

Knowledge Compilation Properties of Tree-of-BDDs

Sathiamoorthy Subbarayan
IT University of Copenhagen, Denmark
sathi@itu.dk

Lucas Bordeaux and Youssef Hamadi
Microsoft Research, Cambridge, UK
lucasb,youssefh@microsoft.com

Abstract

We present a CNF to Tree-of-BDDs (ToB) compiler with complexity at most exponential in the tree width. We then present algorithms for interesting queries on ToB. Although some of the presented query algorithms are in the worst case exponential in the tree width, our experiments show that ToB can answer non-trivial queries like clausal entailment in reasonable time for several realistic instances. While our ToB-tool compiles all the used 91 instances, d-DNNF compilation failed for 12 or 8 of them based on the decomposition heuristic used. Also, on the succeeded instances, a d-DNNF is up to 1000 times larger than the matching ToB. The ToB compilations are often an order of magnitude faster than the d-DNNF compilation. This makes ToB a quite interesting knowledge compilation form.

Introduction

The characteristics of a real-world system modeled in propositional theory could be used for reasoning the behavior of the system. Knowledge compilation focuses on methods for improving such reasoning. A recent focus has been to compile the theory representing a system into a *form* such that arbitrary number of interesting queries on the system behavior can be quickly answered. The forms which have been studied for compilation include OBDDs (Bryant 1986), DNNF (Darwiche 2001) and d-DNNF (Darwiche & Marquis 2002). Applications of knowledge compilation include configuration, model-based diagnosis and Bayesian inference.

The Tree-of-BDDs (ToB) (Subbarayan 2005) is a recently introduced form in the context of configuration systems. We present the properties of ToB, which makes it a quite interesting form. Specifically, we show that a system modeled in a CNF can be compiled into a canonical ToB with compilation complexity exponential in a tree decomposition width of the CNF. Hence, bounded treewidth CNF instances can be compiled into ToBs using linear time and space. We then present algorithms for several interesting queries on ToBs. Although some of the presented query algorithms are not polytime, our experiments show the viability of ToBs for knowledge compilation.

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Our experiments on 91 realistic instances show that ToB compilations often use an order of magnitude lesser time and space than d-DNNF. In fact, while all the 91 instances are compiled into ToBs within 926 seconds, several d-DNNF compilations fail with 3600 seconds time allowed for each instance. A d-DNNF successfully compiled is up to 1000 times larger than the matching ToB. Also, we observe that non-trivial queries like clausal entailment can be answered by ToBs in reasonable time even for several instances whose d-DNNF compilation fails.

The next section defines the ToB and presents a CNF to ToB compiler. The following section presents the algorithms for queries on ToB. Subsequently, the empirical results are presented followed by some concluding remarks.

The Tree-of-BDDs

We consider that the model to be compiled is given as a CNF. But the presented techniques can be easily extended for more general representations. Let the pair (V, C) denote a CNF, where V is a set of variables and C a set of clauses over V . Let the set of variables occurring in a clause c be denoted by $vars(c)$. The terms like OBDD (DNNF) may refer to a specific OBDD (DNNF), or the function represented by an OBDD (a DNNF), or the set of all possible OBDDs (DNNFs), which will be clear from the context.

Definitions

An *ordered binary decision diagram* (OBDD) (Bryant 1986) is a rooted directed acyclic graph with two terminal nodes, marked true (1) and false (0). Each internal node will have two outgoing edges, one solid and another dashed. Each internal node will be associated a variable, such that in any path from the root to a terminal node the variables associated with the nodes in the path occur at most once and occur in a linear order. We only consider a special *reduced* version of OBDDs in which a node n_1 with both of its outgoing edges reaching the same node n_2 will be removed, and all the incoming edges of n_1 will be made to directly reach n_2 . Also, *isomorphic* nodes, i.e., nodes $\{n_1, \dots, n_i\}$, associated with a same variable, whose solid and dashed edges reach the same nodes n_s and n_d respectively, will be merged into a single node n , with all the incoming edges of $\{n_1, \dots, n_i\}$ now reaching n . Given an OBDD b defined over propositional variables P , and an assignment A to P , the *corre-*

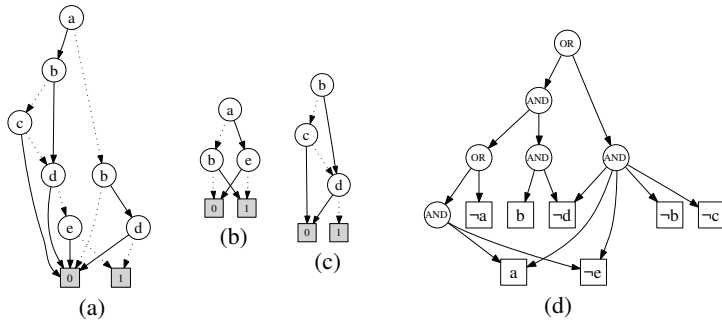


Figure 1: (a) An OBDD, (b) $B(0)$, (c) $B(1)$ and (d) A d-DNNF.

sponding path in b is the unique path from the root node of b to a terminal node, such that the path contains the solid outgoing edge of a node iff the variable corresponding to the node takes true value in A . A function f is represented by an OBDD iff the path corresponding to any *solution* (i.e., a satisfying assignment) of f reaches the true terminal. The size of an OBDD b , denoted $|b|$, is measured in terms of the number of internal nodes or edges.

The operations on OBDDs we use are *conjunction* (\wedge), *disjunction* (\vee), *restriction* ($|$) and *existential quantification* of a set of variables S ($\exists S$). The time and space complexity of \wedge and \vee operations between two OBDDs b_1 and b_2 is $O(|b_1||b_2|)$. The $|$ -operation takes linear time. The time and space complexity of \exists -operation of a variable is $O(|b|^2)$. Given an OBDD b representing a function over variables V , the *projection* of b over a set of variables S is obtained by $\exists(V \setminus S).b$. Given a CNF and a variable order, the OBDD of the CNF could be compiled by \wedge -operations over trivial OBDDs representing the clauses.

A *tree decomposition* (Robertson & Seymour 1986; Dechter 2003) of a CNF (V, C) is a tree (N, A) , such that N is the set of nodes and A the set of edges. Each node $n \in N$ will be associated $vars(n) \subseteq V$. For any $n \in N$, let $clauses(n) = \{c \mid c \in C \wedge vars(c) \subseteq vars(n)\}$. Additionally, (N, A) has to satisfy two conditions: (1) for each clause $c \in C$, $\exists n \in N. c \in clauses(n)$ and (2) for each $v \in V$, the set of nodes $\{n \mid n \in N \wedge v \in vars(n)\}$ form a connected subtree in (N, A) . By the first condition, for every clause $c \in C$, at least one (need not be unique) node $n \in N$ is such that: $vars(c) \subseteq vars(n)$. The second condition specifies that all the nodes that share a variable form a connected subtree. The *width* (w) of (N, A) is $\max_{n \in N} \{|vars(n)| - 1\}$.

A *Tree-of-BDDs* (ToB) (Subbarayan 2005) of a given CNF (V, C) is a triple (N, A, B) , where (N, A) is a decomposition of (V, C) , and B is a mapping such that $\forall n \in N$, the $B(n)$ is an OBDD representing the projection of the CNF on $vars(n)$. That is if the OBDD b represents the CNF, then $\forall n \in N. B(n) = \exists(V \setminus vars(n)).b$. Note, all the $B(n)$ OBDDs use a common variable order. Also, by the properties of tree decomposition, conjunction of the OBDDs of each node will be equivalent to the CNF, i.e., $(b = \wedge_{n \in N} B(n))$. The size of a ToB (N, A, B) is $\sum_{n \in N} |B(n)|$.

Similar to OBDDs each ToB also represents a function. A

```

CNF2ToB  $((V, C), (N, A))$ 
1:  $\forall n \in N. B(n) := \wedge_{c \in clauses(n)} cl2bdd(c)$ 
2: Propagate  $((N, A, B))$ 
3: return  $(N, A, B)$ 

```

```

Propagate  $((N, A, B))$ 
1: for  $i := |N|$  to 2 // except the root node r
2:  $n := BFOOrder(i); p := parent(n)$ 
3:  $B(p) := B(p) \wedge \exists(V \setminus vars(p)).B(n)$ 
4: for  $i := 1$  to  $|N|$ 
5:  $n := BFOOrder(i); H := children(n)$ 
6:  $\forall h \in H. B(h) := B(h) \wedge \exists(V \setminus vars(h)).B(n)$ 

```

Figure 2: The CNF to ToB compiler.

ToB represents a function f iff for any assignment A satisfying f , for each $n \in N$, the path in $B(n)$ corresponding to A reaches the true terminal.

An example Lets consider an example CNF (V, C) with $V = \{a, b, c, d, e\}$ and $C = \{(\neg a \vee \neg e), (a \vee b), (b \vee \neg c), (c \vee \neg d), (\neg b \vee \neg d)\}$. The matching OBDD using the order $a < b < c < d < e$ is in Figure 1a. A matching ToB is (N, A, B) with $N = \{0, 1\}$, $A = \{(0, 1)\}$, $vars(0) = \{a, b, e\}$ and $vars(1) = \{b, c, d\}$. The Figures 1b and 1c show the $B(0)$ and $B(1)$ OBDDs. The size of the OBDD is 14 edges, while the size of the ToB is 12 edges.

We state the following theorem. The theorem follows immediately from the canonicity (uniqueness) of an OBDD for a function up to the used variable order.

Theorem 1. *Given a CNF (V, C) , a tree decomposition of the CNF (N, A) , a linear order $<$ of V , the corresponding ToB (N, A, B) using $<$ is canonical.*

The compiler

Let $BFOOrder$ for a decomposition (N, A) be an order the nodes in N are visited by a breadth-first traversal of (N, A) starting at an arbitrary node r as root. Let $BFOOrder(i)$ denote the i^{th} element in the BFOOrder. Then, for any non-root node $n \in N$, let $parent(n)$ denote the unique neighbor of n which occurs before n in the BFOOrder and let $children(n)$ denote the neighbors of n which occur after n in the BFOOrder.

Let $cl2bdd(c)$ be a function that takes a clause c as input and returns the equivalent OBDD. The pseudo-code of the function $CNF2ToB$ which takes a CNF (V, C) and a tree decomposition (N, A) as input, and returns a ToB (N, A, B) is shown in Figure 2. There are two phases in the function. In the first phase, for each node n in the tree the $clauses(n)$ are compiled into equivalent OBDDs and conjoined to form an initial $B(n)$. In the next phase, the *Propagate* function is called which results in a ToB (N, A, B) . In the lines 1-3 of the *Propagate* function, in the reverse BFOOrder, each node projects its solution space on its parent node. In the lines 4-6, in the BFOOrder, each node projects its solution space over its children, which results in a ToB. The correctness of the pseudo-code $CNF2ToB$ follows from the fact that the code is just a CNF adaptation of a similar compiler in (Subbarayan 2005), which in turn is a compressed way of handling the

well-known *tree-clustering* (Dechter 2003) process.

The following theorem shows that the complexity of CNF2ToB is exponential in the width of the input tree decomposition and hence linear for bounded treewidth CNFs.

Theorem 2. *The complexity of CNF2ToB is $O(|V|w^22^{2w})$.*

Proof. The main cost of the CNF2ToB procedure is due to projection and conjunction operations on $B(n)$ OBDDs. Since each $\text{vars}(n)$ has at most $w + 1$ variables, the matching $B(n)$ never has more than 2^{w+1} paths (solutions) from root to true terminal node. Since each path in a $B(n)$ can have at most $w + 1$ nodes, the number of nodes in a $B(n)$ is at most $(w + 1)2^{w+1}$ at any moment. Also, in a good tree decomposition $|N| \leq |V|$. Hence by using the complexity of the OBDD operations used, the complexity of CNF2ToB is $O(|V|w^22^{2w})$. \square

Queries: Algorithms and Complexity

In this section, following (Darwiche & Marquis 2002), we define the NNF form, and present its subsets. We then present algorithms for interesting queries on ToBs.

A *negation normal form* (NNF) is a rooted directed acyclic graph with leaf nodes associated with either true, false or a literal, and non-leaf nodes will be either an \wedge (AND) node or an \vee (OR) node. A NNF represents the function obtained by recursively defining the functionality of OR and AND nodes to be \vee - and \wedge - operations respectively, over the functions of the child nodes. The size of a NNF is the number of edges.

A NNF may be an OBDD, a DNNF or a d-DNNF based on whether it satisfies some of the four properties: *decomposable* (D), *deterministic* (d), *decision*, and *ordering* ($<$). An AND node of a NNF is *decomposable* if the two functions corresponding to any of the two conjuncts of the AND node do not have a common variable. An OR node of a NNF is *deterministic* if the two functions corresponding to any two disjuncts of the OR node are inconsistent. A NNF is *decomposable* (*deterministic*) if all of its AND (OR) nodes are *decomposable* (*deterministic*). An OR node is a *decision-node* if the OR node has precisely two disjuncts and the disjuncts differ in a variable. That is a decision-node is of the form $(\alpha \wedge x \vee \beta \wedge \neg x)$, where x is a variable called *decision-variable*. Note, a decision-node is also deterministic. A NNF has *decision* property if all of its OR nodes are decision-nodes. A NNF has the *ordering* property if in all the paths from the root to a leaf node, the variables corresponding to the decision nodes in the path occur in a specific linear order.

It has been shown in (Darwiche & Marquis 2002) that, a DNNF is a NNF that is decomposable, a d-DNNF is a DNNF that is also deterministic and an OBDD is a d-DNNF that has both decision and ordering property. Note, while a DNNF or a d-DNNF is directly a NNF, a NNF equivalent to an OBDD can be obtained by converting each node n in the OBDD, with x being the variable associated with n , into the form $(\alpha \wedge x \vee \beta \wedge \neg x)$, where α and β correspond to the functions represented by the solid edge and dashed edge child of the node n . The Figure 1d shows a d-DNNF of size 15 edges, which represents the CNF example.

The following lemma shows that like OBDD, ToB also defines a subset of NNF.

Lemma 3. *ToB defines a subset of NNF*

Proof. We show that for a given ToB we can obtain a NNF representing the same function as the ToB in polytime and hence ToB is a subset of NNF. Let $Obdd2Nnf$ be a function that maps an OBDD to its equivalent NNF. Let the given ToB be (N, A, B) . An equivalent NNF is $\bigwedge_{n \in N} Obdd2Nnf(B(n))$. That is, a NNF equivalent to a given ToB can be obtained by just a conjunction on the NNFs equivalent to each OBDD in the ToB. \square

The eight major queries used in (Darwiche & Marquis 2002) for comparing several compilation forms are: *consistency* (CO), *validity* (VA), *clausal entailment* (CE), *implicant* (IM), *equivalence* (EQ), *sentential entailment* (SE), *model counting* (CT) and *model enumeration* (ME).

Let I_1 and I_2 be instances of a compilation form F , c be a clause and t be a term. The eight queries correspond to: (1) CO : $I_1 = \text{false?}$ (2) VA : $I_1 = \text{true?}$ (3) CE : $I_1 \models c?$ (4) IM : $t \models I_1?$ (5) EQ : $I_1 = I_2?$ (6) SE : $I_1 \models I_2?$ (7) CT : counting the number of models of I_1 (8) ME : an enumeration of the models of I_1 .

A compilation form F *supports* any one of the queries if it can answer the query in time at most polynomial in either the size of the query's input or output.

Now, we show that ToB supports CO and VA .

Theorem 4. *ToB supports CO and VA*

Proof. If a ToB is false (true), then $\forall n \in N. B(n) = \text{false}$ (true). Since checking whether an OBDD is false (true) takes constant time, ToB supports CO (VA). \square

A compilation form supports *conditioning* (CD) if for any instance I_1 in the form and a consistent term t , an instance I_2 in the form which is logically equivalent to $I_1|_t$ can be obtained in polytime.

Now, we state the two following lemmas.

Lemma 5. (Darwiche & Marquis 2002) *If a subset of NNF supports CO and CD , then it also supports ME and CE .*

Lemma 6. (Darwiche & Marquis 2002) *If a subset of NNF supports VA and CD , then it also supports IM .*

Finally, we state the following theorem. The proof of the theorem follows from Theorem 4, and Lemmas 3, 5 and 6.

Theorem 7. *If there exists a polytime CD algorithm for ToB, then ToB supports CE , IM and ME .*

We do not know whether any polytime algorithm exists for conditioning a ToB. Let for a term t and a set of variables P , $t(P)$ denotes restriction of the literals in t to the variables in P . The Figure 3 presents an algorithm for conditioning a ToB. Given a ToB (N, A, B) and a consistent term t , we can condition the ToB by first doing restrict operations: for each $n \in N$. $B(n) := B(n)|_{t(\text{vars}(n))}$. A following *Propagate* operation will result in conditioning the given ToB with t . Following (Darwiche & Marquis 2002), the Figure 3 also presents algorithms for CE and IM . The complexity of the CD , CE and IM algorithms in the figure are dominated by a

Conditioning $((N, A, B), t)$

1 : $\forall n \in N. B(n) := B(n)|_{t(\text{vars}(n))}$
 2 : *Propagate* $((N, A, B))$

Project $((N, A, B), K)$

1 : **for** $i := |N|$ **to** 2 // *except the root node* r
 2 : $n := \text{BFOder}(i); p := \text{parent}(n)$
 3 : $\gamma := V \setminus (\text{vars}(p) \cup K)$
 4 : $B(p) := B(p) \wedge \exists \gamma. B(n)$
 5 : **return** $\exists(\text{vars}(r) \setminus K). B(r)$

IsCE $((N, A, B), c)$

1 : *Conditioning* $((N, A, B), \neg c)$
 2 : **if** $B(r) = \text{false}$ **return true**
 3 : **else return false**

IsIM $((N, A, B), t)$

1 : *Conditioning* $((N, A, B), t)$
 2 : **if** $\forall n \in N. t \models B(n)$ **return true**
 3 : **else return false**

IsEQ $((N_1, A_1, B_1), (N_2, A_2, B_2))$

1 : $\forall n \in N_1$
 2 : $b := \text{Project}((N_2, A_2, B_2), \text{vars}(n))$
 3 : **if** $(b \neq B_1(n))$ **return false**
 4 : $\forall n \in N_2$
 5 : $b := \text{Project}((N_1, A_1, B_1), \text{vars}(n))$
 6 : **if** $(b \neq B_2(n))$ **return false**
 7 : **return true**

Figure 3: The pseudo code for CD, *Project*, CE, IM and EQ.

call to *Propagate*, hence, exponential in the treewidth. Although our conditioning algorithm is not polytime, the empirical results in next section show that ToBs with these algorithms, in reasonable time, can answer CE and IM queries of many instances for which d-DNNF cannot be compiled.

We do not present an algorithm for ME in ToB, since similar to the algorithms for CE and IM, the ME algorithm for d-DNNF in (Darwiche & Marquis 2002) can be naturally extended for ToB with our CD algorithm.

The Figure 3 also presents *Project* an algorithm for obtaining an OBDD which is a projection of the solutions of (N, A, B) over a set of variables K . Essentially, the *Project* existentially quantifies all the variables in $(V \setminus K)$ in a bottom-up fashion, resulting in a projection of the solutions over K . By arguments similar to the proof of Theorem 2, the complexity of the *Project* is $O(|V|(w+|K|)^{2^{2(w+|K|)}})$, where w is the width of (N, A) .

Although a ToB is canonical for a specific (N, A) and ordering, checking the equivalence of two arbitrary ToBs is not trivial. The Figure 3 presents *IsEQ*, a procedure for EQ using the *Project* operation. Given two ToBs (N_1, A_1, B_1) and (N_2, A_2, B_2) using the same variable order, the *IsEQ* projects the solutions of both the ToBs on the variables of each node of the other ToB and checks whether they are equal.

Theorem 8. *The IsEQ $((N_1, A_1, B_1), (N_2, A_2, B_2))$ procedure returns true iff $(N_1, A_1, B_1) = (N_2, A_2, B_2)$.*

Proof. Lets assume the case where the procedure returns false. Wlog, assume that the line(3) of the procedure returns false. Then, there exists a node $n \in N_1$, such that a solution of $B_1(n)$ cannot be extended to a solution of (N_2, A_2, B_2) or vice versa. Hence, the case.

Lets assume the case where the procedure returns true. Then, given any solution S for (N_1, A_1, B_1) , then for each $n \in N_2$, S is also a solution for $B_2(n)$, hence S is also a solution for (N_2, A_2, B_2) . Similarly, a solution for (N_2, A_2, B_2) is also a solution for (N_1, A_1, B_1) . \square

The complexity of the *IsEQ* is dominated by the projection process. If w_1 and w_2 are the widths of (N_1, A_1, B_1) and (N_2, A_2, B_2) respectively, then the complexity of *IsEQ* is $O(|V|(w_1 + w_2)^{2^{2(w_1 + w_2)}})$. Hence, checking the equality of two bounded width ToBs take polytime.

From the *IsEQ* procedure, we can obtain a procedure for SE by just removing the lines 4-6 of *IsEQ* and replacing the condition in the *if* statement in line 3 to $\neg(B_1(n) \models b)$.

Form	CO	VA	CE	IM	EQ	SE	CT	ME
DNNF	✓	○	✓	○	○	○	○	✓
ToB _{<}	✓	✓	†	†	□	□	?	†
d-DNNF	✓	✓	✓	✓	?	○	✓	✓
OBDD _{<}	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Complexity of queries on compilation forms. ✓: polytime, ○: not polytime unless P=NP, †: polytime if CD is polytime, ?: unknown, <: some specific ordering, □: possible in $O(|V|(w_1 + w_2)^{2^{2(w_1 + w_2)}})$. *Note:* All the presented ToB methods are at most exponential in treewidth.

The complexity of the important queries on DNNF, d-DNNF, OBDD and ToB are listed in Table 1. The information for DNNF, d-DNNF and OBDD_< are from (Darwiche & Marquis 2002). In the table, only for ToB the equivalent NNF need not have decomposable property. Remember that, a NNF equivalent to a ToB will have an AND node at the root and it need not be decomposable. Hence, if decomposability is hard for some instances, then compiling ToB will not suffer from that hardness.

The succinctness relation between ToB and DNNF (d-DNNF) is open. But ToB is at least as succinct as OBDD, as given an OBDD b , we can create an equivalent ToB with only one node r in the tree, with $B(r) = b$.

Experiments

The objective of our experiments is to compare the compilation time and size of ToB against d-DNNF, and to check whether the non-trivial queries like CE and IM can be answered by ToBs for realistic instances in reasonable time.

The constituents of the 91 instances used in our experiments are CNFs modeling: 7 ISCAS85 circuits, 17 ISCAS89 circuits, 14 ISCAS93 Addendum circuits, 13 ISCAS99 circuits, 5 bounded model checking problems and 35 Mercedes car configuration problems. We have implemented the CNF2ToB compiler in C++. The source code of our CNF2ToB tool and the used instances, along with links to their origin, are available online¹.

In our experiments we compare our ToB-tool against *c2d²* (version 2.20), the state-of-the-art CNF to d-DNNF compiler. All our experiments are done in an Intel Xeon 3.2 GHz

¹CNF2ToB: <http://www.itu.dk/people/sathi/tob/>

²c2d: <http://reasoning.cs.ucla.edu/c2d/>

Instance				d-DNNF (hg)		d-DNNF (mf)		ToB		ToB - CE Time			ToB - IM Time		
Name	V	C	w (mf)	Time	Size	Time	Size	Time	Size	Mean	Max	>1s%	Mean	Max	>1s%
c1908	751	2053	46	269	26.11	TO		0.1	0.02	0.01	0.06	0	0.01	0.04	0
c7552	3185	8588	44	371	28.89	TO		1.02	0.22	0.06	0.46	0	0.06	0.49	0
s1269	623	1616	40	119	10	TO		0.18	0.05	0.02	0.13	0	0.02	0.14	0
s3271	1714	4269	35	TO		TO		0.55	0.15	0.04	0.19	0	0.04	0.19	0
s4863	2495	6434	33		FSE		FSE	5.09	1.31	0.29	6.13	6	0.31	4.49	5
s6669	3392	8423	26	TO		348	57.53	0.44	0.16	0.03	0.31	0	0.05	0.33	0
cnt09	9207	30678	23	TO		168	0.01	1.86	0.09	0.02	0.07	0	0.02	0.07	0
cnt10	20470	68561	26	TO		TO		5.41	0.21	0.04	0.17	0	0.04	0.16	0
C168FW	1909	7477	110	TO		296	5.54	8.05	0.21	0.02	0.23	0	0.04	0.22	0
C170FR	1874	10610	203	TO		365	5.9	15	0.58	0.04	0.77	0	0.11	0.75	0
C202FS	1990	8883	88	TO		TO		9.58	0.41	0.04	0.75	0	0.1	0.76	0
C202FW	2038	11342	99	TO		TO		65	0.94	0.09	2.62	2	0.21	2.26	4
C208FC	1922	7518	194	TO		243	9.56	18	0.93	0.17	1.42	3	0.17	1.37	3
C210FS	1990	7982	84	TO		2370	51.34	12	0.91	0.09	2.16	2	0.22	2.17	5
C210FW	2024	9705	98	TO		3352	71.63	663	3.64	0.38	10.7	4	1.01	10.42	30

Table 2: Instances for which at least one of the d-DNNF compilations fail. w (mf): Min-fill heuristic width. TO: 3600 seconds time limit exceeded. FSE: File system limit exceeded abort message. Size in 10^6 edges. >1s%: % of 10000 queries requiring more than one second to respond.

Linux machine with 4GB RAM. Each experiment on an instance was given 3600 seconds time limit. The two criteria used for comparison are the time taken for compilation and the size of the resulting representation (a d-DNNF or ToB).

We did not compare our tool against an OBDD compiler as it has been established in (Huang & Darwiche 2005; Subbarayan 2005) that d-DNNFs and ToBs are usually very small when compared with OBDDs. As we are not aware of any publicly available DNNF compiler, similar to (Huang & Darwiche 2005), we do not use a DNNF compiler for comparison.

We use *Min-fill* (Kjærulff 1990) heuristic for decompositions as we found that the *Hinge method* (Gyssens, Jeavons, & Cohen 1994) used in (Subbarayan 2005) gives quite larger width decompositions than min-fill. In fact, experiments using *hinge* failed for most of the instances.

Our ToB-tool just uses the lexicographic ordering of variables. Using any good heuristic for variable ordering may only improve our results. Our experiments using c2d are done with two decomposition methods in c2d: hypergraph partitioning (c2d-hg) and min-fill heuristic (c2d-mf).

	d-DNNF (hg)	d-DNNF (mf)	ToB
#compiled	79	83	91
#failed	12	8	0
Total time [†]	13656.33	13176.43	926.26
Total size [†]	412736839	602249675	17591544

Table 3: Overview. †: for successful compilations.

The Table 3 presents an overview of the compilations. While our ToB tool is able to compile all the instances, c2d-hg (c2d-mf) fails to compile 12 (8) instances. The *Total Time* entry in the table mentions the total time taken by a tool to compile all the instances, excluding time taken for the failed compilations. Even then, for the successful compilations the c2d takes around 13000 seconds with each of the two options. But, all the 91 instances are compiled by the ToB-tool in just 926 seconds. The 91 ToBs require just 17.6 million

edges, while the c2d-hg (c2d-mf) requires 412.7 (602) million edges to represent 79 (83) d-DNNFs.

In Table 2, we list the compilation details of the presumably hard instances: for which at least one of the d-DNNF compilations fail. In case of the *c1908* instance, c2d-mf fails while c2d-hg results in a d-DNNF more than 1000 times larger than the matching ToB.

As shown in Table 2, the ToB compilation for C210FW instance requires 663 seconds. Since the total time taken by ToB-tool for all the 91 instances is 926 seconds, the 90 ToBs other than that of C210FW are compiled in just 263 seconds.

The Table 2 also lists the *mean* and *maximum* response time for 10000 random CE and IM queries for each instance. Each random CE (IM) query will use a random clause (term) with at most $|V|/10$ consistent literals. The table shows that on the average ToB takes at most one second even for the listed hard instances for d-DNNF. In case of C210FW, ToB takes 10 seconds to respond in the worst case, but the size of ToB is around 20 times smaller than the d-DNNF-mg that we definitely get a trade-off, even if d-DNNF gives shorter response time.

In Figure 4, we plot the compilation time taken by ToB versus d-DNNF. In Figure 5, we have a similar plot comparing size. The figures show that the ToB compilations often require relatively very small time and space.

Related Work

Although BDD-trees (McMillan 1994) and ToB have a similar name, they are quite different. The BDD-trees representation uses a binary tree decomposition, while the decomposition used by ToB need not be a binary tree decomposition. Each node in a ToB maps to precisely one OBDD, while each node in a BDD-trees will have a *two-dimensional matrix* of OBDDs. If the left and right child functions of a node in a BDD-trees has m and n equivalence classes of solutions respectively, then the matrix of the node will be of size $m*n$. Each entry in the matrix will be an OBDD. Hence, BDD-trees are quite complex compared to ToB.

Each node in an AND/OR BDD (Mateescu & Dechter

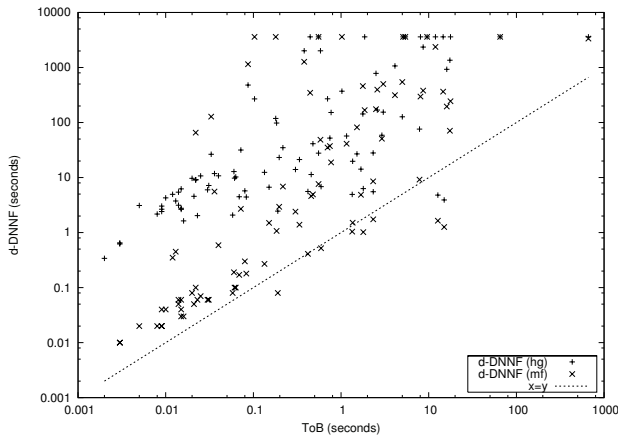


Figure 4: Compilation time: ToB vs. d-DNNF.

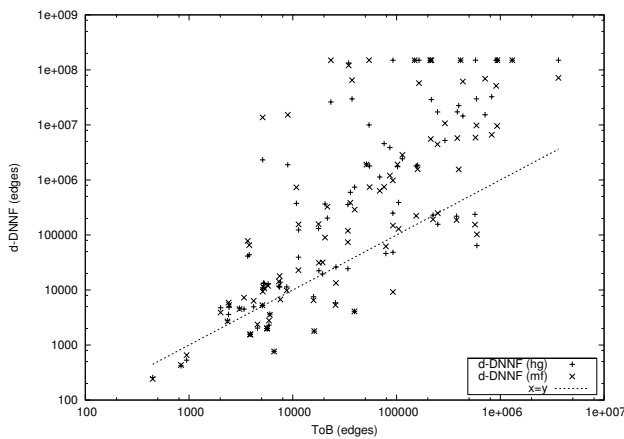


Figure 5: Representation size: ToB vs. d-DNNF.

2006) represents a function of the form: $((\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_{n_1}) \wedge x \vee (\beta_1 \wedge \beta_2 \wedge \dots \wedge \beta_{n_2}) \wedge \neg x)$, where α_i s and β_i s are functions represented by the children of the node and x is the decision variable of the node. Hence, AND/OR BDDs define a subset of d-DNNFs and satisfy the decomposable property. The AND/OR MDDs (Mateescu & Dechter 2006) are multi-valued versions of AND/OR BDDs.

Both the *tree-driven automata* (Fargier & Vilarem 2004) and the *Ordered DDGs* (Fargier & Marquis 2006) are decomposable, and hence different from ToBs.

Conclusion and Future Work

We have presented a CNF to ToB compiler with compilation complexity exponential in the width of a tree decomposition of the CNF. Hence, the compilation takes linear time and space for bounded width CNFs. We have also presented algorithms for several interesting queries on ToBs.

We have presented experiments on several instances, which show that ToB compilation is usually fast and results in smaller representations than d-DNNF compilation. In some instances d-DNNF compilation fails, while ToB

compilation succeeds all the used instances. Even in the instances for which d-DNNF compilation succeeds, a resulting d-DNNF is up to 1000 times larger than the matching ToB.

Our experiments show that ToBs are able to answer non-trivial queries like clausal entailment in reasonable time even for several realistic instances whose d-DNNF compilation fails. This makes ToB a quite interesting form for knowledge compilation.

Avenues for future work include studying the *succinctness* of ToBs, and the *transformations* supported by ToBs. For example, the *negation* transformation of a ToB is trivial, by just switching the two terminal nodes, while it is not known to be polytime for d-DNNFs. Also, the ToB could be adapted for exploiting multi-core CPUs. For instance, if there are four cores in a CPU, then the child subtrees of the root node can be partitioned into four groups and the processing in each group can be allocated to a core.

References

- Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *Transactions on Computers* 8:677–691.
- Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *JAIR* 17:229–264.
- Darwiche, A. 2001. Decomposable negation normal form. *Journal of ACM* 48(4):608–647.
- Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.
- Fargier, H., and Marquis, P. 2006. On the use of partially ordered decision graphs in knowledge compilation and quantified boolean formulae. In *AAAI*, 35–40.
- Fargier, H., and Vilarem, M.-C. 2004. Compiling CSPs into tree-driven automata for interactive solving. *Constraints* 9(4):263–287.
- Gyssens, M.; Jeavons, P. G.; and Cohen, D. A. 1994. Decomposing constraint satisfaction problems using database techniques. *AIJ* 66(1):57–89.
- Huang, J., and Darwiche, A. 2005. DPLL with a trace: From SAT to knowledge compilation. In *IJCAI*, 156–162.
- Kjærulff, U. 1990. *Triangulation of Graphs: Algorithms Giving Small Total State Space*. Technical report, University of Aalborg.
- Mateescu, R., and Dechter, R. 2006. Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In *CP*, 329–343.
- McMillan, K. 1994. Hierarchical representations of discrete functions, with application to model checking. In *CAV*, 41–54.
- Robertson, N., and Seymour, P. 1986. Graph minors. II: Algorithmic aspects of tree-width. *Journal of algorithms* 7(3):309–322.
- Subbarayan, S. 2005. Integrating CSP decomposition techniques and BDDs for compiling configuration problems. In *CP-AI-OR*, 351–365.