

An Improved Decision-DNNF Compiler

Jean-Marie Lagniez and Pierre Marquis

CRIL, U. Artois & CNRS, F-62300 Lens, France,

e-mail: {lagniez, marquis}@cril.univ-artois.fr

Abstract

We present and evaluate a new compiler, called D4, targeting the Decision-DNNF language. As the state-of-the-art compilers C2D and Dsharp targeting the same language, D4 is a top-down tree-search algorithm exploring the space of propositional interpretations. D4 is based on the same ingredients as those considered in C2D and Dsharp (mainly, disjoint component analysis, conflict analysis and non-chronological backtracking, component caching). D4 takes advantage of a dynamic decomposition approach based on hypergraph partitioning, used sparingly. Some simplification rules are also used to minimize the time spent in the partitioning steps and to promote the quality of the decompositions. Experiments show that the compilation times and the sizes of the Decision-DNNF representations computed by D4 are in many cases significantly lower than the ones obtained by C2D and Dsharp.

1 Introduction

Knowledge compilation (KC) is acknowledged as a challenging approach for circumventing the intractability of many practical reasoning problems based on propositional representations (see [Cadoli and Donini, 1998; Darwiche, 2014; Marquis, 2015] for surveys). Since its very beginning, research on KC has been developed following a number of directions, including the identification of the compilability of some problems, the definition and the study of new target languages for KC, the design and the evaluation of compilers, and the use of the KC languages, compilers and reasoners in various applications.

As the previous works [Darwiche, 2004; Muise *et al.*, 2012; Oztok and Darwiche, 2014; 2015a], we are concerned with the third research direction in this paper: our objective is to describe a new compiler, called D4.¹ D4 is a top-down compiler associating with every input CNF formula an equivalent Decision-DNNF representation. Like the state-of-the-art compilers C2D and Dsharp targeting the same language, D4

is a tree-search algorithm exploring the whole space of propositional interpretations. Accordingly, D4 takes advantage of the techniques used in C2D and Dsharp for efficiency reasons (mainly, disjoint component analysis, conflict analysis and non-chronological backtracking, component caching). However, the decomposition heuristics used for guiding the exploration of the search space in D4 differs from the ones considered in C2D and in Dsharp, while trying to keep the best of them. Thus, like Dsharp, D4 computes a *dynamic* decomposition of the input CNF formula under the current partial assignment (disjoint components are not computed prior to the search as with C2D). However, like C2D, D4 partitions the dual hypergraph associated with the input CNF formula under the current partial assignment to find a decomposition (while the branching heuristics of Dsharp is based on the VSADS score of the variables). What is brand new in D4 is that the decomposition is achieved in a parsimonious way (hypergraph partitioning is not used at each decision step) and that some simplification rules are used to minimize the time spent in the partitioning steps and to promote the quality of the resulting decompositions. In order to assess D4 and to compare it with C2D and Dsharp, we performed experiments on many benchmarks, coming from several families. The results obtained clearly show the benefits offered by D4.

The rest of the paper is organized as follows. After some formal preliminaries (Section 2), our compiler D4 is presented in Section 3. An empirical evaluation is provided in Section 4, before the concluding section (Section 5). The runtime code of D4, the benchmarks used, and some additional empirical results are available on line from <https://www.cril.univ-artois.fr/KC/>.

2 Formal Preliminaries

We consider a propositional language $PROP_{PS}$ defined from a finite set PS of propositional symbols and the standard connectives. $PROP_{PS}$ is interpreted in a classical way. For every formula Σ in $PROP_{PS}$, $Var(\Sigma)$ is the set of propositional variables occurring in Σ . A CNF formula Σ is a conjunction of clauses, where a clause is a disjunction of literals. Every CNF is viewed as a set of clauses, and every clause is viewed as a set of literals. For every literal ℓ , $var(\ell)$ denotes the variable x of ℓ ($var(x) = x$ and $var(\neg x) = x$), and $\sim\ell$ denotes the complementary literal of ℓ (i.e., for every variable x , $\sim x = \neg x$ and $\sim\neg x = x$). The conditioning of a CNF for-

¹D4: a Decision-DNNF compiler based on Dynamic Decomposition.

mula Σ by a literal $\ell = x$ (resp. $\ell = \neg x$) is the CNF formula $\Sigma \mid \ell$ obtained by removing from Σ every clause containing x (resp. $\neg x$) and removing from the remaining clauses every occurrence of $\neg x$ (resp. x).

The *dual hypergraph* of a CNF formula Σ is $DH(\Sigma) = (N_d, H_d)$ where $N_d = \Sigma$ and $H_d = \{\{\delta \in \Sigma \mid x \in \text{Var}(\delta)\} \mid x \in \text{Var}(\Sigma)\}$. The nodes of N_d correspond to the clauses of Σ and the hyperedges of H_d are labelled by sets of variables of Σ (for each variable x , the corresponding hyperedge is the set of clauses $\text{Cls}_\Sigma(x)$ of Σ containing x).

Example 1 Let $\Sigma = (a \vee b) \wedge (a \vee \neg c) \wedge (a \vee \neg d) \wedge (b \vee \neg c) \wedge (b \vee \neg d)$. We have $DH(\Sigma) = (N_d, H_d)$, with $N_d = \{n_1, n_2, n_3, n_4, n_5\}$ (n_1 (resp. n_2, n_3, n_4, n_5) corresponds to $a \vee b$ (resp. $a \vee \neg c, a \vee \neg d, b \vee \neg c, b \vee \neg d$) and $H_d = \{\{n_1, n_2, n_3\}, \{n_1, n_4, n_5\}, \{n_2, n_4\}, \{n_3, n_5\}\}$ (labelled respectively by $\{a\}$, $\{b\}$, $\{c\}$, and $\{d\}$).

BCP denotes a Boolean Constraint Propagator [Zhang and Stickel, 1996; Moskewicz *et al.*, 2001], which is a key component of many solvers. $\text{BCP}(\Sigma)$ returns $\{\emptyset\}$ if there exists a unit refutation from the clauses of the CNF formula Σ , and it returns the set of literals (unit clauses) which are derived from Σ using unit propagation in the remaining case. As a side effect, BCP “virtually” simplifies Σ by conditioning it by the unit clauses which are generated. For efficiency reasons, such a simplification is not “physically” performed on the CNF (instead the set of literals derived using unit propagation is maintained).

d-DNNF is the language consisting of the Boolean circuits with a single output (its root), where each input is a literal or a Boolean constant, and each internal gate is either a decomposable \wedge gate of the form $N = \wedge(N_1, \dots, N_k)$ (“decomposable” means here that for each $i, j \in \{1, \dots, k\}$ with $i \neq j$ the subcircuits of N rooted at N_i and N_j do not share any common variable) or deterministic \vee gate of the form $N = \vee(N_1, \dots, N_k)$ (“deterministic” means here that for each $i, j \in \{1, \dots, k\}$ with $i \neq j$ the subcircuits of N rooted at N_i and N_j are jointly inconsistent).

Decision-DNNF is defined in the same way, except that deterministic \vee gates are replaced by decision gates of the form $N = \text{ite}(x, N_1, N_2)$. x is the decision variable at gate N , it does not occur in the subcircuits N_1, N_2 , and ite is a ternary connective whose semantics is given by $\text{ite}(X, Y, Z) = (\neg X \wedge Y) \vee (X \wedge Z)$ (“ite” means “if ... then ... else ...: if X then Z else Y ”).

Example 2 (Example 1 cont’ed) The Decision-DNNF representations given on Figure 1 are equivalent to the CNF formula Σ considered in Example 1 (the nodes labelled by \wedge are decomposable \wedge nodes and the circled nodes are decision nodes).

Decision-DNNF representations, also known as decomposable decision graphs [Fargier and Marquis, 2006], can be turned in linear time into specific d-DNNF representations. Indeed, when one replaces in a Decision-DNNF representation a decision node of the form $N = \text{ite}(x, N_1, N_2)$ by $N = (\neg x \wedge N_1) \vee (x \wedge N_2)$, the two \wedge nodes which are conveyed by the replacement are decomposable ones (x appears neither in N_1 nor in N_2) and the \vee node is deterministic ($(\neg x \wedge N_1) \wedge (x \wedge N_2)$ is inconsistent).

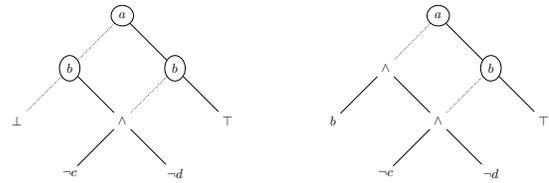


Figure 1: Two Decision-DNNF representations.

3 An Improved Decision-DNNF Compiler: D4

Our top-down compiler D4 associates with every input CNF formula Σ an equivalent Decision-DNNF representation. It elaborates over C2D and Dsharp.

In C2D (reasoning.cs.ucla.edu/c2d/) [Darwiche, 2001; 2004], the generation of the decision nodes in the resulting Decision-DNNF representation is guided by a decomposition tree (dtree) of the input CNF instance Σ , which is computed first. Both the time required to generate a Decision-DNNF representation of Σ and the size of the representation are linear in the size of Σ , yet exponential in the width of the dtree which is used.

In Dsharp (www.haz.ca/research/dsharp/) [Muise *et al.*, 2012], the generation of the decision nodes is not guided by a decomposition tree, but a dynamic decomposition approach is considered instead. During the search, the variable selected by the branching heuristics is one with a maximal Variable State Aware Decaying Sum (VSADS) score [Sang *et al.*, 2005], as in the (exact) model counters sharpSAT (sites.google.com/site/marcthurley/sharpsat) [Thurley, 2006] and Cachet (www.cs.rochester.edu/~kautz/Cachet/index.htm) [Sang *et al.*, 2004]. Implicit BCP is used at every decision point in order to find some failed literals, i.e., literals which are falsified in every model of the formula corresponding to the decision point. The literals which belong to clauses that became binary in the most recent call to BCP are selected as candidates for failed literals. A conflict clause is learned for each failed literal found. Dsharp also incorporates a restricted form of backbone detection, based on BCP.

As it is the case for the previous compilers C2D and Dsharp [Huang and Darwiche, 2007; Muise *et al.*, 2012], the Decision-DNNF representation of Σ which is computed by D4 corresponds to the trace of a solver. We used our own solver *solve*, which is based on the state-of-the-art SAT solver MiniSAT 2.2 [Eén and Sörensson, 2003]. *solve* is called at each decision point, and if a model is found, instead of stopping, the search backtracks up to the level of the last decision node which has been created. Assumptions are exploited in *solve*, so that the clauses which are learnt at each call to *solve* can be kept for the subsequent calls. D4 exploits the same techniques as the ones used in C2D and Dsharp for efficiency reasons (mainly, disjoint component analysis, conflict analysis and non-chronological backtracking, component caching²). However, the decomposition heuristics used by

²In our cache implementation, following the approach already

D4 for guiding the exploration of the search space is different from the ones considered in C2D and Dsharp.

The pseudo-code of D4. Algorithm ?? provides the pseudo-code of the compiler D4. The compilation of a given CNF formula Σ is achieved by calling $D4(\Sigma, \emptyset)$.

Algorithm 1: $D4(\Sigma, LV)$

```

input  : a CNF formula  $\Sigma$ 
input  : a list of variables  $LV$ 
output : the root node  $N$  of a Decision-DNNF
        representation of  $\Sigma$ 

1  $S \leftarrow \text{solve}(\Sigma)$ ;
2 if  $S = \{\emptyset\}$  then return  $\text{leaf}(\perp)$ ;
3 if  $\text{Var}(\Sigma) = \emptyset$  then return  $\text{aNode}(S, [\text{leaf}(\top)])$ ;
4 if  $\text{cache}(\Sigma) \neq \text{nil}$  then return  $\text{aNode}(S, [\text{cache}(\Sigma)])$ ;
5  $\text{comps} \leftarrow \text{connectedComponents}(\Sigma)$ ;
6  $LN_d \leftarrow []$ ;
7 foreach  $c \in \text{comps}$  do
8    $LV_c \leftarrow \text{restrict}(LV, \text{Var}(c))$ ;
9   if  $LV_c = \emptyset$  or
       $\#(\text{Var}(S) \cap \text{Var}(c)) > \frac{1}{10} \#(\text{Var}(c))$  then
       $LV_c \leftarrow \text{sort}(\text{HGP}(c))$ ;
10   $v \leftarrow \text{head}(LV_c)$ ;
11   $LV_c \leftarrow \text{tail}(LV_c)$ ;
12   $N_d \leftarrow \text{ite}(v, D4(c|\neg v, LV_c), D4(c|v, LV_c))$ ;
13   $LN_d \leftarrow \text{add}(N_d, LN_d)$ ;
14  $N_\wedge \leftarrow \text{aNode}(S, LN_d)$ ;
15  $\text{cache}(\Sigma) \leftarrow N_\wedge$ ;
16 return  $N_\wedge$ ;
```

At line 1, `solve` is called on Σ . S is equal to $\{\emptyset\}$ if Σ is inconsistent, and in this case a Decision-DNNF representation of it ($\text{leaf}(\perp)$) is generated at line 2. Otherwise S is equal to $\text{BCP}(\Sigma)$ (please keep in mind that BCP simplifies Σ). The second base case is addressed at line 3: if Σ (once simplified by BCP) contains no variable, then a \wedge node is generated; $\text{aNode}(\{\ell_1, \dots, \ell_k\}, [N_1, \dots, N_m])$ generates a \wedge node with $k + m$ children: the k literals from the set $\{\ell_1, \dots, \ell_k\}$ and the m nodes from the list $[N_1, \dots, N_m]$. Hence $\text{aNode}(S, [\text{leaf}(\top)])$ returns a \wedge node gathering all the literals of S (completed by a \top leaf for handling the case S is empty). At line 4, one first determines whether the current formula Σ has already been encountered or not during the search. One takes advantage of the cache function which associates CNF formulae with Decision-DNNF representations given by their root nodes. If Σ has already been found, then the algorithm simply returns the root node of the corresponding stored Decision-DNNF representation. At line 5 `connectedComponents` returns a set comps of CNF formulae, the connected components of Σ . The loop at line 7 considers every element c of comps successively. For a given c , the variables of LV are restricted to those in c . One then tests at line 9 whether the resulting list LV_c is empty or

used in our MDDG compiler [Koriche *et al.*, 2015], we do not store in the current bucket the clauses which have not been shortened.

not, and if the number of literals obtained by applying BCP on Σ when projected onto the set of variables of the current component c is large enough (at least 10% of the number of variables of c). If at least one of the two conditions is satisfied, then one updates the current list LV_c of variables. $\text{HGP}(c)$ computes a cutset of the dual hypergraph of c after applying to it some simplification rules. The variables of this cutset are sorted according to their VSADS score (the largest score first) to produce an updated list $\text{sort}(\text{HGP}(c))$. Then one selects the first variable v of LV_c (line 10) and remove it from LV_c (line 11). A decision node N_d with decision variable v is computed next, via two recursive calls to D4 corresponding to the two ways of conditioning v in c (line 12). N_d is then added to the list LN_d of nodes at line 13, which has been initialized to the empty list before the loop (line 6). Then a \wedge node N_\wedge gathering the literals of S and the decision nodes of LN_d is generated (line 14), and added to the cache (associated with the entry Σ) (line 15), and finally returned as the result of the main call (line 16).

Dynamic decomposition based on hypergraph partitioning. Like Dsharp, D4 computes a *dynamic decomposition* of the input formula under the current partial assignment (disjoint components are computed during the search and not prior to the search). However, like C2D -dt_method 1, D4 partitions the dual hypergraph associated with the current formula to find a decomposition. Basically, the HGP procedure takes advantage of the PaToH partitioner – Partitioning Tools for Hypergraph, v. 3.2 (<http://bmi.osu.edu/~umit/software.html>) [Catalyürek and Aykanat, 2011] to do the job. PaToH is similar to hMETIS (the partitioner exploited by C2D) but, unlike hMETIS, it runs on the 64 bit architecture used for our experiments. Given the dual hypergraph $DH(\Sigma) = (N_d, H_d)$ of Σ , PaToH roughly looks for a subset C of H_d containing as few elements as possible such that removing the cutset C from H_d leads to a hypergraph containing at least two disjoint components having sizes as close as possible (see [Catalyürek and Aykanat, 2011] for more details). When the variables corresponding to the elements of C are assigned (whatever the way they are assigned) it is guaranteed that the current formula conditioned by the corresponding assignment has at least two disjoint components, so that a decomposable \wedge node can be generated in the compiled form.

When C2D -dt_method 1 is used, a *dtree* is computed entirely at start using hypergraph partitioning: a first cutset C of the dual hypergraph (N_d, H_d) of Σ is computed, and then one recursively looks for a decomposition of each of the two hypergraphs (the resulting connected components) obtained from the removal of C in (N_d, H_d) , until a hypergraph with a single node is obtained. Contrastingly, the decomposition process followed by D4 is dynamic: for each assignment of the variables of corresponding to the hyperedges of C , the hypergraph associated with Σ conditioned by the assignment (and possibly simplified further thanks to BCP) is a candidate for a further decomposition. The variables of C are ordered according to their VSADS score (the ones with a maximal score being considered first as in Dsharp). That way, D4 tries to keep the best of the decomposition heuristics used by

the two compilers C2D and Dsharp.

Each of the static and dynamic decomposition approaches has some pros and some cons. On the one hand, computing a static decomposition as in C2D -dt_method 1 limits the number of calls to the hypergraph partitioner since there is no need to compute a cutset for each assignment. This property is not shared when a dynamic decomposition approach is considered, as in D4. However, limiting the number of calls to the hypergraph partitioner is important since hypergraph partitioning is time consuming. Especially, profiling the code of D4, we observed that in average for the instances considered in our experiments at least 50% of the computation time of D4 is consumed by PaToH (and for some instances the part of the time consumed by PaToH exceeds 90%). On the other hand, the static decomposition approach does not lead in general to the most efficient decompositions; indeed, for each cutset C , the chosen assignment of the variables corresponding to the hyperedges of C (and the additional propagations which can be made from it) can have a huge impact both on the size and the structure of the resulting formula once simplified, so that a different, and actually much better, decomposition can be reached for it, compared to the ones associated with the other assignments of the variables of C .

Example 3 (Example 1 cont'ed) $C = \{\{n_1, n_2, n_3\}, \{n_1, n_4, n_5\}\}$ is a cutset of $DH(\Sigma)$: removing the elements of C from H_d leads to two disjoint hypergraphs since the remaining hyperedges $\{n_2, n_4\}$, $\{n_3, n_5\}$ do not share any node. $\{n_1, n_2, n_3\}$ (resp. $\{n_1, n_4, n_5\}$) are labelled by the sets of variables $\{a\}$ (resp. $\{b\}$), thus assigning those two variables a and b in Σ (whatever the assignment) leads to two disjoint components (one of them consists of the clause $b \vee \neg c$ simplified by the assignment and the other one of the clause $b \vee \neg d$ simplified by the assignment). This can be observed on Figure 1 (left). However, one can also observe on Figure 1 (right) that assigning all the variables corresponding to the hyperedges of C is not necessary to generate disjoint components: setting a to false is enough to get some disjoint components, independently of the way b will be assigned. Thus considering b (which corresponds to the remaining hyperedge of C) as the next branching variable when a has been set to false is not guaranteed to be the best choice. For each connected component resulting from the assignment of a to false, some new decompositions should be looked for.

Improving the hypergraph partitioning steps. In order to circumvent the complexity of the hypergraph partitioning steps, the strategy used in D4 is twofold. On the one hand, we avoid calling HGP at each recursion step or each time a decision node must be generated. Thus, a new partition is computed only if the condition at line 9 is satisfied, i.e., when the current list of variables LV_c is empty, or when the number of propagations obtained via BCP (projected onto the set of variables of the current component c) is "large enough". On the other hand, we designed some specific rules which are used inside HGP and aim at simplifying the hypergraph (both in terms of the number of its nodes and in terms of the number of its hyperedges) associated with the current formula c before calling PaToH on it. The simplification achieved can also lead PaToH to find better decompositions. It guar-

antees that the CNF formula $BCP(c \mid \gamma_c)$ contains at least two disjoint components when γ_c is any total assignment of the variables from the list LV_c computed at line 9. One first exploits an algorithm for the detection of literal equivalences based on BCP. This algorithm, which is a by-product of the algorithm equivSimpl reported in [Lagniez and Marquis, 2014], is used for determining for each literal ℓ appearing in the current formula c a list of equivalent literals (its equivalence class). Those lists are generated by computing for each ℓ , $BCP(c \cup \{\ell\})$ and $BCP(c \cup \{\sim \ell\})$.³ Once this has been done, for each $var(\ell)$, the variables $v_1^\ell, \dots, v_k^\ell$ associated with the literals of the equivalence class of ℓ (but $var(\ell)$ itself) are suppressed from the labels of the hyperedges of $DH(c)$, the hyperedges without any label are then removed from this set, and the hyperedge $Cls_c(var(\ell))$ is replaced by $Cls_c(var(\ell)) \cup \bigcup_{i=1}^k Cls_c(v_i^\ell)$. When all $var(\ell)$ have been considered, two rules are applied on the resulting hypergraph in a systematic fashion in order to simplify it further. The simplification process stops when no rule can be applied or when the current hypergraph has a single hyperedge. The first rule concerns the nodes of this hypergraph. Suppose that the current set of nodes contains two distinct nodes n_i, n_j such that every hyperedge of the current set of hyperedges containing n_i also contains n_j . Then n_i can be safely removed from the current set of nodes, and from the hyperedges which contain it. The labels of the hyperedges containing n_i are marked and added to the labels of the hyperedges containing n_j . The second rule concerns the unmarked labels of hyperedges of size 1 which can obviously be removed.

Example 4 Let $\Sigma = (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee a) \wedge (a \vee c \vee d)$. We have $DH(\Sigma) = (N_d, H_d)$, with $N_d = \{n_1, n_2, n_3, n_4\}$ (n_1 (resp. n_2, n_3, n_4) corresponds to $\neg a \vee b$ (resp. $\neg b \vee c, \neg c \vee a, a \vee c \vee d$) and $H_d = \{\{n_1, n_3, n_4\}, \{n_1, n_2\}, \{n_2, n_3, n_4\}, \{n_4\}\}$ (labelled respectively by $\{a\}$, $\{b\}$, $\{c\}$, and $\{d\}$). The first step leads to identify the equivalence class of a as $\{a, b, c\}$. The resulting hypergraph is $(\{n_1, n_2, n_3, n_4\}, \{\{n_1, n_2, n_3, n_4\}, \{n_4\}\})$. The set of labels of $\{n_1, n_2, n_3, n_4\}$ is $\{a\}$ and the set of labels of $\{n_4\}$ is $\{d\}$. Then the nodes n_1, n_2, n_3 are removed, a is marked and added to the set of labels of $\{n_4\}$ which becomes $\{a, d\}$. Finally, label d is removed from this set. The resulting hypergraph is thus $(\{n_4\}, \{\{n_4\}\})$ where $\{n_4\}$ is labelled by $\{a\}$.

4 Empirical Evaluation

We have considered 703 CNF instances from the SAT LIBrary www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html, gathered into 8 data sets, as follows: BN (Bayesian networks) (192), BMC (Bounded Model Checking) (18), Circuit (41), Configuration (35), Handmade (58), Planning (248), Random (104), Qif (7) (Quantitative Information Flow

³Running this algorithm is also useful for determining that some literals are fixed in c , while they are not necessarily detected as such when running BCP on c . Thus, if $BCP(c \cup \{\ell\}) = \{\emptyset\}$, then we have $c \models \neg \ell$; if $\ell' \in BCP(c \cup \{\ell\})$ and $\ell' \in BCP(c \cup \{\sim \ell\})$, then $c \models \ell'$. In D4, such literals are used to simplify c for the subsequent computations.

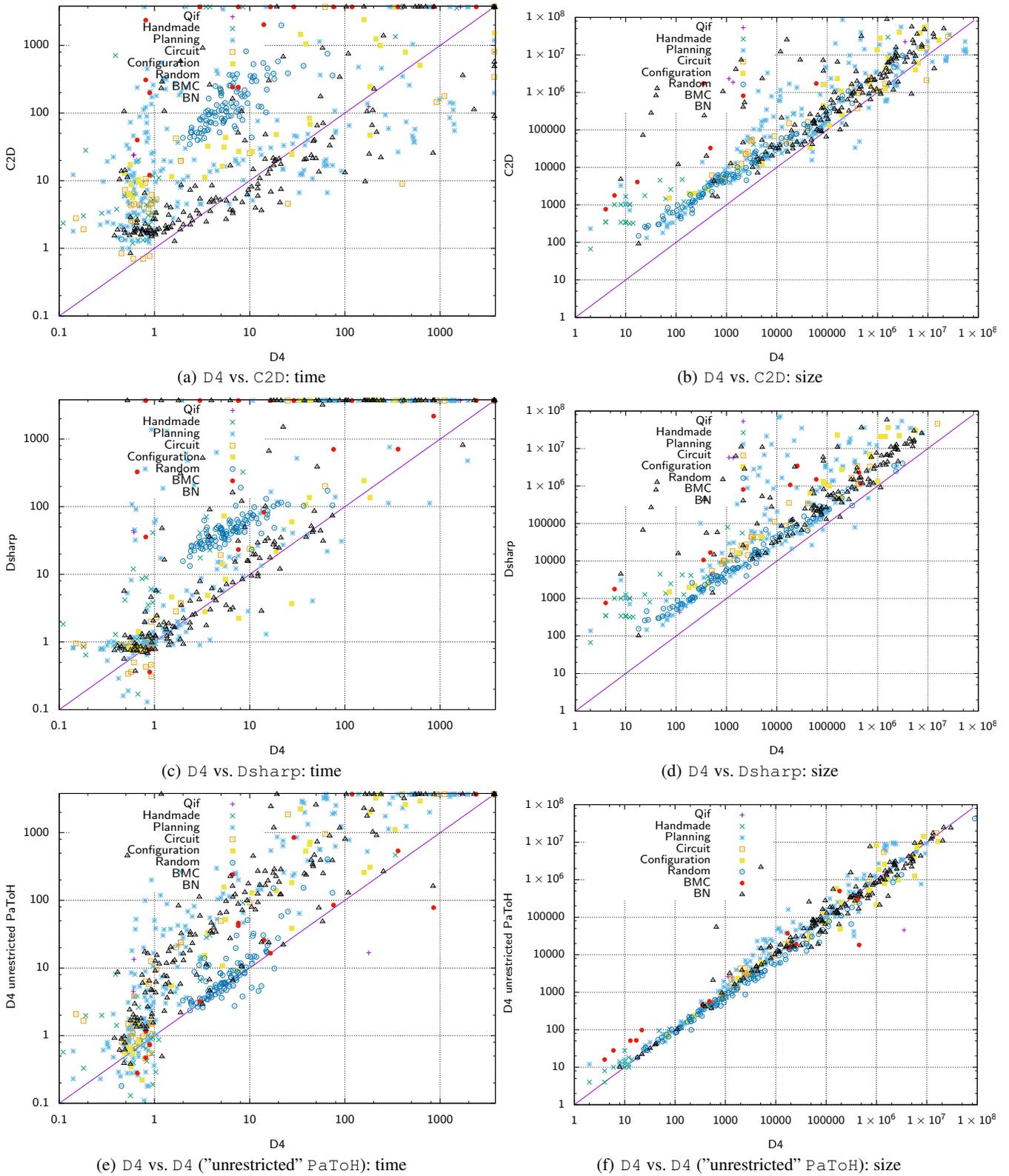


Figure 2: Comparing D4 with C2D, Dsharp, and a version of D4 where PaToH is applied at each decision node.

analysis - security). The experiments have been conducted on Intel Xeon E5-2643 (3.30 GHz) processors with 32 GiB RAM on Linux CentOS. A time-out of 1h and a memory-out of 7.6 GiB has been considered for each instance.

We have computed the compilation times and the sizes of the corresponding Decision-DNNF compiled forms obtained by using D4, C2D `-dt_method 1`, and Dsharp, on each instance. The results are reported on the scatter plots given in Figure 2. Each dot represents an instance; the time (in seconds) needed to solve it or the size (in number of edges) of the resulting compiled form, using the compiler corresponding to the x -axis (resp. y -axis), is given by its x -coordinate (resp. y -coordinate). Logarithmic scales are used for both coordinates. In part (a) and (b) of the figure, the x -axis corresponds to D4 while the y -axis corresponds to C2D. In part (a) compilation times are reported while in part (b), sizes of compiled forms are reported. Parts (c) and (d) present the corresponding results when Dsharp is used instead of C2D. In (d) the number of edges reported for Dsharp is the number of edges in the compressed representations.⁴ The numbers of instances solved within the time and memory limits are 574 (over 703) for D4, 546 for C2D `-dt_method 1`, and 467 for Dsharp.

The experimental results obtained show that the compilation times of D4 are significantly lower than the ones obtained by C2D and Dsharp on many instances. More importantly, the sizes of the Decision-DNNF representations computed by D4 are substantially lower for the great majority of instances (sometimes by several orders of magnitude) than the ones obtained by C2D and Dsharp.

An important issue was to determine the very reasons of the efficiency of D4 compared to C2D and Dsharp. Indeed, beyond its specific decomposition heuristics, D4 is based on a more recent SAT solver than the ones considered in those two compilers. Does this explain the computational benefits achieved by D4? In order to answer this question, we developed a Dsharp-like compiler based on D4, and we evaluated this compiler on the same benchmarks set and considering the same resource bounds as above. It turns out that the Dsharp-like compiler has been able to solve 515 instances, which is much more than Dsharp, but far less than the 574 instances solved by D4. So it clearly appears that the gain offered by D4 is not solely due to the use of a solver based on MiniSAT (even if this helps). Assessing the impact of using PaToH (with default setting, as used in D4) instead of hMETIS was also an issue. Interestingly, a comparison of PaToH and hMETIS has already been done, see <http://bmi.osu.edu/umit/PaToH/table1.html>. It turns out PaToH and hMETIS have similar performances, especially as to the quality of the decompositions found. Similarly, in order to assess the improvements obtained by limiting the time consumed by PaToH by using it sparingly and after the application of simplification rules (as implemented in D4), we have also compared D4 with a version of it where PaToH is called in an unrestricted fashion (i.e., when the condition at line 9

⁴Contrastingly, no reduction rules are applied to the Decision-DNNF representations computed by D4, so that the sizes which are reported correspond to uncompressed representations.

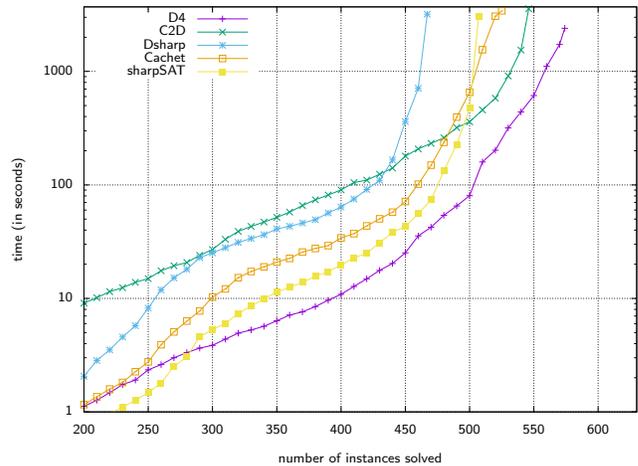


Figure 3: Number of instances solved by D4, C2D, Dsharp, Cachet, and sharpSAT in a given time.

is true) and no simplification rules are applied before calling PaToH. This version of D4 with "unrestricted" use of PaToH solved "only" 531 instances, and does not lead to size improvements. More detailed results are reported in parts (e) and (f) of Figure 2.

Finally, we wanted to assess the performances of D4 as a model counter, and to compare it with miniC2D, Cachet and sharpSAT on the same benchmarks set and with the same resource bounds. miniC2D (<http://reasoning.cs.ucla.edu/minic2d/>) [Oztok and Darwiche, 2015b] is a compiler targeting the language Decision-SDD. While it is based on a more recent SAT solver than the ones used in C2D and Dsharp, it has been able to solve only 414 instances (thus, far less than C2D). Cachet (resp. sharpSAT) solved 525 (resp. 507) instances (over 703) within the time and memory limits. The cactus plots in Figure 3 give for D4, C2D, Dsharp, Cachet, and sharpSAT the number of instances solved in a given time. The results show that while D4 (as a compiler) requires some computational overhead for generating compiled forms, it often proves quite efficient as an exact model counter.

5 Conclusion

We have described D4, a new top-down, tree-search compiler targeting the Decision-DNNF language. In D4, a dynamic decomposition of the input CNF formula under the current partial assignment is computed during the search. This decomposition is based on hypergraph partitioning, used sparingly. Specific hypergraph simplification rules are also used in order to minimize the time spent in each partitioning step and to promote the quality of the decompositions. Experiments have shown that substantial computational savings can be obtained using D4, both in terms of compilation times and (even more) in terms of the sizes of the compiled forms which are generated. D4 also appears as a quite competitive model counter for many instances (despite the computational overhead due to the generation of the compiled form).

References

- [Cadoli and Donini, 1998] Marco Cadoli and Francesco M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3-4):137–150, 1998.
- [Catalyürek and Aykanat, 2011] Umit V. Catalyürek and Cevdet Aykanat. *PaToH (Partitioning Tool for Hypergraphs)*, pages 1479–1487. Encyclopedia of Parallel Computing. 2011.
- [Darwiche, 2001] Adnan Darwiche. Decomposable negation normal form. *Journal of the Association for Computing Machinery*, 48(4):608–647, 2001.
- [Darwiche, 2004] Adnan Darwiche. New advances in compiling cnf into decomposable negation normal form. In *Proc. of ECAI’04*, pages 328–332, 2004.
- [Darwiche, 2014] Adnan Darwiche. *Tractable Knowledge Representation Formalisms*, pages 141–172. Tractability – Practical Approaches to Hard Problems. Cambridge University Press, 2014.
- [Eén and Sörensson, 2003] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proc. of SAT’03*, pages 502–518, 2003.
- [Fargier and Marquis, 2006] H el ene Fargier and Pierre Marquis. On the use of partially ordered decision graphs in knowledge compilation and quantified Boolean formulae. In *Proc. of AAAI’06*, pages 42–47, 2006.
- [Huang and Darwiche, 2007] Jinbo Huang and Adnan Darwiche. The language of search. *Journal of Artificial Intelligence Research (JAIR)*, 29:191–219, 2007.
- [Koriche *et al.*, 2015] Fr ed eric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. Compiling constraint networks into multivalued decomposable decision graphs. In *Proc. of IJCAI’15*, pages 332–338, 2015.
- [Lagniez and Marquis, 2014] Jean-Marie Lagniez and Pierre Marquis. Preprocessing for propositional model counting. In *Proc. of AAAI’14*, pages 2688–2694, 2014.
- [Marquis, 2015] Pierre Marquis. Compile! In *Proc. of AAAI’15*, pages 4112–4118, 2015.
- [Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC’01*, pages 530–535, 2001.
- [Muise *et al.*, 2012] Christian J. Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proc. of AI’12*, pages 356–361, 2012.
- [Oztok and Darwiche, 2014] Umut Oztok and Adnan Darwiche. On compiling CNF into Decision-DNNF. In *Proc. of CP’14*, pages 42–57, 2014.
- [Oztok and Darwiche, 2015a] Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Proc. of IJCAI’15*, pages 3141–3148, 2015.
- [Oztok and Darwiche, 2015b] Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Proc. of IJCAI’15*, pages 3141–3148, 2015.
- [Sang *et al.*, 2004] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of SAT’04*, 2004.
- [Sang *et al.*, 2005] Tian Sang, Paul Beame, and Henry A. Kautz. Performing Bayesian inference by weighted model counting. In *Proc. of AAAI’05*, pages 475–482, 2005.
- [Thurley, 2006] Marc Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proc. of SAT’06*, pages 424–429, 2006.
- [Zhang and Stickel, 1996] Hantao Zhang and Mark E. Stickel. An efficient algorithm for unit propagation. In *Proc. of ISAIM’96*, pages 166–169, 1996.