Fast d-DNNF Compilation with sharpSAT

Christian Muise Sheila McIlraith J. Christopher Beck Eric Hsu
Department of Computer Science, University of Toronto, Toronto, Canada. M5S 3G4.
{cjmuise, sheila, eihsu}@cs.toronto.edu jcb@mie.utoronto.ca

Abstract

Knowledge compilation is a valuable tool for dealing with the computational intractability of propositional reasoning. In knowledge compilation, a representation in a source language is typically compiled into a target language in order to perform some reasoning task in polynomial time. One particularly popular target language is Deterministic Decomposable Negation Normal Form (d-DNNF). d-DNNF supports efficient reasoning for tasks such as consistency checking and model counting, and as such it has proven a useful representation language for Bayesian inference, conformant planning, and diagnosis. In this paper, we exploit recent advances in #SAT solving in order to produce a new state-of-the-art CNF $\rightarrow$ d-DNNF compiler. We evaluate the properties and performance of our compiler relative to C2D, the de facto standard for compiling to d-DNNF. Empirical results demonstrate that our compiler is generally one order of magnitude faster than C2D on typical benchmark problems while yielding a d-DNNF 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 of the task. This compilation process, popularly referred to as *knowledge compilation*, has proven an effective tool for dealing with many practical reasoning problems (e.g., (Darwiche and Marquis 2002)).

Perhaps the best known target language is the language captured by Ordered Binary Decision Diagrams (OBDDs), a data structure that is commonly used in circuit synthesis and verification (Ranjan et al. 1995). Here we are interested in Deterministic Decomposable Negation Normal Form (d-DNNF), a strict superset of OBDDs that is also more succinct. d-DNNF 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 and Geffner 2006), and diagnosis (Siddiqi and Huang 2008).

The de facto standard for CNF $\rightarrow$ 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 $\rightarrow$ d-DNNF compilation can still be slow. Knowledge compilation has traditionally been characterized as an off-line process and therefore its processing time can often be rationalized by amortizing it over numerous subsequent queries. However, more recent use of d-DNNF in tasks such as planning and diagnosis has been problem specific, challenging this characterization and emphasizing the need for fast compilation.

Motivated by this need, in this paper we propose a new CNF $\rightarrow$ d-DNNF compiler, DSHARP (available online at `http://www.haz.ca/research/dsharp/`). Our compiler builds on the research results by Huang and Darwiche showing that we can extract target languages such as d-DNNF 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 $\rightarrow$ d-DNNF compilers: dynamic decomposition, and implicit binary constraint propagation.

Our objective in constructing DSHARP was to develop a state-of-the-art CNF $\rightarrow$ d-DNNF compiler that was faster than C2D, while preserving the size of the output. We evaluated the performance of our compiler on 300 problem instances over eight problem domains taken from SatLib[2] and the Fifth International Planning Competition.[3] DSHARP solved more problem instances than C2D in the time allowed, showing a significant improvement in run time. The size of the resulting d-DNNF representation was maintained, and was on average five times smaller. In addition to these experiments, we also delved deeper into the workings of our compiler to attempt to determine the components that contributed significantly to this improved performance. To this end we did extensive ANOVA testing, identifying several components of our system as being critical to this impressive speedup.

In Section 2, we review some basic terminology related to d-DNNF. We follow in Section 3 with a review of the relationship between knowledge compilation and the search trace of an execution of the Davis, Putnam, Logemann, and Loveland algorithm (DPLL) for determining Satisfiability (Davis, Logemann, and Loveland 1962), and a discussion of our approach to developing DSHARP. In Section 4 we present our experimental results, and conclude with a discussion in Section 5.

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

[2] `http://www.satlib.org/`

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

2 Preliminaries

In (Darwiche and Marquis 2002) the authors proposed a so-called *knowledge compilation map*, an analysis of a number of target compilation languages with respect to two key features: succinctness of the target language, and the class of queries and transformations that the language supports in polytime. The target languages that they analyzed were not restricted to classical "flat" normal forms such as CNF or DNF, but also include a relatively large class of languages based on directed acyclic graphs (DAGs). This class of languages included both OBDDs and d-DNNF, and helps highlight the benefit that can be yielded by an alternative characterization of languages in terms of a graph structure.

The knowledge compilation map proposed a hierarchy of target languages. The root of the map is Negation Normal Form (NNF), a DAG 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 ($\vee$). While NNF is technically not a target language itself (since it does not permit a polytime clausal entailment test) there are two distinct subsets of NNF whose members are target languages– a flat subset and a nested subset. Our interest here is with the nested subset. We distinguish members of the nested subset by their properties including decomposability, determinism, and smoothness. From these properties, a subset relation is induced among the languages. The languages are then characterized with respect to the tasks they enable in polytime. The set of tasks considered includes consistency, validity, clausal entailment, implicant checking, equivalence, sentential entailment, model counting, and model enumeration.

Here we study compilation to d-DNNF. d-DNNF is the subset of NNF satisfying decomposability and determinism. More precisely, let $Vars(n)$ be the propositional variables that appear in the subgraph rooted at n, and let $\triangle(n)$ denote the formula represented by n and its descendants. Decomposability holds when $Vars(n_i) \cap Vars(n_j) = \emptyset$ for any two children n_i and n_j of an *and* node of n. Determinism holds when $\triangle(n_i) \wedge \triangle(n_j)$ is logically inconsistent for any two children n_i and n_j of an *or* node of n.

Alternatively, we can understand d-DNNF as a set of well-formed formulae of the following form. 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. Decomposable 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 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, described in the section to follow.

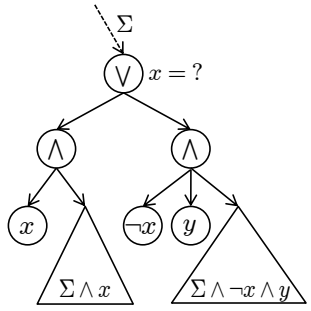


Figure 1: Partial d-DNNF from an exhaustive DPLL trace.

3 DSHARP

The primary objective of this work is to develop a state-of-the-art CNF → d-DNNF compiler that exploits recent advances in #SAT technology, with the aim of reducing compilation time while maintaining d-DNNF size relative to existing compilers. We use a result of Huang and Darwiche that shows we can extract target languages such as d-DNNF from the trace of an exhaustive search of a propositional theory. More specifically, we exploit the exhaustive search performed by the #SAT solver, sharpSAT. In Section 3.1 we review the Huang and Darwiche result. Then in Section 3.2 we discuss how we employ the features of sharpSAT to instead generate d-DNNF within our new DSHARP compiler.

3.1 d-DNNF from an Exhaustive DPLL Trace

In order to perform the CNF → d-DNNF compilation, we use the approach introduced in (Huang and Darwiche 2005) to record the search space of an exhaustive DPLL procedure. The exhaustive DPLL algorithm consists of a DPLL algorithm modified to find all solutions and, therefore, to implicitly explore the entire search space. Each node in the DPLL search tree corresponds to a decision in the exhaustive DPLL search (i.e., a variable selection and a choice of assignment 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. Figure 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 $\Sigma \wedge \neg x \wedge y$. If any unit propagation occurs due to the variable being set, we record the implied literals under the appropriate *and* node. For example, Figure 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.

3.2 DSHARP Components

The sharpSAT solver is the current state-of-the-art solver for the problem of #SAT. DSHARP, by being built on top of sharpSAT, uses the algorithm components that lead to its strong performance. Specifically, we have adapted: dynamic decomposition, implicit binary constraint propagation, conflict analysis, non-chronological backtracking, preprocessing, and component caching. In this section we describe each component and the modifications required to produce a sound CNF → d-DNNF compiler.[4]

Dynamic Decomposition When we can partition a theory in CNF into sets of clauses such that no two sets share variables, then the theory is disjoint and we refer to each set of clauses as a component. We can compile each component individually and combine the results, a technique called *disjoint component analysis*.

When used as part of a d-DNNF compiler, disjoint component analysis changes the structure of the d-DNNF representation; we treat each component as an individual theory, with a corresponding d-DNNF, and add the d-DNNF for each component as a child to the *and* node where the theory was found to be disjoint. For example, consider Figure 2. After making the decision $x_1 = \text{TRUE}$, 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 used for disjoint component analysis. In *static decomposition*, the solver computes disjoint components prior to search while in *dynamic decomposition*, the solver computes the components during search. There is a trade-off between the two approaches in terms of simplicity, computational difficulty, and effectiveness. C2D uses a form of static decomposition while DSHARP uses the dynamic decomposition of sharpSAT.

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. If either assignment causes unit propagation to derive an inconsistency, the solver soundly infers the opposite assignment. We test each variable in the chosen set for both TRUE and FALSE.

If IBCP infers a setting, we add the corresponding literal as a child to the appropriate *and* node. For example, the literal l_3 in Figure 2 could be created by IBCP, regular unit propagation, unit propagation of a conflict clause, or any combination of these – DSHARP views all forms as equivalent for the compilation.

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 these temporary implications and includes them when a variable setting is kept, discarding them otherwise.

[4]Further information regarding features of the sharpSAT solver that do not pertain to the modifications required for DSHARP can be found in (Thurley 2006).

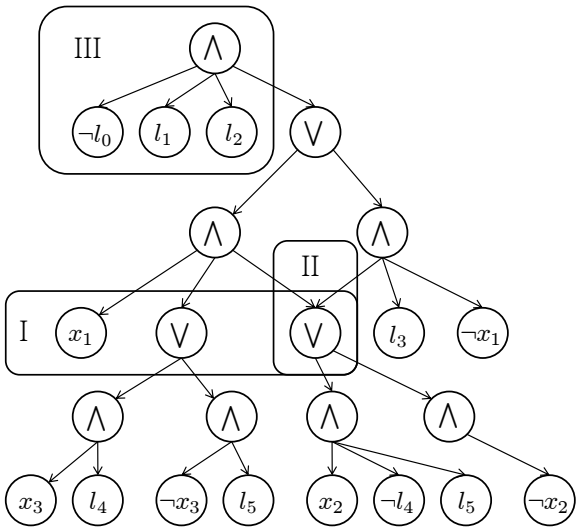


Figure 2: Example d-DNNF representation as DSHARP may generate.

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 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 the exhaustive DPLL procedure.

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 directed acyclic graph, a more concise form of representing the d-DNNF. Figure 2 (II) shows an example of the 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 pre-processing does find 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. Figure 2 (III) is an example of the results of pre-processing: literals $\neg l_0$, l_1, and l_2 were inferred in the pre-processing phase.

4 Experimental Analysis

To evaluate the DSHARP system, we conducted two experiments measuring both compilation speed and the size of the output representation: (i) a comparison of the performance of DSHARP with that of C2D and (ii) a fully crossed evaluation of the parameter settings of DSHARP.

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.

4.1 DSHARP vs. C2D

We tested DSHARP using a wide range of benchmarks and compared the results of both run time and output size to that of C2D. 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.[5] We used the number of edges in the resulting d-DNNF as an indication of the size of the generated result. This measure is typically used to gauge the size of d-DNNF representations (e.g., (Huang and Darwiche 2005)).

The benchmarks we used are: uniform random 3SAT (uf), structured problems encoded as CNF (blocksworld, bw; bounded model checking, bmc; flat graph colouring, flat; and logistics, log), and conformant planning problems converted to CNF as described in (Palacios et al. 2005) (emptyroom, empr; grid; and sortnet, stnt).

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

Table 1 presents the results for each domain and over all instances. Similar to the plots in Figure 3, the runtime comparisons were done on all problems solved by at least one solver, with a resource violation (time or memory) recorded as taking 1800 seconds. Size comparisons only consider instances where both DSHARP and C2D were able to find a solution.[6] On these instances, we present the number of problems solved in each domain, the mean improvement of run time (size), the mean ratio of run time (size) between C2D and DSHARP, and the number of problems DSHARP does better or worse for run time (size).

The significance of the differences in mean run time and size was tested using a randomized paired t-test (Cohen 1995) on a per domain basis and over all problems with a significance level of $p \leq 0.01$. A positive value indicates DSHARP performed better (faster or smaller), and results marked with a $\star$ are statistically significant. The mean ratio is the arithmetic mean of the individual ratio's for each problem — the run time (size) of C2D divided by the run time (size) of DSHARP. A value of k indicates that DSHARP was k times faster (or smaller) than C2D. We have marked all metrics where DSHARP outperforms C2D in boldface.

DSHARP solved more instances than C2D in five of the eight domains and an equal number in the remaining three. In most domains, DSHARP solved only a few more problem instances, with the exception of the grid and log domains where DSHARP solved substantially more problems than C2D. 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 overall. DSHARP was at least one order of magnitude faster in all but one domain (empty room).

The results for d-DNNF size are more even, as should be expected from Figure 3(b). 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.

4.2 The Impact of Parameter Settings

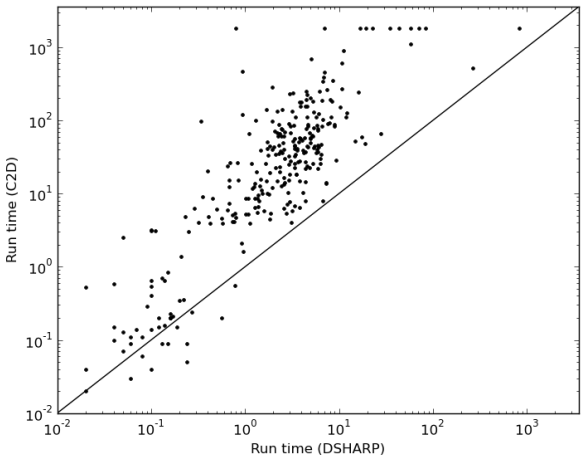
In order to better understand DSHARP's performance, we investigated the impact the various parameter settings have on both compilation time and the size of the generated d-DNNF representation. Recall that each parameter setting indicates whether DSHARP uses a given algorithm component or not. With the exception of dynamic decomposition we can switch each component on or off independently. The components are: implicit binary constraint propagation (IBCP), conflict analysis (CA), non-chronological backtracking (NCB), component caching (CC), and pre-processing (PP).

To measure the impact of each component, we performed a fully crossed Analysis Of Variance (ANOVA): DSHARP was run on every problem instance for every combination of parameter settings. ANOVA is a standard statistical tool for testing the null hypothesis that two or more distributions have equal mean. In our case, we separately tested the distribution of run times and output size for each of the parameter settings. For any parameter or parameter interaction that was deemed significant by the ANOVA (i.e., for which the null-hypothesis was rejected), a Tukey Honest Significance Difference (TukeyHSD) was performed to determine the best parameter setting.[7] A summary of the ANOVA and

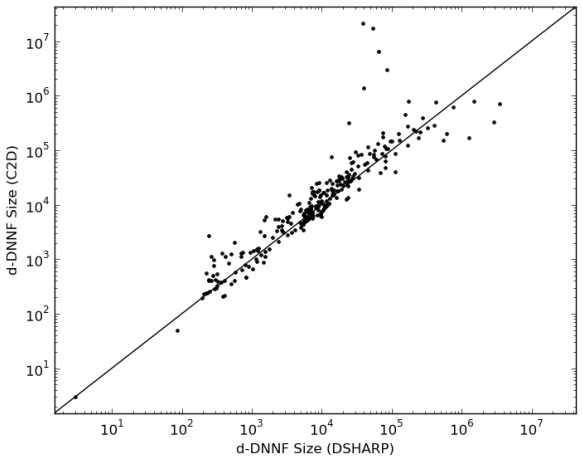
[5] A full analysis of many C2D parameter settings is provided in the appendix.

[6] While 1800s is a reasonable lower bound for time, we have no such lower bound on the size of the C2D's output.

[7] All statistical tests in this section were performed using the R



(a) Scatter plot of run time (in seconds) for each problem instance using C2D (y-axis) or DSHARP (x-axis).



(b) Scatter plot of the number of edges in the generated d-DNNF for each problem instance using C2D (y-axis) or DSHARP (x-axis).

Figure 3: Run time and size comparison of all problems. Points above the line represent problems where DSHARP was faster or smaller. Note that all axes are log-scale.

	Solved		Run time (sec.)			d-DNNF Size (edges)		
Domain	DSHARP	C2D	MI	MRI	+ / -	MI	MRI	+ / -
log (4)	2	0	n/a	n/a	n/a	n/a	n/a	n/a
grid (33)	32	26	**392.38***	**54.61**	**26 / 0**	**1455368.54**	**36.27**	8 / 18
stnt (12)	10	9	**206.58***	**27.92**	**7 / 2**	**889301.00***	**17.72**	**8 / 0**
bw (7)	7	6	**190.14**	**22.42**	**4 / 2**	-521.83	0.70	1 / 5
uf (100)	100	100	**88.38***	**20.91**	**100 / 0**	**6697.87***	**1.67**	**80 / 20**
bmc (13)	4	3	**758.16***	**14.65**	**3 / 0**	-1158787.00	0.75	1 / 2
flat (100)	100	100	**40.25***	**13.46**	**100 / 0**	**13005.24***	**1.42**	**82 / 18**
empr (31)	31	31	**0.094***	**2.89**	**23 / 7**	-1323.83*	0.83	0 / 31
all (300)	286	275	**123.93***	**27.73**	**263 / 11**	**161065.71**	**5.25**	**180 / 94**

Table 1: Comparative performance of DSHARP relative to C2D. MI is the **m**ean **i**mprovement of DSHARP over C2D, where a positive number indicates an improvement in seconds (for run time) or number of edges (for d-DNNF size). MRI is the **m**ean **r**elative **i**mprovement, the arithmetic mean of C2D's run time (output size) over that of DSHARP. An MRI > 1 indicates DSHARP performed better. The '+ / -' column indicates the number of problem instances for which DSHARP performed better ('+') or worse ('-'), over all instances that both systems solved (identical results are not included). Bold text indicates where DSHARP outperformed C2D, and $\star$ indicates the MI values that are statistically significant at $p \leq 0.01$. Other than the '+ / -' column, all runtime calculations were performed using all problems in the domain where at least one system found a solution. A failed run was assigned the time-out value of 1800 seconds (a lower bound on the true run time).

TukeyHSD results is shown in Table 2. For the runtime data, we included all problem instances solved by at least one setting. Where a setting was unable to find a solution within the resource limits, we used the time-out (1800 seconds) as its run time. This run time is a lower-bound on the true run time. For the output size data, we only included instances solved by all parameter settings as we do not have a reasonable lower-bound on output size for the unsolved instances. Only those domains where DSHARP was able to solve more than half of the problems are shown.

In terms of run time, CA and IBCP appear most often as significant factors for low run times. These results do not come as a surprise given their contribution to improving the speed of #SAT solving (Thurley 2006). CC sometimes helps (e.g., in grid, empr, and overall). The other components appear to have little significant impact.

For the compiled d-DNNF size, CC is the prevailing factor with significance over most problem instances. This effect is reasonable since DSHARP reuses parts of the d-DNNF when CC is able to recognize repeated sub-problems. It is interesting to note that on the uf instances, CA (alone or in combination) resulted in larger output. We do not yet have an understanding of this result though it raises the important point of a possible trade-off between compilation speed and output size given the positive impact CA had on run time.

We should also note that the dynamic decomposition em-

statistical package (R Development Core Team 2006).

Domain	Sig. (Time)	Sig. (Size)
uf (100,27)	CA+ IBCP+ CA+:IBCP+	CA- CC+ CA-:CC+
bw (7,5)	CA+	PP-
flat (100,7)	CA+ IBCP+ CA+:IBCP+	CC+
grid (33,27)	CC+	CC+
sortnet (12,10)	PP-	
empr (31,5)	CC+	CC+
all (283,81)	CA+ CC+ IBCP+ CA+:IBCP+	CC+

Table 2: Results of the ANOVA and TukeyHSD tests for each parameter of DSHARP. All significant ($p \leq 0.01$) parameters and interactions in terms of either run time or d-DNNF representation size are shown. For each parameter we indicate whether using it was advantageous ('+') or not using it was advantageous ('-'). The "Domain" column includes (x,y) where x is the number of instances solved by at least one setting and y is the number solved by all settings.

ployed by DSHARP empowers the component caching that takes place during compilation. Without this decomposition scheme, component caching would be far less effective since the number of components available is greatly reduced — DSHARP would effectively be recording large disjoint theories as a single component.

5 Conclusions

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. This in turn is increasing the demand for CNF $\rightarrow$ d-DNNF compilers to be fast while continuing to produce high quality representations. In this paper 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 #SAT solver, sharpSAT. Our system, DSHARP, exploits the DPLL trace constructed for model counting to instead construct a d-DNNF representation of the boolean theory. DSHARP exploits the latest advances in #SAT technology, most notably dynamic decomposition and IBCP, but also conflict analysis, NCB, component caching, and pre-processing.

We tested DSHARP on 300 problems in eight domains taken from benchmark problem sets in SAT solving and planning. DSHARP solved more problem 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. We also took a deeper look at DSHARP's behaviour, performing an Analysis Of Variance to try to identify components of the DSHARP system that contributed most significantly to its performance. We found that conflict analysis and implicit binary constraint propagation appeared most frequently as contributing significantly to reducing the run time, and that component caching was vital in generating d-DNNF representations of a reasonable size. We were unable to evaluate the impact of dynamic decomposition because it could not be disabled. Nevertheless, we conjecture that it plays a significant role in improving performance, particularly in concert with component caching, as it has done with sharpSAT. In future work, we plan to experiment with further optimizations of our compiler and with its use in more diverse AI applications.

6 Acknowledgements

The authors gratefully acknowledge funding from the Ontario Ministry of Innovation and the Natural Sciences and Engineering Research Council of Canada (NSERC). We would also like to thank the anonymous referees for useful feedback on earlier drafts of the paper.

References

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.

Cohen, P. R. 1995. *Empirical Methods for Artificial Intelligence*. The MIT Press, Cambridge, Mass.

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*.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the ACM* 5(7):394–397.

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

Palacios, H., and Geffner, H. 2006. Mapping conformant planning into SAT through compilation and projection. *Lecture Notes in Computer Science* 4177:311–320.

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.

R Development Core Team. 2006. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Ranjan, R. K.; Aziz, A.; Brayton, R. K.; Plessier, B.; and Pixley, C. 1995. Efficient BDD algorithms for FSM synthesis and verification. *International Workshop on Logic Synthesis*.

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*.

Appendix

C2D Settings				Run time (sec.)			d-DNNF Size (edges)		
Reduced	Smooth	DT Method	Solved	MI	MRI	+ / -	MI	MRI	+ / -
no	no	4	275	**123.93**⋆	**27.73**	**263 / 11**	**161065.71**	**5.25**	**180 / 94**
no	-smooth	4	275	**124.45**⋆	**27.77**	**261 / 13**	**161065.71**	**5.25**	**180 / 94**
no	-smooth_all	4	275	**124.66**⋆	**27.77**	**264 / 10**	**161065.71**	**5.25**	**180 / 94**
yes	no	4	275	**124.67**⋆	**27.83**	**261 / 11**	**161065.71**	**5.25**	**180 / 94**
yes	-smooth_all	4	275	**124.70**⋆	**27.83**	**263 / 12**	**161065.71**	**5.25**	**180 / 94**
yes	-smooth	4	275	**124.73**⋆	**28.55**	**267 / 8**	**161065.71**	**5.25**	**180 / 94**
no	no	3	275	**202.59**⋆	**55.78**	**255 / 17**	**218277.51**⋆	**9.89**	**193 / 81**
yes	-smooth_all	3	275	**202.84**⋆	**56.12**	**258 / 17**	**218277.51**⋆	**9.89**	**193 / 81**
no	-smooth	3	275	**202.91**⋆	**56.17**	**260 / 15**	**218277.51**⋆	**9.89**	**193 / 81**
yes	no	3	275	**203.00**⋆	**56.01**	**260 / 13**	**218277.51**⋆	**9.89**	**193 / 81**
yes	-smooth	3	275	**203.01**⋆	**56.38**	**264 / 10**	**218277.51**⋆	**9.89**	**193 / 81**
no	-smooth_all	3	275	**203.14**⋆	**56.03**	**255 / 17**	**218277.51**⋆	**9.89**	**193 / 81**
no	no	1	269	**116.06**⋆	**58.16**	**263 / 6**	**22609.21**	**1.83**	**143 / 123**
yes	-smooth	0	269	**118.96**⋆	**73.78**	**265 / 4**	-4114.50	**1.30**	**149 / 118**
no	-smooth_all	1	268	**119.35**⋆	**63.46**	**263 / 4**	**6818.32**	**1.90**	**144 / 121**
no	-smooth	0	268	**121.74**⋆	**78.76**	**262 / 6**	**21295.75**	**2.96**	**135 / 130**
no	-smooth	1	268	**123.02**⋆	**79.46**	**263 / 5**	-3655.66	**1.39**	**142 / 124**
no	no	0	268	**127.16**⋆	**89.87**	**264 / 4**	**24688.42**	**1.34**	**134 / 131**
yes	no	0	268	**129.17**⋆	**87.48**	**264 / 4**	**41385.28**	**1.46**	**137 / 129**
yes	no	1	267	**123.31**⋆	**59.56**	**261 / 6**	**11253.97**	**1.97**	**147 / 117**
yes	-smooth	1	267	**126.37**⋆	**77.90**	**262 / 5**	-2894.02	**1.35**	**145 / 120**
yes	-smooth_all	0	267	**127.49**⋆	**92.81**	**263 / 4**	**41530.84**	**1.45**	**139 / 125**
yes	-smooth_all	1	267	**132.15**⋆	**86.59**	**262 / 5**	-1790.32	**1.35**	**140 / 124**
no	-smooth_all	0	266	**132.97**⋆	**98.88**	**262 / 4**	-863.08	**1.41**	**135 / 127**
no	no	2	242	**401.27**⋆	**609.56**	**234 / 6**	**552529.57**⋆	**73.26**	**225 / 16**
no	-smooth_all	2	242	**401.86**⋆	**613.84**	**238 / 4**	**552529.57**⋆	**73.26**	**225 / 16**
yes	no	2	242	**401.94**⋆	**613.19**	**238 / 4**	**552529.57**⋆	**73.26**	**225 / 16**
no	-smooth	2	242	**401.99**⋆	**612.80**	**238 / 4**	**552529.57**⋆	**73.26**	**225 / 16**
yes	-smooth_all	2	242	**402.00**⋆	**612.91**	**238 / 4**	**552529.57**⋆	**73.26**	**225 / 16**
yes	-smooth	2	242	**402.13**⋆	**614.52**	**237 / 5**	**552529.57**⋆	**73.26**	**225 / 16**

Here we present results for many of the C2D settings: **Reduce** indicates if the -reduce option was used; **Smooth** indicates if -smooth or -smooth_all (or neither) was used; **DT Method** indicates what decomposition tree method was used (via -dt_method). The **Solved** column indicates the number of problems (out of all 300) that C2D was able to solve. The last six columns correspond to the calculations made in the last six columns of Table 1. The results are for all domains, and thus correspond to the final row in Table 1.