DPLL with a Trace: From SAT to Knowledge Compilation

Jinbo Huang and Adnan Darwiche
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
{jinbo, darwiche}@cs.ucla.edu

Abstract

We show that the trace of an exhaustive DPLL search can be viewed as a compilation of the propositional theory. With different constraints imposed or lifted on the DPLL algorithm, this compilation will belong to the language of d-DNNF, FBDD, and OBDD, respectively. These languages are decreasingly succinct, yet increasingly tractable, supporting such polynomial-time queries as model counting and equivalence testing. Our contribution is thus twofold. First, we provide a uniform framework, supported by empirical evaluations, for compiling knowledge into various languages of interest. Second, we show that given a particular variant of DPLL, by identifying the language membership of its traces, one gains a fundamental understanding of the intrinsic complexity and computational power of the search algorithm itself. As interesting examples, we unveil the "hidden power" of several recent model counters, point to one of their potential limitations, and identify a key limitation of DPLL-based procedures in general.

1 Introduction

Knowledge compilation has been a key direction of research in automated reasoning [Selman and Kautz, 1991; Marquis, 1995; Selman and Kautz, 1996; Cadoli and Donini, 1997; Darwiche and Marquis, 2002]. When propositional theories are compiled into a suitable target language, some generally intractable queries may be answered in time polynomial in the size of the compilation. Compiling combinational circuits into OBDDs, for example, allows functional equivalence to be tested in polynomial time (or constant time if the same variable order is used) [Bryant, 1986]. More recent applications of compilation can be found in the fields of diagnosis and planning, involving the use of the DNNF language [Darwiche, 2001a; Barrett, 2004; Palacios *et al.*, 2005].

Propositional Satisfiability (SAT), on the other hand, has been an area of no less importance, or activity. Aside from its theoretical significance as the prototypical NP-complete problem, SAT finds practical applications in many areas of artificial intelligence and computer science at large. While SAT algorithms have substantially improved over the decades, many of them continue to build on what is known as the DPLL search [Davis *et al.*, 1962] (for examples, see complete SAT solvers in the 2004 SAT Competition: http://satlive.org/SATCompetition/2004/).

This paper sets out to demonstrate a deep connection between SAT solving and knowledge compilation. In the first direction, we show how advances in search-based SAT algorithms will carry over, via an exhaustive version of DPLL, to compiling propositional theories into one of several tractable languages. We start by pointing out that the trace of an exhaustive DPLL search, recorded compactly as a DAG, can be viewed as a compiled representation of the input theory. With different constraints imposed or lifted on the search process, we then show that the compilation will be in the language of d-DNNF, FBDD, and OBDD, respectively. These languages are known to decrease in succinctness: A propositional theory may have a polynomial-size representation in d-DNNF, but not in FBDD; or in FBDD, but not in OBDD [Darwiche and Marquis, 2002].

In the second direction, we formulate two principles by which the intrinsic complexity and computational power of a DPLL-based exhaustive search is related to the language membership of its traces. Applying these principles, we point out that several recent model counters are doing enough work to actually compile theories into d-DNNF. We also discuss a potential limitation on the efficiency of these model counters, as well as knowledge compilers using similar algorithms, based on the fact that these algorithms only generate traces in a specific subset of d-DNNF. Finally, we note that the efficiency of all DPLL-based exhaustive search algorithms are inherently limited by their inability to produce traces beyond d-DNNF. This realization is significant because some important computational tasks, such as existential quantification of variables and computation of minimum-cardinality models, could be efficiently accomplished with a weaker representation known as DNNF, a strict superset of d-DNNF.

Under our uniform DPLL-based framework for knowledge compilation, the power of successful modern SAT techniques is harnessed, including sophisticated conflict analysis, clause learning, faster detection of unit clauses, and new branching heuristics. We also discuss caching methods specific to the needs of compilation and tailored to the desired target compilation language, as well as structure-based complexity guarantees. We finally relate our experimental results on imple-

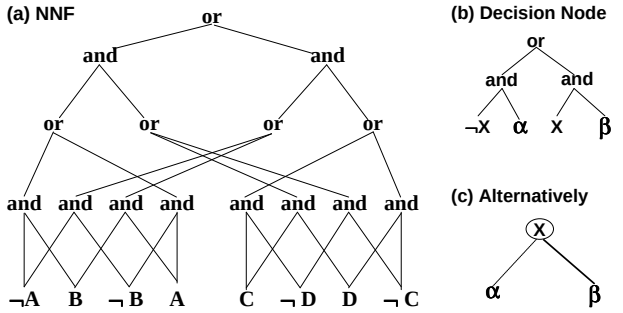

Figure 1: An NNF sentence and a decision node.

mentations of the three respective compilers to the theoretical succinctness relations of the corresponding languages.

The remainder of the paper is organized as follows. Section 2 reviews a number of propositional languages concerned in this work and their theoretical roles and relations in knowledge compilation. In Section 3, we describe a uniform framework for knowledge compilation, with regard to these languages, based on recording the trace of an exhaustive DPLL search, and discuss its implications on our understanding of the complexity and computational power of various search algorithms. We report experimental results in Section 4 and conclude in Section 5.

2 Target Compilation Languages

A logical form qualifies as a target compilation language if it supports some set of nontrivial queries, usually including clausal entailment, in polynomial time. We will review in this section several target compilation languages relevant to the present paper, and refer the reader to [Darwiche and Marquis, 2002] for a more comprehensive discussion.

The languages we shall describe are all subsets of the more general Negation Normal Form (NNF). A sentence in NNF is a propositional formula where conjunction (and, $\wedge$), disjunction (or, $\vee$) and negation (not, $\neg$) are the only connectives and negation only appears next to a variable; sharing of subformulas is permitted by means of a rooted DAG; see Figure 1a.

We start with the DNNF language, which is the set of all NNF sentences that satisfy **decomposability**: conjuncts of any conjunction share no variable. Our next language, d-DNNF, satisfies both decomposability and **determinism**: disjuncts of any disjunction are pairwise logically inconsistent. The formula of Figure 1a, for example, is in d-DNNF.

The FBDD language is the subset of d-DNNF where the root of every sentence is a **decision** node, which is defined recursively as either a constant (0 or 1) or a disjunction in the form of Figure 1b where X is a propositional variable and α and β are decision nodes. Note that an equivalent but more compact drawing of a decision node—shown in Figure 1c—is widely used in the formal verification literature, where FBDDs are equivalently known as Binary Decision Diagrams (BDDs) that satisfy the **test-once** property: each variable appears at most once on any root-to-sink path [Gergov and Meinel, 1994]. See Figure 2c for an FBDD example using this more compact drawing.

Table 1: Polytime queries supported by a language. $\circ$ means "not supported unless P=NP" and ? means "do not know."

Lang.	CO	VA	CE	IM	EQ	SE	CT	ME
DNNF	√	$\circ$	√	$\circ$	$\circ$	$\circ$	$\circ$	√
d-DNNF	√	√	√	√	?	$\circ$	√	√
FBDD	√	√	√	√	?	$\circ$	√	√
OBDD	√	√	√	√	√	$\circ$	√	√
$\text{OBDD}_<$	√	√	√	√	√	√	√	√

The OBDD language is the subset of FBDD where all sentences satisfy the **ordering** property: variables appear in the same order on all root-to-sink paths [Bryant, 1986]. See Figure 2d for an OBDD example. For a particular variable order $<$, we also write $\text{OBDD}_<$ to denote the corresponding OBDD subset where all sentences use order $<$.

Having an option among target compilation languages is desirable, despite their succinctness relations which may be known. The reason is that the succinctness of a language often runs counter to its tractability—that is, the set of queries supported in polynomial time—and the best choice may depend on the task at hand. Given this trade-off between the two criteria, the rule of thumb, according to [Darwiche and Marquis, 2002], is to choose the most succinct language that supports the desired set of queries in polynomial time (in some cases the support of *transformations*, such as Boolean operations, is also a consideration).

Table 1 lists a set of polynomial-time queries of interest supported by each of these languages; the two-letter abbreviations stand for the following eight queries, respectively: **CO**nsistency, **VA**lidity, **C**lausal **E**ntailment, **IM**plicant, **EQ**uivalence, **S**entential **E**ntailment, model **C**oun**T**ing, **M**odel **E**numeration [Darwiche and Marquis, 2002].

Interestingly, this table offers one explanation for the popularity of OBDDs in formal verification where efficient equivalence testing, among other things, is often critical. Although more succinct, d-DNNF and FBDD are not known to admit a polynomial-time equivalence test (a polynomial-time probabilistic equivalence test is possible [Darwiche and Huang, 2002; Blum *et al.*, 1980]). Note also that although there is no difference between d-DNNF and FBDD to the extent of this table, the question mark on the equivalence test (EQ) could eventually be resolved differently for the two languages.

In the following section we will use the notion of recording the trace of a DPLL search to establish an important link between SAT and knowledge compilation, providing a uniform framework for compiling knowledge into some of these languages. From this point of view we will then discuss our new understanding of the complexity and computational power of algorithms based on exhaustive DPLL.

3 DPLL with a Trace

We start with the basic DPLL search as in [Davis *et al.*, 1962]. To facilitate our subsequent discussion of variants of this algorithm, we will omit unit resolution from the pseudocode. (All our discussions, however, are valid in the presence of unit resolution; see also the following two footnotes.)

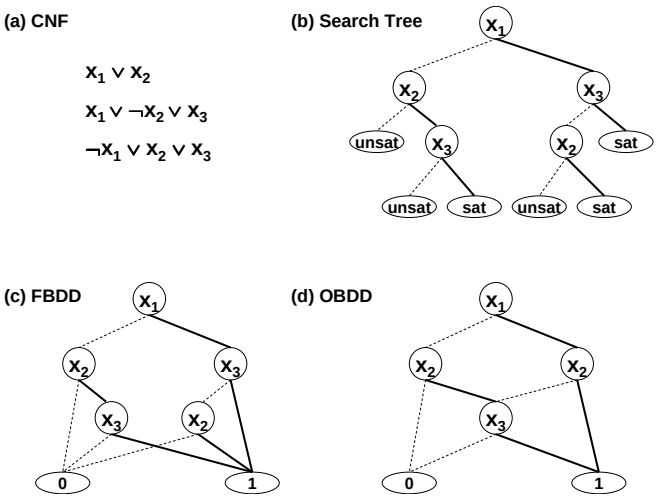

Figure 2: Exhaustive DPLL with a trace.

Algorithm 1 is a summary of DPLL for SAT. It works by recursively doing a case analysis on the assignment to a selected variable (Lines 5&6): The theory is satisfiable if and only if either case results in a satisfiable subtheory. $\Delta|_{x=0}$ ($\Delta|_{x=1}$) denotes the CNF obtained by replacing all occurrences of x with 0 (1) in Δ and simplifying the formula accordingly. In effect, this algorithm performs a search in the space of variable assignments until it finds one that satisfies the given CNF formula or realizes that no satisfying assignment exists.[1]

Now consider extending Algorithm 1 so that it will enumerate all satisfying assignments—by always exploring both branches of Line 6—rather than terminate on finding the first one. Figure 2b depicts the search tree of this exhaustive version of DPLL on the CNF of Figure 2a, under some particular variable ordering. We are using a dotted (solid) line to denote setting the variable to 0 (1), and will refer to the corresponding branch of the search as the *low* (*high*) branch. Note that each leaf of this tree gives a partial variable assignment that satisfies the theory regardless of the values of any unassigned variables, and the whole tree characterizes precisely the set of all satisfying assignments.

[1]To incorporate unit resolution into this simplified picture where all assignments are by decision, one can assume that when a chosen decision variable (Line 5 of Algorithm 1) has been implied by unit resolution, the algorithm will simply proceed down the right branch according to the implied value of the variable, noting that the other branch leads to unsatisfiability. One may also assume that in choosing a decision variable, those already implied by unit resolution will be favored, although this would represent a restriction on the search traces, which we describe shortly, that can be possibly generated.

Algorithm 1 DPLL(Δ): returns satisfiability of CNF Δ

```
1: if there is an empty clause in Δ then
2:     return 0
3: if there is no variable in Δ then
4:     return 1
5: select variable x of Δ
6: return DPLL(Δ|x=0) or DPLL(Δ|x=1)
```

As the title of this paper suggests, we like to think of this complete search tree, also known as a *termination tree* or *decision tree*, as the *trace* left by the exhaustive DPLL search. Such a trace corresponds to the portion of the search space actually explored by a particular execution of the algorithm. Furthermore, the trace can be viewed as a compiled representation of the original CNF formula, because it uniquely identifies the propositional theory—by specifying its models.

From the viewpoint of knowledge compilation, however, a search trace recorded in its present form may not be immediately useful, because it will typically have a size proportional to the amount of work done to produce it. Answering even a linear-time query on such a compilation, for example, would be as if one were running the whole search over again.

This problem can be remedied by reducing the trace from a tree to a DAG, with repeated applications of the following two rules: (i) Isomorphic nodes (i.e., nodes that have the same label, same low child, and same high child) are merged; (ii) Any node with identical children is deleted and pointers to it are redirected to either one of its children [Bryant, 1986].

If we apply these reduction rules to the tree of Figure 2b, and rename "sat/unsat" to "1/0", we will get the DAG of Figure 2c (in this particular example the first rule only applies to the terminal nodes and the second rule does not apply). Observe that this DAG is none other than a propositional theory in the language of FBDD. And this is no accident!

3.1 Compiling CNF into FBDD

We are in fact in possession of a CNF-to-FBDD compiler, described more formally in Algorithm 2. The main difference from the original DPLL is on Line 6: We always explore both branches, and the newly introduced function, **get-node**, provides a means of recording the trace of the search in the form of a DAG. Specifically, get-node will return a decision node labeled with the first argument, having the second argument as the low child, and having the third argument as the high child. Note that Lines 2&4 have been modified to return the terminal decision nodes, instead of the Boolean constants.

We point out that the amount of space required by Algorithm 2 to store the FBDD is only proportional to the size of the final result. In other words, it will never create redundant nodes to be reduced later. This is because the two reduction rules are built-in by means of a **unique nodes** table, well-known in the BDD community [Somenzi, Release 240]. Specifically, all nodes created by get-node are stored in a hash table and get-node will not create a new node if (i) the node to be created already exists in the table (that existing node is returned); or (ii) the second and third arguments are the same (either argument is returned).

Algorithm 2 dpll$_f$(Δ): compiles CNF Δ into FBDD (dpll is lowercased to distinguish it from Algorithm 3, its full version)

```
1: if there is an empty clause in Δ then
2:     return 0-sink
3: if there is no variable in Δ then
4:     return 1-sink
5: select variable x of Δ
6: return get-node(x, dpll_f(Δ|x=0), dpll_f(Δ|x=1))
```

Caching. Despite the use of unique nodes which controls the space complexity, Algorithm 2 still has a time complexity proportional to the size of the tree version of the search trace: Portions of the DAG can end up being explored multiple times. To alleviate this problem, one resorts to *formula caching* [Majercik and Littman, 1998].

Algorithm 3 describes the same CNF-to-FBDD compiler, but now with caching: The result of a recursive call DPLL$_f(\Delta)$ will be stored in a cache (Line 10) before being returned, indexed by a key (computed on Line 5) identifying Δ; any subsequent call on some Δ' will immediately return the existing compilation for Δ from the cache (Line 7) if Δ' is found to be equivalent to Δ (by a key comparison on Line 6).

In practice, one normally focuses on efficiently recognizing formulas that are *syntactically* identical (i.e., have the same set of clauses). Various methods have been proposed for this purpose in recent years, starting with [Majercik and Littman, 1998] who used caching for probabilistic planning problems, followed by [Darwiche, 2002] who proposed a concrete formula caching method in the context of knowledge compilation, then [Bacchus *et al.*, 2003; Sang *et al.*, 2004] in the context of model counting, and then [Huang and Darwiche, 2004; Darwiche, 2004] who proposed further refinements on [Darwiche, 2002].

3.2 From FBDD to OBDD

We now turn to OBDD as our target compilation language. Note that in Algorithm 3, DPLL is free to choose any variable on which to branch (Line 8). This corresponds to the use of a dynamic variable ordering heuristic in a typical SAT solver, in keeping with the spirit of FBDD compilation.

Not surprisingly, a CNF-to-OBDD compiler can be obtained by switching from dynamic to static variable ordering: The new compiler will take a particular variable order π as a second argument, and make sure that this order is enforced when constructing the DAG. See Line 8 of Algorithm 4.

Caching. Naturally, any general formula caching method, such as the ones we described earlier, will be applicable to Algorithm 4. For this more constrained compiler, however, a special method is available where shorter cache keys can be used to reduce the cost of their manipulation. The reader is referred to [Huang and Darwiche, 2004] for details of this method, which allows one to bound the number of distinct cache keys, therefore providing both a space and a time complexity bound. In particular, due to this specific caching scheme, the space and time complexity of compiling OBDDs was shown to be exponential only in the *cutwidth* of the given CNF. A variant caching scheme allows one to show a parallel complexity in terms of the *pathwidth* (cutwidth and pathwidth are not comparable).

Classical OBDD Construction. We emphasize here that Algorithm 4 represents a distinct way of OBDD construction, in contrast to the standard method widely adopted in formal verification where one recursively builds OBDDs for components of the system (or propositional theory) to be compiled and combines them using the *Apply* operator [Bryant, 1986]. A well-known problem with this latter method is that the intermediate OBDDs that arise in the process can grow so large as to make further manipulation impossible, even when the final result would have a tractable size. Considering that the final OBDD is really all that one is after, Algorithm 4 affords a solution to this problem by building exactly it, no more and no less (although it may do more work than is linear in the OBDD size, both because inconsistent subproblems do not contribute to the OBDD size, and because the caching is not complete). An empirical comparison of this compilation algorithm and the traditional OBDD construction method can be found in [Huang and Darwiche, 2004].

3.3 From FBDD to d-DNNF

Although any FBDD is also a d-DNNF sentence by definition, it remains a reasonable option to compile propositional theories into d-DNNF only, given its greater succinctness. Considering that d-DNNF is a relaxation of FBDD, we can obtain a d-DNNF compiler by relaxing a corresponding constraint on Algorithm 3. Specifically, immediately before Line 8, we need not insist any more that a case analysis be performed on some variable x of the formula; instead, the following technique of **decomposition** can be utilized. See Algorithm 5.[2]

As soon as variable instantiation finishes without contradiction, we will examine the remaining CNF formula, and partition it into subsets that do not share a variable (Line 5). These subsets can then be recursively compiled into d-DNNF (Lines 7–9) and conjoined as an **and-node** (Line 10). Note that decomposition takes precedence over case analysis: Only when no decomposition is possible do we branch on a selected variable as in regular DPLL (Lines 14&15).

[2] When unit resolution and clause learning are both integrated into this algorithm, an issue arises regarding implications via learned clauses that span otherwise disjoint components. See [Sang *et al.*, 2004] for more discussion on this issue.

Algorithm 3 DPLL$_f(\Delta)$: compiles CNF Δ into FBDD

```
 1: if there is an empty clause in Δ then
 2:     return 0-sink
 3: if there is no variable in Δ then
 4:     return 1-sink
 5: key = compute-key(Δ)
 6: if (result = cache-lookup(key)) ≠ null then
 7:     return result
 8: select variable x of Δ
 9: result = get-node(x, DPLL_f(Δ|_{x=0}), DPLL_f(Δ|_{x=1}))
10: cache-insert(key, result)
11: return result
```

Algorithm 4 DPLL$_o(\Delta, \pi)$: compiles CNF Δ into OBDD$_\pi$

```
 1: if there is an empty clause in Δ then
 2:     return 0-sink
 3: if there is no variable in Δ then
 4:     return 1-sink
 5: key = compute-key(Δ)
 6: if (result = cache-lookup(key)) ≠ null then
 7:     return result
 8: x = first variable of order π that appears in Δ
 9: result = get-node(x, DPLL_o(Δ|_{x=0}, π), DPLL_o(Δ|_{x=1}, π))
10: cache-insert(key, result)
11: return result
```

Note that our relaxation of Algorithm 3 has resulted in a new type of node, returned by **get-and-node** on Line 10. The old get-node function (Line 15) still returns decision nodes (in a relaxed sense, as their children now are not necessarily decision nodes) in the form of Figure 1c. The unique nodes technique can also be extended in a straightforward way so that isomorphic and-nodes will not be created.

We point out here that Algorithm 5 only **produces sentences in a subset of d-DNNF**, because it only produces a special type of disjunction nodes—decision nodes (again in the relaxed sense). Recall that d-DNNF allows any disjunction as long as the disjuncts are pairwise logically inconsistent. We will come back to this in the next subsection.

Static vs. Dynamic Decomposition. Algorithm 5 suggests a dynamic notion of decomposition, where disjoint components will be recognized after each variable split. This dynamic decomposition was initially proposed and utilized by [Bayardo and Pehoushek, 2000] for model counting and adopted by a more recent model counter [Sang *et al.*, 2004]. [Darwiche, 2002; 2004] proposed another (static) method for performing the decomposition by preprocessing the CNF to generate a *decomposition tree (dtree)*, which is a binary tree whose leaves correspond to the CNF clauses. Each node in the dtree defines a set of variables, whose instantiation is guaranteed to decompose the CNF into disjoint components. The rationale is that the cost of dynamically computing a partition (Line 5) many times during the search is now replaced with the lesser cost of computing a static and recursive partition once and for all. This method of decomposition allows one to provide structure-based computational guarantees as discussed later, and can be orders of magnitude more efficient on some benchmarks, including the ISCAS85 circuits.[3]

[3]One may obtain results to this effect by running the model counter [Sang *et al.*, 2004], version 1.1, available at http://www.cs.washington.edu/homes/kautz/Cachet/, on benchmarks used in [Darwiche, 2004]. It should be noted that the two programs differ in other aspects, but the decomposition method appears to be the major difference. Note also that using DPLL for compilation incurs higher overhead than for model counting due to the bookkeeping involved in storing the DPLL trace.

Algorithm 5 DPLL$_d(\Delta)$: compiles CNF Δ into d-DNNF

```
1: if there is an empty clause in Δ then
2:    return 0-sink
3: if there is no variable in Δ then
4:    return 1-sink
5: components = disjoint partitions of Δ
6: if |components| > 1 then
7:    conjuncts = {}
8:    for all Δ_c ∈ components do
9:       conjuncts = conjuncts ∪ {DPLL_d(Δ_c)}
10:   return get-and-node(conjuncts)
11: key = compute-key(Δ)
12: if (result = cache-lookup(key)) ≠ null then
13:   return result
14: select variable x of Δ
15: result = get-node(x, DPLL_d(Δ|_{x=0}), DPLL_d(Δ|_{x=1}))
16: cache-insert(key, result)
17: return result
```

Caching. Several caching methods have been proposed for d-DNNF compilation, the latest and most effective of which appeared in [Darwiche, 2004]. However, we refer the reader to [Darwiche, 2001b] for a caching scheme that is specific to the dtree-based decomposition method. This scheme is not competitive with the one in [Darwiche, 2004] in that it may miss some equivalences that would be caught by the latter, yet it allows one to show that the space and time complexity of d-DNNF compilation is exponential only in the *treewidth* of the CNF formula (as compared to the pathwidth and cutwidth in OBDD compilation). Interestingly, no similar structure-based measure of complexity appears to be known for FBDDs.

Relation to AND/OR Search. Recent work has explored the long established notion of AND/OR search to process queries on belief and constraint networks [Dechter and Mateescu, 2004b; 2004a]. An AND/OR search is characterized by a search graph with alternating layers of and-nodes and decision-nodes, the former representing decomposition and the latter branching. The DAGs produced by Algorithm 5 are indeed AND/OR graphs and, conversely, the AND/OR search algorithms described in [Dechter and Mateescu, 2004b; 2004a] can be used to **compile** networks into the multi-valued equivalent of d-DNNF. This implies that these AND/OR search algorithms are capable of many more tasks than what they were proposed for—model counting (or other equivalent tasks such as computing the probability of a random variable assignment satisfying the constraint query). We discuss this further in the following subsection.

3.4 Understanding the Power and Limitations of DPLL

The main proposal in this paper has been the view of exhaustive-DPLL traces as sentences in some propositional language. This view provides a unified framework for knowledge compilation as we have shown earlier, but we now show another major benefit of this framework: By using known results about the succinctness and tractability of languages, one can understand better the intrinsic complexity and computational power of various exhaustive DPLL procedures.

Consider a particular variation of DPLL, say DPLL$_x$, and suppose that its traces belong to language L_x. We then have:

1. DPLL$_x$ will not run in polynomial time on formulas for which no polynomial-size representation exists in L_x.
2. If DPLL$_x$ runs in polynomial time on a class of formulas, then DPLL$_x$ (with some trivial modification) can answer in polynomial time any query on these formulas that is known to be tractable for language L_x.

Take for example the model counters recently proposed in [Bayardo and Pehoushek, 2000; Sang *et al.*, 2004], which employ the techniques of decomposition and (the latter also) caching. A simple analysis of these model counters shows that their traces are in the d-DNNF language (for specific illustrations, see the DDP algorithm of [Bayardo and Pehoushek, 2000] and Table 1 of [Sang *et al.*, 2004]). Therefore, neither of these model counters will have a polynomial time complexity on formulas for which no polynomial-size representations exist in d-DNNF.

In fact, one can take this analysis one step further as follows. These model counters, and the compiler of [Darwiche, 2004], actually produce traces in a strict subset of d-DNNF, call it d-DNNF$'$, which employs a syntactic notion of determinism; that is, every disjunction in their trace has the form $(x \wedge \alpha) \vee (\neg x \wedge \beta)$, where x is a splitting variable. The d-DNNF language, however, does not insist on syntactic determinism: It allows disjunctions $\eta \vee \psi$ where $\eta \wedge \psi$ is logically inconsistent, yet η and ψ do not contradict each other on any particular variable x. If d-DNNF$'$ turns out to be not as succinct as d-DNNF, then one may find another generation of model counters and d-DNNF compilers that can be exponentially more efficient than the current ones.

As an example of the second principle above, consider the query of testing whether the minimization of a theory Δ implies a particular clause α, $\min(\Delta) \models \alpha$, where $\min(\Delta)$ is defined as a theory whose models are exactly the minimum-cardinality models of Δ. This query is at the heart of diagnostic and nonmonotonic reasoning and is known to be tractable if Δ is in d-DNNF. Therefore, this query can be answered in polynomial time for any class of formulas on which the model counters in [Bayardo and Pehoushek, 2000; Sang *et al.*, 2004] have a polynomial time complexity. Similarly, a probabilistic equivalence test can be performed in polynomial time for formulas on which these models counters run in polynomial time.

Beyond DPLL. DPLL traces are inherently bound to be NNF sentences that are both deterministic and decomposable. Decomposability alone, however, is sufficient for the tractability of such important tasks as clausal entailment testing, existential quantification of variables, and cardinality-based minimization [Darwiche and Marquis, 2002]. DPLL cannot generate traces in DNNF that are not in d-DNNF, since variable splitting (the heart of DPLL) amounts to enforcing determinism. It is this property of determinism that provides the power needed to do model counting (#SAT), which is essential for applications such as probabilistic reasoning. But if one does not need this power, then one should go beyond DPLL-based procedures; otherwise one would be solving a harder computational problem than is necessary.

4 Experimental Results

By way of experimentation, we compiled a set of CNF formulas into OBDD, FBDD and d-DNNF, both to show the practicality of the DPLL-based compilation approach and to relate the results to the theoretical succinct relations of the three languages. The benchmarks we used include random 3-CNF and graph coloring problems from [Hoos and Stützle, 2000], and a set of ISCAS89 circuits.[4] The compilation was done with the OBDD compiler of [Huang and Darwiche, 2004], using the MINCE variable order [Aloul *et al.*, 2001], an FBDD compiler we implemented according to Subsection 3.1 using the VSIDS variable ordering heuristic [Moskewicz *et al.*, 2001], and the d-DNNF compiler of [Darwiche, 2004].

[4]The CNF formulas used for these sequential circuits model the functionality of their combinational parts; they also define the transition relations of the circuits. Compilations of these theories will thus be useful, for example, for reachability analysis of the circuits.

Table 2: Compiling CNF into OBDD, FBDD, and d-DNNF.

CNF	Number of	OBDD		FBDD		d-DNNF	
Name	Models	Size	Time	Size	Time	Size	Time
uf75-01	2258	10320	0.14	3684	0.02	822	0.02
uf75-02	4622	22476	0.15	14778	0.04	1523	0.03
uf75-03	3	450	0.02	450	0.02	79	0.01
uf100-01	314	2886	2.22	2268	0.01	413	0.02
uf100-02	196	1554	0.91	1164	0.07	210	0.04
uf100-03	7064	12462	0.78	9924	0.12	1363	0.02
uf200-01	112896	8364	651.04	7182	35.93	262	3.66
uf200-02	1555776	–	–	12900	33.72	744	2.64
uf200-03	804085558	–	–	662238	56.61	86696	10.64
flat75-1	24960	23784	0.16	10758	0.04	2273	0.01
flat75-2	774144	13374	0.28	8844	0.04	1838	0.01
flat75-3	25920	84330	0.29	26472	0.07	4184	0.04
flat100-1	684288	62298	0.78	37704	0.10	3475	0.03
flat100-2	245376	88824	1.57	39882	0.30	6554	0.09
flat100-3	11197440	15486	0.15	21072	0.09	2385	0.02
flat200-1	5379314835456	–	–	–	–	184781	56.86
flat200-2	13670940672	–	–	134184	7.07	9859	23.81
flat200-3	15219560448	–	–	358092	4.13	9269	3.28
s820	8388608	1372536	72.99	364698	0.69	23347	0.07
s832	8388608	1147272	76.55	362520	0.70	21395	0.05
s838.1	73786976294838206464	87552	0.24	–	–	12148	0.02
s953	35184372088832	2629464	38.81	1954752	4.01	85218	0.26
s1196	4294967296	4330656	78.26	4407768	12.49	206830	0.44
s1238	4294967296	3181302	158.84	4375122	12.14	293457	0.94
s1423	24758800785707605497982484448	–	–	–	–	738691	4.75
s1488	16384	6188427	50.35	388026	1.14	51883	0.19
s1494	16384	3974256	31.67	374760	1.07	55655	0.18

The results of these experiments are shown in Table 2, where the running times are given in seconds based on a 2.4GHz CPU. The size of the compilation reflects the number of edges in the NNF DAG. A dash indicates that the compilation did not succeed given the available memory (4GB) and a 900-second time limit. It can be seen that for most of these propositional theories, the compilation was the smallest in d-DNNF, then FBDD, then OBDD; a similar relation can be observed among the running times. Also, the number of instances successfully compiled was the largest for d-DNNF, then FBDD, then OBDD. This tracks well with the theoretical succinctness relations of the three languages. (However, note that FBDD and d-DNNF are not canonical representations and therefore compilations smaller than reported here are perfectly possible; smaller OBDD compilations are, of course, also possible under different variable orderings.)

We close this section by noting that the implementations of these knowledge compilers bear witness to the advantage of the DPLL-based framework we have described. The first compiler is based on an existing SAT solver [Moskewicz *et al.*, 2001], and the other two on our own implementation of DPLL, all three benefiting from techniques that have found success in SAT, including conflict analysis, clause learning, and data structures for efficient detection of unit clauses.

5 Conclusion

We established an important relationship between SAT and knowledge compilation, based on studying the trace of an ex-

haustive DPLL search. This relationship provides a uniform framework for compiling propositional theories into various languages of interest, and throws light on the intrinsic complexity and computational power of various DPLL-based algorithms. As interesting examples, we unveiled the "hidden power" of several recent model counters and discussed one of their potential limitations. We also pointed out the inability of exhaustive DPLL to produce traces in strict DNNF, which limits its power from a knowledge compilation point of view.

Acknowledgments

We thank the reviewers for commenting on an earlier version of this paper. This work has been partially supported by NSF grant IIS-9988543 and MURI grant N00014-00-1-0617.

References

[Aloul *et al.*, 2001] Fadi Aloul, Igor Markov, and Karem Sakallah. Faster SAT and smaller BDDs via common function structure. In *International Conference on Computer Aided Design (ICCAD)*, pages 443–448, 2001.

[Bacchus *et al.*, 2003] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 340–351, 2003.

[Barrett, 2004] Anthony Barrett. From hybrid systems to universal plans via domain compilation. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 44–51, 2004.

[Bayardo and Pehoushek, 2000] Roberto Bayardo and Joseph Pehoushek. Counting models using connected components. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI)*, pages 157–162, 2000.

[Blum *et al.*, 1980] Manuel Blum, Ashok K. Chandra, and Mark N. Wegman. Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters*, 10(2):80–82, 1980.

[Bryant, 1986] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[Cadoli and Donini, 1997] Marco Cadoli and Francesco M. Donini. A survey on knowledge compilation. *AI Communications*, 10:137–150, 1997.

[Darwiche and Huang, 2002] Adnan Darwiche and Jinbo Huang. Testing equivalence probabilistically. Technical Report D-123, Computer Science Department, UCLA, 2002.

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

[Darwiche, 2001a] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

[Darwiche, 2001b] Adnan Darwiche. On the tractability of counting theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001.

[Darwiche, 2002] Adnan Darwiche. A compiler for deterministic decomposable negation normal form. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, pages 627–634, 2002.

[Darwiche, 2004] Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, pages 328–332, 2004.

[Davis *et al.*, 1962] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Journal of the ACM*, (5)7:394–397, 1962.

[Dechter and Mateescu, 2004a] Rina Dechter and Robert Mateescu. The impact of AND/OR search spaces on constraint satisfaction and counting. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 731–736, 2004.

[Dechter and Mateescu, 2004b] Rina Dechter and Robert Mateescu. Mixtures of deterministic-probabilistic networks and their AND/OR search spaces. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 120–129, 2004.

[Gergov and Meinel, 1994] J. Gergov and C. Meinel. Efficient analysis and manipulation of OBDDs can be extended to FBDDs. *IEEE Transactions on Computers*, 43(10):1197–1209, 1994.

[Hoos and Stützle, 2000] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In *I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000*, pages 283–292. IOS Press, 2000. SATLIB is available online at *www.satlib.org*.

[Huang and Darwiche, 2004] Jinbo Huang and Adnan Darwiche. Using DPLL for efficient OBDD construction. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 127–136, May 2004.

[Majercik and Littman, 1998] Stephen M. Majercik and Michael L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pages 954–959, 1998.

[Marquis, 1995] Pierre Marquis. Knowledge compilation using theory prime implicates. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 837–843, 1995.

[Moskewicz *et al.*, 2001] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*, pages 530–535, 2001.

[Palacios *et al.*, 2005] Hector Palacios, Blai Bonet, Adnan Darwiche, and Hector Geffner. Pruning conformant plans by counting models on compiled d-DNNF representations. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, 2005.

[Sang *et al.*, 2004] Tian Sang, Fahiem Bacchus, Paul Beame, Henry Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 20–28, 2004.

[Selman and Kautz, 1991] Bart Selman and Henry Kautz. Knowledge compilation using horn approximation. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI)*, pages 904–909, 1991.

[Selman and Kautz, 1996] Bart Selman and Henry Kautz. Knowledge compilation and theory approximation. *Journal of the ACM*, 43(2):193–224, 1996.

[Somenzi, Release 240] Fabio Somenzi. CUDD: CU Decision Diagram Package. Release 2.4.0.