Symmetry Breaking and Knowledge Compilation

Andrea Balogh
Confirm Centre for Smart Manufacturing
School of Computer Science & IT, University College Cork, Ireland

Barry O'Sullivan
Confirm Centre for Smart Manufacturing and Insight SFI Research Centre for Data Analytics
School of Computer Science & IT, University College Cork, Ireland

Abstract

Constraint programming is a powerful method for solving combinatorial problems. Diagnosis, planning, and product configuration, are example use cases. While reasoning about the solution space of combinatorial problems is usually intractable, compilation methods are often used to pre-compute a representation that can answer queries in time that is polynomial in the representation size. Symmetry breaking constraints can be added to a combinatorial problem to eliminate symmetries, in the expectation that this will speed-up search and reduce the number of solutions. Finding compact representations is often the bottleneck of compilation methods. In this paper we investigate if breaking symmetries always leads to a smaller compiled representation. We considered four compilers and three highly symmetrical problems. A reduction is observed in all the problems for the compilation size when we break symmetries, with top-down compilers obtaining more reduction.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming.; Computing methodologies → Knowledge representation and reasoning.

Keywords and phrases Symmetry Breaking, Knowledge Compilation.

Acknowledgements This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Numbers 16/RC/3918 and 12/RC/2289-P2.

1 Introduction

Constraint programming (CP) is a powerful method to solve combinatorial problems, but due to a large search space, solving can be very time consuming. Diagnosis, planning, product configuration are example use cases. These systems are often used in an online format answering queries, such as "How many possible configurations exist for a product?" or "Is this configuration valid?". Compilation methods were developed to deal with the complexity of solving the problems offline and create a representation that is able to answer queries in polynomial time [6].

Symmetry in CP is defined as a bijective function mapping solutions to solutions and non-solutions to non-solutions [3]. Symmetry breaking (SB) is the addition of constraints to eliminate symmetrical solutions. 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 was that when we eliminate symmetries, sub-graphs that represent the difference between the two solutions get eliminated and thus paths that represents the two symmetrical solutions get merged into one. Thus, 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 does this occur? 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 and Balanced Incomplete Block Design (BIBD). We introduced symmetry breaking

constraints to these models and compiled them using four different tools: PySDD, ZSDD, D4 and MiniC2D.

To 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, such as dynamic symmetry breaking during compilation [2] or changing the compiled representation to deal with symmetry [11]. Dynamic symmetry breaking during compilation has been explored in [2] by extending the languages FBDD and DDG to Sym-FBDD and Sym-DDG respectively. Variable Shift SDDs [11] were 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 VS-SDD has an offset of k associated to it. Vtree nodes expressing the same Boolean formulas are merged and the offset is used to associate literals to this. The effect of symmetry breaking at CNF level and at domain level with automated tools and manual formulation has been studied on model counting [17].

The paper is structured as follows: Section 2 gives an overview of symmetry breaking in CP. Section 3 introduces knowledge compilation, the four tools and the representations they compile to. Finally, Section 4 describes the experiment setups and results of them.

2 Symmetry breaking in Constraint Programming

Symmetry breaking in CP has been widely recognized as having a large impact, but research around it is still sparse compared to other aspects in CP. A good overview of the different definitions and types of symmetry can be found in [3]. Generally, symmetry is defined as a bijective function of the variable-value pairs, mapping solutions to solutions, non-solutions to non-solutions. It can be inherent in the problem or introduced while modelling. A set of symmetrical solutions belong a group. Breaking symmetries mean that through additional constraints we eliminate solutions, such that at least one representative of the symmetry group remains a solution.

Depending on the problem at hand, recognizing 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 CNF level `saucy` [4] or `nauty` [10] can be used to recognize automorphisms in graphs. A graph is created such that vertices represent SAT literals and edges possible combinations of these. Shatter [1] and BreakID [7] were implemented using `saucy` to identify symmetries, then generate predicates to eliminate them.

Savile Row [12] is a modelling assistant for CP which also has an option for symmetry breaking. It takes as input a constraint model defined in a high-level language, ESSENCE', and transforms this to various CP and SAT models. It uses a graph automorphism solver to detect symmetries and breaks them using the lex-leader method. Such tools create auxiliary variables while breaking symmetries, thus for our setup using these tools is difficult. Our hypothesis, that sub-graphs that are occurring multiple times due to symmetries would be joined, is not possible in a model where we reformulate the problem with additional literals. To this end, we hand model the symmetry breaking constraints.

Symmetry breaking is not always beneficial, for example local search performs better when we do not break symmetries [16]. Also for some problems, with few symmetries, the overhead of extra constrains might not be beneficial enough.

3 Knowledge Compilation

Knowledge compilation covers techniques to deal with the computational complexity of propositional reasoning, such that a propositional theory is compiled into a target representation, one that has properties that allow the user to perform operations in polytime. The most well known such representation is Ordered Binary Decision Diagram (OBDD). SAT models are often compiled to OBDDs so that this representation can be queried instead of solving the SAT model again.

There is a trend that more succinct representations are less tractable, thus selecting the appropriate representation can have a huge impact. The Knowledge Compilation Map [6] was created to define a structure that describes the relationship between some representations to help make an informed decision. 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 [6].

Negation Normal Form (NNF) is the base of the compilation map, a DAG that has as leaves $true$, $false$, literal X or $\neg X$ and internal nodes represent OR and AND operations. NNF does not qualify as a target compilation language, but many of its subset do.

There are two ways to compile a knowledge base (KB) such as CNF formulas: bottom-up and top-down compilation. Bottom-up compilation takes fragments of the KB, such as CNF clauses, compiles these and uses the `apply` operation to join the representations together efficiently. Top-down compilation takes the whole KB and recursively compiles fragments of it using conditioning, such as the trace of DPLL using a SAT solver. Top-down compilers work only on CNF inputs, whereas bottom-up compilers `apply` operation is able to deal with other input types as well. Top-down compilation has been compared to bottom-up compilation for OBDDs [8] and for SDDs [14], in both cases top-down compilation being faster.

In the following, we consider the four compilers we used and the representations they generate: PySDD[1] ZSDD[2], MiniC2D[3] and D4.[4]

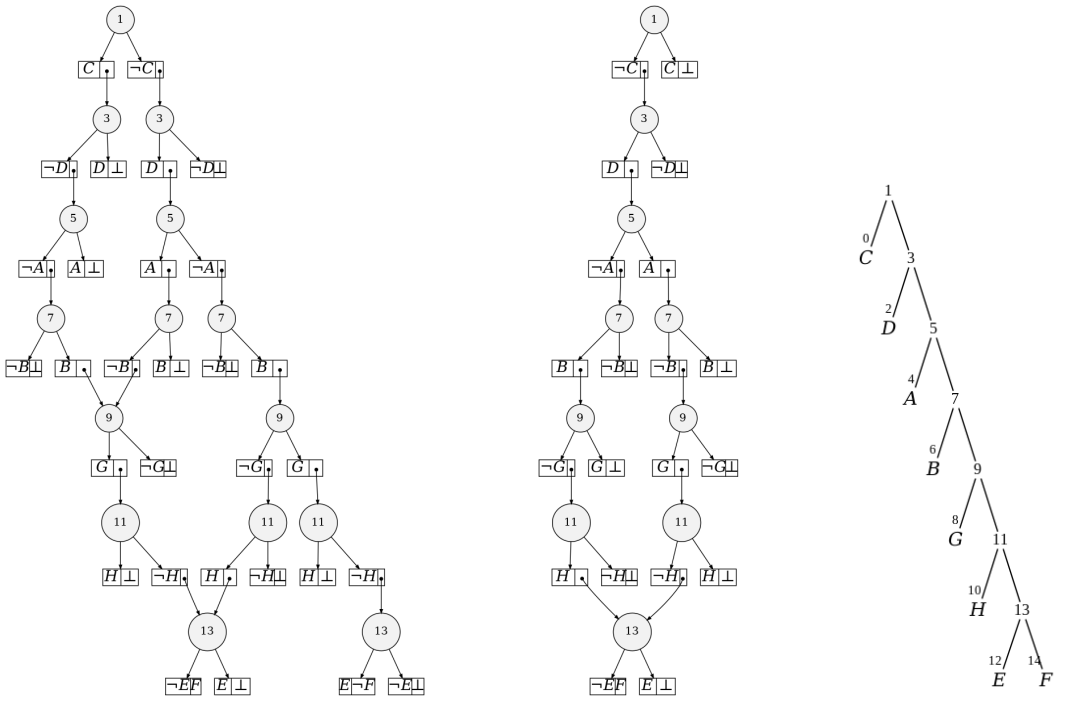
3.1 D4 : Decision-DNNF

D4 [9] is a top-down compiler with the target representation of Decision-DNNF. The deterministic-DNNF(d-DNNF) is an extension of NNF with decomposability and determinism properties. Decomposability is satisfied if for all conjunctions variables are not shared, that is, children of AND-nodes do not share variables. 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 order of the variables is defined by a dtree, defined later in Section 3.4

3.2 PySDD : Sentential Decision Diagram(SDD)

PySDD [5] is a Python wrapper for the open-source C SDD package that compiles CNF/DNF to Sentential Decision Diagram (SDD). This is an extension of NNF with structured decom-

[1] `https://github.com/wannesm/PySDD`
[2] `https://github.com/nsnmsak/zsdd/`
[3] `http://reasoning.cs.ucla.edu/minic2d/`
[4] `https://www.cril.univ-artois.fr/KC/d4.html`



(a) compiled SDD without SB **(b)** compiled SDD with SB **(c)** vrtee used

Figure 1 SDD of the direct encoded CSP: $v_1 + v_2 + v_3 = v_4$ with domains $\{1, 3\}$ for v_1, v_2, v_3 and $\{7, 9\}$ for v_4. We break the symmetry by adding $v_1 \leq v_2$, $v_2 \leq v_3$. Literal A denotes the assignment $v_1 = 1$, B denotes $v_1 = 3$ etc.

posability and strong determinism. These properties make it a strict subset of d-DNNF and a strict superset of OBDDs.

An SDD is a DAG (e.g. Figure 1a), where decision nodes (circles) represent (X,Y)-decompositions, i.e. an OR operation on sentences. These correspond to the nodes in the vtree associated to the SDD (e.g. Figure 1c). A vtree is a binary tree whose leafs correspond to literals. The rectangles in the SDD are represent 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 1a. Figure 1 contains the SDD without and with symmetry breaking of a sum constraint: $v_1 + v_2 + v_3 = v_4$. Using the same vtree, we can observe which paths in the SDD get eliminated when adding SB, that is, which symmetrical solutions are removed.

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. Similarly to OBDDs, where the variable ordering defines the structure, with SDDs vtrees do. OBDDs are canonical with respect to variable ordering, SDDs are with respect to vtrees. Using a right linear vtree makes the SDD equivalent to an OBDD.

3.3 ZSDD : Zero Suppressed Sentential Decision Diagram

Zero Suppressed Sentential Decision diagrams (ZSDDs) are an extension to SDDs such that while terminal symbols for SDDs are the literal X or $\neg X$, ZSDDs have terminals ϵ and $\bot$,

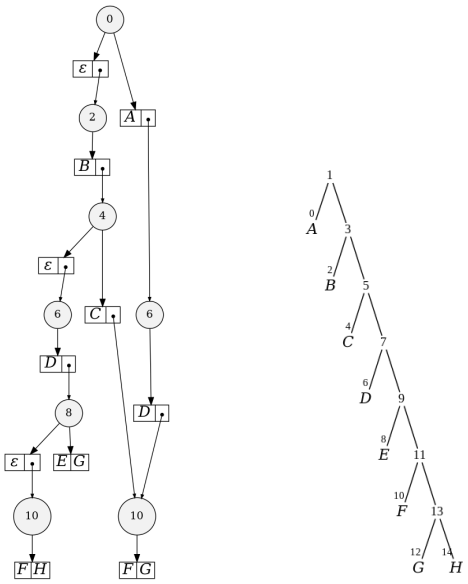


Figure 2 ZSDD representing the CSP from Figure 1 and the vtree used

where ϵ represents Boolean functions that returns 1 if all variables are 0 [13]. 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 in these. Solution counting runs in time linear to size of ZSDD. In ZSDD a decision node is removed if the corresponding function return true only when all variables from either left or right subtrees are assigned 0. With this in mind, the expectation is that when we break symmetries, decision nodes get removed. The ZSDD also uses bottom-up compiler, and the representation can be used as a more compact alternative to SDDs [13]. Figure 2 visualizes a ZSDD for the same CSP as Figure 1.

3.4 MiniC2D : Decision-SDD

MiniC2D uses a top-down approach to compile CNF to Decision-SDD [14]. A vtree node whose left child is a leaf is called Shannon node, otherwise it is a decomposition node. An SDD with any vtree can be a Desicion-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 $s_1 = \neg s_2$. A special form of vtree called dtree guarantees and SDD to be a Decision-SDD. 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 dtree if every clause is compatible with only Shannon nodes. Thus, for every clause we identify the lowest vtree node that is a common root to the literals, the left child of this node has to be a leaf node. An example representation is found in Figure 3.

A vtree with all Shannon nodes is called right-linear vtree, and an SDD constructed with such a vtree is equivalent to an OBDD. 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.

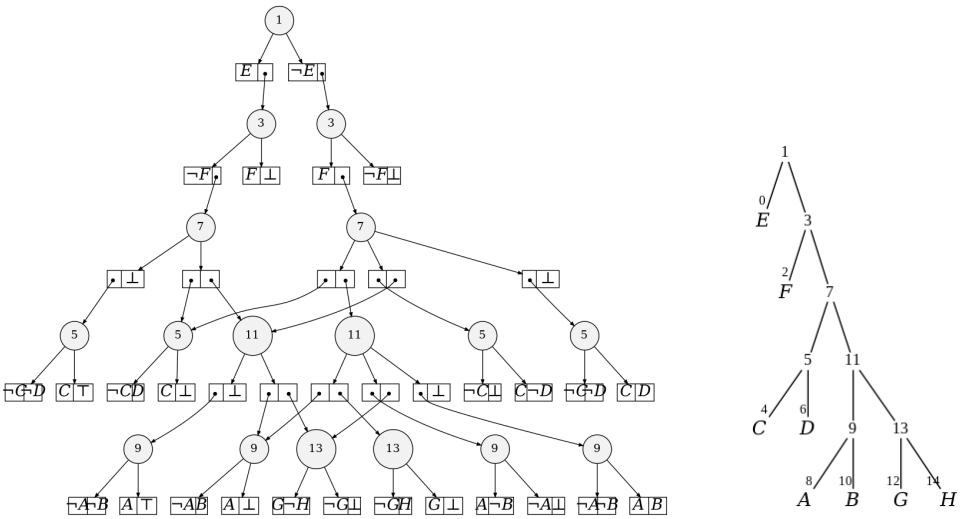


Figure 3 Decision-SDD representing the CSP from Figure 1 and the vtree used

4 Results

We considered four compilers: PySDD, ZSDD, D4 and MiniC2D. For all experiments a single machine was used with 11th Gen Intel Core i7-1165G7 @ 2.80GHz × 8 running Ubuntu 20.04.3 LTS. 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 used the direct encoding to formulate the problems as CNF input. Each experiment was run five times and averaged results are shown. PySDD used by default a balanced initial vtree and post-compilation vtree search. D4 is the only compiler that is not deterministic, as its default parameters include a random factor. In each of the following experiments we looked at compilation time and size, defined by node count. We use the terms size and node count interchangeably, as well as solution and model count.

Cliques of Not-Equals. As a first experiment we reconstructed the alldifferent constraint using a clique of not-equal constraints with and without symmetry breaking. We created 10 CSPs with pairwise not-equal constraints amongst 10 variables $X = \{x_1, x_2, ..., x_{10}\}$ and 10 to 20 domain values for each x_i $D = \{D_1 = \{1, ..10\}, D_2 = \{1, ..11\}, ..., D_5 = \{1, ..20\}\}$. We break the symmetry by ordering the variables such that $x_1 < x_2 < x_3 < ... < x_{10}$. The number of literals ranges from 100 to 200, and number of clauses from 910 to 11360.

Table 1 presents the results with and without symmetry breaking, time expressed in seconds. For all instances there is a large 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 SB, these models compile within a few seconds. We ignore ZSDD, as compilation fails due to excessive memory use. No tool was able to compile the problem with domain size 18 and no SB, with only PySDD succeeding for domain sizes 19 and 20.

N-queens Problems. 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 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. These contain 16 to 144 literals and from 80 to 2680 clauses.

Table 1 Time and size measurements for compiling clique of not-equals

	Domain size	PySDD		MiniC2D		D4		Model count
		Time(s)	Size	Time(s)	Nodes	Time(s)	Nodes	
No symmetry breaking	10	7.60	19768	19.03	668743.4	19.15	2616751	3628800
	11	19.35	41929	16.15	2238849.8	69.02	9260088	39916800
	12	57.68	81518	32.78	3320898.6	169.28	22268278	239500800
	13	169.42	187757	30.49	7516849.8	369.00	50075258	1037836800
	14	199.44	195015	65.95	16547016.4	652.62	91864534	3632428800
	15	531.12	899580	85.47	26786678.8	1019.86	151085371	10897286400
	16	170.63	307686	139.00	44592981	1321.55	246739107	29059430400
	17	-	-	141.09	33779490	-	-	70572902400
	18	-	-	-	-	-	-	-
	19	376.82	897798	-	-	-	-	335221286400
	20	1685.89	5208404	-	-	-	-	670442572800
With symmetry breaking	10	1.02	272	0.46	199	0.05	90	1
	11	1.18	441	0.49	439.2	0.03	313	11
	12	1.74	653	0.50	622.6	0.03	498	66
	13	2.29	826	0.64	1075.4	0.04	746	286
	14	2.80	1030	0.68	3632.4	0.05	1020	1001
	15	4.10	1301	0.80	16621.8	0.05	1487	3003
	16	5.20	1409	1.00	36275.8	0.06	1945	8008
	17	7.58	1656	1.20	47154.8	0.07	2933	19448
	18	9.91	2041	1.52	74144.8	0.08	3570	43758
	19	8.45	2366	1.66	98443.4	0.09	4193	92378
	20	12.60	2522	2.19	257089.6	0.11	5789	184756

Table 2 shows time and node count of each compiler, without and with symmetry breaking. Around $50-60\%$ of the solutions get eliminated when breaking symmetry. For the 10-queens problem, ZSDD compiles in only 7.3 minutes without symmetry breaking but when SB is added, instead of 1480 we have 1530 clauses, compilation times out within 30 minutes. For 15-queens, only D4 was able to compile the formulation with SB, in 16.02 minutes. The top-down compilers perform better, D4 obtains the most reduction both in time and node count, MiniC2D following it for node count. Time reduction for all compilers ranges from no reduction to 40%, with time for $N < 10$ being below 2 seconds. ZSDD achieves the most reduction but is only able to compile the first 7 instances, and for N=9,10 significantly slower than other tools.

VS-SDD [11] also look at the N-queens problem with N=8,9,10,11. The size of the compiled SDDs differ, that can be caused by a different order of the CNF clauses, but they are relatively close. 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 SB gets 61.3%, 62.1% , 56.9% and 57.1%.

Balanced Incomplete Block Designs. Next we looked at the Balanced Incomplete Block Design (BIBD) problem, 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. We took the CNF formulations from CSPLib[5], with only instance (7,7,3,3,1) compiling within 30 minutes. This is expressed using 833 literals and 7028 clauses and 10080 clauses for symmetry breaking added.

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 summarizes the compilation results.

[5] `https://www.csplib.org/Problems/prob028/models/`

Table 2 Time and node count measurements for compiling the N-queens problem

	N	PySDD		ZSDD		MiniC2D		D4		Model count
		Time(s)	Nodes	Time(s)	Nodes	Time(s)	Nodes	Time(s)	Nodes	
No symmetry breaking	4	0.039	35	0.551	7	0.071	54.4	0.025	35	2
	5	0.079	154	0.533	44	0.093	238	0.018	204	10
	6	0.138	117	0.631	23	0.148	190.4	0.018	150	4
	7	0.380	571	0.875	218	0.202	1350.8	0.023	1240	40
	8	0.821	976	2.524	456	0.274	3650.2	0.034	3308	92
	9	2.551	2229	88.300	1651	0.411	15097.8	0.087	13076	352
	10	11.312	4032	438.197	3833	0.605	37655.6	0.222	32986	724
	11	169.036	10855	-	-	1.407	157622.2	0.932	128993	2680
	12	478.861	35047	-	-	8.581	873364	5.323	693163	14200
	13	-	-	-	-	59.813	4688929.4	30.369	3556955	73712
	14	-	-	-	-	762.773	24971462.2	224.056	18161401	365596
	15	-	-	-	-	-	-	-	-	-
With symmetry breaking	4	0.038	18	0.477	3	0.064	31	0.018	16	1
	5	0.070	91	0.534	22	0.099	151.4	0.017	106	5
	6	0.111	73	0.636	11	0.149	117.6	0.017	74	2
	7	0.307	405	0.892	103	0.196	789.8	0.021	678	19
	8	0.665	615	3.646	203	0.267	1608	0.027	1476	35
	9	1.861	1432	88.147	829	0.398	7639.2	0.054	6404	158
	10	6.700	2380	-	-	0.565	17155.6	0.124	14767	289
	11	75.475	6522	-	-	1.408	73984.6	0.482	61224	1133
	12	240.976	18214	-	-	5.092	381331.4	2.475	294589	5564
	13	-	-	-	-	49.265	2230340.8	16.634	1597131	31051
	14	-	-	-	-	617.260	11157575.4	112.269	7352257	141988
	15	-	-	-	-	-	-	961.469	48590750	940824

Table 3 Compilation of the instance (7,7,3,3,1), where - denotes timeout

	PySDD		ZSDD		MiniC2D		D4		Model count
	Time(s)	Size	Time(s)	Edges	Time(s)	Edges	Time(s)	Edges	
No SB	1800	-	67.4	-	1800	-	874.3	-	-
With SB	1800	-	66.7	-	560.91	67153530	52.6	22716852	151200

5 Conclusion and future work

In general the top-down compilers reduce most the compilation size when breaking symmetry and they also compile faster. It is worth keeping in mind that the tools obtain compiled representations with different properties, so comparison among each other is not the focus. Rather we focus on compilations with and without symmetry breaking. In these highly symmetrical problems compilation size 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 gap exist for the different compilers.

In SDDs a decision node is removed if the corresponding Boolean function does not depend on variables in either the left or right subtree. So for instance if a subset of variable-value pair is compatible with all solutions and these literals have a common root in the vtree without any other literal in that tree, decision nodes with this common root can be eliminated. But if we break the symmetry and eliminate some combinations of these pairs the common root cannot be eliminated anymore. This could be one case where symmetry breaking could increase the SDD size, especially if no vtree is found to overcome this.

Formulating symmetry breaking clauses at CNF level is not an easy task, especially without introducing auxiliary variables. The maximal encoding [15] on the other hand, breaks interchangeability symmetry by formulation. As next steps we will investigate the relation between knowledge compilation and the maximal encoding.

References

1. 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.
2. 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.
3. David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints An Int. J.*, 2006.
4. 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, San Diego, CA, USA, June 7-11, 2004*. ACM, 2004.
5. 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.
6. Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 2002.
7. 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, 2016, Proceedings*. Springer, 2016.
8. 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, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.
9. Jean-Marie Lagniez and Pierre Marquis. An Improved Decision-DNNF Compiler. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017*. ijcai.org, 2017.
10. Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 2014.
11. 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*.
12. Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artif. Intell.*, 2017.
13. 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.
14. Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*. AAAI Press, 2015.
15. Steven D. Prestwich. Full dynamic substitutability by SAT encoding. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*. Springer, 2004.
16. 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, Prague, Czech Republic, May 30 - June 1, 2005, Proceedings*. Springer, 2005.
17. 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 2020, Dublin, Ireland, Proceedings, Part I*. Springer, 2020.