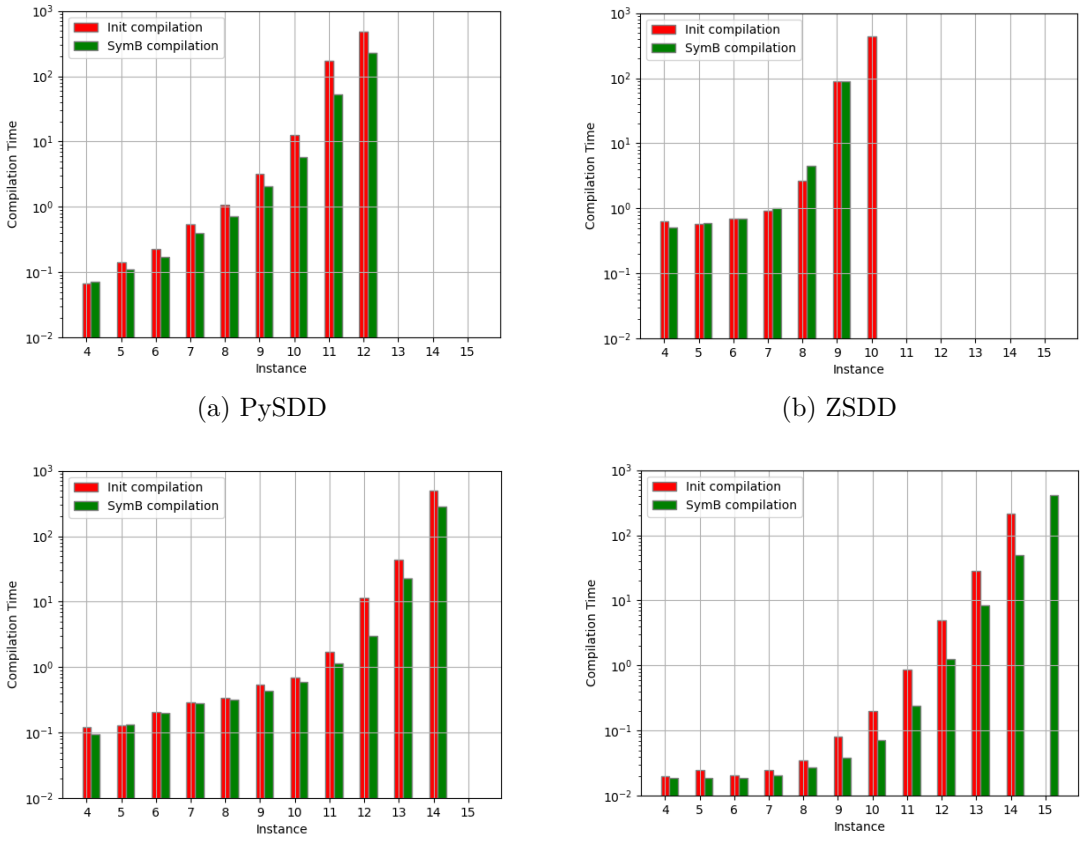
Table 3.5: Time measurements in seconds for compiling the N-queens problem

N	PySDD		ZSDD		MiniC2D		D4		Model Count	
	Init	SymB	Init	SymB	Init	SymB	Init	SymB	Init	SymB
4	0.068	0.071	0.637	0.507	0.121	0.096	0.020	0.019	2	1
5	0.141	0.112	0.579	0.588	0.131	0.132	0.025	0.019	10	2
6	0.226	0.169	0.694	0.705	0.205	0.203	0.020	0.019	4	1
7	0.547	0.394	0.925	1.005	0.294	0.285	0.025	0.021	40	8
8	1.087	0.721	2.625	4.469	0.343	0.323	0.035	0.027	92	20
9	3.237	2.066	89.539	89.293	0.539	0.443	0.080	0.038	352	68
10	12.554	5.772	434.801	-	0.689	0.597	0.199	0.072	724	130
11	171.494	53.345	-	-	1.719	1.129	0.870	0.245	2680	521
12	477.989	229.246	-	-	11.533	3.014	4.947	1.252	14200	2688
13	-	-	-	-	44.106	22.947	28.706	8.467	73712	13751
14	-	-	-	-	504.559	287.593	216.923	49.299	365596	67410
15	-	-	-	-	-	-	-	415.412	-	419515

Table 3.6: Size measurements for compiling the N-queens problem

N	PySDD		ZSDD		MiniC2D		D4		Model Count	
	Init	SymB	Init	SymB	Init	SymB	Init	SymB	Init	SymB
4	35	18	7	3	53.4	31	35	16	2	1
5	154	44	44	9	238.6	80	204	47	10	2
6	117	51	23	5	185.4	71	150	35	4	1
7	571	216	218	44	1381.8	394.6	1240	250	40	8
8	976	455	456	119	3616.6	1046.8	3308	826	92	20
9	2229	998	1651	403	14909.4	3802.4	13076	3092	352	68
10	4032	1686	3833	-	38220.4	8582.8	32986	7350	724	130
11	10855	4093	-	-	160396.4	36070	128993	30151	2680	521
12	35047	12265	-	-	876562	200923.4	693163	152828	14200	2688
13	-	-	-	-	4648915.8	1058456.6	3556955	810053	73712	13751
14	-	-	-	-	24793661.2	5696624.6	18161401	3842090	365596	67410
15	-	-	-	-	-	-	-	24283307	-	419515

eliminated when breaking symmetry. We do not break all the combinations of rotation and flipping symmetries for each instance. The number of unique solutions, without symmetry, can be found under the sequence A002562 in the OEIS [9].



(a) PySDD (b) ZSDD

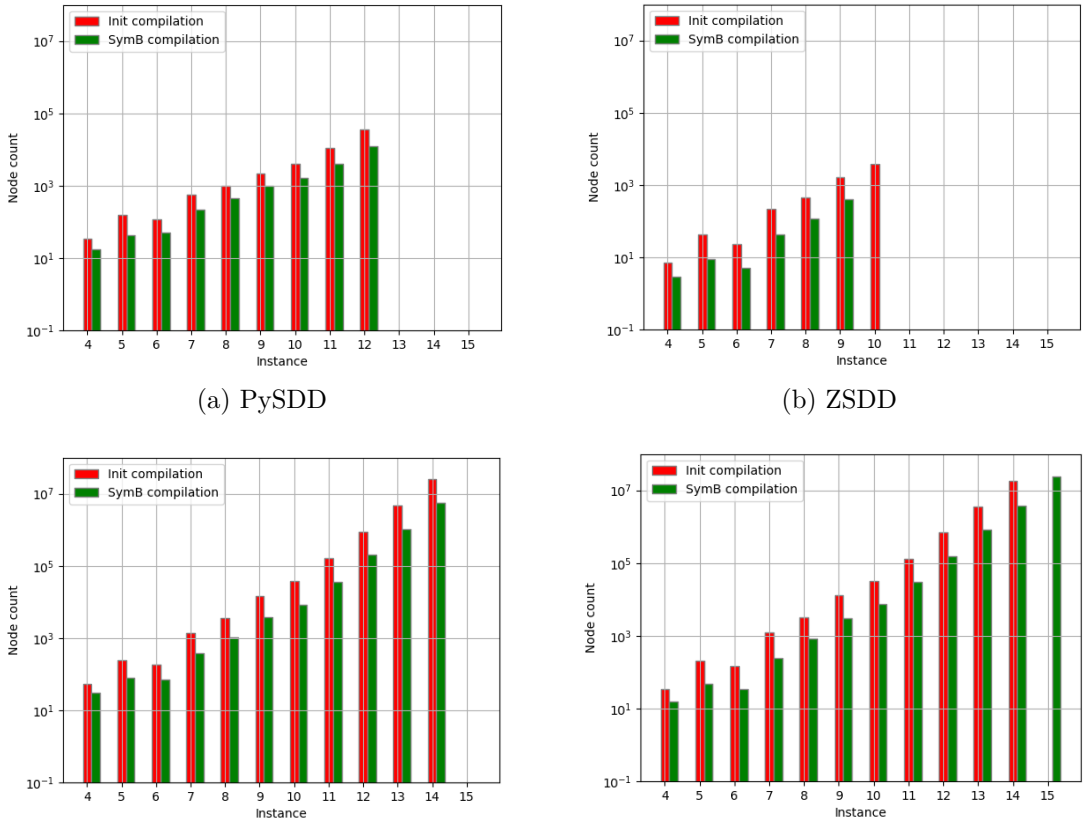
(c) MiniC2D (d) D4

Figure 3.7: Compilation time for each tool for the initial problem and with symmetry breaking for the N-queens problem.

[9] `https://oeis.org/A002562`

For the 10-queens problem, ZSDD compiles in only 7.24 minutes without symmetry breaking but when it is added, instead of 1480 we have 1686 clauses, compilation times out within 30 minutes. For 15-queens, only D4 was able to compile the formulation with symmetry breaking within roughly 7 minutes. The top-down compilers perform better, D4 obtains the most reduction both in time and node count, MiniC2D following it for node count. ZSDD achieves good reduction but is only able to compile the first 7 instances, for N=9,10 it is significantly slower than other tools.

Figure 3.7 shows the compilation time for each tool, comparing the time needed to compile the initial problem, without symmetry breaking and the one with added symmetry breaking constraints. Figure 3.8 shows the number of nodes for each instance with and without symmetry breaking. Missing bars in both plots indicate that the compilation timed out.



(a) PySDD (b) ZSDD

(c) MiniC2D (d) D4

Figure 3.8: Number of nodes for each tool for the initial problem and with symmetry breaking for the N-queens problem.

VS-SDD [NDN20] also look at the N-queens problem with N=8,9,10,11. The size of the compiled SDDs differs, which can be caused by a different order of the CNF

clauses. They report the ratio between VS-SDD size and SDD size to be 73.1%, 85.8%, 88.5% and 94.3% for N=8,9,10 and 11 respectively. Our compilation using PySDD and symmetry breaking gets 44.95%, 41.86%, 38.76% and 34.79%. According to this, our method achieves more reduction but we obtain a different size for initial compilation and it is not clear what level of symmetry breaking is achieved in [NDN20]. Note that these percentages compare the size, not the node counts presented in Table 3.6.

3.6 Graceful Graph Labelling

We consider the graceful labelling of graphs [Pet]. It is important to note that not all graphs can be labelled gracefully. Given a graph with n nodes($\{N_1, \ldots, N_n\}$) and q edges, the labelling of the nodes is graceful if each node has a unique label from $\{0, 1, \ldots, q\}$ and each edge, $E_{i,j}$ connecting nodes N_i and N_j, is labelled with the absolute difference between the labels of N_i and N_j. Both node and edge labels have to be unique within node and edge labels respectively.

There are two types of symmetries in this problem. Firstly the graph itself can have symmetry. For example if a graph is a clique any permutation of the node labels will be a valid solution. If the graph is a path, the node labels can be reversed. Secondly, a symmetry exists from the fact that we can replace the value of node label N_i with its complement $q - N_i$ and still obtain a valid solution. These can also be combined. For our experiments we take into consideration a few well known graphs for graceful labelling from the literature [PS03a, Gal00]. The list of the graphs we looked at and the symmetry breaking constraints we added to them can be found in Figure 3.11. We took graph examples from the classes of path graphs P_i, star graphs S_i, circular graphs C_i, wheel graphs W_i. Some instances from these classes have been proven to have a graceful labelling, and thus we focus on these. In [PS03b] the authors describe symmetry breaking for some of the used graphs.

3.6.1 Model

Problem 3.3 (Graceful Graph Labelling)**.** *We define the Graceful Graph Labelling CSP on a graph G with n nodes and q edges:*

- *variables denote the labels for the set of nodes $\mathcal{X} = \{N_1, N_2, \ldots N_n\}$ and labels for the set of edges $\mathcal{Y} = \{E_{i,j} |$ such that N_i and N_j are connected by an edge in graph $G\}$,*

- *domains:* $\mathcal{D}_X = \{D_{N_1}, D_{N_2}, \ldots, D_{N_n}\}$ *such that* $D_{N_1} = D_{N_2} = \ldots D_{N_n} = \{0, 1, \ldots, q\}$ *and* $\mathcal{D}_Y = \{D_{E_{i,j}} | \forall E_{i,j} \in \mathcal{Y}\}$ *where* $D_{E_{i,j}} = \{1, 2, \ldots, q\}$,
- *constraints:*

  $alldifferent([N_1, N_2, \ldots N_n])$

  $E_{i,j} = |N_i - N_j|$

  $alldifferent([E_{i,j} | \forall E_{i,j} \in \mathcal{Y}])$.

Example 3.5 (Graceful Graph Labelling)**.** *As an example, we instantiate the Graceful Graph Labelling problem with* $q = 2$ *and* $n = 3$ *variables using the graph found in Figure 3.9. Solutions to this problem are visualised in Figure 3.10.*

- *variables:* $\mathcal{X} = \{N_1, N_2, N_3\}$ $\mathcal{Y} = \{E_{1,2}, E_{2,3}\}$,
- *domains:* $\mathcal{D}_X = \{D_{N_1}, D_{N_2}, D_{N_3}\}$ *such that* $D_{N_1} = D_{N_2} = D_{N_3} = \{0, 1, 2\}$ *and* $\mathcal{D}_Y = \{D_{E_{1,2}}, D_{E_{2,3}}\}$ *such that* $D_{E_{1,2}} = D_{E_{2,3}} = \{1, 2\}$,
- *constraints:*

  $alldifferent([N_1, N_2, N_3])$

  $E_{1,2} = |N_1 - N_2|$

  $E_{2,3} = |N_2 - N_3|$

  $alldifferent([E_{1,2}, E_{2,3}])$,
- *symmetry breaking:* $N_1 < N_3$ *and* $N_2 < N_3$.

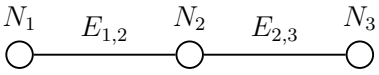


Figure 3.9: Example of a path graph P_3 with 3 nodes and 2 edges.

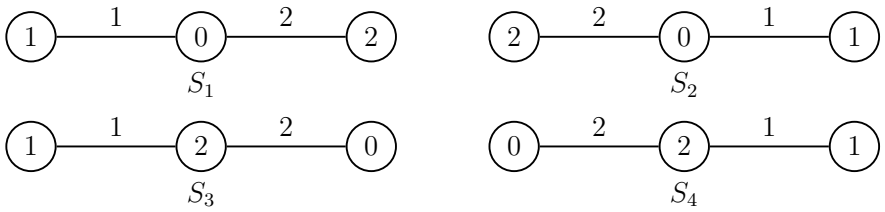
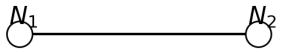


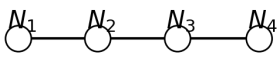
Figure 3.10: All solutions to graceful labelling of P_3 graph shown in Figure 3.9

Figure 3.10 presents the four solutions P_3 graph can be labelled according to the Graceful labelling constraints. We break the symmetry by adding constraints to order $N_1 < N_3$ and $N_2 < N_3$. After symmetry breaking only S_1 remains a valid solution.

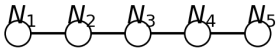


(a) $P_2 : N_1 < N_2$

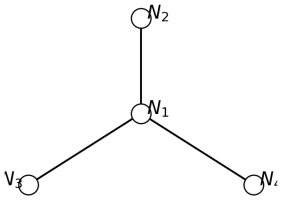
(b) $P_3 : N_1 < N_3$, $N_2 < N_3$



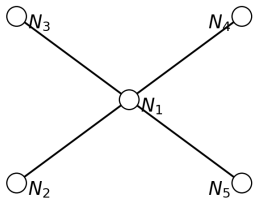
(c) $P_4 : N_1 < N_3$, $N_1 < N_4$



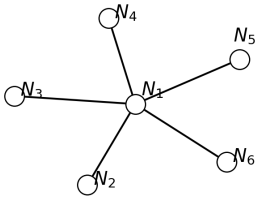
(d) $P_5 : N_1 < N_5$, $N_4 < N_5$, $N_2 < N_1$



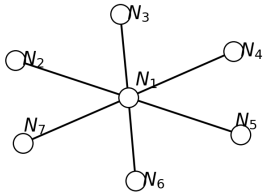
(e) $S_4 : N_1 = 0$, $N_2 < N_3 < N_4$



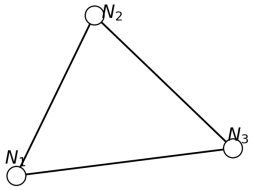
(f) $S_5 : N_1 = 0$, $N_2 < N_3 < N_4 < N_5$



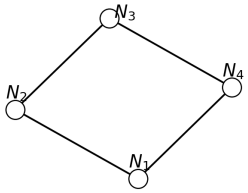
(g) $S_6 : N_1 = 0$, $N_2 < N_3 < N_4 < N_5 < N_6$



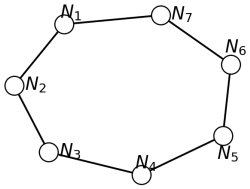
(h) $S_7 : N_1 = 0, N_2 < N_3 < N_4 < N_5 < N_6 < N_7$



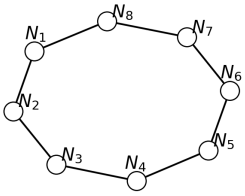
(i) $C_3 : N_1 < N_2 < N_3$



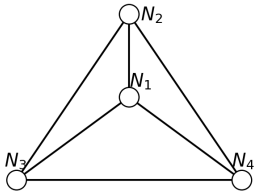
(j) $C_4 : N_1 = 0$, $N_2 < N_4$



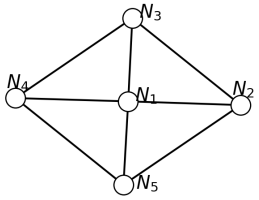
(k) $C_7 : N_1 = 0, N_2 = 7$



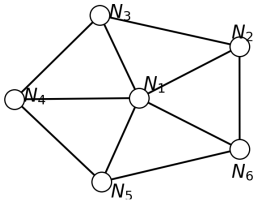
(l) $C_8 : N_1 = 0, N_2 = 8$



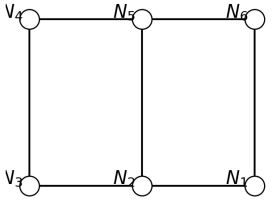
(m) $W_4 : N_1 = 0$, $N_2 < N_3 < N_4$



(n) $W_5 : N_1 \leq q/2, N_2 < N_3, N_2 < N_4, N_2 < N_5, N_3 < N_5$



(o) $W_6 : N_1 \leq q/2, N_2 < N_3, N_2 < N_4, N_2 < N_5, N_2 < N_6, N_3 < N_6$



(p) $G_{2,3} : N_1 < N_6, N_1 < N_3, N_1 < N_4$

Figure 3.11: Graphs used for graceful labelling along with the constraints used to break symmetries.

Fig. 3.11 presents the graphs we considered, as well as the constraints used to break symmetries. Similarly as before we use the direct encoding to transform the CSP to CNF. The number of literals for the instances we looked at ranges from 5 to 166 and clauses from 9 to 12467.

3.6.2 Results

Table 3.7 shows the reduction of time and Table 3.8 shows the reduction of node count when breaking symmetries. We use a similar notation as before for columns "Init" and "SymB" denoting the compilation without and with symmetry breaking respectively. PySDD seems to be one of the slower compilers but still has a 41.7% median time reduction when comparing initial compilations with the partial ones. ZSDD has very little time improvement, in most cases even taking longer to compile the problems with symmetry breaking. Similarly, MiniC2D has a very small time improvement, 3.5% median, but most instances compile within less than a second. D4 compiles the fastest, almost all instances under 1 second but still improving at a median of 45.4% reduction when breaking symmetry. Looking at total run time, it took around 41 minutes for PySDD to compile all graphs whereas for D4 only 3.7 seconds, for MiniC2D 37.3 seconds and for ZSDD 12.3 minutes. Similarly as before the top-down compilers are faster. Looking at node count reduction, they are more proportional to solution reduction. ZSDD failed to compile the instance W_6 with and without symmetry breaking as well,

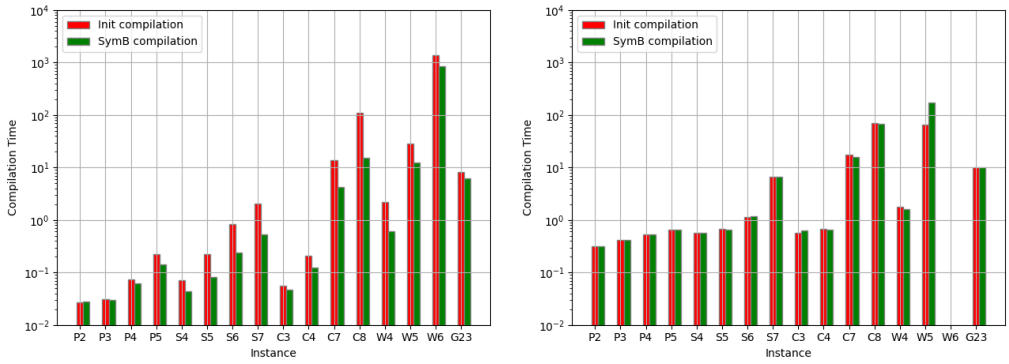
Table 3.7: Time measurements in seconds for compiling the Graceful labelling problem

Graph	PySDD		ZSDD		MiniC2D		D4		Model Count	
	Init	SymB	Init	SymB	Init	SymB	Init	SymB	Init	SymB
P_2	0.027	0.028	0.318	0.315	0.039	0.039	0.015	0.015	2	1
P_3	0.031	0.03	0.415	0.417	0.065	0.065	0.015	0.015	4	1
P_4	0.074	0.063	0.524	0.525	0.098	0.091	0.016	0.016	4	1
P_5	0.224	0.141	0.651	0.651	0.197	0.2	0.018	0.017	8	1
S_4	0.071	0.044	0.561	0.562	0.102	0.094	0.017	0.016	12	1
S_5	0.22	0.083	0.666	0.663	0.205	0.198	0.02	0.016	48	1
S_6	0.84	0.236	1.157	1.179	0.409	0.396	0.031	0.016	240	1
S_7	2.031	0.525	6.583	6.625	0.777	0.318	0.079	0.017	1440	1
C_3	0.056	0.047	0.575	0.635	0.064	0.072	0.018	0.016	12	2
C_4	0.212	0.124	0.669	0.661	0.161	0.17	0.018	0.017	16	2
C_7	13.996	4.301	17.346	15.577	1.095	0.334	0.135	0.029	168	12
C_8	111.896	15.518	69.596	67.614	3.040	0.696	0.511	0.042	384	24
W_4	2.232	0.603	1.805	1.588	0.251	0.217	0.033	0.018	48	2
W_5	28.498	12.424	65.864	171.528	1.927	2.202	0.276	0.117	64	4
W_6	1395.551	863.520	-	-	5.431	7.354	0.549	0.142	240	12
$G_{2,3}$	8.111	6.184	10.254	10.236	0.478	0.358	0.101	0.049	128	32

Table 3.8: Size measurements for compiling the Graceful labelling problem

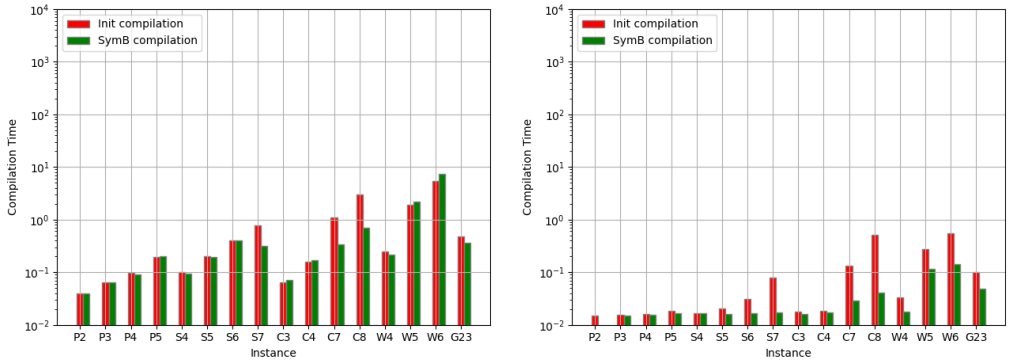
Graph	PySDD		ZSDD		MiniC2D		D4		Model Count	
	Init	SymB	Init	SymB	Init	SymB	Init	SymB	Init	SymB
P_2	9	6	5	2	17	9	9	0	2	1
P_3	32	17	15	4	67.2	25	46	14	4	1
P_4	80	28	23	6	130.8	49	99	18	4	1
P_5	201	58	54	8	338.4	81	331	51	8	1
S_4	124	35	55	6	269.2	49	208	17	12	1
S_5	398	50	231	8	992.8	81	915	29	48	1
S_6	1046	86	1211	10	4154.6	121	3720	49	240	1
S_7	2392	138	7183	12	34864	169	19194	71	1440	1
C_3	111	38	49	10	224.4	66.8	208	31	12	2
C_4	256	57	88	14	505.8	106.6	478	63	16	2
C_7	2161	525	1353	128	10861	1109.2	11618	728	168	12
C_8	3569	816	3344	284	30412	2568	33207	1741	384	24
W_4	650	137	291	18	1993.8	199.2	2267	88	48	2
W_5	1373	344	509	50	5146.2	560.8	5473	386	64	4
W_6	3835	937	-	-	30714	2027	30144	1737	240	12
$G_{2,3}$	1659	893	959	343	7657.8	2327	7659	1957	128	32

due to memory issues. Both tables also present the number of solutions for both the initial compilation and the one with symmetry breaking.



(a) PySDD

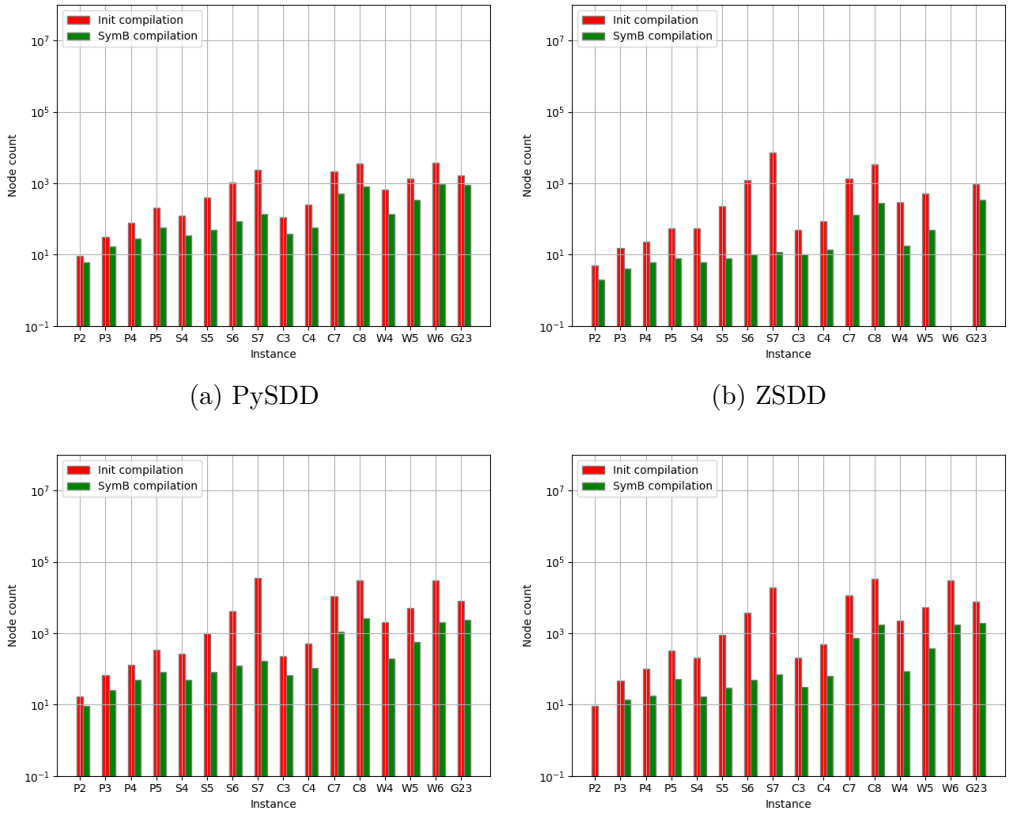
(b) ZSDD



(c) MiniC2D

(d) D4

Figure 3.12: Compilation time for each tool for the initial problem and with symmetry breaking for the Graceful Graph Labelling problem



(a) PySDD

(b) ZSDD

(c) MiniC2D

(d) D4

Figure 3.13: Node count for each tool for the initial problem and with symmetry breaking for the Graceful Graph Labelling problem

Figure 3.12 and Figure 3.13 show the compilation time and node count for each tool and instance with and without symmetry breaking.

3.7 Balanced Incomplete Block Designs

We looked at the Balanced Incomplete Block Design (BIBD) problem [MT01, Pre02, Pre]. It originated from design theory, initially used in the design of statistical experiments then also in network reliability and cryptography. A block design is defined by the parameters (v, b, r, k, λ). The problem looks at the arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks.

3.7.1 Model

A design can be represented by a $v \times b$ binary matrix where there is exactly r 1 values per row and k 1 values per column. Additionally, the scalar product of every pair of rows is λ. Three types of constraints have to be defined on this matrix:

- enforce r number of 1s for each row,
- enforce k number of 1s for each column,
- make sure each pair of rows scalar product is equal to λ.

A detailed model can be found in [Pre02]. They propose a binary CSP where variables are introduced, similar to the N-queens problem, to denote the index of row/column where a 1 or a 0 is placed. There are four sets of variables to represent this and an additional one to represent the λ values. Symmetry in this problem emerges from the fact that given any solution any two rows or columns can be exchanged to form another valid solution. In [Pre02], three levels of symmetry breaking are defined. The first level pairwise orders consecutive rows/columns in the five sets of variables. The second and third level uses implied constraints to enforce symmetry breaking. A solution has $v! \times b!$ symmetric equivalents: one for each permutation of the rows and or columns of the matrix.

We took the CNF formulations from CSPLib [Pre], the instance defined by (7,7,3,3,1) with and without symmetry breaking. The instance is expressed using 833 literals, 7028 clauses and 10080 clauses when symmetry breaking is added. We used a timeout of 30 minutes for compilations.

3.7.2 Results

Here too, the top-down compilers, D4 and MiniC2D achieve better results. Both compile within 30 minutes but only with symmetry breaking. ZSDD is terminated due to memory, whereas PySDD due to time. Table 3.9 gives a summary of the compilation results.

Table 3.9: Compilation of the BIBD instance (7,7,3,3,1), where - denotes no compilation due to timeout or memory

	PySDD		ZSDD		MiniC2D		D4		Model Count
	Time(s)	Nodes	Time(s)	Nodes	Time(s)	Nodes	Time(s)	Nodes	
No SymB	1800	-	67.4	-	1800	-	874.3	-	-
With SymB	1800	-	66.7	-	560.91	33578179	52.6	22528998	151200

3.8 Summary

In this chapter we examined the effects of symmetry breaking on some knowledge compilation representations. For this we considered four highly symmetrical problems and four different compilers, each using a different representation. In all instances we considered we obtained a smaller representation when removing symmetries. Most cases in significantly less time too. A few exceptions are for cases where the compilation time of an instance without symmetry breaking is within less than a second. In some cases, only instances with symmetry breaking were compiled due to time out or a memory limit reached. The percentage of reduction in size and in number of solutions are roughly the same. Ideally, we would hope for a greater reduction in the size than the number of solutions. This is not the case for most of the instances we considered, but the nature of this approach provides some exceptions. Firstly, knowing what symmetries we eliminated reconstructing the symmetrical solutions can be done without much overhead. Secondly, in many applications the symmetrical solutions are not only removed to speed up solving but also because from a problem perspective there is not much value in the symmetrical solution.

The approach presented and the results on a set of instances demonstrate that there is potential to obtain compact and efficient partial solutions by eliminating symmetrical solutions. However, due to the limitations we have seen perhaps a more general approach for reducing the number of solutions would lead to better partial compilation. The next two chapters explore other approaches of reducing the solutions represented in order to obtain compact partial compilations.

Chapter 4

Selective Backbones for Partial Compilation of CSPs

Abstract. *We explore the possibility of obtaining a partial compilation by restricting the set of solutions to represent using a few carefully selected variable assignments. We introduce the concept of the selective backbone as the set of such assignments. We describe a set of heuristic functions used to obtain selective backbones and evaluate them. This chapter is based on our paper from ECAI 2023: Partial Compilation of SAT Using Selective Backbones [BEO23].*

4.1 Introduction

In this chapter we explore another way of obtaining partial compilations of constraint problems. Previously, we used symmetry breaking constraints to restrict the solution set to be represented in the partial compilation. Similar to the previous chapter, this approach is inspired by a property of Constraint Satisfaction Problems (CSPs): backbones. The backbone of a CSP is a set of variables that is common for all solutions [MZK+99]. We relax this with an objective to obtain a set of assignments that is common for a large subset of the solutions. We call this the *selective backbone* of the problem. Using this we explore a few different heuristic methods to obtain such a set of assignments and evaluate the compilation of the problem where we fix these assignments.

We chose to relax the backbone of an instance as backbones have been widely used. A common application for backbones is to characterize the difficulty of a

problem [AGKS00, KSTW05, MZK+99, SW01], but they have also been previously used to reduce instance size [SFM+96]. Any variable that is represented in the backbone will take up little space in a compilation of the instance. Indeed, the only information that we need to keep about this variable is the one value that it takes in all solutions. Since all variables represented in a backbone can be completely documented within a space of linear size in their number, large backbones are highly desirable in the context of instance compilation. This is a notion that has some similarities with the notion of streamlining constraints for search [GS04] where constraints are added to characterise a subset of solutions that can be found efficiently, but which limits the solutions that can be found. Specifically, those solutions that are inconsistent with the streamlining constraints are no longer available. Streamlining is different from the approach we propose, as it was introduced to boost the effectiveness of propagation mechanisms. It is not clear how compilation size would be affected by streamlining.

Now that we defined the criteria that select the subset of solutions for compact partial compilation the next question is how to obtain this. For this we have to decide how many variables should be assigned and which ones. The more variables we fix the fewer possible solutions we have. To this end, we want to identify the assignments that optimise the partial compilation. The definition of the optimal partial compilation depends on what that compilation is going to be used for. In this work, we focus on optimising the partial compilation with respect to the percentage of information kept as well as the size of the representations. We use model counting (MC) the query on an instance about the number of solutions, to measure how much information is kept for a partial compilation. For now, we are only concerned by the number of solutions kept, assuming each has the same value of information. Later on in Chapter 5 (page 90) we explore the possibility where solutions have different importance.

Defining the global best partial compilation is difficult since there is a trade-off between how much information we keep and how large the size of the representation is, so which is more important is up to the user and the use case of the compilation. For this reason, we implemented an iterative heuristic approach and evaluated the progression of the partial compilations defined by the selective backbones.

Example 4.1. *We define a problem on variables:* $x_1 \in \{1, 2\}$, $x_2, x_3 \in \{1, 2, 3\}$ *with solutions in Table 4.1*

In Table 4.1 we observe that the assignment $x_1 = 2$ covers 9 out of 11 solutions.

Table 4.1: All solutions for a random CSP instance denoted by Example 4.1

	x_1		x_2			x_3		
	=1	=2	=1	=2	=3	=1	=2	=3
	A	B	C	D	E	F	G	H
S1	1	0	0	0	1	0	0	1
S2	1	0	0	1	0	0	0	1
S3	0	1	0	0	1	0	1	0
S4	0	1	0	0	1	0	0	1
S5	0	1	0	0	1	1	0	0
S6	0	1	0	1	0	0	1	0
S7	0	1	0	1	0	0	0	1
S8	0	1	0	1	0	1	0	0
S9	0	1	1	0	0	0	1	0
S10	0	1	1	0	0	0	0	1
S11	0	1	1	0	0	1	0	0

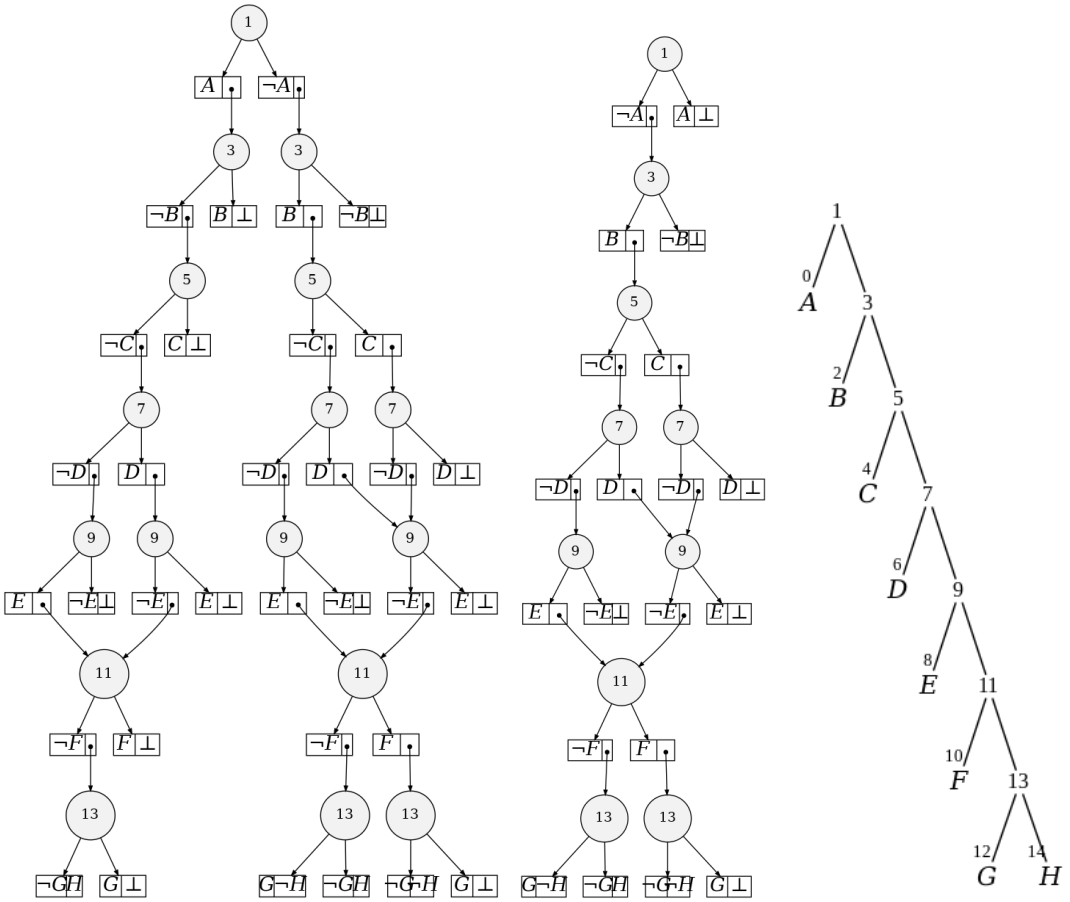
This makes this assignment a good candidate for a selective backbone. Figure 4.1a shows the compilation for all 11 solutions to a Sentential Decision Diagram (SDD). The SDD has a size of 34 and 17 nodes, where size denotes the number of AND nodes and the node count refers to the number of OR nodes. Using a right linear vtree as shown in Figure 4.1c leads to an SDD what is equivalent to a BDD. Figure 4.1b shows the SDD for the subset of solutions defined by selective backbone having a size of 20 and a node count of 10. We obtain the SDD in Figure 4.1b by restricting the variable x_1 to value 2. This only eliminates 2 solutions but with keeping the same vtree there is a size reduction from 34 to 20. Looking at Figure 4.1 the whole subtree of node 1 is eliminated.

With the above small hand-crafted example we showed that we can keep most of the solutions 9 out of 11 (81.81%) with a compilation that is significantly smaller, 20 compared to 34 (58.8%).

4.2 Related Work

Since one of the criteria when reducing the solution space is to keep the most solutions, we could categorize our approach of partial compilation as an approximate model counting technique. This is not the main purpose of it and we do not provide any guarantees of the approximation as other approaches do.

In [Lai18] they explore an extension of the Decision-DNNF as a partial compilation. They explore the possibility of randomly generating a Decision-DNNF and using these generated partial assignments as samples of the solution space.



(a) SDD of Example 4.1

(b) SDD of Example 4.1 with selective backbone

(c) vtree used

Figure 4.1: SDD of Example 4.1 and its reduction with a selective backbone

They use importance sampling and also consider some ISCAS instances for this. They introduce a new leaf vertex type, called unknown vertex, which represents a sub-formula defined by a partial assignment. The subformula represented by the unknown vertex is part of the problem that has not been compiled yet. They introduce an algorithm, PartialKC that is used for approximate model counting with the use of compiling the initial formula to a partial Decision-DNNF [Lai18]. We discuss more about approximate model counters in Section 2.2.2 (page 27). Before this, in [GD12] they explore a similar approach but using an AND/OR sample graph to approximate model count. The sampling method is also somewhat different. Our goal is not to define an approximate model counter but to obtain a compact partial compilation. The method defined in [Lai18] prunes subgraphs, but not necessarily all subgraphs of a variable, it acts more like ignoring constraints of the initial problem. Opposed to this we keep the integrity of the constraints but assign variables in order to limit the number of solutions

represented in the partial compilation. Since we define our partial compilation to represent as many solutions as possible one can think of using it for approximating the model count. However, since this is not our goal we do not compare our approach with approximate model counters or provide any theoretical guarantees on the model count given the partial compilation.

The size of Decision-DNNF often explodes when representing CNFs with high tree width. In [LMY21] they introduce a new representation, Constrained Conjunction & Decision Diagram (CCDD), a generalization of Decision-DNNF, used for model counting. A kernelized conjunction node is introduced to capture literal equivalence. They show that CCDD is tractable with respect to a number of queries (implicant check, model counting, consistency check, validity check, and model enumeration) and outperforms state-of-the-art model counters but do not mention the size of the representations.

Projected model counting is useful when some variables are irrelevant to the count, for example when auxiliary variables are used for modelling and these are not functionally defined [ACMS15]. An example is evaluating the robustness of a solution. In this case we want to count the number of changes we can make to a subset of variables such that the problem is still satisfiable. Our method differs from Projected Model Counting (PMC) in the sense that we assign variables a single value whereas PMC ignores variables. In our case, model count reduces because we eliminate solutions with a specific variable value assignment. In a sense, the set of solutions defined by the selective backbone is a subset of the solutions of the projected model count, where the same set of variables are in the selective backbone and in the set of projected variables. Our approach of selective backbone restriction is a stronger restriction than projection. The benefit of selective backbone is that we know exactly what variable-value assignments were not explored. With projection we have no full assignment.

In [CW17] they define a *pseudo backbone* for MAXSAT problems as variable assignments shared by local optima. They use these to decompose the problem.

4.3 Selective Backbones

We focus on Boolean Satisfiability (SAT) instances, as defined in Section 2.1.2 (page 17). Before compilation, SAT instances are given in Conjunctive Normal Form (CNF). A CNF consists of literals and clauses that define the problem at hand. The formal definition of CNF is found in Definition 2.4 (page 18). To obtain

Table 4.2: All solutions for the instance RE.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
S_1	**1**	**1**	**1**	0	**1**	0	**1**	0	**1**
S_2	**1**	**1**	0	0	0	0	**1**	**1**	**1**
S_3	0	**1**	**1**	0	**1**	0	**1**	0	0
S_4	**1**	0	0	0	**1**	0	**1**	0	0
S_5	0	0	**1**	**1**	0	0	**1**	**1**	0
S_6	0	0	0	0	**1**	0	**1**	0	0
S_7	**1**	0	**1**	0	**1**	**1**	**1**	0	**1**
S_8	0	0	0	**1**	0	**1**	**1**	0	**1**
S_9	**1**	0	**1**	**1**	0	**1**	**1**	0	**1**
S_{10}	**1**	0	**1**	**1**	0	**1**	**1**	**1**	**1**

adequate partial compilations, we will be employing selective backbones. Every selective backbone is a partial assignment to some instance. As per Definition 2.5, a partial assignment is a set of p assignments. If the partial assignment satisfies all clauses it is a partial solution. We denote a partial assignment as P. If P assigned all variables of an instance I and satisfies all clauses it is a *solution*, or *model*, for I as per Definition 2.3. If the number of solutions (also called *model count*) for an instance I with n variables is s, then compiling all solutions for I can be trivially done by listing $s \times n$ assignments. However, it is sometimes possible to represent the same information in a more compact way, for example when some variables are assigned the same value in all solutions. The backbone of an instance I is the set of variables that takes the same value in each solution as defined in Definition 2.8.

Example 4.2 (Running Example)**.** *Let RE be an instance with 9 Boolean variables such that the solutions of RE are the 10 solutions listed in Table 4.2. There is exactly one variable, x_7, which takes the same value in all solutions for RE, therefore the backbone of RE is the set $\{x_7 = 1\}$.*

Because backbones are defined so strictly, instances with more than a few solutions will usually have small or non-existent backbones. To be able to use the general idea of a backbone in instances with many solutions, we instead relax the definition and apply the notion to a subset of the solutions for an instance. We call this relaxation the selective backbone.

Definition 4.1 (Selective Backbone)**.** *Let I be a constraint instance and let $\mathcal{S}$ be a set of solutions (not necessarily exhaustive) for I. The* selective backbone *B is a set of assignments of variables in I such that it is the intersection of all solutions in $\mathcal{S}$.*

Example 4.3 (Selective Backbone)**.** *Let RE be the instance from Example 4.2 with solutions in Table 4.2. Consider the set* $\mathcal{S} = \{S_4, S_5, S_6\}$ *of some solutions for RE. There are exactly four variables,* x_2*,* x_6*,* x_7 *and* x_9*, that take the same value in all solutions of* $\mathcal{S}$*. Therefore the selective backbone of* $\mathcal{S}$ *is the set* $\{x_2 = 0, x_6 = 0, x_7 = 1, x_9 = 0\}$*.*

In the same manner that every set of solutions uniquely defines a selective backbone, every partial assignment uniquely defines a particular solution set. We say that the set of solutions $\mathcal{S}$ is defined by the selective backbone B.

Definition 4.2 (Selected Solution Set)**.** *Let* B *be a set of variable assignments for a constraint instance* I*. The* selected solution set *of* B *is the set of all solutions for* I *that are supersets of* B*.*

Remark 4.1. *If* B *is a partial assignment with a non-empty selected solution set, then the selective backbone of the selected solution set of* B *is a superset (not always strict) of* B*.*

Example 4.4 (Selective Backbone)**.** *Let RE be the running example instance with solutions in Table 4.2. Consider the partial assignments* $B_1 = \{x_2 = 0, x_6 = 0, x_7 = 1, x_9 = 0\}$*,* $B_2 = \{x_2 = 0, x_6 = 1, x_7 = 1, x_9 = 1\}$ *and* $B_3 = \{x_2 = 1, x_6 = 0, x_7 = 1, x_9 = 1\}$*, all with the same set of variables* $\{x_2, x_6, x_7, x_9\}$ *being assigned, just to different values. The selective solution set of* B_1 *is* $\{S_4, S_5, S_6\}$*, the selective solution set of* B_2 *is* $\{S_7, S_8, S_9, S_{10}\}$ *and the selective solution set of* B_3 *is* $\{S_1, S_2\}$*. Note that* B_3 *is not the selective backbone of* $\{S_1, S_2\}$*, which is* $\{x_1 = 1, x_2 = 1, x_4 = 0, x_6 = 0, x_7 = 1, x_9 = 1\}$*, but it is a subset of it.*

An instance has many selective backbones and selecting the right one has a large impact. We define some criteria that we use to evaluate a selective backbone. It is important to keep in mind what we want to use the selective backbones for. Our goal is to be able to define a partial solution set to an instance I whose compilation is beneficial. By beneficial we mean the following: first, that the most important or largest set of solutions is kept, and second that the compilation of the selected solution set is smaller than the initial compilation, preferably inverse proportionally to the solution loss. By the most important solution set, we refer to the idea we will introduce in Chapter 5, where solutions have different weights. The current setting is a particular case of that, such that each solution has the same weight. When ranking selective backbones of the same size we have two objectives to optimise: the model count and the compilation size. We want to maximise the model count and minimise the compilation size. This means when

creating the ranking the better selective backbone has to either have a higher model count or a lower compilation size.

For example, we can define the best selective backbone of size p by the size of the selected solution set, or by the ratio between the size of the selected solution set and the size of the representation it compiles to. On the one hand, the second formulation captures both sides of the compromise between minimal loss of solutions and compiled size, while the first formulation is only concerned with the number of solutions. On the other hand, the first formulation is independent of the form in which an instance is compiled, while the second formulation depends on the representation chosen.

Example 4.5 (Best Selective Backbone)**.** *Let us present a scenario with three possible selective backbones B_1, B_2 and B_3 each with the selected solution set size (MC) and compilation size as presented in Table 4.3.*

Table 4.3: Example scenario for ranking selective backbones

	MC	Compilation size
B1	1000	800
B2	200	1500
B3	700	500

When ranking these selective backbones if we only consider the size of the selected solution set we obtain the ordering: B_1, B_3, B_2. In case we rank them using the above mentioned ratio the ordering will be B_3, B_1, B_2, with scores of 1.4, 1.25 and about 0.13 respectively.

Note that in any case, B_2 is the lowest ranked since it covers the least number of solutions and has the highest compilation size. A good selective backbone has a high solution count to minimise information loss and low representation size. The selective backbone B_2 is the worst performing in both criteria, thus no matter what scoring function we use it should be ranked as the worst choice.

4.4 Building a Selective Backbone

In our experiments, we compile instances into Binary Decision Diagrams (BDD), defined in Section 2.3 (page 29). While the initial construction of a BDD is expensive, once it has been built it allows for quick answers to many otherwise computationally demanding questions, such as model counting. It is important

to keep in mind we do not guarantee the smallest compiled representation. The size of the BDD depends on the variable ordering used, we do not optimise this more than the default of a given tool.

In the rest of this section, we focus on two main topics: what metrics we use to compare selective backbones to select the best one, and what type of heuristics we implement. We call the metrics used to evaluate selective backbones the scoring function. We define two scoring functions described in Section 4.4.1: model count and ratio between model count and representation size.

We denote the size of a selective backbone, that is the number of variables we assign with p. For an instance with n variables the number of potential selective backbones of size p is $\binom{n}{p} \times 2^p$, so examining all possible selective backbones to find the optimal one is infeasible for large instances. For this reason, we instead consider greedy iterative heuristics. We start by assigning one variable defined by the maximal score and expand this until we reach p assignments. Since updating an already existing BDD with one variable instantiation does not require recomputing the whole BDD from scratch, building the answer for p upon the answer for $p-1$ allows for finding the answers for each selective backbone size from 0 to the total number n of variables in the instance with only one expensive BDD construction, the initial one.

We present three heuristic types: *random_selection* in Section 4.4.2.1, *static* in Section 4.4.2.2 and *dynamic* in Section 4.4.2.3. As mentioned the presented heuristics are iterative approaches, so by the type of the heuristic we mean the manner in which the instance is updated from one iteration to another. We also introduce three versions of a random heuristic that we use as a baseline. To evaluate the heuristics and scoring functions we look at the adjusted ratio measure defined in Section 4.5 (page 80).

4.4.1 Scoring Functions

We look at two scoring functions that evaluate a variable assignment to decide if it gets added to the selective backbone. To do this we evaluate all variables that have not been assigned yet. We give all such variable instantiations a score and choose the highest score to extend the selective backbone with. Let I be an instance and let B be the selective backbone of some subset of the solutions for I. Let I' be the instance obtained from I after making all variable instantiations in B. We define the two scoring functions:

- *mc*: the score of B is the size of its selected solution set, that is the model count of instance I'
- *ratio*: the score of B is the size of its selected solution set divided by the size of I' in the chosen compiled representation for the problem.

Example 4.6 (Best Selective Backbone)**.** *Let RE be the running example instance with solutions in Table 4.2. We want a set of two variable assignments with the largest possible selected solution set. No pair of distinct variable assignments are present in all solutions for RE, nor even in all but one or all but two solutions, but there are multiple pairs of assignments that are present in seven out of the ten solutions for RE. One such pair of assignments is* $\{x_2 = 0, x_7 = 1\}$*, which is present in the solutions* $\{S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$*. Therefore the set* $\{x_2 = 0, x_7 = 1\}$ *is the best selective backbone of size 2 for the instance RE using the* mc *score.*

The *mc* score of a selective backbone composed of only one variable instantiation can be seen as an exact measure of the variable bias [HMBM08] of its singleton support. When the chosen compilation representation is a BDD, we define its size by the number of nodes reachable from the root of the tree.

Definition 4.3 (Winning Variable Instantiation)**.** *Let B be a selective backbone and let Q be a set of variable instantiations such that every variable present in Q is absent from B (but a variable can appear more than once in Q). We say that the* winning variable instantiation *of B, Q and the scoring function f, is the instantiation $q \in Q$ such that for every variable instantiation $q' \in Q$ with $q' \neq q$, the score f of $B \cup q$ is higher than the score of $B \cup q'$.*

Example 4.7 (Winning Variable Instantiation)**.** *Consider the running example instance RE and the* mc *scoring function. Let* $B = \{x_8 = 0, x_9 = 1\}$ *and let* $Q = \{x_1 = 0, x_6 = 0, x_6 = 1, x_7 = 0\}$*. For simplicity, here Q is only a subset of all remaining assignments.*

- *The selected solution set of* $B \cup \{x_1 = 0\}$ *is the set of solutions where x_8 is assigned 0, x_9 is assigned 1, and x_1 is assigned 0. This is the singleton set* $\{S_8\}$*.*
- *The selected solution set of* $B \cup \{x_6 = 0\}$ *is* $\{S_1\}$*.*
- *The selected solution set of* $B \cup \{x_6 = 1\}$ *is* $\{S_7, S_8, S_9\}$*.*
- *Since no solution for RE assigns 0 to x_7, the selected solution set of* $B \cup \{x_7 = 0\}$ *is empty.*

The winning variable instantiation of B and Q is therefore $x_6 = 1$, because it leads to the largest selected solution set when being added to B.

4.4.2 Heuristic Types

We present three heuristic types: *random_selection*, *static* and *dynamic*. Along with these we use as baseline a completely *random* set of variable assignments. Each presented heuristic type is iterative, extending the CNF model with one variable assignment at each step. The set of these assignments forms the selective backbone. The heuristic types differ on what information they have when calculating the score of a variable assignment. *Static* and *random* create a variable ordering given the initial CNF and apply the assignments. *Dynamic* and *random_selection* calculate the score of a variable assignment given an updated CNF with the previous variable assignments.

The baseline method *random* simply selects p random variables and one random value for each one. Because it is fully random, it can select, sometimes early on, a set of variable instantiations that cannot be extended to a solution for the whole instance. The inductive nature of the method will then keep the size of the selected solution set at 0 until the end. To avoid an empty selected solution set, we proposed a new heuristic, *random_selection*. This refines the selection of the new variable instantiation by giving a score to all possible candidates. The scoring functions we use in these heuristics have been defined previously in Section 4.4.1. This way we are able to directly compare different selective backbones, and we can determine for a given one which next variable instantiation would lead to the highest score.

We present the remaining three methods in order from fastest to slowest. As we show in the next section, the fastest method (*random_selection*) gives the worst overall results of the three, while the slowest (*dynamic*) usually finds higher scoring selective backbones than the other two.

4.4.2.1 Random Selection

The first heuristic type is *random_selection*, described in Algorithm 1. Just like *random*, it picks the next variable randomly. However, the value chosen to assign to the variable is the one that achieves the highest score. This ensures that the selected solution set of the answer will never be empty, as long as the original instance has at least one solution.

Algorithm 1 Algorithm for *random_selection*.

Parameters:

- p: desired size of the selective backbone
- I: a CNF instance with n variables
- f: a scoring function

```
1: Q = { the n variables of the instance }
2: B ← ∅
3: for i ← 1 to p do
4:    x ← random variable from Q
5:    l ← winning_variable_instantiation(B, {x = 0, x = 1}, f)
6:    B ← B ∪ {l}
7:    add the unit clause l to I
8:    Q ← Q \ {l, ¬l}
9: end for
10: return B
```

Example 4.8 (Random selection)**.** *Let us apply* random_selection *on the running example instance RE with solutions in Table 4.2 with* mc *scoring. Suppose that the first variable randomly chosen is* x_2*. This variable is assigned 0 more often than it is assigned 1, so the answer for* $p = 1$ *is* $\{x_2 = 0\}$*.*

If the second variable randomly chosen is x_4*, we then need to compare the scores of the two selective backbones* $B = \{x_2 = 0, x_4 = 0\}$ *and* $B' = \{x_2 = 0, x_4 = 1\}$*. The selected solution set of* B *is* $\mathcal{S} = \{S_4, S_6, S_7\}$*, and the selected solution set of* B' *is* $\mathcal{S}' = \{S_5, S_8, S_9, S_{10}\}$*. Since* $\mathcal{S}'$ *is larger than* $\mathcal{S}$*, the answer for* $p = 2$ *is* B'*.*

4.4.2.2 Static

The second method is *static*, described in Algorithm 2. It starts by ranking all $2n$ variable instantiations, of instance I with n variables, in decreasing order of the score that they would get if they were composing a selective backbone on their own. Ties between scores are broken arbitrarily. We do not distinguish if a variable is a backbone by forming a unit clause, but given the scoring functions we look at, it is highly likely we prioritise them. The algorithm then returns the first p variable instantiations, avoiding repeated variables. Adding these assignments to the original CNF we can evaluate the selective backbone B.

Example 4.9 (Static)**.** *Let us apply* static *on the running example instance RE with solutions in Table 4.2 and with* mc *score. If each of the* 2×9 *variable instantiations was a selective backbone on its own, the ones with the highest score would be:*

Algorithm 2 How to obtain a selective backbone with a static heuristic.

Parameters:

- p: desired size of the selective backbone
- I: a CNF instance with n variables
- f: a scoring function on literals

```
1: Rank all 2n literals l_1, ..., l_2n in I in decreasing order of the value of f(l_i)
2: Q = { all 2n literals in order of their ranking }
3: B ← ∅
4: for i ← 1 to p do
5:    l ← highest-ranking eligible literal from Q
6:    B ← B ∪ {l}
7:    add the unit clause l to I
8:    Q ← Q \ {l, ¬l}
9: end for
10: return B
```

1. $\{x_7 = 1\}$ *with all 10 solutions for RE in the selected solution set.*

2.-3. $\{x_2 = 0\}$ *and* $\{x_8 = 0\}$*, each with a selected solution set of size 7.*

4.-8. $\{x_1 = 1\}$, $\{x_3 = 1\}$, $\{x_4 = 0\}$, $\{x_6 = 0\}$, *and* $\{x_9 = 1\}$*, each with a selected solution set of size 6.*

So the answer for $p = 1$ *would be* $\{x_7 = 1\}$*, the answer for* $p = 2$ *would be either* $\{x_7 = 1, x_2 = 0\}$ *or* $\{x_7 = 1, x_8 = 0\}$*, and the answer for* $p = 3$ *would be* $\{x_7 = 1, x_2 = 0, x_8 = 0\}$*. The answers for* $p = 4$ *to* $p = 8$ *would add variable instantiations on variables* x_1*,* x_3*,* x_4*,* x_6 *and* x_9 *so the instantiation on variable* x_5 *would not be picked until* $p = 9$*.*

The method *static* only computes the score associated with each variable instantiation once, at the beginning. This saves time, but it means that once a variable instantiation is picked the scores of the remaining instantiations are still based on solutions that might have been already eliminated.

4.4.2.3 Dynamic

The third heuristic type, *dynamic*, takes into account the variable instantiations previously chosen when computing the winning variable instantiation, so all scores are up to date. The method is described in Algorithm 3.

The procedure for choosing the first literal of the selective backbone is the same for *static* and *dynamic* heuristic types: compute the value of the scoring function for every literal, then pick the literal with the highest score. The difference is that *static* heuristics do not make any expensive operation beyond this first step, they

Algorithm 3 How to obtain a selective backbone with a dynamic heuristic.

Parameters:

- p: desired size of the selective backbone
- I: a CNF instance with n variables
- f: a scoring function on literals

1: $B \leftarrow \emptyset$
2: Q = { all $2n$ literals in I to the pool of eligible literals}
3: **for** $i \leftarrow 1$ **to** p **do**
4: $l \leftarrow$ winning_variable_instantiation(B, Q, f)
5: $Q \leftarrow Q \setminus \{l, \neg l\}$
6: $I \leftarrow extend(I, l)$
7: $B \leftarrow B \cup \{l\}$
8: **end for**
9: **return** B

just follow the ranked list of literals. While this prevents them from acquiring updated information about the instance, it also means that from the second pick onwards they are much faster than *dynamic* heuristics. Algorithm 3 depicts this difference with the use of *winning_variable_instantiation()* method, that ranks all remaining eligible assignments according to the scoring function and returns the highest ranked. Here too, ties are broken arbitrarily. The *extend(I,l)* method makes sure the instance is updated in each iteration with the assignment added to the selective backbone. In this chapter this method is implemented by using the *apply* operation on the compilation of instance I and the representation of the new assignment. The *apply* operation is described in Section 2.3 and in here is used to efficiently combine a compilation with a variable assignment.

Example 4.10 (Dynamic)**.** *Let us apply* dynamic *on the running example instance RE for the* mc *score. Table 4.2 reminds us of all solutions for the problem. Table 4.5 presents the first few iterations of both* static *and* dynamic *heuristics with* mc *scoring. In Table 4.6 the last three columns present the solutions from the initial problems which are still part of the selected solution set given the selective backbone defined in Table 4.5. The following variables are selected for each iteration for the two heuristics:*

$p = 1$ *: For* $p = 1$*, the best variable instantiation to pick is* $\{x_7 = 1\}$*, which appears in all solutions for the original instance. The selective backbone for* static *and* dynamic *is denoted by* BS_1 *and* BD_1 *respectively, both equal to* $\{x_7 = 1\}$ *as shown in Table 4.5.*

$p = 2$ *: For the second variable instantiation, given the previous assignment of*

$\{x_7 = 1\}$, *for both* static *and* dynamic $\{x_2 = 0\}$ *and* $\{x_8 = 0\}$ *are tied winners. Let us pick the latter, giving us an answer of* $BS_2 = \{x_7 = 1, x_8 = 0\}$ *and* $BD_2 = \{x_7 = 1, x_8 = 0\}$ *for* $p = 2$. *For now both* static *and* dynamic *behave the same way. As seen in Table 4.6 in the second column both heuristics have the same set of selected solutions.*

$p = 3$: *Given the previous assignment of* $BS_2 = BD_2 = \{x_7 = 1, x_8 = 0\}$ *we look at calculating the* mc *score of the rest of the variables in order to extend the selective backbone to size 3. At this point* static *and* dynamic *obtain different* mc *scores for the rest of the variables. This is because* static *calculates the scores in the very beginning and does not update them through the previous iterations. These scores are available in Table 4.4. The third highest score from here is for the assignment* $\{x_2 = 0\}$ *so the selective backbone for* static *gets extended with this. Note that when we calculate the size of the selected solution set of* $BS_3 = \{x_7 = 1, x_8 = 0, x_2 = 0\}$ *it is not equal to 7, the score of the latest assignment but it is 5. This is because the* static *heuristic did not know that the previous iterations already eliminated some solutions. When we look at the same iteration for the* dynamic *heuristic a different assignment wins as shown in Table 4.5. Here* $\{x_5 = 1\}$ *is the winning one as it satisfies 5 solutions out of the 7 solutions from the previous iteration. Thus the selective backbone for* dynamic *of size 3 is* $BD_3 = \{x_7 = 1, x_8 = 0, x_5 = 1\}$.

$p = 4$: *For* static *the next options with score 6 are* $\{x_1 = 1\}$, $\{x_3 = 1\}$ *and* $\{x_9 = 1\}$. *Let us pick the first one, extending* BS_3 *to* $BS_4 = \{x_7 = 1, x_8 = 0, x_2 = 0, x_1 = 1\}$. *For* dynamic *the score is calculated given the selected solution set* $\{S_1, S_3, S_4, S_6, S_7\}$ *defined by* BD_3. *The assignment* $\{x_4 = 0\}$ *satisfies all 5 previous solutions, so it is the best candidate to extend* BD_3.

Table 4.4: All *mc* score for *static* heuristic on instance RE

variable	x_1		x_2		x_3		x_4		x_5		x_6		x_7		x_8		x_9	
value	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
mc score	4	6	7	3	4	6	6	4	5	5	6	4	0	10	7	3	4	6

Note that it is possible to never pick $x_2 = 0$ with *dynamic*, while from Example 4.9 we know that this variable assignment is always picked by *static* for $p = 3$ at the latest.

4.5 Evaluation Metrics

The goal of the heuristics is to find the best variable instantiation set that covers most of the solutions but also has a smaller compiled size. To be able to compare the heuristics across instances with different initial values of model count and compilation size, we define the adjusted ratio AR:

$$AR = \frac{|S|/|compilation|}{|S_{init}|/|compilation_{init}|}$$

with

- $|S|$ the size of the current solution set,
- $|compilation|$ the size of the current compilation,
- $|S_{init}|$ the size of the initial solution set,
- $|compilation_{init}|$ the size of the initial compilation.

In this chapter $|S|$ is equivalent to the model count (MC) and $|compilation|$ to the size of a BDD.

This ratio represents how much better (if greater than 1) or worse (if smaller than 1) the compilation obtained from the current selective backbone is compared to the initial compilation. We focus on answering two questions:

- **Q1**: Is there a compilation such that the adjusted ratio is higher than 1?
- **Q2**: Is there a compilation that is significantly more compact if we want to keep at least $x\%$ of the initial solution set?

Table 4.5: Example of selective backbones for instance RE, with MC_i denoting the number of the selected solutions set for $p = i$. Note that the static score denotes the values calculated initially, before any assignments are made.

	p	Selective Backbone	MC_i	score
static	1	$BS_1 = \{x_7 = 1\}$	10	10
	2	$BS_2 = \{x_7 = 1, x_8 = 0\}$	7	7
	3	$BS_3 = \{x_7 = 1, x_8 = 0, x_2 = 0\}$	5	7
	4	$BS_4 = \{x_7 = 1, x_8 = 0, x_2 = 0, x_1 = 1\}$	3	6
dynamic	1	$BD_1 = \{x_7 = 1\}$	10	10
	2	$BD_2 = \{x_7 = 1, x_8 = 0\}$	7	7
	3	$BD_3 = \{x_7 = 1, x_8 = 0, x_5 = 1\}$	5	5
	4	$BD_4 = \{x_7 = 1, x_8 = 0, x_5 = 1, x_4 = 0\}$	5	5

Table 4.6: Selective backbones for $p \geq 2$.

	$p=2$		$p=3$		$p=4$	
S_1	BS_1	BD_1		BD_2		BD_3
S_2						
S_3	BS_1	BD_1		BD_2		BD_3
S_4	BS_1	BD_1	BS_2	BD_2	BS_3	BD_3
S_5						
S_6	BS_1	BD_1	BS_2	BD_2		BD_3
S_7	BS_1	BD_1	BS_2	BD_2	BS_3	BD_3
S_8	BS_1	BD_1	BS_2			
S_9	BS_1	BD_1	BS_2		BS_3	
S_{10}						

4.6 Results

In this section we describe the setup of the experiments, the datasets used and we analyse the results.

4.6.1 Setup

Experiments were run on a machine with an Intel(R) Xeon(R) CPU E5620 @ 2.40GHz running Ubuntu 22.04.2. Three types of benchmarks were tested: a generated dataset[1] from a construction used to produce challenging model counting instances [EO22], the ISCAS circuit suites[2] and some instances from the planning benchmark.[2] The ISCAS circuit design and the planning benchmarks have been extensively used in the field of knowledge compilation and model counting as well [CLM21, LLM20, OD15].

The generated sets contain 10 instances with 15 variables and 45 clauses (Dataset A) and 10 instances with 30 variables and 90 clauses (Dataset B). Within the ISCAS benchmark, we looked at instances from ISCAS89, ISCAS93 and ISCAS99 containing problems with 26-252 variables and 66-639 clauses. Within planning we looked at the following instance types: blocks, bomb, coins, comm, emptyroom, flip, ring, safe, sort and uts. Instances in here contain 5-600 variables and 10-1901 clauses. For all cases we only considered CNFs with at most 600 variables. We ran all experiments with 1 hour timeout. We excluded instances that did not compile within the hour as for the proposed evaluations we need the initial compilation. Later on, in Chapter 5 we introduce scoring methods that do not rely on the initial compilation. All code necessary to reproduce the experiments

[1] https://github.com//baloghAndi//partialKC_using_selective_backbones
[2] https://www.cril.univ-artois.fr/KC/benchmarks.html

is available online[1].

We implemented the previously mentioned heuristics in Python 3 and used CUDD for BDD compilations [3]. More specifically we used part of Temporal Logic Planning (TuLiP) toolbox[4] that serves as a Python and Cython wrapper of CUDD. During compilation after each *apply* operation we apply group sifting in order to find good variable ordering for the new BDD.

4.6.2 Methodology

For our experiments we looked at three greedy heuristics types, *random_selection*, *static*, *dynamic*. For each heuristic we looked at the scoring function *mc* and *ratio*. In total that is six different heuristics denoted by the combination of the methods type and the scoring function: *random_selection_mc*, *random_selection_ratio*, *static_mc*, *static_ratio*, *dynamic_mc* and *dynamic_ratio*. We also compared these with a completely random set of variable instantiations, denoted as *random*. Since this does not calculate the score of assignments no scoring function is applied or added to its name. As mentioned above, all heuristics are incremental, the selection of p variable instantiations contains the selection of $p-1$ variable instantiations extended with a new variable instantiation. With each variable instantiation, the size of the solution set is monotonically decreasing.

In some cases, using the *mc* scoring, multiple variable instantiations of size p exist that cover the same number of solutions but a different solution set. In these cases, we break these ties arbitrarily. The introduction of the *ratio* scoring changes this to favouring the p variable instantiations that have a smaller compilation. This approach allows us to first select the variable instantiations that impact the BDD size but with minimal solution loss. This is visible from the fact that *dynamic_ratio* achieves the best trade-offs. Also in case the problem has proper backbones (Definition 2.8) *dynamic_ratio* will not necessarily select them at first as they do not change the size of the BDD. Therefore problems with large backbones would have fewer opportunities for a good selective backbone. In our experiments only 41 out of the total 168 instances have proper backbones, with one from ISCAS and the rest from the planning benchmarks as seen in Table 4.7. The sizes of the proper backbones range between 0.48% to 30% of the number of variables. Table 4.7 contains more information based on instance types, about

[3] https://web.archive.org/web/20150215010018/ http://vlsi.colorado.edu/ fabio/CUDD/cuddIntro.html

[4] https://github.com/tulip-control/dd

Table 4.7: Best ratios achieved for each type of instances. For each dataset, Column 2 (Nb) is the number of instances in the set, and Column 3 (Nb with backbone) is the number of instances in the set with a non-empty backbone.

Instance type	Nb	Nb with backbone	Best adjusted ratio			
			min	max	average	median
Dataset A	10	0	1.776	3.662	2.54	2.66
Dataset B	10	0	5.383	13.94	8.68	9.01
ISCAS89	5	1	1.264	4.944	2.47	1.94
ISCAS93	2	0	11.501	26.701	19.10	19.10
ISCAS99	7	0	1.151	4.053	2.64	2.93
blocks	8	7	1.064	5.049	2.65	2.54
bomb	8	8	1.016	2.32	1.60	1.60
coins	7	9	1.2	1.851	1.39	1.34
comm	4	4	1	15.863	5.02	1.60
emptyroom	21	0	1	7.469	1.66	1.00
flip	20	0	1	1.095	1.00	1.00
ring	16	0	1	3.2	1.64	1.45
safe	26	0	1	1.873	1.35	1.42
sort	11	0	1.852	5.244	3.83	3.89
uts	13	12	1.005	5.51	3.49	3.94

the number of instances for each type (Column 2) and how many of them have a non-empty backbone (Column 3).

Table 4.8 summarises the average time per instance spent on the initial BDD compilation and for all the heuristics to explore n selective backbones. Most of the runtime is spent on the initial compilation, this is a required first step for all the heuristics. The next seven columns show the time spent on finding n selective

Table 4.8: Average compilation times in seconds for instance types.

Instance type	Init comp	random	random_ selection mc	random_ selection ratio	static mc	static ratio	dynamic mc	dynamic ratio
Dataset A	0.217	0.309	0.313	0.43	0.362	0.34	0.333	0.326
Dataset B	2.613	2.946	3.076	3.24	3.277	3.287	3.421	3.243
ISCAS89	1290.631	216.865	221.961	295.039	239.691	295.71	374.492	317.405
ISCAS93	295.331	90.589	96.605	104.362	138.229	135.261	180.098	159.95
ISCAS99	156.584	45.901	47.435	50.009	51.184	47.731	78.151	63.092
blocks	1119.291	1051.07	1047.16	1025.909	1085.312	1058.3	1230.959	1160.956
bomb	386.295	232.778	265.839	265.896	295.138	302.942	1470.723	1265.708
coins	2333.587	194.944	214.181	554.477	254.396	560.743	556.6	441.289
comm	316.963	360.663	386.863	999.783	436.769	450.071	866.599	635.998
emptyroom	2999.494	321.221	336.584	334.526	365.968	360.129	640.034	616.192
flip	1.691	0.638	0.745	0.711	0.88	0.885	0.958	1.001
ring	1060.66	305.258	312.894	323.228	327.277	322.515	408.546	348.009
safe	18.332	194.905	220.732	226.521	293.156	248.284	483.846	462.226
sort	337.266	419.213	439.866	415.04	497.05	425.432	689.139	541.916
uts	205.234	145.488	152.462	152.579	189.794	161.606	357.711	248.549

backbones for each heuristic. More time is taken by *dynamic* and *dynamic_ratio* since both heuristics at each step explore at least $2(n-p)$ possible variable instantiations for the p selective backbone problem. For each possible instantiation *dynamic* and *dynamic_ratio* call the *apply* function in order to calculate next BDD.

4.6.3 Existence of compact compilations

As defined earlier in Section 4.5, we use the adjusted ratio to measure the quality of the selective backbones. The adjusted ratio is defined as a ratio between the compactness of the current model and the initial model. We define compactness in terms of the ratio between model count and size of compilation. Because instances have different number of variables, we show 50 data points for each heuristic, with the i^{th} data point being the median of the values achieved after $2i\%$ of the variables have been assigned for each instance. We proposed two research questions in Section 4.5. The first one, **Q1** asks if there exists a compact compilation, one whose adjusted ratio is higher than 1. To answer **Q1** the last columns of Table 4.7 highlight information about the best adjusted ratio for each instance type. There are 36 instances where there is no improvement for the adjusted ratio, some are due to the initially small number of solutions.

A visual representation of the adjusted ratio can be found in Figure 4.2, Figure 4.3 and Figure 4.4, showing the median adjusted ratio for each benchmark respectively. The first data point denotes the $p = 0$ problem for which the adjusted ratio is 1. The higher the adjusted ratio is, the better the selection is. Since

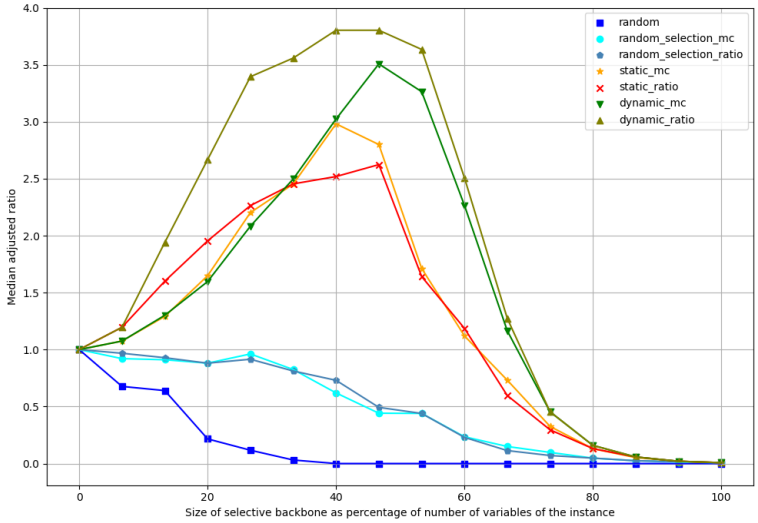


Figure 4.2: Median Adjusted Ratio over Dataset A and Dataset B.

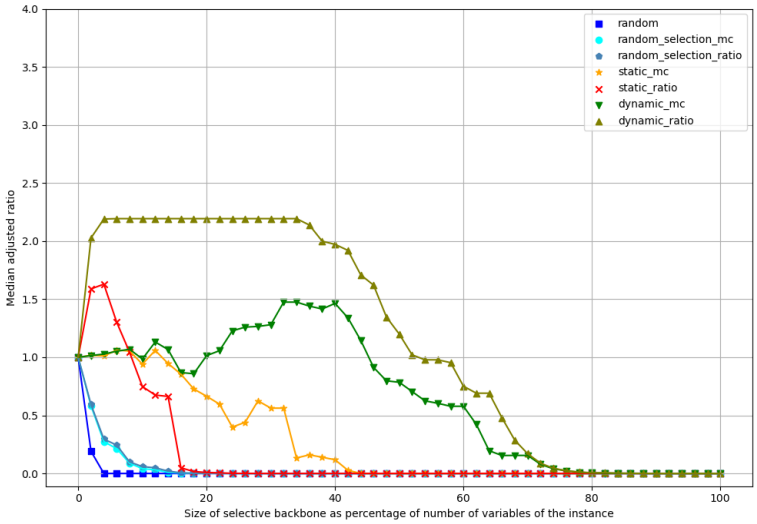


Figure 4.3: Median Adjusted Ratio over ISCAS instances.

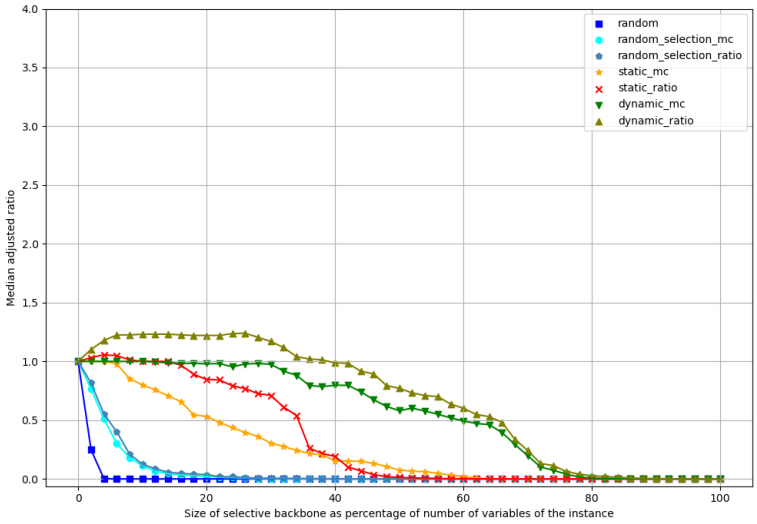


Figure 4.4: Median Adjusted Ratio over planning instances.

each instance has a different number of variables the maximum value of p is also different, so we calculated the median adjusted ratio according to the percentage of the selective backbone size divided by the total number of variables. The most gain can be seen in the generated instances, as the ratio curve reaches higher values, close to 4 times more compact than the initial compilation. The baseline *random* and *random_selection* drop very quickly below 1, indicating no compact compilation is obtained. In each figure *dynamic_ratio* performs best, followed by *static_ratio* for the first few selections, which is then dominated by *dynamic_mc*. In Figure 4.2 we observe a clear peak for *dynamic_ratio* at 40% of the variables assigned. In Figure 4.3 and Figure 4.4 there is more of a plateau where the ratio is roughly the same for a few variable instantiations for *dynamic_ratio*. That

is because the ISCAS and planning instances have variables that split the solution space less uniformly than in the generated dataset. For all experiments *random* performed worse, reaching a set of variable instantiations without any solutions quite early. Due to the incremental property the selection never improves. The model count is monotonically non-increasing but the compiled representation size is not guaranteed to exhibit this behaviour. When the representation size increases between iterations, the adjusted ratio creates a drop, but subsequent iterations might reach a smaller compilation. This can happen due to the fact that we do not optimize the representation size, but also highlight that not any small solution set is guaranteed a small compiled representation with default compilation parameters. This behaviour can be seen mostly in Figure 4.3.

4.6.4 Compact compilation with a threshold

The next research question, **Q2**, looks for the existence of a compact compilation given a specific threshold for solution loss. To answer **Q2** we look at Figure 4.5, Figure 4.6 and Figure 4.7 which show the median efficiency for each heuristic for each benchmark set. The 45° grey line denotes the area where the solution set size decrease is directly proportional to the BDD size decrease, that is there is no significant benefit. By representing $x\%$ of the solutions the BDD compilation will be $x\%$ of its original size too. Therefore, we want to be on the right of this diagonal, as close as possible to the top of the plot. For this metric the generated instance results show the most significant benefits as seen in Figure 4.5, the efficiency is quite far from the diagonal. It is important to note that each data point might represent multiple partial compilations if they reach the same model count and size reduction. These plots are useful to check for the existence of a partial compilation given a threshold of solution loss, but are not suitable to compare iterations of the different heuristics. They also offer a perspective where it is clear separately how much the partial compilation size and partial solution set changed with respect to the initial problem. In each Figure 4.5, Figure 4.6 and Figure 4.7 all three random based heuristics perform poorly, as they are all under the grey diagonal. In Figure 4.5 all four non random based heuristics seem to behave very similarly. The scoring function influences this in a way that static and dynamic heuristics of the same scoring function behave closest to each other. A similar behaviour is present in Figure 4.6 and Figure 4.7 but the heuristics with the model count scoring function perform worse.

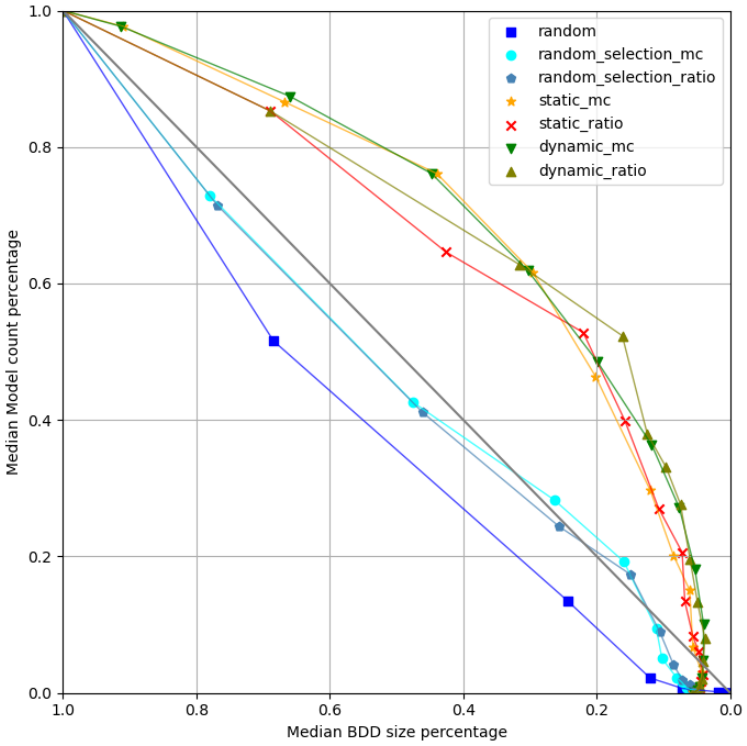


Figure 4.5: Median efficiency over dataset A and B.

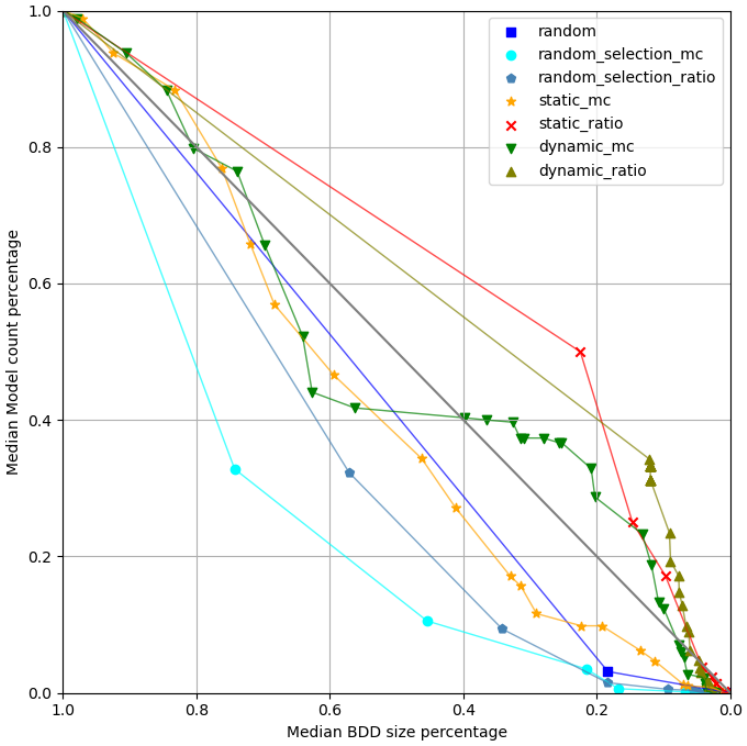


Figure 4.6: Median efficiency over ISCAS instances.

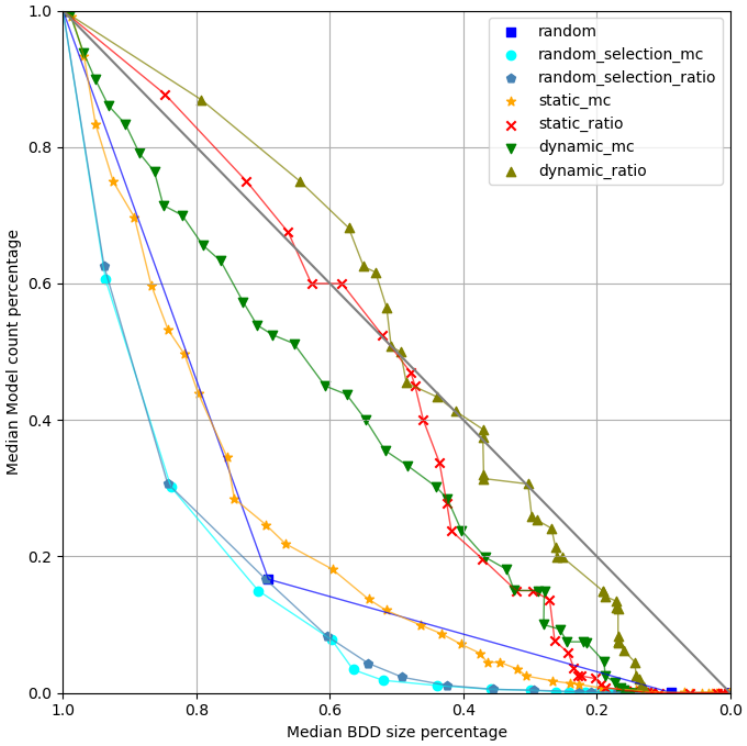


Figure 4.7: Median efficiency over planning instances.

Using these figures one can determine that for representing $x\%$ of the solution set what is the smallest representation any of the heuristics can find. For example, let us consider representing at least $x = 80\%$ of the solutions of the A and B datasets. We look at the area above 0.8 median model count in Figure 4.2 and select the heuristic most to the right. In this case the *dynamic_mc* heuristic, obtains the smallest compilation representing at least 80% of the solutions.

The ratio figures offer a detailed view of the progression of the adjusted ratio with respect to the increasing selective backbone. The efficiency plots highlight the existence of compilation that represents $x\%$ of all solutions but the compilation size is less than a given $x\%$ of the initial compilation size. Note that we did not exhaustively optimize the BDDs, so our results do not state that there is no smaller BDD with the same solution set. We show that even with minimal effort, such compilations exist.

4.7 Summary

In this chapter we explored the use of a relaxation of backbones, called selective backbones, in knowledge compilation for constraint problems. We used selective

backbones to restrict the set of solutions, and by only compiling this subset of solutions we obtained a partial compilation. We defined six different heuristics to obtain a good selective backbone that would yield a compact compilation. We evaluated the compactness and usefulness of the partial compilation by looking at the percentage of solutions kept and the size reduction. Ideally, there is more reduction in the size than information, that is solutions not represented. We obtained such evidence by looking at a set of instances from a generated datasest, a circuit design and a planning dataset. For 78.57% of the instances, we obtained at least one compact compilation with the adjusted ratio larger than 1. The median of the best adjusted ratio over all instances was 1.48, a value that indicates a compact compilation.

Seeing such good results the question arose whether we were keeping the correct and most interesting subset of solutions. To address this in the next chapter we look at weighted combinatorial problems, where we consider the weights as the importance or preference of certain solutions. This way not only can we see the size of the represented solutions in the partial compilation but also the importance of them. Since the heuristics we presented in this chapter relied on expensive computation, the desire to improve on that led to the introduction of new heuristic methods. Also since the heuristics take an iterative approach, defining a good stopping point for the iterations is important. In the next chapter we also explore a simple way to achieve this.

Chapter 5

Improved Constructions of Selective Backbones

Abstract *In this chapter we consider defining and obtaining selective backbones for weighted constraint satisfaction problems. With this solution restricting technique we evaluate the obtained partial compilations. We propose improved heuristics for obtaining partial compilations. We focus on determining if the most important subset of solutions is kept by looking at the Weighted Model Count (WMC) of the subset of solutions kept. Also, we propose an approach to identify the size of a good selective backbone, that is the number of variables to assign. This chapter is based on our paper from CPAIOR 2025: Determining the Most Promising Selective Backbone Size for Partial Knowledge Compilation.*

5.1 Introduction

In this chapter we extend the previous work on *selective backbones* with three aspects: we look at weighted problems, we introduce new computationally less expensive heuristics to obtain selective backbones, and we propose an approach to determine a good size for the selective backbone.

In order to deem our approach successful two criteria must be satisfied: there is a significant reduction in size and an important subset of solutions is kept. Since many subsets of solutions exist, representing the most preferred, most likely, most compact, or most important, ones for the user is important. For this reason in this

dissertation, we explore various techniques for selecting the subset of solutions to be compiled. We consider different ways to define the importance of a variable. In case of unweighted instances, all variables have the same importance, whereas for weighted instances weights represent a preference or probability. We want the most important subset to be selected, not just the largest one.

In this chapter, we look at a different compiler and a slightly different definition of what a good selective backbone is. We look at the d4 compiler [LM17], which is a top-down compiler that has ranked highly in the Weighted Model Counting competitions in the past years. D4 compiles to a representation called Decision-DNNF which we formally define in Section 2.3.4 (page 38). Deterministic Decomposable Negational Normal Form (d-DNNF), a superset of Decision-DNNF, became a popular representation since it is a superset of OBDDs, and probabilistic inference and Maximum a-Posteriori (MAP) can be answered in polynomial time in the size of the representation [ALM22]. A variety of problems in machine learning, expert systems, social network analysis, bioinformatics and information theory can be formulated as probabilistic maximum a posteriori inference. This deals with the problem of computing the most likely assignment to given variables in a distribution. Conditional marginal probability calculation can be reduced to weighted model counting, by summing all weights of solutions that satisfy the constraints [CD08].

We introduce new heuristics that behave similarly to the ones introduced in Chapter 4. We tailor them for weighted problems as well as use less expensive approximate methods to determine the next variable assignment to expand the selective backbone with. Using statistical information about an instance we estimate which assignment would bring most benefit. We also introduce a hybrid method that combines the simple estimates with the computationally expensive evaluation at each iteration in a heuristic. Detailed description of these can be found in Section 5.5 (page 99) and Section 5.4 (page 95).

The third major topic of this chapter is the identification of the size of a good selective backbone with respect to a heuristic. This is important because knowing the size p we can perform p iterations of a given heuristic without trying to compile the problem at each step to check if the representation is small enough or even the compilation finishes successfully. We present an approach for this and its evaluation in Section 5.7. In the next section we present the related works, dealing with partial compilation especially in the context of weighted problems.

5.2 Related Work

Similar to the KC map, a compilation map is created for conditional preference statement-based languages such as CP-nets [FM21].

Knowledge Compilation has also been explored as a method for uniform sampling [SGRM18]. A given CNF formula is compiled into d-DNNF form, and then performing only two passes over the d-DNNF representation they generate as many identically and independently distributed samples as specified by the user. However, this approach needs the initial compilation as well.

Weighted projected model counting is an extension of weighted model counting such that it considers a set of variables Y to be forgotten. The tool d4Max [ALM22], extends d4 to deal with such problems as well. Our approach differs from this since we do not forget variables, but fix their values and eliminate solutions that are not consistent with the assignment.

An algorithm has been developed for obtaining the top k models, given a d-DNNF [BDD+22]. Our approach is different from this as we aim to obtain the top models without compiling the initial problem, only the subproblem that represents these solutions. In [FdB18] collapsed compilation has been proposed for approximate inference for discrete probabilistic graphical models. The algorithm incrementally selects which variable to sample next based on the partial compilation obtained so far, represented as an SDD. The context of their algorithm is quite different from our work, as well as they iteratively instantiate variables in the SDD to decrease its size given a threshold, but only take into account the partial problem.

Another compilation map [FMNS14] focuses on classifying real-valued functions. Valued decision diagrams (VDDs) are data structures that represent functions mapping variable-value assignments to non-negative real numbers. In a way this can be seen as weights for SAT literals that denote CP variables. VDDs are useful to compile cost functions, utility functions, or probability distributions.

As mentioned before, we consider partial compilation as a compiled representation that omits some solutions from the initial set of solutions. Partial compilations have been an area of interest for multiple reasons [OP06]. For example, they can act as bounds in an optimization problem. A partial compilation can be looked at as a restriction on the original decision diagram, this way used as an upper bound. Decision Diagrams (DDs) have been used as upper and lower bounds for optimization problems. They have shown significant advantages compared to

the widely used linear relaxations, but the size of DDs heavily depends on the variable ordering used for them, which optimization problem is known to be NP-hard. To tackle this problem [CBR+22] look at Reinforcement Learning in order to learn the best bounds. We do not explore different variable orderings. Our approach for partial compilation is by fixing part of the problem and eliminating some solutions.

Approximate MDDs have also been explored as generators for lower bounds. A top-down compilation method has been developed for this purpose that merges nodes when the width of the representation gets larger than a limit [BvHH11]. Both MDDs and BDDs have been used as domain stores for Constraint Programming systems.

By relaxing or reducing the decision diagram the problem becomes less constrained, allowing more solutions. This way the decision diagram decreases in size since some of the branches can be merged. Another way to decrease the size of the decision diagram is by restricting it and thus eliminating some solutions. In this case, parts of the decision diagram that represent the eliminated solutions can also be eliminated, leading to a smaller representation. Both relaxation and restriction have a wide range of usage. Explanations have also been explored as a means to achieve approximate weighted model counting [RKdBR14].

Bayesian Networks have become a popular method to deal with probabilistic inference. However, they do not take advantage of some properties that would make inference more scalable. Compiled representations can take advantage but the cost of compiling sometimes is still too high for it to be useful in some scenarios. In [DLHL21] they propose a framework for probabilistic inference that partitions the compilation into subproblems, which are recomposed in the inference query. This is called Compositional Weighted Model Counting (CWMC). They transform Bayesian Networks using the one hot encoding, which assigns a literal to a unique variable value pair, to a Boolean Formula.

5.3 Selective Backbone for Weighted Problems

In this section we extend the definition of selective backbones to apply to weighted problems. *Weighted model counting* (WMC) is the equivalent of *model counting* in a weighted problem setting. These instances are expressed in CNF with added weights, $w(l)$, for every literal l. The *weight of a model/solution* is the product of the weights of the literals present in the model. The *weighted model count* of an

instance is the sum of the weights of the models to the instance. Weighted model counting has many applications, most notably probabilistic inference problems such as Bayesian Networks can be reduced to weighted model counting [CD08]. For example the joint probability distribution of the variables is represented by the product of the weights of the variable assignments and the probability of some evidence is the sum of weights for models consistent with the evidence. The model counting problem is #P-Complete, even when no clause is allowed to have more than two literals [Val79]. The WMC problem is #P-Complete, like its unweighted version (because the latter is a special case of the former where all literals weigh 1).

We aim to remove models from an instance so that its representation becomes much more compact while maintaining the weighted model count as high as possible. The set $\mathcal{S}$ of models to keep is characterised by the set of literals that appear in all models from $\mathcal{S}$. This set of literals is called a *selective backbone* (Definition 4.1, page 70). The selective backbone is still a set of variable assignments. For weighted problems, a good selective backbone B is such that removing all models that are not supersets of B significantly decreases the size of the compiled representation, and the sum of the weights of the models that are an extension of B is as close as possible to the original weighted model count. Note that this selective backbone B is a backbone of the instance obtained after removing all models that do not include B.

Given the running example RE from the previous chapter we modify it such that we add weights to each variable assignment as denoted in Table 5.1.

Example 5.1 (Weighted Running Example)**.** *Let WRE be a weighted instance with 9 Boolean variables each assigned weights as shown in Table 5.1 and all solutions to the problem are shown in Table 5.2 along with the weight of each solution in the last column.*

Table 5.1: Weights for instance WRE

variable / value	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
0	0.304	0.714	0.773	0.449	0.281	0.577	0.019	0.315	0.519
1	**0.696**	**0.286**	**0.227**	**0.551**	**0.719**	**0.423**	**0.981**	**0.685**	**0.481**

Note that Table 5.2 and Table 4.2 present the same set of solutions, in order to illustrate the differences we can obtain with weights.

Table 5.2: All solutions for the instance WRE.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	WMC
S_1	**1**	**1**	**1**	0	**1**	0	**1**	0	**1**	0.0013
S_2	**1**	**1**	0	0	0	0	**1**	**1**	**1**	0.0036
S_3	0	**1**	**1**	0	**1**	0	**1**	0	0	0.0006
S_4	**1**	0	0	0	**1**	0	**1**	0	0	0.0115
S_5	0	0	**1**	**1**	0	0	**1**	**1**	0	0.0015
S_6	0	0	0	0	**1**	0	**1**	0	0	0.0050
S_7	**1**	0	**1**	0	**1**	**1**	**1**	0	**1**	0.0023
S_8	0	0	0	**1**	0	**1**	**1**	0	**1**	0.0016
S_9	**1**	0	**1**	**1**	0	**1**	**1**	0	**1**	0.0011
S_{10}	**1**	0	**1**	**1**	0	**1**	**1**	**1**	**1**	0.0024

Table 5.3: Assignment weights and weighted model counts for each solution in the selected solution set of $B = \{x_7 = 1, x_2 = 0\}$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	WMC
S4	**0.696**	0.714	0.773	0.449	**0.719**	0.577	**0.981**	0.315	0.519	0.0115
S5	0.304	0.714	**0.227**	**0.551**	0.281	0.577	**0.981**	**0.685**	0.519	0.0015
S6	0.304	0.714	0.773	0.449	**0.719**	0.577	**0.981**	0.315	0.519	0.0050
S7	**0.696**	0.714	**0.227**	0.449	**0.719**	**0.423**	**0.981**	0.315	**0.481**	0.0023
S8	0.304	0.714	0.773	**0.551**	0.281	**0.423**	**0.981**	0.315	**0.481**	0.0016
S9	**0.696**	0.714	**0.227**	**0.551**	0.281	**0.423**	**0.981**	0.315	**0.481**	0.0011
S10	**0.696**	0.714	**0.227**	**0.551**	0.281	**0.423**	**0.981**	**0.685**	**0.481**	0.0024

Example 5.2 (Selective Backbone)**.** *Given instance WRE as defined in Example 5.1 and a selective backbone* $B = \{x_7 = 1, x_2 = 0\}$ *the weighted model count (WMC) of its selected solution set* $\mathcal{S} = \{S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$ *is* $0.0115 + 0.0015 + 0.0050 + 0.0023 + 0.0016 + 0.0011 + 0.0024 = 0.0254$. *The WMC for each solution is found in Table 5.3.*

5.4 Scoring Functions

To obtain selective backbones for weighted instances we follow a similar methodology as in Section 4.4. The scoring function calculates a metric that given a CNF and a selective backbone of size p evaluates a possible variable assignment. Note that when we present an example we define a selective backbone B of size p and the score is calculated on the instance obtained after assigning B.

In the rest of this section, we define the scoring functions used: *actual_WMC* in Section 5.4.1, *relative_weight* in Section 5.4.2 and *estimated_WMC* in Section 5.4.3.

5.4.1 Actual WMC

The scoring function *actual_WMC* is the equivalent to the *model count* in an unweighted problem setting. It calculates the weighted model count given the selected subset of solutions defined by a selective backbone. The weighted model count is the sum of each solution weight, which is the product of the literals participating in the solution.

Definition 5.1 (Actual WMC)**.** *Given an instance I and a literal l from I, let I' be the instance obtained after adding the unit clause l to I. We say that $actual\_WMC(l)$ is the weighted model count of I'.*

Example 5.3 (WMC Score)**.** *Given the instance WRE as defined in Example 5.1 and a selective backbone $B = \{x_7 = 1, x_2 = 0\}$ the score of each remaining possible assignment is presented in decreasing order in Table 5.4. The selected solution set of B is $\mathcal{S} = \{S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$ with the* actual_WMC *score of* 0.0254*. When extending this with $\{x_8 = 0\}$ the new selected solution set will be $\mathcal{S}' = \{S_4, S_6, S_7, S_8, S_9\}$ which has the score* $0.0115 + 0.0050 + 0.0023 + 0.0016 + 0.0011 = 0.0215$

Table 5.4: Actual_WMC score for each remaining variable assignment with Example 5.1 and the selective backbone $B = \{x_7 = 1, x_2 = 0\}$

rank	assignment	*actual_WMC*	rank	assignment	*actual_WMC*
1.	$x_8 = 0$	0.0215	8.	$x_1 = 0$	0.0082
2.	$x_4 = 0$	0.0188	9.	$x_6 = 1$	0.0074
3.	$x_5 = 1$	0.0188	10.	$x_9 = 1$	0.0074
4.	$x_3 = 0$	0.0181	11.	$x_3 = 1$	0.0073
5.	$x_6 = 0$	0.0180	12.	$x_4 = 1$	0.0067
6.	$x_9 = 0$	0.0180	13.	$x_5 = 0$	0.0067
7.	$x_1 = 1$	0.0173	14.	$x_8 = 1$	0.0039

This scoring function returns the sum of the weights of the models that contain the given literal. Calling it is equivalent to computing the weighted model count of an instance that is about the same size as the original instance, so it can be costly when the input instance is large, especially for dynamic heuristics that run it for each eligible literal at every step. For large instances, it might therefore be wiser to use simpler scoring functions that only look at the basic structure of the CNF instance and do not require any kind of model counting or compilation.

5.4.2 Relative Weight

This scoring function evaluates a literal based on the weight given by the user. We now define the *relative weight* of a literal. Given a literal l, we call $W(l)$ the weight of l, and $W_{rel}(l) = \frac{W(l)}{W(l)+W(\neg l)}$ the weight of l relative to its opposite $\neg l$. Because $W_{rel}(l) + W_{rel}(\neg l) = 1$ for every literal l, we can treat the cases where the sum of the weight of a literal and the weight of its opposite is not the same for all variables.

Definition 5.2. *Given a literal l, we say that relative_weight(l) is equal to $W_{rel}(l)$.*

Example 5.4 (Relative Weight Score)**.** *Given the instance WRE as defined in Example 5.1 and a selective backbone $B = \{x_7 = 1, x_2 = 0\}$ the relative weight score for the remaining variables is presented in Table 5.5.*

Table 5.5: *Relative_weight* score for each remaining variable assignment with Example 5.1 and the selective backbone $B = \{x_7 = 1, x_2 = 0\}$

rank	assignment	*relative_weight*	rank	assignment	*relative_weight*
1.	$x_3 = 0$	0.773	8.	$x_9 = 1$	0.481
2.	$x_5 = 1$	0.719	9.	$x_4 = 0$	0.449
3.	$x1 = 1$	0.696	10.	$x_6 = 1$	0.423
4.	$x_8 = 1$	0.685	11.	$x_8 = 0$	0.315
5.	$x_6 = 0$	0.577	12.	$x_1 = 0$	0.304
6.	$x_4 = 1$	0.551	13.	$x_5 = 0$	0.281
7.	$x_9 = 0$	0.519	14.	$x_3 = 1$	0.227

Note that this scoring function is not affected by the CNF clauses, only by the weights associated with the literals. This means it is very quick to calculate, but also contains little information. All emphasis is on what the user defines as importance for the literals. However, importance of a variable given by the user might not be indicative of solutions. It would help pick the highest weighted solution but not guarantee that the highest weights will lead to valid solutions.

5.4.3 Estimated WMC

For the third scoring function, we compute an estimate of the number of models that contain a literal l. We introduce a very simple estimation based on the occurrence of literals in clauses. In general, a rough estimate $Est_{inst}(I)$ of the model count of a CNF instance I with m clauses $C_1, C_2, \ldots, C_m$ is:

$$Est_{inst}(I) = 2^n \prod_{i=1}^{m} (1 - (\frac{1}{2})^{k_i})$$

where k_i is the number of literals of C_i. Indeed, the number of assignments on the k_i variables that appear in the i^{th} clause is 2^{k_i}, and exactly one of them is forbidden by the clause, so we can expect the clause to multiply the number of models by $\frac{2^{k_i}-1}{2^{k_i}} = 1-(\frac{1}{2})^{k_i}$. This number is of course only a crude approximation, but it is easy to compute.

In the same manner we can compute $Est_{lit}(l)$, an estimate of the number of models containing literal l:

$$Est_{lit}(l) = \prod_{i=1}^{m(l)} (1 - (\frac{1}{2})^{k_i - 1})$$

where $m(l)$ is the number of clauses containing $\neg l$, and k_i is the number of literals in the i^{th} such clause. There are three differences between the formula for Est_{inst} and the formula for Est_{lit}:

1. The factor 2^n does not appear in the formula for Est_{lit}. This is because within an instance the value of n not change so the term 2^n becomes a large constant that we can ignore.

2. Not all clauses are considered in Est_{lit}. This is because if neither l nor $\neg l$ appears in a clause C, then C will have the same impact on the expected number of models containing l as on the expected number of models containing $\neg l$. Furthermore, if l appears in a clause, then this clause will be satisfied by all models containing l, so we can ignore it.

3. The exponent in Est_{lit} is $k_i - 1$ instead of k_i. This is because we already know that the literal $\neg l$ that appears in all considered clauses is not present in any model containing l, so the number of possible assignments to the variables of one of these clauses is only 2^{k_i-1}.

We can now introduce the scoring function *estimated_WMC*:

Definition 5.3. *Given a literal l, we say that*

$$estimated\_WMC(l) = W_{rel}(l) \times \frac{Est_{lit}(l)}{Est_{lit}(l) + Est_{lit}(\neg l)}$$

Example 5.5 (Estimated WMC Score)**.** *Given the instance WRE as defined in Example 5.1 and a selective backbone $B = \{x_7 = 1, x_2 = 0\}$ the* estimated_WMC *score for the remaining variables are presented in Table 5.6. This score depends on the CNF formulation of the problem. We simply negated all possible value*

combinations that are not present in Table 5.2 and added a unit clause to make $x_7 = 1$ *a trivial backbone.*

Table 5.6: *Estimated_WMC* score for each remaining variable assignment with Example 5.1 and the selective backbone $B = \{x_7 = 1, x_2 = 0\}$

rank	assignment	*estimated_WMC*	rank	assignment	*estimated_WMC*
1.	$x_3 = 0$	0.3835	8.	$x_9 = 1$	0.2424
2.	$x_5 = 1$	0.3567	9.	$x_4 = 0$	0.2227
3.	$x_1 = 1$	0.3507	10.	$x_6 = 1$	0.2132
4.	$x_8 = 1$	0.3344	11.	$x_8 = 0$	0.1612
5.	$x_6 = 0$	0.2862	12.	$x_1 = 0$	0.1508
6.	$x_4 = 1$	0.2777	13.	$x_5 = 0$	0.1416
7.	$x_9 = 0$	0.2575	14.	$x_3 = 1$	0.1144

5.5 Heuristics

We look at a similar set of heuristic types as in Chapter 4. We expand the definitions of *static* and *dynamic* and introduce a slightly different *random* baseline. The main concepts of these algorithm types are the same, but some implementation differences are present mostly due to using two different compilers. We introduce a new heuristic type *hybrid*, which combines the expensive accurate score with a fast approximation.

5.5.1 Static

The *static* heuristic type is very similar to the definition in Algorithm 2 (page 77). All $2n$ variables are evaluated with the specified scoring functions and ordered in a decreasing order given the score. At each iteration the selective backbone is extended with a new variable assignment. This is continued until the required size of the selective backbone is reached. The difference compared to Algorithm 2 is that trivial backbones defined by unit clauses are given priority and ties are broken according to literal weights. Static heuristics only prioritise trivial backbones of the initial instance, since the ordering is defined only at the very beginning.

Example 5.6 (Static)**.** *Given the instance WRE from Example 5.1 we present the first few iterations of* static *heuristic type with* actual_WMC *scoring function. Table 5.7 presents the* actual_WMC *scores of each variable assignment in decreasing order:*

- *for* $p = 1$ *: since* $x_7 = 1$ *is a proper backbone of the problem it gets assigned. Thus* $B_1 = \{x_7 = 1\}$*,*

- *for $p = 2$: we extend B_1 with the second highest ranked assignment $x_2 = 0$.* $B_2 = \{x_7 = 1, x_2 = 0\}$,
- *for $p = 3$: we extend B_2 with the third highest ranked assignment $x_4 = 0$.* $B_3 = \{x_7 = 1, x_2 = 0, x_4 = 0\}$.

Table 5.7: Ranking and *actual_WMC* score of all 2×9 variable assignments

rank	assignment	*actual_WMC*	rank	assignment	*actual_WMC*
1.	$x_7 = 1$	0.030894	10.	$x_9 = 1$	0.012281
2.	$x_2 = 0$	0.025433	11.	$x_5 = 0$	0.010275
3.	$x_4 = 0$	0.024239	12.	$x_3 = 1$	0.009152
4.	$x_6 = 0$	0.023485	13.	$x_1 = 0$	0.008771
5.	$x_8 = 0$	0.023350	14.	$x_8 = 1$	0.007544
6.	$x_1 = 1$	0.022123	15.	$x_6 = 1$	0.007409
7.	$x_3 = 0$	0.021742	16.	$x_4 = 1$	0.006655
8.	$x_5 = 1$	0.020619	17.	$x_2 = 1$	0.005461
9.	$x_9 = 0$	0.018613	18.	$x_7 = 0$	0

Note that Example 5.1 and Example 4.2 have the same set of solutions with the only difference that literals in Example 5.1 have a weight associated to them. We could consider that literals in Example 4.2 also have weights, all of them being equal to 1. We compare the assignment rankings with the *static* heuristic type on these two examples in Table 5.8. Note that for the unweighted problem, ties are broken arbitrarily thus the assignments of x_8 and x_2 could be interchanged as well as the ordering of x_9, x_6, x_4, x_3 and x_1. Both $x_2 = 0$ and $x_8 = 0$ have the same amount of solutions thus the unweighted version of the problem will see both of these assignments as equivalent. Looking at the weighted version $x_2 = 0$ has a higher weight than $x_8 = 0$ as seen in Table 5.1. Because of this, the weighted problem formulation will differentiate between these two, prioritising the assignment with the higher weight. This way, not only the user can break ties meaningfully but also use weights to prioritise assignments even if that leads to fewer solutions. Depending on how weights are distributed a lot can be changed.

Even in this small example, the variable x_3 gets assigned 1 for the unweighted problem but for the weighted problem it is assigned 0. This is due to the fact

Table 5.8: Comparison of assignment orderings for the weighted and unweighted problem using the *static* heuristic type

mc	assign	$x_7 = 1$	$x_8 = 0$	$x_2 = 0$	$x_9 = 1$	$x_6 = 0$	$x_4 = 0$	$x_3 = 1$	$x_1 = 1$	$x_5 = 1$
	score	10	7	7	6	6	6	6	6	5
wmc	assign	$x_7 = 1$	$x_2 = 0$	$x_4 = 0$	$x_6 = 0$	$x_8 = 0$	$x_1 = 1$	$x_3 = 0$	$x_5 = 1$	$x_9 = 0$
	score	0.0309	0.0254	0.0242	0.0235	0.0234	0.0221	0.0217	0.0206	0.0186

Algorithm 4 *extend*:

Parameters:

- I: a CNF instance
- Q: set of eligible literals
- l: a literal

```
1: for every clause c in I do
2:    if c contains l then
3:       remove c from I
4:    else if c contains ¬l then
5:       remove ¬l from c
6:    end if
7: end for
8: add the unit clause l to I
9: Q ← Q \ {l, ¬l}
10: return I, Q
```

that $x_3 = 0$ has a much higher weight, 0.773 compared to the weight of $x_3 = 1$ which is 0.227 but both assignments have almost the same amount of solutions.

5.5.2 Dynamic

The previous definition of the *dynamic* heuristic type is presented in Algorithm 3 (page 78). For this chapter, two changes are made: trivial backbones defined by a unit clause are given priority and the *extend* function has a slightly different implementation. Literals that form a unit clause in the instance are given priority because, since these literals appear in all models for the instance, we know that picking one of these literals cannot decrease the weighted model count. In case we use an exact weighted model count as score this difference will make no impact, but in case of estimated scores it makes a huge impact. If no such literal is present in the instance, then we pick the literal that has the highest score according to some given metric f, breaking ties by picking the literal with the highest weight. The *extend* function is defined as updating the clauses of the CNF instance as shown in Algorithm 4.

Once a decision has been made on which literal to assign, Algorithm 4 is called, which removes from the instance clauses that are satisfied by the chosen literal, and removes its opposite from the remaining clauses. We also introduce a unit clause comprising the literal that was picked, to record that the corresponding variable has been assigned and is not unconstrained. Finally, we remove both the chosen literal and its negation from consideration for future iterations. This ensures that no variable is assigned twice.

Because dynamic heuristics modify the instance every time they add a literal to the selective backbone, they need to update the score of each literal before each pick, and also to keep track of clauses that have become unit clauses after the opposites of previously chosen literals have been pruned out. On the one hand, the information uncovered in the early steps of dynamic approaches can be made available for later steps, improving the quality of the selective backbones that are built. On the other hand, if the scoring function is expensive to compute, then determining its value for each remaining eligible literal at every step could be a burden on the runtime of dynamic methods. In contrast, static heuristics compute the scoring function only once for each literal, and at each iteration pick the remaining literal that achieved the highest score in this initial ranking.

Example 5.7 (Dynamic)**.** *Given the instance WRE from Example 5.1 we present the first few iterations of* dynamic *heuristic type with* actual_WMC *scoring function:*

- *for* $p = 1$ *: this is equivalent to* static *as shown in Example 5.6:* $B_1 = \{x_7 = 1\}$*,*
- *for* $p = 2$ *: since the first assignment did not remove any solutions and there is no other proper backbone, the second iteration is also equivalent to the* static *selection, thus* $B_2 = \{x_7 = 1, x_2 = 0\}$*,*
- *for* $p = 3$ *: since* B_2 *reduced the selected solution set to* $\mathcal{S}_2 = \{S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$ dynamic *and* static *heuristic types will behave differently. Here the highest* actual_WMC *is 0.0215 obtained by* $x_8 = 0$*. This is calculated by summing the weighted model count of the selected solution set* $\mathcal{S}_3 = \{S_4, S_6, S_7, S_8, S_9\}$ *using the Table 5.3.* $B_3 = \{x_7 = 1, x_2 = 0, x_8 = 0\}$*,*
- *for* $p = 4$ *: given* B_3 *the highest weighted model count to extend this selective backbone is either* $x_4 = 0$ *or* $x_5 = 1$*. They both lead to the same selected solution set* $\mathcal{S}_4 = \{S_4, S_6, S_7\}$ *with* actual_WMC *of 0.0187. Since* $x_5 = 1$ *has a higher weight this will be selected,* $B_4 = \{x_7 = 1, x_2 = 0, x_8 = 0, x_5 = 1\}$*.*

If we compare the *dynamic* heuristic type for the weighted problem in Example 5.1 and its unweighted version in Example 4.2 we notice some differences. At $p = 4$ the unweighted problem will have the selective backbone $BD_4 = \{x_7 = 1, x_8 = 0, x_5 = 1, x_4 = 0\}$ and the weighted problem will have $B_4 = \{x_7 = 1, x_2 = 0, x_8 = 0, x_5 = 1\}$. The set of variables assigned are similar, but not exactly the same. Also for $p = 2$ the unweighted problem has two equivalent choices $x_2 = 0$

and $x_8 = 0$ but once it picks one the other is not the next choice anymore. In the unweighted problem there is no tie because even though they have the same number of solutions the weighted model count of each solution set is different, thus yielding a different score for the two assignments.

Note when combining the *relative_weight* with *static* and *dynamic* heuristic types the only difference is that *dynamic* recognises trivial backbones defined by unit clauses during the iterations. If a literal does not become a unit clause there is no difference in its score between the *static* and *dynamic* types or between the different iterations.

5.5.3 Random

The baseline heuristic, *random*, simulates returning a random score by picking a completely random literal. This heuristic is an extension of the *random* from Section 4.4.2. This heuristic has been updated to work similarly to the updated version of *dynamic*, with the difference that instead of choosing the best assignment according to a score a random assignment is chosen. Because of this, we refer to this heuristic as *dynamic random*.

The difference with respect to *random* from Chapter 4 is that this heuristic picks unit clauses when available, so it is possible for it to build selective backbones of decent sizes before reaching a conflict. This type of prioritization was not added to the previous heuristics defined for the unweighted problems. One reason for this is that with the model count scoring function, the order would not have changed, since a unit clause indicates the variable assignment is a proper backbone, thus has maximal model count compared to other variable assignments. Ideally, the same behaviour would happen with the ratio scoring function as well. As the model count does not change when assigning a backbone, the only difference would have come from the size, which probably would not have changed much as well. Another reasoning is that in Chapter 4 we used the compiled representation to extend with a new clause, whereas in this chapter we keep track of the CNF. This way, observing trivial backbones, defined by unit clauses, is done without any overhead. For example Table 5.12 shows that on average around 74% of variables are assigned before leading to a conflict.

This heuristic type updates the instance at every iteration as *dynamic* rather than imposing an ordering at the very beginning as *static* and *random* from Chapter 4. We found in our early experiments that the static heuristic with random literal ranking often reaches a conflict in the first few steps, and therefore most of the

selective backbones that it obtains lead to a weighted model count of 0 and an empty instance representation. For this reason, we will only present the results for the version *dynamic random*.

5.5.4 Hybrid

The last heuristic type, *hybrid*, is a *dynamic* heuristic type combining a fast approximation with an expensive scoring function. We look at the combination of *estimate_WMC* and *actual_WMC* scoring function. This heuristic is described in Algorithm 5. Each iteration starts with checking if there is a unit clause in the instance I. If there is one its literal gets added to the selective backbone, otherwise we get the next assignment based on the scoring functions. It first computes *estimated_WMC* (Definition 5.3) for all remaining literals as shown in Line 7 in Algorithm 5. The function *winning_variable_instantiation()* is according to Definition 4.3. Let e be the literal with the highest score obtained using *estimated_WMC* scoring. In Line 8 we compute $E = \{x \mid estimated\_WMC(x) \geq \frac{90}{100} \times estimated\_WMC(e)\}$. The heuristic then computes *actual_WMC* (Definition 5.1) for all literals in E, and picks the literal of E that got the highest score with *actual_WMC* as shown in Line 9. This hybrid heuristic is faster than computing *actual_WMC* for every single literal and, as we will show in the next

Algorithm 5 How to obtain a selective backbone with a hybrid heuristic.

Parameters:
- p: desired size of the selective backbone
- I: a CNF instance with n variables
- f: the *estimate_WMC* scoring function
- g: the *actual_WMC* scoring function

```
1:  B ← ∅
2:  Q = { all 2n literals in I to the pool of eligible literals }
3:  for i ← 1 to p do
4:    if I contains at least one eligible unit clause then
5:      l ← an eligible unit clause literal
6:    else
7:      e ← winning_variable_instantiation(B, Q, f)
8:      E = {x | x ∈ Q and f(x)/f(e) ≥ 90/100}
9:      l ← winning_variable_instantiation(B, E, g)
10:   end if
11:   I, Q ← extend(I, Q, l)
12:   B ← B ∪ {l}
13: end for
14: return B
```

Section, it obtains higher ratios than *estimated_WMC*. This heuristic makes some calls to an exact model counter, but these calls are a lot fewer than the calls made by the two most expensive heuristics.

Example 5.8 (Hybrid WMC)**.** *Given the instance WRE from Example 5.1 we present the first few iterations of the* hybrid *heuristic.*

- *for* $p = 1$ *: this is equivalent with the first iteration of* dynamic *and* static *since* $x_7 = 1$ *is a trivial backbone since we defined it that way.* $B_1 = \{x_7 = 1\}$*,*
- *for* $p = 2$ *: the ranking using the* estimated_WMC *score is shown in Table 5.9. Out of these, only the top 4 satisfy the criteria of the estimated score being at least* 90% *of the best estimated score. The scores of these 4 assignments are recalculated using the* actual_WMC *function, obtaining: 0.0217 for* $x_3 = 0$*, 0.0254 for* $x_2 = 0$*, 0.0206 for* $x_5 = 1$ *and 0.0221 for* $x_1 = 1$*. This changes the ordering of the candidates with* $x_2 = 0$ *being the highest. Thus* $B_2 = \{x_7 = 1, x_2 = 0\}$ *just like for the* dynamic *heuristic but only doing the expensive calculations 4 times rather than 16 times,*
- *for* $p = 3$ *: the initial ranking of* estimated_WMC *and the ranking of the top solutions using* actual_WMC *can be found in Table 5.10. This time only the top 3 assignments satisfy the condition to be evaluated with* actual_WMC*. The fourth assignment is actually* $x_8 = 0$ *the assignment chosen for* $p = 3$ *for* dynamic*, but its estimated score is only about* 88% *of the best score so it does not get evaluated. It is important to note the* 90% *is an arbitrary number defined by us, would it be lower* hybrid *would select the same assignment for this iteration as well. But with this setup* $B_3 = \{x_7 = 1, x_2 = 0, x_1 = 1\}$*.*

Table 5.9: Ranking of assignments for $p = 2$ Example 5.8

rank	assignment	*estimated WMC*	*actual WMC*	rank	assignment	*estimated WMC*
1.	$x_3 = 0$	0.3850	0.0217	9.	$x_9 = 1$	0.2414
2.	$x_2 = 0$	0.3598	0.0254	10.	$x_4 = 0$	0.2254
3.	$x_5 = 1$	0.3595	0.0206	11.	$x_6 = 1$	0.2107
4.	$x_1 = 1$	0.3494	0.0221	12.	$x_8 = 0$	0.1587
5.	$x_8 = 1$	0.3398		13.	$x_1 = 0$	0.1514
6.	$x_6 = 0$	0.2896		14.	$x_2 = 1$	0.1419
7.	$x_4 = 1$	0.2744		15.	$x_5 = 0$	0.1405
8.	$x_9 = 0$	0.2585		16.	$x_3 = 1$	0.1139

Table 5.10: Ranking of assignments for $p = 3$ Example 5.8

rank	assignment	*estimated_WMC*	rank	assignment	*actual_WMC*
1.	$x_3 = 0$	0.3850	1.	$x_1 = 1$	0.0221
2.	$x_5 = 1$	0.3595	2.	$x_3 = 0$	0.0217
3.	$x_1 = 1$	0.3494	3.	$x_5 = 1$	0.0206
4.	$x_8 = 1$	0.3398			
5.	$x_6 = 0$	0.2896			
6.	$x_4 = 1$	0.2744			
7.	$x_9 = 0$	0.2585			

5.6 Evaluating Selective Backbones on Weighted Problems

In this section we describe the dataset, the details of the experiments conducted and their results. We evaluate eight heuristics: *static* and *dynamic actual_WMC*, *hybrid_WMC*, *static* and *dynamic estimated_WMC*, *static* and *dynamic relative_weight* and *dynamic random*.

5.6.1 Setup

We looked at a similar dataset as in Chapter 4: a set of generated instances denoted as AB containing 10 instances with 15 variables and another 10 with 30 variables; a subset of the ISCAS suite containing 14 instances; and 134 instances from the planning suite. The ISCAS suite contains instances with 26-252 variables and the planning suite contains instances with 5-577 variables. The number of models for all instances ranges from 4e0 to 6.2e85, with a median of 8.1e6. These benchmarks have previously been used by numerous other papers about knowledge compilation and model counting [OD15, LLM20, CLM21]. From the ISCAS and planning instances we only take into account instances that compiled within the set timeout. Since in Chapter 4 we used a different compiler instances behave slightly different so the datasets do not completely overlap.

We generated a weight w between 0 and 3 for each positive literal following the uniform distribution and assigned its negation the weight of $3-w$. The choice of 3 is an arbitrary scaling factor since the weights 0-1 were producing small weighted model count values leading to underflow problems. We used preprocessing as described in Section 5.6.2.

We evaluate eight approaches to obtain a compact compilation with respect to weighted model counting. In the previous chapter we focused on comparing the

static and *dynamic* heuristic types. In this chapter the emphasis is more on the scoring function used. As defined earlier we look at two types of heuristics, *static* and *dynamic*, each with the scoring functions: *actual_WMC*, *estimated_WMC* and *relative_weight*. We also look at *hybrid_WMC* as defined in Section 5.5.4. As a baseline, we use the *dynamic random* heuristic as defined in Section 5.5. We refer to both *static* and *dynamic estimated_WMC* and *relative_weight*, as well as *dynamic random* as simple heuristics.

We used d4 [1] on the command line with its default parameters to compile the instances and perform weighted model counting and model counting as well. This tool compiles to the decision-DNNF representation described in Section 2.3.4 (page 38). All experiments were performed on a machine with Intel(R) Xeon(R) CPU E5620 @2.40GH running Ubuntu 22.04.2 LTS. Each instance was run for 1 hour with each heuristic.

5.6.2 Preprocessing

Before applying the heuristics, we repeat the following operations for each instance:

- If a clause contains the same literal more than once, we remove all but one occurrence of this literal in the clause.
- If a clause contains both a literal and its negation, we remove the clause.
- If two clauses are composed of the exact same set of literals, regardless of the order in which literals appear in the clause, we remove one of the clauses.
- If there is a trivial backbone in the form of a unit clause l, we eliminate it, remove all clauses containing l, and remove all occurrences of $\neg l$ from the other clauses.
- We eliminate any variable that is not constrained by any clauses.

We do the above until the instance satisfies all of the following properties:

1. No variable occurs more than once in a clause.
2. No two clauses are composed of the exact same set of literals.
3. There is no unit clause.

[1] https://github.com/crillab/d4

4. All variables appear in at least one clause.

Note that although we used the same dataset as in Chapter 4 we did not perform preprocessing there. One reason is that the scoring functions in Chapter 4 are not affected by these sorts of properties since they do not rely on literal counts from the CNF. The only preprocessing that would have impacted it is the elimination of proper backbones, defined by unit clauses. However, even this, as we mentioned before, especially using the model count scoring function, would have been fixed in the first iterations.

5.6.3 Methodology

As defined earlier each heuristic implements an iterative approach, where each variable gets assigned the best value according to the scoring function used. Satisfiability of the model before making the assignment at every iteration is only ensured for the dynamic heuristics with *actual_WMC* scoring function. Every other combination of heuristic and scoring function could lead to making an assignment that would lead to a conflict, making the model unsatisfiable. In case a heuristic reaches zero model count or times out within one hour, for the rest of the iterations that it did not complete, zero is taken as a value for its model count and weighted model count. This is important so that for each instance we compare the same iterations.

For all eight heuristics, we compiled the instances and calculated the weighted model count and model count at each iteration for post-analysis only. We use the same evaluation metric as defined in Section 4.5 (page 80): the adjusted ratio. The difference in the definition of the adjusted ratio for this chapter is that the size of the solution set is expressed as the weighted model count and the compilation size is the size of a different representation. Since in this chapter we use a different compiler that compiles to a different representation, the compilation size in the adjusted ratio formula refers to the size of this representation, the d-DNNF. Informally the adjusted ratio is the ratio between the compactness of the current model and the initial problem. Compactness is expressed as the ratio between weighted model count and compilation size.

Runtime for each heuristic can be found in Table 5.11. Note that in the table runtime for using the scoring function *actual_WMC* includes an extra step for performing model counting at each iteration. In each iteration, this heuristic spends time compiling each possible assignment which contains writing out CNF files so that d4 can be called from the command line. The extra step of one model

Table 5.11: Execution time of each heuristic expressed in seconds

		Min	**Max**	**Average**	**Median**
static	actual_WMC	1.246	395.674	66.577	43.581
dynamic		5.857	3597.082	1906.339	1895.844
	hybrid_WMC	0.001	639.823	65.457	32.174
static	estimated_WMC	0.002	0.471	0.078	0.046
dynamic		0.003	27.239	1.563	0.515
static	relative_weight	0.001	0.079	0.014	0.010
dynamic		0.003	2.441	0.145	0.049
dynamic	random	0.001	3.017	0.145	0.040

counting operation per iteration is not as significant as the $2 \times n$ compilations to obtain the best next assignment. We can see there are huge differences in runtimes between the heuristics. Dynamic heuristics tend to run longer than their static version since they update the CNF at each iteration and also spend some time calculating the new scores. On average it takes very little time to run the simple heuristics so comparing them to each other is not as relevant. Since we stop experiments when a conflict is reached for each heuristic we take into account the runtime while the subproblem remains satisfiable. The *static* heuristics often reach conflict since the assignments are ordered at the very beginning and are not updated as they get applied. This means an assignment might have a great score at the very beginning but the solutions with that assignment might be eliminated by previous assignments. In terms of average runtime *hybrid_WMC* behaves similarly to *static actual_WMC*, but its median is lower.

Table 5.12 presents an aggregated view of the percentage of variables that have been assigned as well as the number of instances that have assigned all variables for each heuristic individually. Out of the total 168 instances only *dynamic actual_WMC* times out, for 56 instances. No other heuristic reaches the maximum timeout as shown in Table 5.11. The reason the rest of the instances may not assign all variables is that they reach a conflict at some iteration. Static heuristics are more prone to this behaviour since they do not choose the next assignment based on the most current CNF. As seen in Table 5.12 *static relative_weight* reaches a conflict for all instances, *static estimated_WMC* only completes assigning all variables for 4 instances and *static actual_WMC* reaches a conflict for 74 instances. As seen in Table 5.12, the heuristics that seem the best at successfully assigning variables are *static actual_WMC* and *hybrid_WMC*. The former has the higher minimum and average number of assigned variables, always assigning more than 60% of the variables, while the latter also has a high average

Table 5.12: Variables assigned before conflict and/or timeout

		Percentage of variables assigned				NB
		Min	Max	Average	Median	finished
static	actual_WMC	61.21	100	96.00	100	94/168
dynamic		2.71	100	73.78	100	112/168
	hybrid_WMC	11.7	100	94.24	100	146/168
static	estimated_WMC	2.46	100	29.02	18.56	4/168
dynamic		4.93	100	87.12	100	127/168
static	estimated_WMC	0.69	66.67	10.55	3.97	0/168
dynamic		0.69	100	72.58	100	94/168
dynamic	random	0.55	100	74.31	100	98/168

and is the heuristic that completes the most instances. The dynamic heuristics tend to complete more instances, which is understandable since they propagate the assignments after each iteration, thus a more informed decision of the actual state of the CNF is made at each step. That is the reason the *dynamic random* heuristic seemingly performs well, completing assignments for a high number of instances, but we will later see that the quality of those partial compilations is not good. Looking at the percentage of variables assigned is interesting from the point of view that we do not know at what percentage the most efficient compilation happens.

Another interesting analysis to look at is the percentage of unit clauses assigned with respect to the number of assignments made by each heuristic, presented in Table 5.13. This is only relevant for dynamic heuristics as they prioritise assigning unit clauses when present and then propagate this, while static heuristics only do that at the first iteration. The simple heuristics *dynamic relative_weight* and *dynamic random* assign around 70% of the variables because they are unit clauses not because their score is high. Later on, when looking at the efficiency and compactness of the partial compilation we can observe that the *dynamic*

Table 5.13: Percentage of variables assigned being chosen because of it being a unit clause with respect to number of assignments made. Note: static heuristics do not identify unit clauses within iterations

	Heuristic	**Min**	**Max**	**Average**	**Median**
dynamic	actual_WMC	0	69.02	25.00	29.14
	hybrid_WMC	0	92.57	56.75	60.33
dynamic	estimated_WMC	0	93.02	57.09	60.55
dynamic	relative_weight	0	99.43	72.37	77.78
dynamic	random	0	99.30	72.10	76.11

relative_weight leads to a different selective backbone than *dynamic random*, one that provides a better partial compilation. Interestingly both *dynamic estimated_WMC* and *hybrid_WMC* reveal more unit clauses than *dynamic actual_WMC*.

5.6.4 Efficiency

The previous tables shine a light on some behaviours of the heuristics but not on the performance of the partial compilations. In order to evaluate the performance of the heuristics we look at the efficiency and the adjusted ratio. The aim of the heuristics is to obtain the best selective backbone that leads to the most compact compilation. We define compactness in terms of the ratio between WMC and the size of the representation, maximising the WMC and minimising the representation size. Since the size of the representation is canonical with respect to the variable ordering used, and we are not optimising the variable ordering, in general when the representation gets smaller it is due to fewer models being represented.

In Chapter 4 we show that there can be a partial compilation that decreases more the size of the representation than the number of models it does not represent. In this chapter, we aim to show that there exist partial compilations of weighted problems such that the tradeoff between smaller WMC and size of representation is not directly proportional. To show this we plotted the percentage reduction of WMC with respect to the initial WMC against the percentage of reduction of size with respect to the initial representation size. We call this the efficiency and the plots in Figure 5.1, Figure 5.2, Figure 5.3 show the median efficiency of dataset *AB*, ISCAS and *planning* instances respectively. The grey 45 degree line denotes the efficiency when the loss of WMC and representation size is directly proportional. We want to be above this line, as close as possible to the top right corner, in order to obtain a compact compilation. Both timeout and inconsistency are treated the same way, weighted model count and size are taken as 0. Because instances have vastly different number of variables, we show 50 data points for each heuristic, with the i^{th} data point being the median of the values achieved after $2i\%$ of the variables have been assigned for each instance.

In all Figure 5.1, Figure 5.2, Figure 5.3 *dynamic random* is below the 45 degrees line, meaning it does not achieve worthy partial compilations. The *static* and *dynamic actual_WMC* heuristics perform the best, with static occasionally outperforming dynamic, due to the fact that in some cases the dynamic times out,

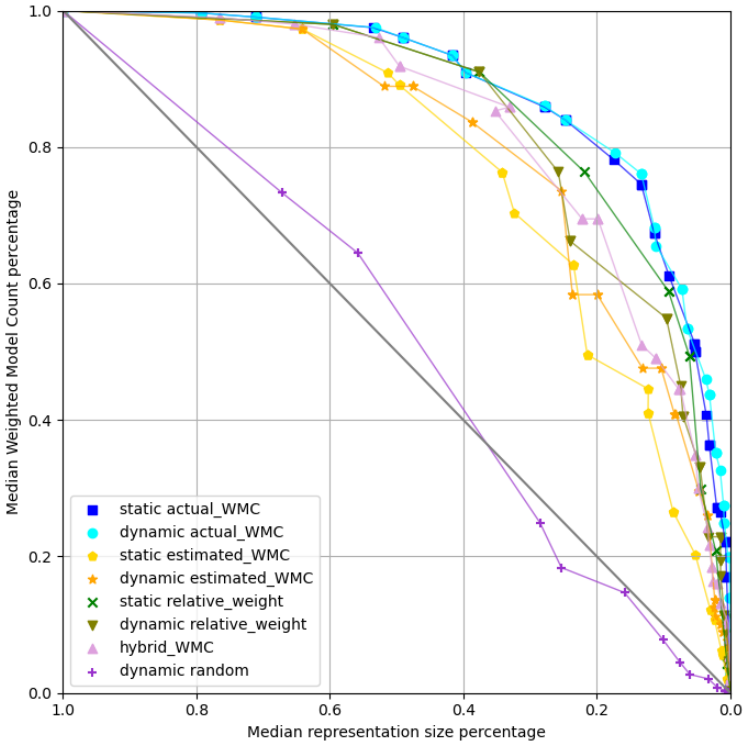


Figure 5.1: Median efficiency over Dataset A and Dataset B.

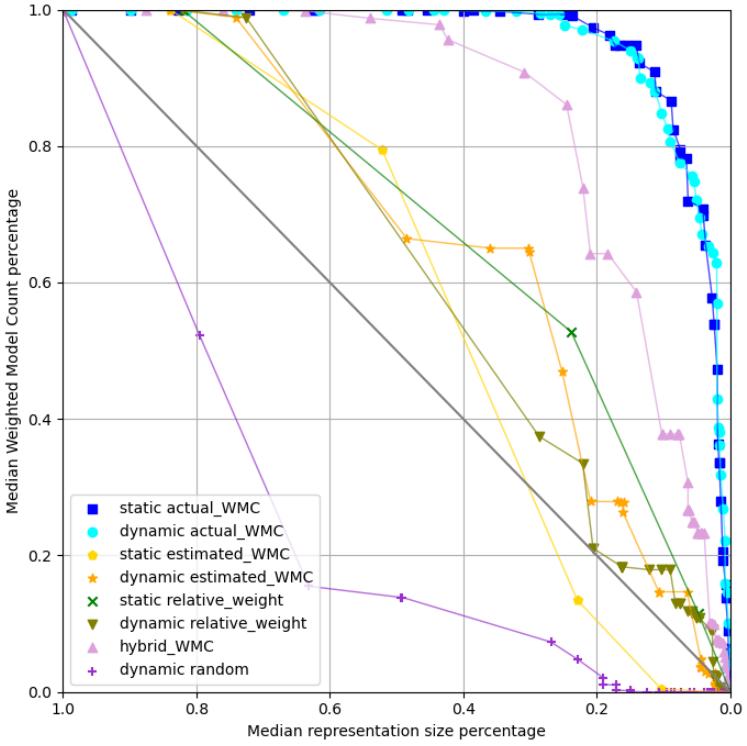


Figure 5.2: Median efficiency over ISCAS instances.

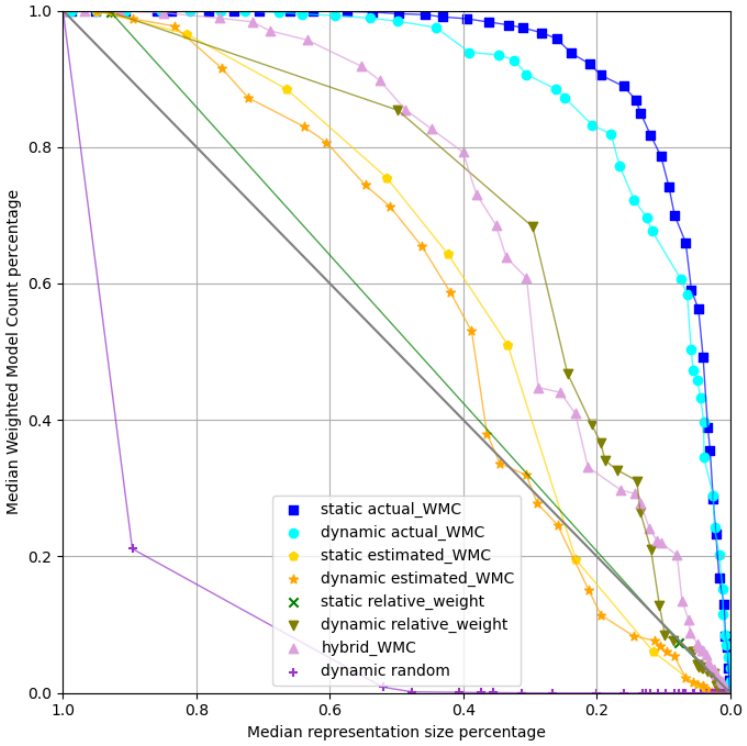


Figure 5.3: Median efficiency over planning instances.

bringing its median value down. This is due to the fact that their behaviour is very similar for a lot of instances but dynamic times out 33.3% of the time within all 168 instances, with a range of 2.71% to 100% of the variables assigned. Static does not time out but it reaches inconsistency 44% of the time between all instances, but this happens at the earliest after 61.21% of variable assignments. The next best heuristics are the *dynamic relative_weight* and *hybrid_WMC*, with the latter one better since more observations occur closer to the high WMC percentage. The rest of the heuristics, *relative_weight* and *estimated_WMC* both static and dynamic seem to behave very similarly, with no clear best. For Dataset *AB* the simple heuristics are performing better, whereas for the ISCAS suite it is the largest gap to the best heuristic.

In Figure 5.4 we compare the efficiency of the dynamic heuristics with respect to model counting and weighted model counting. Data points represented with a square denote the WMC percentage reduction, whereas the ones with a cross denote the MC percentage reduction. All heuristics use a scoring function that uses the weights of the literals when ranking selective backbones. Because of this as expected the efficiency with respect to the WMC will be better than with respect to only MC. Except *dynamic random* all efficiencies of WMC are on the right hand side of the 45 degree line. This shows that these heuristics truly select

the partial compilation that represents the most important solutions, not just the largest solution set.

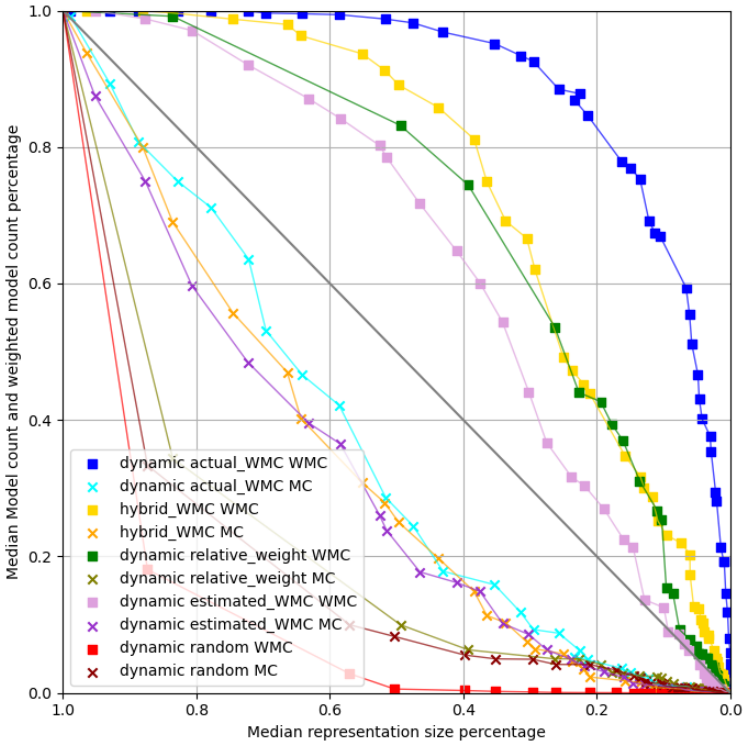


Figure 5.4: Median efficiency of MC and WMC over all instances.

5.6.5 Adjusted ratio

The efficiency plots are a good way to visualise the evolution of the compactness of the compilations but are not suitable to show how instances behave with the same sized selective backbone. For this reason in Figure 5.5, Figure 5.6, Figure 5.7 we present the median of the adjusted ratio. Ratio 1 will be equivalent to the compactness of the initial model count, whereas the higher the ratio the more compact the partial compilation. Here too *static* and *dynamic actual_WMC* reach the best performance, with static outperforming dynamic due to the timeout of the dynamic heuristic. Among the other heuristics, the best one is *hybrid_WMC*, which keeps a somewhat steady adjusted ratio.

For Dataset *A and B* in Figure 5.5 we can see that *dynamic estimated_WMC* reaches a higher peak than the rest of the simple heuristics, but until 40% of the selective backbone size *dynamic relative_weight* is outperforming it. For the ISCAS and *planning* instances as seen in Figure 5.6, Figure 5.7 there is a larger difference between the heuristics. *Static actual_WMC* and *dynamic*

actual_WMC obtains the highest ratio, third best is *hybrid_WMC*, *dynamic relative_weight* slightly outperforms *dynamic estimated_WMC* and the static version of both simple heuristics and *dynamic random* achieve the lowest ratio.

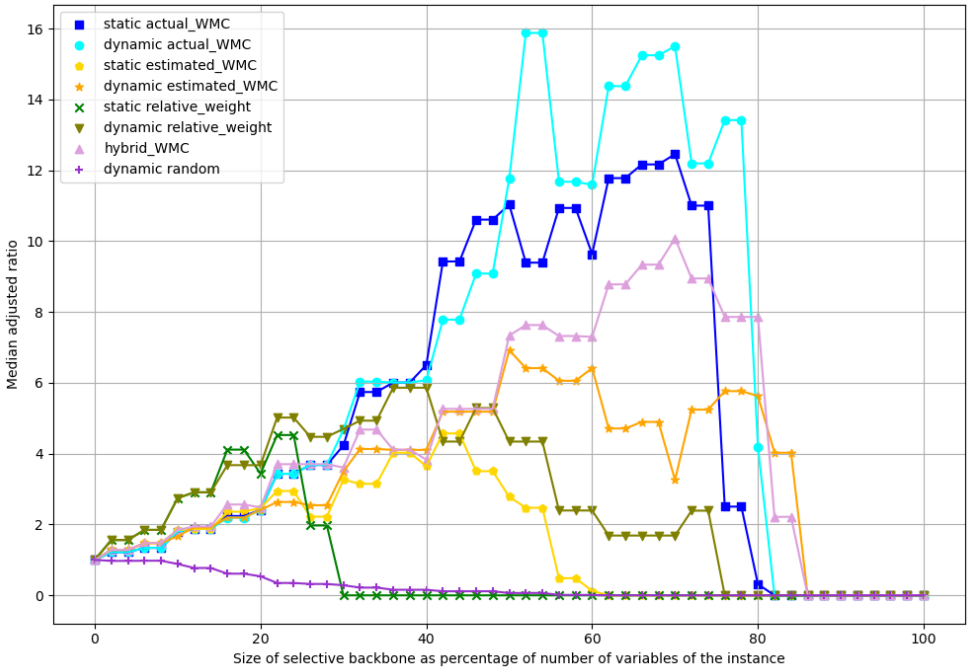


Figure 5.5: Median adjusted ratio over Dataset A and Dataset B.

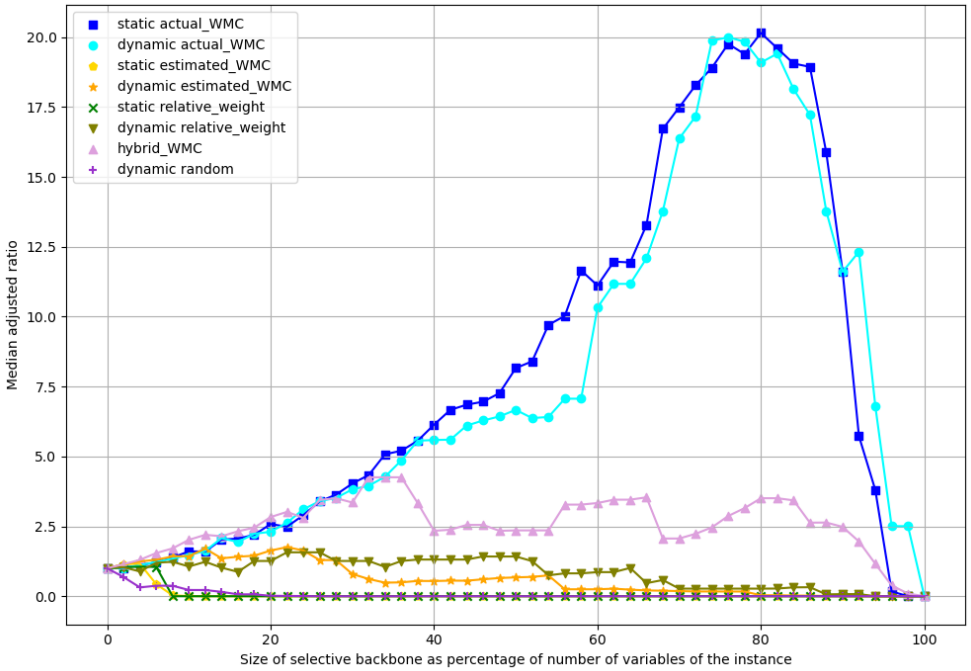


Figure 5.6: Median adjusted ratio over ISCAS instances.

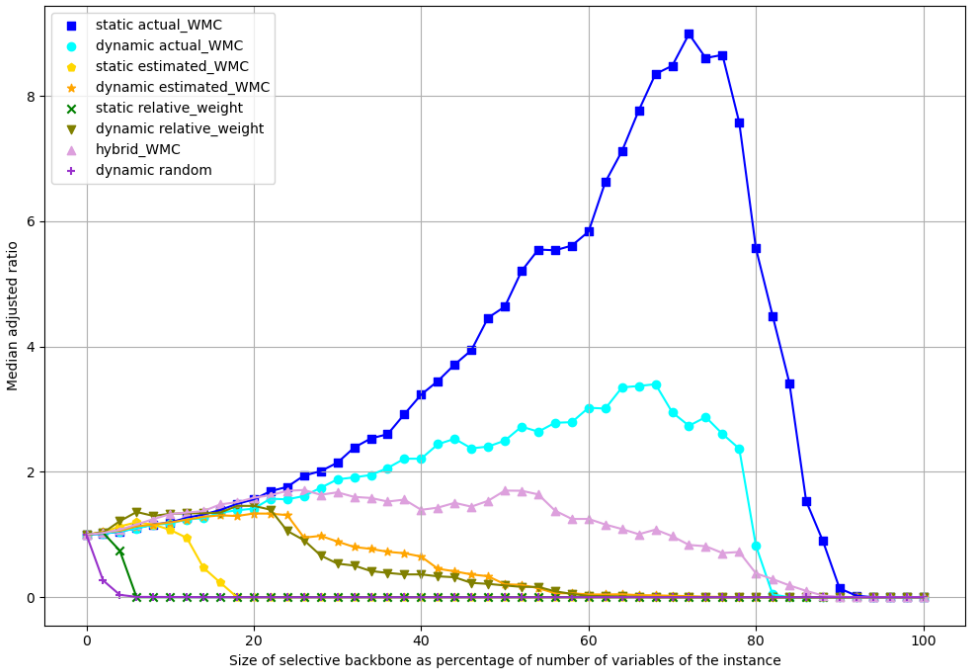


Figure 5.7: Median adjusted ratio over planning instances.

Table 5.14 presents the number of times each heuristic was the one that reached the highest adjusted ratio of an instance, excluding instances where more than one heuristic reached the highest adjusted ratio. The heuristics that most often get the highest adjusted ratio are unsurprisingly the two most computationally expensive ones, but *hybrid_WMC* also manages to reach a higher adjusted ratio than any other heuristic for quite a few instances, while it is very rare that a simple heuristic is the one with the best result.

Table 5.14: Number of best adjusted ratio per heuristic

	Number of times best AR	Percent of times best AR
static actual_WMC	63	45.99%
dynamic actual_WMC	55	40.15%
hybrid_WMC	11	8.03%
all five simple heuristics combined	8	5.84%

Figure 5.8 is similar to Figure 5.4 in the sense that here too we compare model counting and weighted model counting the heuristics obtained in each iteration. Data points represented with a square denote the adjusted ratio with WMC and data points with a cross denote the adjusted ratio obtained with MC. As previously, all heuristics take into consideration the weights of the literals when ranking selective backbones. Except *dynamic random* all heuristics obtain an adjusted

ratio with WMC that is better than the initial ratio, whereas the adjusted ratio with MC decreases from the very beginning. As expected solutions with high weights are prioritised, even if they lead to a smaller solution set represented in the partial compilation.

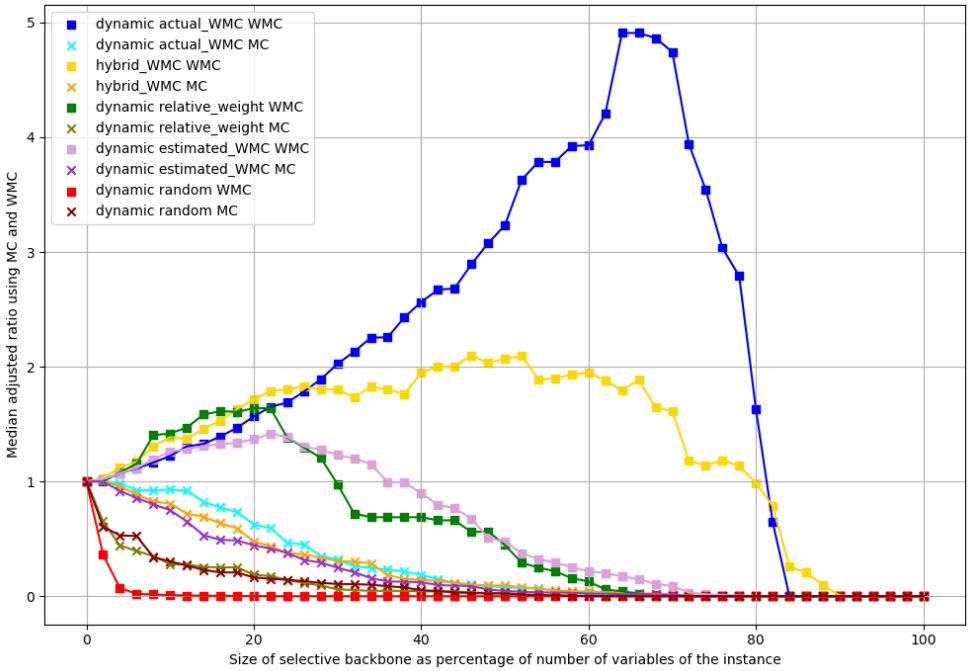


Figure 5.8: Median adjusted ratio of MC and WMC over all instances.

5.7 Determining a Size for the Selective Backbone

In this section we present the approach we took to infer from a set of small instances the size of the selective backbone that is most promising for larger instances too. The proposed iterative approaches restrict the compiled solution set by assigning a new variable at each iteration. When we assign a variable we add it to the selective backbone, increasing its size by one. Thus defining the size of the selective backbone directly impacts the partial compilation.

Instead of trying to compile the instance at every iteration and evaluating if it is adequate we propose a method to define what iteration should lead to an efficient partial compilation. Section 5.7.1 gives the motivation, Section 5.7.2 describes the approach we took and Section 5.7.3 evaluates the approach on a dataset with larger instances.

5.7.1 Motivation

Until now we talked about iterative heuristics, observing the behaviour at all iterations. This is useful to see the progression of the selective backbone to be able to evaluate the usefulness of such an approach. However, in a more realistic scenario, one would want to execute a single compilation, the one that would result in the best partial compilation. It is not necessary to compile or evaluate each iteration of the heuristics but until now there is no other measurement that would indicate when to stop the heuristic. In case we assign too many variables the partial compilation might get very small which is great, but we might lose too many solutions, obtaining a useless partial compilation. In case we assign too few assignments, we might not reduce the size of the compilation enough. This is why it is important to have some intuition about how many variables we should fix, that is what is a good size for the selective backbone.

Using the selective backbone approach to obtain a partial compilation and the previously discussed methods to obtain them, we have three main components we can change: the heuristic type, the scoring function and the size of the selective backbone. In previous sections we discussed about the heuristic types and the scoring functions, but always looking at all iterations. In this section we focus on defining an approach that could provide the number of iterations one should execute, that is equivalent to the size of the selective backbone.

Researchers have looked at methods to predict run times of SAT problems as well as satisfiability [LHHX14, ATO16]. Features such as the number of variables and clauses, constraint graph statistics, etc. are used to train machine learning methods to predict solving time of a problem or whether it can be satisfied. Portfolio approaches exist for solving SAT instances where a main algorithm decides which SAT solver to use for a specific instance given a portfolio of solvers [XHHL08]. In the machine learning community algorithm selection and hyper parameter tuning methods were developed to help with the choice of a parameter value of algorithm usage [KTH+17].

We develop a very simple method to generalise from a small set of instances. We assume that the behaviour of our heuristics is similar between structurally similar instances but size wise larger. Thus we propose a method that takes a dataset of small instances and identifies the best size for the selective backbone from it. Using this p percentage of variables to assign one only needs to run any of the heuristics for p iterations and compile the obtained CNF. To identify p we propose a very simple approach. We use the same evaluation metric as previously,

the adjusted ratio. We aggregate this across all instances at each iteration and identify the highest adjusted ratio, the location of this being p, the size of the best selective backbone.

5.7.2 Methodology

Our goal is to produce partial compilations that are as compact as possible with regards to the compilation of the original instance. A formal measure for this relative compactness is what we call the *adjusted ratio*, which we define in Section 4.5. An adjusted ratio of 1 is equivalent to the compactness of the initial instance, whereas the higher the adjusted ratio the more compact the partial compilation.

We aim to build a selective backbone for a large instance I with a high adjusted ratio, without any intermediate compilation of I. This means that we do not compute the adjusted ratio at each iteration to check whether it is good enough to stop, instead, we decide on the number of variables to assign before starting to build the selective backbone. The only two compilations of I that are done are of the initial instance and the instance obtained from I by assigning the selective backbone. These two compilations are not needed to compute the selective backbone, they are only done to measure the improvement.

We determine and evaluate the selective backbone size that is most likely to lead to a high adjusted ratio in the following manner. Let $\mathcal{I}$ be a set of instances. We partition $\mathcal{I}$ into two sets $\mathcal{I}_{small} = \{I_1, I_2, \ldots, I_{n_1}\}$ and $\mathcal{I}_{large} = \{J_1, J_2, \ldots, J_{n_2}\}$. For a given heuristic h, we run h on all n_1 instances from $\mathcal{I}_{small}$. For each $i \in [1, n_1]$, for each percentage p (sampled depending on the number of variables of instances in $\mathcal{I}_{small}$), we compute the adjusted ratio $AR_{i,p}$ obtained after assigning $p\%$ of the variables of I_i with h. Computing the adjusted ratio requires compilation, but because the instances in $\mathcal{I}_{small}$ are small, this should not take too much time. Let p_{max} be the percentage of variables assigned where the highest median adjusted ratio is observed.

For each $j \in [1, n_2]$, we run h to compute a selective backbone B_j of size $p_{max}\%$ of the number of variables in J_j. Using h we run $|B_j|$ number of iterations to obtain the required partial model. We only compute the adjusted ratio for the instance J'_j obtained from J_j after assigning B_j, so we only need to do two compilations for each j: one for J_j and one for J'_j. In fact we only compile the initial problem J_j in order to prove the compactness of the partial compilation J'_j. At this stage we can run the heuristic h for $|B_j|$ iteration on any instance to obtain a promising

selective backbone.

This prediction approach is very simple but effective since instances seem to have a similar behaviour when looking at the progression of the adjusted ratio. Ideally, instances from $\mathcal{I}_{small}$ and $\mathcal{I}_{large}$ should be part of the same dataset, representing similar problems on smaller and larger scales. If this is the case, then the trend of the adjusted ratio of the larger instances should be similar to the smaller instances trend. This way the location of the best adjusted ratio from the small instances should be relevant for the large instances as well.

5.7.3 Results

In this section, we test whether the approach described previously can successfully pick a selective backbone size with a high adjusted ratio for large instances, without compiling any of these large instances. It is important to note that the obtained $p\%$ of variables to fix is not a universal number that should yield good results in any problem. With this approach, we do not guarantee the best p, we offer a good and fast estimation, which is highlighted by the result. The experiments were set up in a way that the small and large instances are from the same class of instances, just scaled up. We have also generalised from merging the circuit design and planning instances, whereas in an ideal scenario, determining p from a single class might be more suitable.

In this set of experiments, we look at the heuristic *hybrid_WMC*, but any heuristic could be used. $\mathcal{I}_{small}$ dataset contains a subset of the 168 instances we previously looked at. We eliminate 45 instances that have less than 50 variables since they are too small. The obtained dataset contains 123 instances with 50-300 variables. Given the $\mathcal{I}_{small}$ dataset we observe that the highest median adjusted ratio is 1.79, obtained at $p_{max} = 22\%$ of variables of the specific instance. At this percentage, 68.3% of the instances from $\mathcal{I}_{small}$ have an adjusted ratio higher than 1. Figure 5.9 represents the median adjusted ratio of the $\mathcal{I}_{small}$ instances at each p. We notice this has a slightly different behaviour than the *hybrid_WMC* from Figure 5.5. This is due to the fact that Figure 5.9 represents a different dataset, where smaller than 50 variable instances are eliminated, and the dataset used for Figure 5.5 contains instances with 15 and 30 variables.

After identifying p_{max} on the smaller instances we look at evaluating the partial compilation of the larger instances given B_j. $\mathcal{I}_{large}$ contains 171 instances with the 300-900 variables. We eliminated 7 since they had no initial compilation and there is one instance that times out before obtaining any selective backbone. We

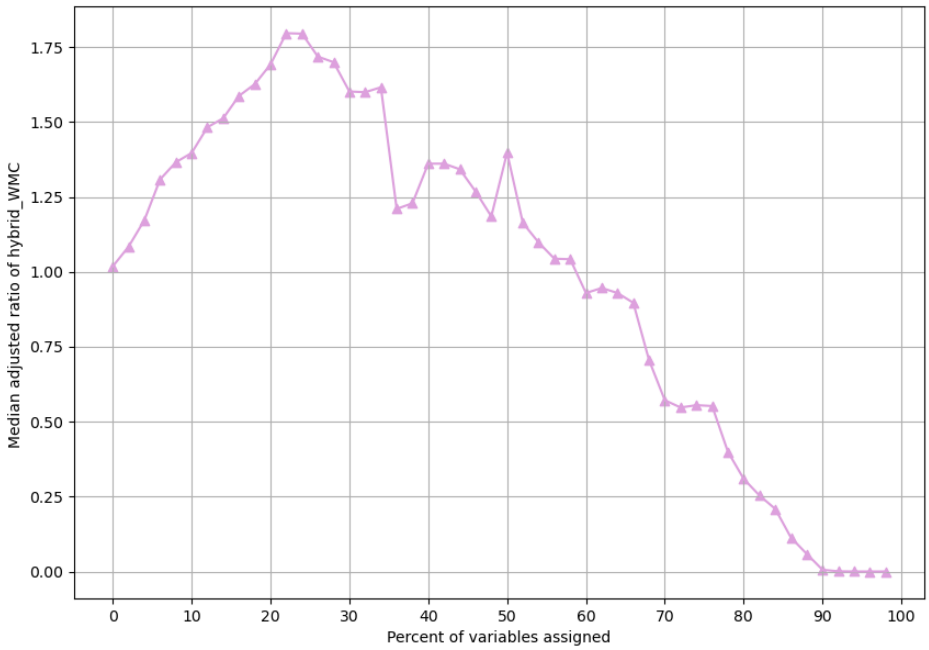


Figure 5.9: Median Adjusted ratio of *hybrid_WMC* over the ISCAS and planning instances with 50-300 variables

define a 6 hour timeout but some instances obtain the selective backbone within 0.8 seconds, with a median time of 376.23 seconds. There are 25 instances that do not reach 22% variables assigned or reach a conflicting assignment. Figure 5.10 represents the adjusted ratio in increasing order for instances in $\mathcal{I}_{large}$ given the partial compilation defined by the selective backbone of size 22% of variables. Each green bar represents the adjusted ratio of an instance with 22% of variables assigned, orange represents the compilation of an instance before a conflicting assignment is reached and blue represents the partial compilation for instances where a timeout occurred before assigning 22% of the variables.

The median of the adjusted ratio of the compilations from $\mathcal{I}_{large}$ is 1.56 and 60% of the instances are compact, with an adjusted ratio larger than 1. These values are close to the ones observed for $\mathcal{I}_{small}$, indicating that the behaviour of selective backbones on small instances can indeed provide an idea of the behaviour of selective backbones on larger instances.

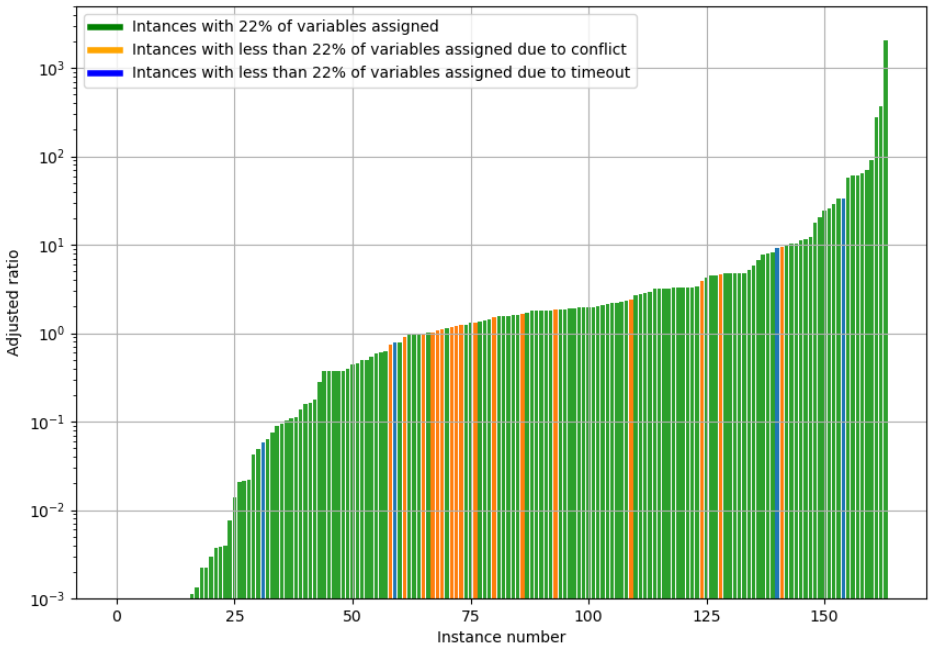


Figure 5.10: Adjusted ratio at 22% Selective Backbone

5.8 Summary

In this chapter we have covered three main topics: partial compilation of weighted problems, more efficient heuristics to obtain selective backbones and a method to identify a stopping criteria for the iterative heuristics that would lead to an efficient partial compilation.

Firstly, we showed that the method of obtaining partial compilation through selective backbones is suitable for weighted problems as well. We considered instances from a generated dataset, circuit design and planning dataset with generated weights. With this we proved that using selective backbones we can represent in a partial compilation the most important solutions not just the largest subset of solution. The selection criteria of a good selective backbone changed to weighted model counting in order to reflect the preferences imposed by the weights. Secondly, we proposed new heuristics that are computationally less expensive to obtain selective backbones. We showed that they also obtain efficient partial compilations and are significantly faster as well. Thirdly, we presented a method to identify an adequate number of iterations to obtain an efficient partial compilation. We consider a partial compilation efficient if it is more compact than the initial compilation, that is the ratio between the compilation size and

the weighted solution count is higher than for the initial compilation. We showed that is it possible to infer from a dataset of smaller instances the number of iterations for a heuristic that would lead to an efficient partial compilation on a set of larger instances as well.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We revisit the thesis statement and summarise its defence. After which we propose some future improvements and directions this research could be expanded to. The thesis statement is as follows:

> "It is possible to develop techniques that can selectively compile combinatorial problems in a way that ensures a more compact representation can be achieved while still maintaining a large subset of the most important solutions."

To prove the existence of a technique to obtain compact partial compilations we explored two main techniques and some alternatives of them. The first approach explored uses symmetry breaking to eliminate symmetrical solutions. The second approach to find such compilations centres around the concept of selective backbone, a notion that we have introduced and is in essence a partial assignment that is present in a given subset of the solutions to an instance.

We introduced symmetry breaking at the modelling phase and used a variety of existing well-established compilers to observe the behaviour. The results of these were promising on a set of classical instances from the symmetry breaking field. We considered a clique of not-equals constraints, the N-queens, the Graceful Graph Labelling and the Balanced Incomplete Block Design problem. For all these, there was a speed-up in compilation when adding symmetry breaking as well as obtaining smaller representations. In some cases without symmetry breaking the compilation did not finish due to reaching memory issues or the

timeout of half an hour. The same instance with symmetry breaking would then compile within seconds. However, this approach might be limited to problems that are highly symmetrical, so there is a need to generalise this.

A more general approach we introduced was to restrict the represented solutions by fixing part of the problem. That is, a few carefully selected variables are assigned in order to obtain a partial compilation. We call the set of assignments the selective backbone. We formulated six iterative greedy heuristics that would build the selective backbone, expanding it one assignment per iteration. We have empirically proven that for many Boolean instances, belonging to various problem types, it is possible to obtain partial compilations that are far more space-efficient than the full one in terms of the number of solutions that are conserved. To showthe effectiveness of these we considered a set of generated instances, a set of circuit design instances (ISCAS) and a set of planning instances. We considered a partial compilation effective if the solution loss is smaller than the compilation size reduction. That is we want a compilation that represents information more densely.

Having shownthe usefulness of selective backbones we expanded our work towards improving the efficiency of the heuristics used to obtain them. We also expanded the problem formulation towards weighted problems, thus representing importance of certain variable-value pair or user preferences. We explored eight heuristics, a random one used as a baseline, two computationally expensive ones based on previous approach for unweighted problems and five computationally less intensive heuristics. The five computationally less intensive heuristics rely on instance properties that can be computed quickly, so they can return a set of models to remove in a much briefer time. It turns out that the selective backbones that characterise the sets of models to keep will lead to drastically different results depending on whether they are applied to the weighted or unweighted instances. Indeed, we have shown that the same selective backbone can result in a weighted model count close to the original one, while discarding most of the models. This shows that our approaches successfully take into account the fact that some models are worth more than others.

Another improvement on the method of obtaining a good selective backbone that is used to obtain an efficient compact compilation was the determination of the number of iterations a heuristic should perform. Since all the proposed heuristics were iterative, determining the number of iterations is an important aspect. We do not want to try and compile a problem at every iteration, hoping that

it obtained a manageable size, but would prefer to have some guidance on what is a good number of iterations to perform. With a simple statistical approach, we determine a good stopping point for a set of smaller instances that we then use on larger instances. With this method, we have shown that it is possible to determine in an efficient manner at which selective backbone size these highly compact partial compilations can be expected to be found. This empirically demonstratesthat a compact representation of a weighted CNF instance can be obtained with minimal information loss without any kind of expensive computation.

6.2 Future Work

In the future, this work could be expanded in a few different directions. More experiments, especially real-world instances with and without user preferences would be a natural extension. Also further exploration with different scoring functions and heuristics could improve on the results. The approaches we introduced are driven by changing the problem formulation, adding constraints to the initial problems to restrict the solutions even more. We have evaluated a few different compilers and thus a few different representations but a more comprehensive study for a larger set of compilers could provide more guarantees the approach is independent of the compilation tools and representations.

The concept of partial compilation could be useful for different reasons. In the next sections, we explore some different methods of obtaining partial compilation. Until now we focused on obtaining partial compilations with the motivation of scalability, so that a smaller, more compact and faster compiled representation is obtained even if not representing the initial problem it compiles. In the next sections we explore different motivations for partial compilations.

6.2.1 Formal proofs

In this dissertation, we empirically demonstrated our thesis that it is feasible to obtain an efficient partial compilation. A future direction of this work could be to formally prove this. Since different compiled representations exist, each satisfying a set of properties it would be interesting to see if our hypothesis applies to all representations and under what conditions. Firstly, we could formalize a theory and its proof for the hypothesis that after symmetry breaking, sub-graphs that represent the different symmetrical solutions merge into one.

Secondly, a guarantee of the number of solutions represented given a selective backbone would be interesting. A partial compilation would gain a lot more trustworthiness if we could prove that although not all solutions are represented, an estimated guarantee that $x\%$ is.

6.2.2 Predicting Selective Backbone size

Another natural future work direction could focus on the expansion of the approach used to determine the size of the selective backbone. In the previous chapter we described a basic method that given an iterative heuristic determined a good number of iterations the heuristic should perform in order to obtain a compact partial solution. An improvement on this could be the addition of using features of the instances that are often used to characterize them such as number of variables, clauses, statistical metrics about the primal graph, probing etc. Implementing a machine learning model using these features could lead to a better prediction.

SATzilla [XHHLB08] has been developed to make an informed decision, based on SAT instance features, about which solver to use. Similar to this concept a tool to analyse an instance and determine how many and which variables to assign such that a compact partial compilation is obtained that represents a large or most important subset of solutions is envisioned.

Two directions could be explored. One, where we use a previously defined heuristic and use a predictive model to determine the number of iterations for the given greedy algorithm. For this a large dataset of instances is needed along with results from running the iterative heuristic on the instances. Each instance could be represented in terms of features that would be indicative of the compilation size they would be compiled to given a specific tool. The success of this method highly depends on the selection of the features representing the instances. A second approach could extend this prediction with not only the number of variables to fix to obtain the partial compilation but also which variables. This would entail that the prediction algorithm is capable of identifying the impact of a variable or a set of variable assignments on both the compilation size and the subset of solutions represented.

6.2.3 Global Selective Backbone

The heuristics we proposed in this dissertation are all iterative. They extend the selective backbone one assignment at a time. Because of this, it is not guaranteed that a selective backbone of size n is globally the best with respect to a criterion such as model count, or weighted model count. It would be an interesting direction to develop approaches that would identify the globally best selective backbone.

6.2.4 Backdoors to Restrict the Problem

The complexity of constraint problems has been researched heavily. A backdoor is a set of variables that once assigned a certain set of values the rest of the problem becomes solvable in polynomial time. Strong backdoors are defined as a set of variables that once instantiated in any way the rest of the problem is solvable in polynomial time [WGS03]. Backdoors have been shown to partition an instance into classes of instances that can be solved in polytime also referred to as "islands of traceability" [GRS17]. Given this, we could explore the possibility of a compiled representation that consists of two parts: the backdoor and the rest of the problem.

A partial compilation could be used to represent the partial solutions defined by a backdoor. Each satisfiable problem has a backdoor of size at most $n - 1$, where n is the number of variables [WGS03]. The smaller the backdoor the more powerful it is, but finding them is expensive. Once the backdoor is fixed the rest of the problem is solvable in polynomial time. With this, a framework could be developed that answers knowledge compilation queries using a partial compilation of a backdoor and a sub-solver for the rest of the problem variables. An interesting study would be to see the effect of the backdoor size on the partial compilation, taking into account the effort of obtaining the backdoor.

6.2.5 Partial compilation as Problem Relaxation

In this dissertation, we defined a partial compilation as a representation that contains a subset of solutions to the original problem. Another interpretation of partial could refer to considering only a subset of the constraints to the partial compilation. In this setting the initial problem is relaxed and the partial compilation has more solutions, some that were not a solution to the initial problem. In case losing solutions is not appropriate for a use case, having a relaxed par-

tial compilation can be used as an upper bound. The use of relaxed Decision Diagrams has also been explored in [CBR+22].

By restricting the solution set the very unique solutions that do not share any paths in a representation with other solutions get eliminated, thus the size is reduced. In case of relaxing the problem, we accept solutions that make unique solutions less unique, thus sharing more of its partial assignment with the rest of the problem. We could achieve this by clustering constraints based on their solution set and ignoring the cluster with the least amount of solutions.

References

[ACMS15] Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise, and Peter J. Stuckey. #∃SAT: Projected model counting. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 121–137. Springer, 2015.

[AFNP14] Jérôme Amilhastre, Hélène Fargier, Alexandre Niveau, and Cédric Pralet. Compiling CSPs: A complexity map of (non-deterministic) multivalued decision diagrams. *Int. J. Artif. Intell. Tools*, 23(4), 2014.

[AGKS00] Dimitris Achlioptas, Carla P. Gomes, Henry A. Kautz, and Bart Selman. Generating satisfiable problem instances. In Henry A. Kautz and Bruce W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, pages 256–261. AAAI Press / The MIT Press, 2000.

[ALM22] Gilles Audemard, Jean-Marie Lagniez, and Marie Miceli. A new exact solver for (weighted) max#sat. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[AMS03] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Shatter: efficient symmetry-breaking for Boolean Satisfiability. In *Proceedings of the 40th Design Automation Conference, DAC 2003*. ACM, 2003.

[ATO16] Alejandro Arbelaez, Charlotte Truchet, and Barry O'Sullivan. Learning sequential and parallel runtime distributions for randomized algorithms. In *28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016, San Jose, CA, USA, November 6-8, 2016*, pages 655–662. IEEE Computer Society, 2016.

[Aze07] Francisco Azevedo. Cardinal: A finite sets constraint solver. *Constraints An Int. J.*, 12(1):93–129, 2007.

[BB19] Kyle E. C. Booth and J. Christopher Beck. A constraint programming approach to electric vehicle routing with time windows. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2019.

[BCFR08] J. Christopher Beck, Tom Carchrae, Eugene C. Freuder, and Georg Ringwelski. A space-efficient backtrack-free representation for constraint satisfaction problems. *Int. J. Artif. Intell. Tools*, 17(4):703–730, 2008.

[BDD+22] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. Pseudo polynomial-time top-k algorithms for d-DNNF circuits. *CoRR*, abs/2202.05938, 2022.

[BDK07] Delbert D. Bailey, Víctor Dalmau, and Phokion G. Kolaitis. Phase transitions of PP-complete satisfiability problems. *Discret. Appl. Math.*, 155(12):1627–1639, 2007.

[BEO23] Andrea Balogh, Guillaume Escamocher, and Barry O'Sullivan. Partial compilation of SAT using selective backbones. In Kobi Gal, Ann Nowé, Grzegorz J. Nalepa, Roy Fairstein, and Roxana Radulescu, editors, *ECAI 2023 - 26th European Conference on Artificial Intelligence, September 30 - October 4, 2023, Kraków, Poland - Including 12th Conference on Prestigious Applications of Intelligent Systems (PAIS 2023)*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 174–181. IOS Press, 2023.

[BF21] Beate Bollig and Martin Farenholtz. On the relation between struc-

tured d-DNNFs and SDDs. *Theory Comput. Syst.*, 65(2):274–295, 2021.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[BHW03] Christian Bessière, Emmanuel Hebrard, and Toby Walsh. Local consistencies in SAT. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2003.

[BKLM14] Anicet Bart, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. Symmetry-driven decision diagrams for knowledge compilation. In *ECAI 2014 - 21st European Conference on Artificial Intelligence*. IOS Press, 2014.

[BL99] Elazar Birnbaum and Eliezer L. Lozinskii. The good old davis-putnam procedure helps counting models. *J. Artif. Intell. Res.*, 10:457–477, 1999.

[BLPN06] Philippe Baptiste, Philippe Laborie, Claude Le Pape, and Wim Nuijten. Constraint-based scheduling and planning. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 761–799. Elsevier, 2006.

[BO23] Andrea Balogh and Barry O'Sullivan. Breaking symmetry for knowledge compilation. In Jiman Hong, Maart Lanperne, Juw Won Park, Tomás Cerný, and Hossain Shahriar, editors, *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, SAC 2023, Tallinn, Estonia, March 27-31, 2023*, pages 991–994. ACM, 2023.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with Ordered Binary Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[BT06] Dimitris Bertsimas and Aurélie Thiele. A robust optimization approach to inventory theory. *Oper. Res.*, 54(1):150–168, 2006.

[BvHH11] David Bergman, Willem Jan van Hoeve, and John N. Hooker. Manipulating MDD relaxations for combinatorial optimization. In Tobias Achterberg and J. Christopher Beck, editors, *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*, volume 6697 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2011.

[CBR+22] Quentin Cappart, David Bergman, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Augustin Parjadis. Improving variable orderings of approximate decision diagrams using reinforcement learning. *INFORMS J. Comput.*, 34(5):2552–2570, 2022.

[CD08] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008.

[CD13a] Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, pages 187–194. AAAI Press, 2013.

[CD13b] Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, 2013*. AAAI Press, 2013.

[CFM+14] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 1722–1730. AAAI Press, 2014.

[CFMV15] Supratik Chakraborty, Dror Fried, Kuldeep S. Meel, and Moshe Y. Vardi. From weighted to unweighted model counting. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJ-*

CAI 2015, Buenos Aires, Argentina, July 25-31, 2015, pages 689–695. AAAI Press, 2015.

[CG01] Joseph C. Culberson and Ian P. Gent. Frozen development in graph coloring. *Theor. Comput. Sci.*, 265(1-2):227–264, 2001.

[CGLR96] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996*, pages 148–159. Morgan Kaufmann, 1996.

[CJJ+06] David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints An Int. J.*, 11(2-3):115–137, 2006.

[CLM21] Florent Capelli, Jean-Marie Lagniez, and Pierre Marquis. Certifying top-down decision-DNNF compilers. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6244–6253. AAAI Press, 2021.

[CMV13] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2013.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[CW17] Wenxiang Chen and L. Darrell Whitley. Decomposing SAT in-

stances with pseudo backbones. In Bin Hu and Manuel López-Ibáñez, editors, *Evolutionary Computation in Combinatorial Optimization - 17th European Conference, EvoCOP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*, volume 10197 of *Lecture Notes in Computer Science*, pages 75–90, 2017.

[Dar01] Adnan Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, 2001.

[Dar04] Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 328–332. IOS Press, 2004.

[Dar11] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, 2011*. IJCAI/AAAI, 2011.

[Dar20a] Adnan Darwiche. Three modern roles for logic in AI. *CoRR*, abs/2004.08599, 2020.

[Dar20b] Adnan Darwiche. Three modern roles for logic in AI. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, pages 229–243. ACM, 2020.

[Dar23] Adnan Darwiche. Logic for explainable AI. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS2023, Boston, MA, USA, June 26-29, 2023*, pages 1–11. IEEE, 2023.

[DBBD16] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference*. Springer, 2016.

[dBD15] Guy Van den Broeck and Adnan Darwiche. On the role of canonicity in knowledge compilation. In *Proceedings of the Twenty-Ninth*

AAAI Conference on Artificial Intelligence, 2015. AAAI Press, 2015.

[Dec90] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.

[DH23] Adnan Darwiche and Auguste Hirth. On the (complete) reasons behind decisions. *J. Log. Lang. Inf.*, 32(1):63–88, 2023.

[DLHL21] Giso H. Dal, Alfons W. Laarman, Arjen Hommersom, and Peter J. F. Lucas. A compositional approach to probabilistic knowledge compilation. *Int. J. Approx. Reason.*, 138:38–66, 2021.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DLSM04] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In *Proceedings of the 41th Design Automation Conference, DAC 2004, USA*. ACM, 2004.

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002.

[DPV20] Jeffrey M. Dudek, Vu Phan, and Moshe Y. Vardi. ADDMC: weighted model counting with algebraic decision diagrams. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1468–1476. AAAI Press, 2020.

[EGS12] Stefano Ermon, Carla P. Gomes, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. In Nando de Freitas and Kevin P. Murphy, editors, *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, August 14-18, 2012*, pages 255–264. AUAI Press, 2012.

[EO22] Guillaume Escamocher and Barry O'Sullivan. Generation and prediction of difficult model counting instances. *CoRR*, abs/2212.02893, 2022.

[FdB18] Tal Friedman and Guy Van den Broeck. Approximate knowledge compilation by online collapsed importance sampling. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 8035–8045, 2018.

[FFH+02] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 2002.

[FFW+19] Liangda Fang, Biqing Fang, Hai Wan, Zeqi Zheng, Liang Chang, and Quan Yu. Tagged sentential decision diagrams: Combining standard and zero-suppressed compression and trimming rules. In David Z. Pan, editor, *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, pages 1–8. ACM, 2019.

[FM21] Hélène Fargier and Jérôme Mengin. A knowledge compilation map for conditional preference statements-based languages. In Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé, editors, *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, pages 492–500. ACM, 2021.

[FMNS14] Hélène Fargier, Pierre Marquis, Alexandre Niveau, and Nicolas Schmidt. A knowledge compilation map for ordered real-valued decision diagrams. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 1049–1055. AAAI Press, 2014.

[FMS22] Hélène Fargier, Jérôme Mengin, and Nicolas Schmidt. Nucleus-satellites systems of OMDDs for reducing the size of compiled forms. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, volume 235 of *LIPIcs*, pages 23:1–23:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[Fre91] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In Thomas L. Dean and Kathleen R. McKeown, editors, *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14-19, 1991, Volume 1*, pages 227–233. AAAI Press / The MIT Press, 1991.

[Gal00] Joseph Gallian. A dynamic survey of graph labeling. *Electron J Combin DS6*, 19, 11 2000.

[Gas01] Christine Gaspin. RNA secondary structure determination and representation based on constraints satisfaction. *Constraints An Int. J.*, 6(2/3):201–221, 2001.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[GD07] Vibhav Gogate and Rina Dechter. Approximate counting by sampling the backtrack-free search space. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 198–203. AAAI Press, 2007.

[GD11] Vibhav Gogate and Rina Dechter. SampleSearch: Importance sampling in presence of determinism. *Artif. Intell.*, 175(2):694–729, 2011.

[GD12] Vibhav Gogate and Rina Dechter. Importance sampling-based estimation over AND/OR search spaces for graphical models. *Artif. Intell.*, 184-185:38–77, 2012.

[GHSS07] Carla P. Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2293–2299, 2007.

[GPP06] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006.

[GRS17] Robert Ganian, M. S. Ramanujan, and Stefan Szeider. Discovering archipelagos of tractability for constraint satisfaction and counting. *ACM Trans. Algorithms*, 13(2):29:1–29:32, 2017.

[GS04] Carla P. Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2004.

[GSK98] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In Jack Mostow and Chuck Rich, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, pages 431–437. AAAI Press / The MIT Press, 1998.

[GSMT98] Carla P. Gomes, Bart Selman, Ken McAloon, and Carol Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In Reid G. Simmons, Manuela M. Veloso, and Stephen F. Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, Pittsburgh, Pennsylvania, USA, 1998*, pages 208–213. AAAI, 1998.

[GSS06] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 54–61. AAAI Press, 2006.

[HD04] Jinbo Huang and Adnan Darwiche. Using DPLL for efficient OBDD construction. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 2004, Canada*, 2004.

[HD05] Jinbo Huang and Adnan Darwiche. On compiling system models for faster and more scalable diagnosis. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 300–306. AAAI Press / The MIT Press, 2005.

[HMBM08] Eric I. Hsu, Christian J. Muise, J. Christopher Beck, and Sheila A. McIlraith. Probabilistically estimating backbones and variable bias: Experimental overview. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, volume 5202 of *Lecture Notes in Computer Science*, pages 613–617. Springer, 2008.

[HTW20] Markus Hecher, Patrick Thier, and Stefan Woltran. Taming high treewidth with abstraction, nested dynamic programming, and database technology. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2020.

[Hua06] Jinbo Huang. Combining knowledge compilation and search for conformant probabilistic planning. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pages 253–262. AAAI, 2006.

[KdBCD14] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Four-*

teenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014. AAAI Press, 2014.

[KSS08] Lukas Kroc, Bart Selman, and Ashish Sabharwal. Leveraging belief propagation, backtrack search, and statistics for model counting. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008.

[KSTW05] Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Backbones and backdoors in satisfiability. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 1368–1373. AAAI Press / The MIT Press, 2005.

[KSW05] Philip Kilby, John K. Slaney, and Toby Walsh. The backbone of the travelling salesperson. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 175–180. Professional Book Center, 2005.

[KTH+17] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in WEKA. *J. Mach. Learn. Res.*, 18:25:1–25:5, 2017.

[LA97] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In Gert Smolka, editor, *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, volume 1330 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 1997.

[Lai18] Yong Lai. Approximate model counting by partial knowledge compilation. *CoRR*, abs/1805.07180, 2018.

[Lee59] Choong Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999,

1959.

[LHHX14] Kevin Leyton-Brown, Holger H. Hoos, Frank Hutter, and Lin Xu. Understanding the empirical hardness of *NP*-complete problems. *Commun. ACM*, 57(5):98–107, 2014.

[LL22] Byung Lee and Joyce Low. A constraint programming approach to capacity planning in container vessels. *Maritime Economics Logistics*, 24:1–24, 06 2022.

[LLM20] Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Definability for model counting. *Artif. Intell.*, 281:103229, 2020.

[LM17] Jean-Marie Lagniez and Pierre Marquis. An improved Decision-DNNF compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017.

[LM21] Jean-Marie Lagniez and Pierre Marquis. About Caching in D4 2.0. In *Workshop on Counting and Sampling 2021*, Barcelona, Spain, July 2021.

[LMY21] Yong Lai, Kuldeep S. Meel, and Roland H. C. Yap. The power of literal equivalence in model counting. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3851–3859. AAAI Press, 2021.

[LMY23] Yong Lai, Kuldeep S. Meel, and Roland H. C. Yap. Fast converging anytime model counting. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 4025–4034. AAAI Press, 2023.

[Mac77] Alan K. Mackworth. Consistency in Networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[MdlBW09] Christopher Mears, Maria Garcia de la Banda, and Mark Wallace. On implementing symmetry detection. *Constraints An Int. J.*, 14(4):443–477, 2009.

[MDM14] Robert Mateescu, Rina Dechter, and Radu Marinescu. AND/OR multi-valued decision diagrams (AOMDDs) for graphical models. *CoRR*, abs/1401.3448, 2014.

[Mic06] Jean Michel. Handbook of computational group theory. *Math. Comput.*, 75(254), 2006.

[Min93] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In Alfred E. Dunlop, editor, *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*, pages 272–277. ACM Press, 1993.

[MJL10] João Marques-Silva, Mikolás Janota, and Inês Lynce. On computing backbones of propositional theories. In Helder Coelho, Rudi Studer, and Michael J. Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 15–20. IOS Press, 2010.

[MMBH12] Christian J. Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In Leila Kosseim and Diana Inkpen, editors, *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, volume 7310 of *Lecture Notes in Computer Science*, pages 356–361. Springer, 2012.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[MP14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 2014.

[MT01] Pedro Meseguer and Carme Torras. Exploiting symmetries within constraint satisfaction search. *Artif. Intell.*, 129(1-2):133–163, 2001.

[MZK+99] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, 1999.

[NAG+17] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artif. Intell.*, 2017.

[NDN20] Kengo Nakamura, Shuhei Denzumi, and Masaaki Nishino. Variable shift SDD: A more succinct sentential decision diagram. In *18th International Symposium on Experimental Algorithms, SEA 2020*, 2020.

[NR16] Peter Nightingale and Andrea Rendl. Essence' description. *CoRR*, abs/1601.02865, 2016.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[NW07] Nina Narodytska and Toby Walsh. Constraint and variable ordering heuristics for compiling configuration problems. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 149–154, 2007.

[NYMN16] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Zero-suppressed sentential decision diagrams. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016*. AAAI Press, 2016.

[OD15] Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3141–3148. AAAI Press, 2015.

[OD17] Umut Oztok and Adnan Darwiche. On compiling DNNFs without determinism. *CoRR*, abs/1709.07092, 2017.

[OP06] Barry O'Sullivan and Gregory M. Provan. Approximate compilation for embedded model-based reasoning. In *Proceedings of AAAI-2006*, 2006.

[PBDG05] Héctor Palacios, Blai Bonet, Adnan Darwiche, and Hector Geffner. Pruning conformant plans by counting models on compiled d-DNNF representations. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*, pages 141–150. AAAI, 2005.

[PD08] Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 517–522. AAAI Press, 2008.

[Pet] Karen Petrie. CSPLib problem 053: Graceful graphs. `http://www.csplib.org/Problems/prob053`. visited 14.04.2025.

[PF] Laurent Perron and Vincent Furnon. Or-tools v9.11. `https://developers.google.com/optimization/`. Google (visited 14.04.2025).

[PF22] Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *J. Open Source Softw.*, 7(78):4708, 2022.

[PR05] Steven D. Prestwich and Andrea Roli. Symmetry breaking and local search spaces. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Second International Conference, CPAIOR 2005,Czech Republic*. Springer, 2005.

[Pre] Steven Prestwich. CSPLib problem 028: Balanced incomplete block designs. `http://www.csplib.org/Problems/prob028`. visited 14.04.2025.

[Pre02] Steven Prestwich. Balanced incomplete block design as satisfiability. 07 2002.

[Pre03] Steven D. Prestwich. Negative effects of modeling techniques on search performance. *Ann. Oper. Res.*, 118(1-4):137–150, 2003.

[Pre04] Steven D. Prestwich. Full dynamic substitutability by SAT encoding. In *Principles and Practice of Constraint Programming - CP 2004, Canada*. Springer, 2004.

[PS03a] Karen E. Petrie and Barbara M. Smith. Symmetry breaking in graceful graphs. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 930–934. Springer, 2003.

[PS03b] Karen E. Petrie and Barbara M. Smith. Symmetry breaking in graceful graphs. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 930–934. Springer, 2003.

[RAL17] Erfan Rahimian, Kerem Akartunali, and John Levine. A hybrid integer and constraint programming approach to solve nurse rostering problems. *Comput. Oper. Res.*, 82:83–94, 2017.

[RG07] Jussi Rintanen and Alban Grastien. Diagnosability testing with satisfiability algorithms. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 532–537, 2007.

[RKdBR14] Joris Renkens, Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Explanation-based approximate weighted model counting for probabilistic logics. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2490–2496. AAAI Press, 2014.

[RN20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[SBB+04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

[SBK05] Tian Sang, Paul Beame, and Henry A. Kautz. Performing Bayesian inference by weighted model counting. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 475–482. AAAI Press / The MIT Press, 2005.

[SCD18a] Yujia Shen, Arthur Choi, and Adnan Darwiche. Conditional PSDDs: Modeling and learning with modular knowledge. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6433–6442. AAAI Press, 2018.

[SCD18b] Andy Shih, Arthur Choi, and Adnan Darwiche. A symbolic approach to explaining Bayesian network classifiers. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 5103–5111. ijcai.org, 2018.

[SCD19] Andy Shih, Arthur Choi, and Adnan Darwiche. Compiling Bayesian network classifiers into decision graphs. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Ar-*

tificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pages 7966–7974. AAAI Press, 2019.

[SDC19] Andy Shih, Adnan Darwiche, and Arthur Choi. Verifying binarized neural networks by Angluin-style learning. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 354–370. Springer, 2019.

[SdV01] Laurent Simon and Alvaro del Val. Efficient consequence finding. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 359–370. Morgan Kaufmann, 2001.

[SFM+96] Johannes Schneider, Christine Froschhammer, Ingo Morgenstern, Thomas Husslein, and Johannes Maria Singer. Searching for backbones — an efficient parallel algorithm for the traveling salesman problem. *Computer Physics Communications*, 96(2):173–188, 1996.

[SGRM18] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. Knowledge compilation meets uniform sampling. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 620–636. EasyChair, 2018.

[SKH+24] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. On the benefits of knowledge compilation for feature-model analyses. In Maxime Cordy, Daniel Strüber, Mónica Pinto, Iris Groher, Deepak Dhungana, Jacob Krüger, Juliana Alves Pereira, Mathieu Acher, Thomas Thüm, Maurice H. ter Beek, Jessie Galasso-Carbonnel, Paolo Arcaini, Mohammad Reza Mousavi, Xhevahire Tërnava, José A. Galindo, Tao Yue, Lidia Fuentes, and José Miguel Horcas, editors, *Proceedings of the 28th ACM International Systems and Software Product Line Conference - Volume A, SPLC 2024, Dommeldange, Luxembourg, September 2-6, 2024*, page 217. ACM, 2024.

[SRSM19] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1169–1176. ijcai.org, 2019.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE Computer Society / ACM, 1996.

[Stu10] Peter J. Stuckey. Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, volume 6140 of *Lecture Notes in Computer Science*, pages 5–9. Springer, 2010.

[SW01] John K. Slaney and Toby Walsh. Backbones in optimization and approximation. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 254–259. Morgan Kaufmann, 2001.

[Thu06] Marc Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006.

[Val79] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[vDWM17] Tom van Dijk, Robert Wille, and Robert Meolic. Tagged BDDs: Combining reduction rules from different decision diagram types. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal*

Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017, pages 108–115. IEEE, 2017.

[VH03] Christos Valouxis and Efthymios Housos. Constraint programming approach for school timetabling. *Comput. Oper. Res.*, 30(10):1555–1572, 2003.

[WGS03] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 1173–1178. Morgan Kaufmann, 2003.

[WS05] Wei Wei and Bart Selman. A new approach to model counting. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 324–339. Springer, 2005.

[WUA+20] Wenxi Wang, Muhammad Usman, Alyas Almaawi, Kaiyuan Wang, Kuldeep S. Meel, and Sarfraz Khurshid. A study of symmetry breaking predicates and model counting. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*. Springer, 2020.

[XBZ+21] Hao Xu, Souheib Baarir, Tewfik Ziadi, Lom-Messan Hillah, Siham Essodaigui, and Yves Bossu. Optimisation for the product configuration system of Renault: towards an integration of symmetries. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, United Kindom, 2021, Volume B*. ACM, 2021.

[XHHL08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008.

[XHHLB08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *J. Artif. Int. Res.*, 32(1):565–606, June 2008.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In Rolf Ernst, editor, *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, pages 279–285. IEEE Computer Society, 2001.

[ZP96] Nevin Lianwen Zhang and David L. Poole. Exploiting causal independence in Bayesian network inference. *J. Artif. Intell. Res.*, 5:301–328, 1996.