DSHARP: Fast d-DNNF Compilation with sharpSAT (Amended Version)*

Christian Muise[1] and **Sheila A. McIlraith**[1] and **J. Christopher Beck**[2] and **Eric Hsu**[1]
[1]Department of Computer Science, University of Toronto, Toronto, Canada.
[2]Department of Mechanical & Industrial Engineering, University of Toronto, Toronto, Canada.
[1]{cjmuise, sheila, eihsu}@cs.toronto.edu
[2]jcb@mie.utoronto.ca

Abstract

Knowledge compilation is a compelling technique for dealing with the intractability of propositional reasoning. One particularly effective target language is Deterministic Decomposable Negation Normal Form (d-DNNF). We exploit recent advances in #SAT solving in order to produce a new state-of-the-art CNF → d-DNNF compiler: DSHARP. The core technique we leverage for DSHARP is to extract the search trace from the complete #SAT solver. Empirical results demonstrate that DSHARP is generally an order of magnitude faster than C2D, the de facto standard for compiling to d-DNNF, while yielding a representation of comparable size.

1. Introduction

To deal with the intractability of propositional reasoning tasks, one can sometimes compile a propositional theory from a source language into a target language that guarantees tractability. This compilation process, popularly referred to as *knowledge compilation*, has proved to be an effective technique for dealing with many practical reasoning problems (Darwiche and Marquis 2002). Here we are interested in Deterministic Decomposable Negation Normal Form (d-DNNF), a language that supports efficient reasoning for tasks such as consistency checking and model counting. d-DNNF has also been exploited more recently for a diversity of AI applications including Bayesian reasoning (Chavira, Darwiche, and Jaeger 2006), conformant planning (Palacios et al. 2005), diagnosis (Siddiqi and Huang 2008), and database queries (Jha and Suciu 2011).

The de facto standard for CNF → d-DNNF compilation is C2D, a tool developed and refined by Darwiche and colleagues over a number of years.[1] Although C2D is well designed and optimized, CNF → d-DNNF compilation can still be slow. Knowledge compilation has traditionally been characterized as an off-line process and its processing time is rationalized by amortizing it over numerous queries. However, recent problem specific use of d-DNNF in tasks such as planning and diagnosis challenges this characterization and emphasizes the need for fast compilation.

We propose a new CNF → d-DNNF compiler, DSHARP.[2] Our compiler builds on the research by Huang and Darwiche showing that d-DNNF can be extracted from the trace of an exhaustive search of a propositional theory (Darwiche 2004). To this end, we construct our compiler by appealing to a state-of-the-art #SAT solver, sharpSAT (Thurley 2006). Our compiler exploits two significant features of sharpSAT that distinguish it from previous CNF → d-DNNF compilers: dynamic decomposition and implicit binary constraint propagation.

We evaluated the performance of DSHARP on 300 problem instances over eight domains taken from SatLib[3] and the Fifth International Planning Competition.[4] DSHARP solved more problem instances than C2D in the time allowed, and showed a significant improvement in run time. The size of the resulting d-DNNF representation was maintained, and was on average five times smaller. We additionally performed an analysis of the DSHARP components that impact the compiler's efficiency. Further details on this experiment and a more in depth analysis of the results can be found in (Muise et al. 2010).

2. Preliminaries

Darwiche and Marquis proposed the *knowledge compilation map*, an analysis of a number of target compilation languages with respect to two key features: succinctness and the class of queries and transformations that the language supports in polytime (Darwiche and Marquis 2002). The set of tasks considered includes consistency, validity, clausal entailment, implicant checking, equivalence, sentential entailment, model counting, and model enumeration. The most general target language of the map is Negation Normal Form (NNF), a directed acyclic graph in which the label of each leaf node is a literal, TRUE, or FALSE, and the label of each internal node is a conjunction ($\wedge$) or a disjunction ($\vee$). Here we study compilation to d-DNNF, the subset of NNF satisfying *decomposability* and *determinism*. We define NNF to be the family of boolean formulae that are built from the operators $\vee$, $\wedge$, and $\neg$, with the added restriction that all $\neg$ operators exist only at the literal level. De-

*A version of this paper also appears in the Proceedings of the 25th Canadian Conference on Artificial Intelligence (CAI-12). The only notable difference is the addition of Section 5.

[1] http://reasoning.cs.ucla.edu/c2d/

[2] http://www.haz.ca/research/dsharp/

[3] http://www.satlib.org/

[4] http://www.ldc.usb.ve/~bonet/ipc5/

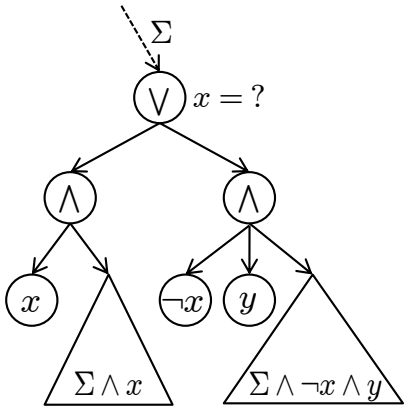


Figure 1: Partial d-DNNF from an exhaustive DPLL trace.

composable Negation Normal Form (DNNF) is the subset of NNF formulae whose members additionally have the property that the formula operands of $\wedge$ do not share variables. Finally, d-DNNF is the subset of DNNF whose members have the additional property that the formula operands of $\vee$ are logically inconsistent. d-DNNF permits polytime (in the size of the representation) processing of clausal entailment, model counting, model minimization, model enumeration, and probabilistic equivalence testing (Darwiche 2004). The conceptualization of d-DNNF as a directed acyclic *and-or* graph helps us understand its relation to the DPLL trace.

Exhaustive DPLL Trace To develop a state-of-the-art CNF $\rightarrow$ d-DNNF compiler, we use a result of Huang and Darwiche that shows we can extract d-DNNF from the trace of an exhaustive search of a propositional theory (Huang and Darwiche 2005). More specifically, we exploit the exhaustive search performed by the #SAT solver, sharpSAT (Thurley 2006). The exhaustive DPLL algorithm is a modification of DPLL used to find all solutions and, therefore, to implicitly explore the entire search space. Each node in the search tree corresponds to a decision in the exhaustive DPLL search (i.e., assigning a variable to either TRUE or FALSE). Decision nodes correspond to *or* nodes in the d-DNNF representation. For each *or* node, we add *and* nodes as children, corresponding to the subtrees for the decision variable's setting and any variable assignments inferred by unit propagation.

Fig. 1 shows an example of part of the d-DNNF at a decision node where variable x has been chosen. The theory as it exists before setting x is Σ, and the theory solved for each subproblem is $\Sigma \wedge x$ and $\Sigma \wedge \neg x \wedge y$ (after unit propagation is run). If any unit propagation occurs due to the variable being set, we record the implied literals under the appropriate *and* node. For example, Fig. 1 shows the literal y as an implication of setting $x =$ FALSE.

Following this approach, we are left with an *and-or* tree with the leaf nodes corresponding to literals of the theory. The tree has all of the required properties to qualify as a representation for the d-DNNF language: it is in negation normal form since the negations are at the literal level, it is decomposable because the children of *and* nodes are disjoint theories, and it is deterministic since the immediate children of every *or* node has both a literal and its negation making the conjunction inconsistent.

3. DSHARP

sharpSAT is a state-of-the-art solver for the problem of #SAT. DSHARP uses the algorithmic components of sharpSAT responsible for its strong performance. Specifically, we have adapted the following to compute a d-DNNF representation: dynamic decomposition, implicit binary constraint propagation, conflict analysis, non-chronological backtracking, pre-processing, and component caching. Here, we describe each component and the modifications we made to produce an efficient CNF $\rightarrow$ d-DNNF compiler.

Dynamic Decomposition A theory in CNF is disjoint if it can be partitioned into sets of clauses (called components) such that no two sets share variables. We can compile each component individually and combine the results, a technique called *disjoint component analysis*. This technique changes the structure of the d-DNNF representation; we treat each component as an individual theory and add the d-DNNF for each component as a child to the *and* node where the theory was found to be disjoint. Consider Fig. 2. After the solver decides that $x_1 =$ TRUE, the theory decomposes into two components (corresponding to the parts of the d-DNNF rooted at each *or* node marked I).

There are two prevailing methods for disjoint component analysis. In *static decomposition*, the solver computes the components prior to search while in *dynamic decomposition*, the solver computes the components during search. There is a trade-off between the two methods in terms of simplicity, computational difficulty, and effectiveness. C2D uses a static decomposition while DSHARP uses dynamic decomposition.

Implicit Binary Constraint Propagation DSHARP employs a simple form of lookahead during search called *implicit binary constraint propagation* (IBCP) (Thurley 2006). In IBCP, a subset of the unassigned variables are heuristically chosen at a decision node and the impact of assigning any one of them is evaluated. We test each variable in the chosen set for both TRUE and FALSE. If either assignment causes unit propagation to derive an inconsistency, the solver soundly infers the opposite assignment.

IBCP, via unit propagation, may infer the assignment of a number of literals during the lookahead. Unless the theory becomes inconsistent, these implications should be ignored since the variable setting will be undone. DSHARP maintains the temporary implications and includes them permanently only when a variable setting is kept.

Conflict Analysis / Non-Chronological Backtracking *Conflict analysis* refers to the use of conflict clauses to reduce search effort. When the solver reaches a dead end in the search space it records a reason for this conflict in the form of a new clause. We add the clause to the theory, and

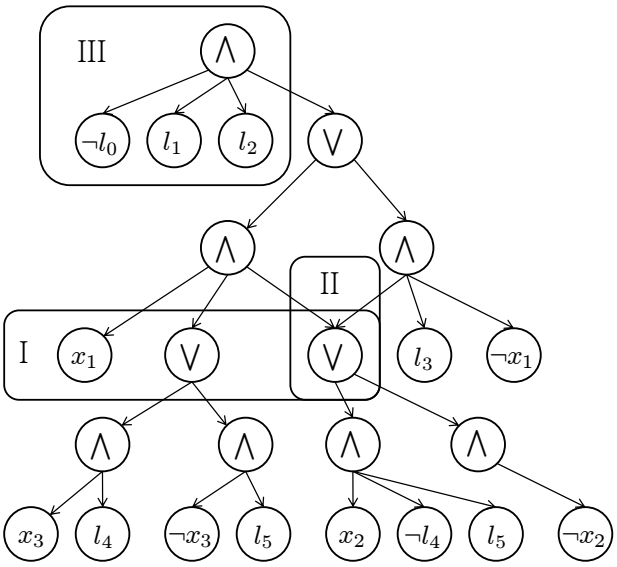


Figure 2: Example d-DNNF representation as DSHARP may generate during search.

subsequently include it in unit propagation and the computation of heuristics. *Non-chronological backtracking* (NCB) uses learned conflict clauses to backtrack past the most recent assignment to the highest decision node possible while remaining sound. Both conflict analysis and NCB are widely used in a variety of SAT-solving applications and solvers (Beame, Kautz, and Sabharwal 2003). The addition of conflict clauses during the solving procedure does not change the structure of the d-DNNF. When DSHARP uses NCB it must step back in the partial d-DNNF to the correct spot before continuing to record, but this does not affect the structure of the d-DNNF representation either.

Component Caching *Component caching* is an extension of disjoint component analysis where the solver stores the d-DNNF result for each component and retrieves it if DSHARP encounters that component again during search. Caching can have substantial savings when the theory naturally decomposes during search. One way of handling component caching in the trace would be to duplicate the repeated d-DNNF subtree when DSHARP re-encounters a component. However, if we relax the assumption that the d-DNNF representation is an *and-or* tree, we can simply link to the part of the d-DNNF corresponding to the repeated component. The d-DNNF representation then becomes a DAG: a more concise form of representing the d-DNNF. Fig. 2 (II) shows an example of a d-DNNF when DSHARP reuses a component through component caching.

Pre-processing Finally, *pre-processing* is a version of IBCP used at the root node to simplify the starting theory. Pre-processing performs the same lookahead as IBCP, but on all variables rather than on a heuristically chosen subset. If a setting to a variable exists such that unit propagation causes the theory to become inconsistent, the solver soundly infers the opposite setting. If pre-processing finds any variables to set, DSHARP records these as leaf nodes under a root *and* node. The search proceeds as usual with the compiled d-DNNF attached as a child to the root node. Fig. 2 (III) shows an example of the result of pre-processing with literals $\neg l_0$, l_1, and l_2 inferred during pre-processing.

4. Experimental Analysis

To evaluate the DSHARP system, we compared both compilation speed and the size of the output representation to that of C2D. Experiments were conducted on a Linux desktop with a two-core 3.0GHz processor. Individual runs were limited to a 30-minute time-out and a 1.5GB memory limit. DSHARP was run with its default settings, and C2D was run with dt_method 4. While there is an extensive range of settings for C2D, we found that this setting performed consistently well. Similar to (Huang and Darwiche 2005), we used the number of edges in the d-DNNF as an indication of the size of the generated result.

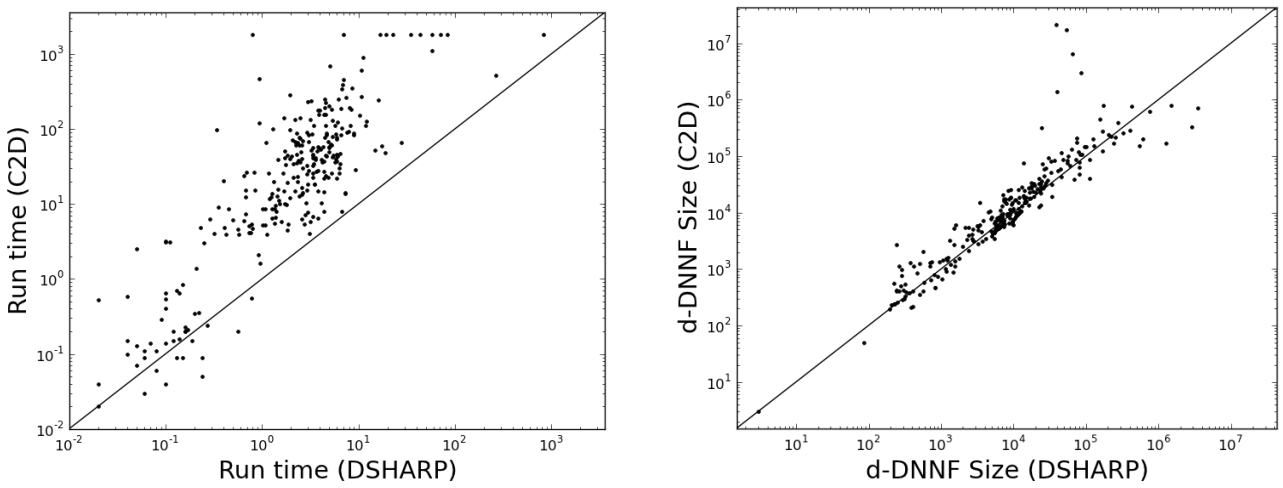
We selected the benchmarks to cover a range of problem types: uniform random 3SAT, structured problems encoded as CNF (blocksworld; bounded model checking; flat graph colouring; and logistics), and conformant planning problems converted to CNF as described in (Palacios et al. 2005) (emptyroom; grid; and sortnet).

Fig. 3 shows a broad picture of the results for compiler run time and resulting size. All problems solved by at least one solver are present in Fig. 3a and all problems that both solved are in Fig. 3b. Points above the $y = x$ line indicate better performance of DSHARP (i.e., smaller run time and smaller size, respectively). Fig. 3a shows that DSHARP achieved a lower run time on almost all of the problem instances (274 of the 286 solved by at least one solver) while Fig. 3b demonstrates that the sizes of the output are comparable, with a few outliers in favour of each solver.

DSHARP solved more instances than C2D in five of the eight domains and an equal number in the remaining three. Overall, DSHARP solved 286 of the 300 instances while C2D only solved 275. DSHARP was significantly faster in all but one domain (blocksworld) and it was 27 times faster on average. When DSHARP was faster, it was by at least one order of magnitude in all but one domain (empty room). The results for d-DNNF size are more even: in three domains DSHARP was significantly smaller and in one domain it was significantly larger. In the remaining domains, the difference in output size was not statistically significant. When considering problems from all domains, we found that C2D produced d-DNNF representations about 5 times larger than DSHARP, though this difference was not statistically significant. Further details on the results and an analysis of the impact of the DSHARP components can be found in (Muise et al. 2010).

5. Related Work and Extensions

Relying on the trace of a complete DPLL based solver is the prevailing technique for CNF $\rightarrow$ d-DNNF compilers; used by both DSHARP and C2D (Darwiche 2004). The DPLL



(a) Run time comparison

(b) Size comparison

Figure 3: Scatter plot of the run time (in seconds) and the number of edges in the generated d-DNNF for each problem instance using C2D (y-axis) or DSHARP (x-axis). Points above the line represent problems where DSHARP was better. All axes are log-scale.

trace has been leveraged for proving bounds on model counting, such as the work of Beame et al. (2013). It has also been used to good effect for proving theoretical results about the worst-case size complexity of the d-DNNF representation (Oztok and Darwiche 2014).

Knowledge compilers (including the DSHARP software presented in this paper) have played a role as subcomponents in larger systems. One example is in the SAT-based analysis of quantification information flow programs, where efficient methods for conditioned model counting using d-DNNF were exploited (Klebanov, Manthey, and Muise 2013). Another example is the Probabilistic Logic Programming framework, ProbLog (Fierens et al. 2015). Here, the model counters were used to provide probabilistic inference for targeted queries compiled from a given logic program.

Finally, since the original publication of this work in 2012 by Muise et al., the DSHARP software has been extended in a variety of ways. Most notably, the software has been extended to do model counting with the assumption of stable model semantics (Aziz et al. 2015b), as well as extended to do projected model counting and knowledge compilation (Aziz et al. 2015a).

6. Concluding Remarks

d-DNNF is proving to be an effective language for a diversity of practical AI reasoning tasks including Bayesian inference, conformant planning, and diagnosis. Many of these applications require the CNF $\rightarrow$ d-DNNF compilation to be performed on a problem-specific basis, and as such compilation time is included in the measure of performance of the overall system. CNF $\rightarrow$ d-DNNF compilers, therefore, need to be fast while continuing to produce high quality representations. We address this need through the development of a new state-of-the-art CNF $\rightarrow$ d-DNNF compiler that builds on #SAT technology, and in particular on advances found in the solver, sharpSAT. Our system, DSHARP, exploits the DPLL trace constructed for model counting to construct a d-DNNF representation of the propositional theory. DSHARP leverages the latest advances in #SAT technology, including dynamic decomposition, IBCP, conflict analysis, NCB, component caching, and pre-processing. We tested DSHARP on a variety of problem sets in SAT solving and planning. DSHARP solved more instances than C2D in the time allowed, averaging an improvement of 27 times in run time while maintaining the size of the d-DNNF generated by C2D. In future work, we plan to experiment with further optimizations of DSHARP and applications to more diverse AI domains.

Acknowledgements

The authors gratefully acknowledge funding from the Ontario Ministry of Innovation and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Aziz, R. A.; Chu, G.; Muise, C.; and Stuckey, P. 2015a. Projected model counting. In *International Conference on Theory and Applications of Satisfiability Testing*.

Aziz, R. A.; Chu, G.; Muise, C.; and Stuckey, P. 2015b. Stable model counting and its application in probabilistic logic programming. In *The 29th AAAI Conference on Artificial Intelligence*.

Beame, P.; Li, J.; Roy, S.; and Suciu, D. 2013. Lower bounds for exact model counting and applications in probabilistic databases. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013, Bellevue, WA, USA, August 11-15, 2013*.

Beame, P.; Kautz, H.; and Sabharwal, A. 2003. Understanding the power of clause learning. In *International*

Joint Conference on Artificial Intelligence, volume 18, 1194–1201.

Chavira, M.; Darwiche, A.; and Jaeger, M. 2006. Compiling relational bayesian networks for exact inference. *International Journal of Approximate Reasoning* 42:4–20.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17:229–264.

Darwiche, A. 2004. New advances in compiling CNF to decomposable negational normal form. In *Proceedings of European Conference on Artificial Intelligence*.

Fierens, D.; den Broeck, G. V.; Renkens, J.; Shterionov, D. S.; Gutmann, B.; Thon, I.; Janssens, G.; and Raedt, L. D. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *TPLP* 15(3):358–401.

Huang, J., and Darwiche, A. 2005. DPLL with a trace: from SAT to knowledge compilation. In *International Joint Conference On Artificial Intelligence*, 156–162.

Jha, A., and Suciu, D. 2011. Knowledge compilation meets database theory: compiling queries to decision diagrams. In *Proceedings of the 14th International Conference on Database Theory*, 162–173. ACM.

Klebanov, V.; Manthey, N.; and Muise, C. 2013. SAT-based Analysis and Quantification of Information Flow in Programs. In *10th International Conference on Quantitative Evaluation of SysTems (QEST 2013)*, 177–192.

Muise, C.; McIlraith, S. A.; Beck, J. C.; and Hsu, E. 2010. Fast d-DNNF compilation with sharpSAT. In *Workshop on Abstraction, Reformulation, and Approximation (AAAI-10)*.

Muise, C.; Mcilraith, S. A.; Beck, J. C.; and Hsu, E. 2012. DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*.

Oztok, U., and Darwiche, A. 2014. On compiling CNF into decision-dnnf. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, 42–57.

Palacios, H.; Bonet, B.; Darwiche, A.; and Geffner, H. 2005. Pruning conformant plans by counting models on compiled d-DNNF representations. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling*, 141–150.

Siddiqi, S., and Huang, J. 2008. Probabilistic sequential diagnosis by compilation. *Tenth International Symposium on Artificial Intelligence and Mathematics*.

Thurley, M. 2006. sharpSAT — counting models with advanced component caching and implicit BCP. In *Ninth International Conference on Theory and Applications of Satisfiability*.