Functional Treewidth: Bounding Complexity in the Presence of Functional Dependencies

Yuliya Zabiyaka and Adnan Darwiche

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095-1596, USA,
{yuliaz,darwiche}@cs.ucla.edu

Abstract. Many reasoning problems in logic and constraint satisfaction have been shown to be exponential only in the treewidth of their interaction graph: a graph which captures the structural interactions among variables in a problem. It has long been observed in both logic and constraint satisfaction, however, that problems may be easy even when their treewidth is quite high. To bridge some of the gap between theoretical bounds and actual runtime, we propose a complexity parameter, called functional treewidth, which refines treewidth by being sensitive to non–structural aspects of a problem: functional dependencies in particular. This measure dominates treewidth and can be used to bound the size of CNF compilations, which permit a variety of queries in polytime, including clausal implication, existential quantification, and model counting. We present empirical results which show how the new measure can predict the complexity of certain benchmarks, that would have been considered quite difficult based on treewidth alone.

1 Introduction

The complexity of a number of problems in logic, constraint satisfaction, and probabilistic reasoning is bounded by the treewidth of their interaction graph [8, 3, 9, 12]. The interaction graph is an undirected graph, with nodes representing variables in the given problem, and edges representing direct interactions between variables. For example, the interaction graph for a CNF contains an edge between two variables iff they appear in the same clause. Treewidth is a graph theoretic parameter, which measures the extent to which the graph resembles a tree [12].

Treewidth, however, appears to be too loose of a complexity bound in some cases. In particular, many problem instances that have large treewidth tend to be solvable in time and space that is much smaller than predicted by treewidth. The reason for the discrepancy between theoretical bounds and actual runtime is due to aspects of a problem structure, in particular determinism, which are not captured in the interaction graph and, hence, do not factor into the notion of treewidth. For example, for CNFs, treewidth is insensitive to the particular

literals appearing in a clause, being only a function of the variables appearing in such a clause.

To bridge some of the gap between theoretical bounds based on treewidth and actual runtime, we propose in this paper a more refined parameter, which we call *functional treewidth,* that is sensitive to other aspects of a problem structure, beyond its interaction graph. In particular, functional treewidth is based on both the interaction graph and functional dependencies that are known to hold for the given problem. A functional dependency is a statement of the form $\mathbf{V} \rightarrow V$, where $\mathbf{V}$ is a set of variables and V is a single variable, indicating that each assignment of values to $\mathbf{V}$ implies a particular value for V.

Functional treewidth dominates treewidth and is therefore no easier to compute than treewidth, which is known to be NP–complete [1]. However, we show in this paper that if a CNF has functional treewidth w^f, then it has a *compilation* which is exponential only in w^f. This compilation is in the form of a deterministic, decomposable negation normal form (d-DNNF), which allows a number of queries to be answered in polytime, including clausal entailment, model counting and existential quantification [7]. In fact, we show that one of the simplest algorithms for compiling CNFs into d-DNNFs is capable of producing compilations that are only exponential in the functional treewidth. We note here that these results apply to the compilation of Bayesian networks as well, which can be reduced to the problem of compiling CNFs into d-DNNFs [5].

This paper is structured as follows. Section 2 introduces the new parameter of functional treewidth. Section 3 discusses the compilation of CNFs into d-DNNFs, showing the existence of d-DNNFs that are only exponential in functional treewidth. Section 4 presents a method for approximating functional treewidth, together with experimental results. Section 5 presents further experimental results, showing how functional treewidth can be used to bound the size of d-DNNF compilations. Finally, Section 6 closes with some concluding remarks.

2 Functional Treewidth

The treewidth of an (interaction) graph is usually defined in terms of secondary structures, such as elimination orders, jointrees[1], or dtrees, which can also be used to drive algorithms whose complexity is only exponential in treewidth. A number of these definitions are discussed in [6], with polytime transformations between these structures. We will base our treatment in this paper on dtrees, since these have been used to drive algorithms for compiling CNFs into d-DNNFs.

A *dtree* (decomposition tree) for a CNF Δ is a full binary tree whose leaves are in one–to–one correspondence with the CNF clauses; see Figure 1. We will now define the *width* of a dtree, where the treewidth of CNF Δ is the width of

[1] Jointrees correspond to tree decompositions as known in the graph theoretic literature.

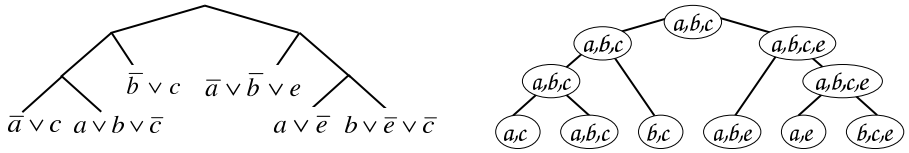

Fig. 1. A dtree (left) for the CNF equivalent to $(a \vee b \equiv c) \wedge (a \wedge b \equiv e)$ and its clusters (right). The width of this dtree is 3.

its best dtree (the one with smallest width). This will also correspond to the treewidth of the interaction graph for CNF Δ.

Before we define the width of a dtree, we need some additional notation. First, as is common with binary trees, we will identify a dtree with its root node. And for a dtree node T, we will use T^l and T^r to denote the left and right children of T, respectively. Moreover, for a leaf node T, the variables of T, $vars(T)$, are just the variables appearing the clause associated with T. For an internal node T, $vars(T) = vars(T^l) \cup vars(T^r)$. We will also use $vars^{\uparrow}(T)$ to denote $\bigcup_{T'} vars(T')$, where T' is a leaf node that is not a descendant of T.

The width of a dtree is defined in terms of the *clusters* of its nodes T, $cluster(T)$. The cluster for a leaf node T is $vars(T)$. The cluster for an internal node T is $(vars(T^l) \cap vars(T^r)) \cup (vars(T) \cap vars^{\uparrow}(T))$. The width of a dtree is then the size of its largest cluster minus one. Figure 1 depicts a dtree with its clusters, leading to a width of 3.

We will next define functional treewidth for a given CNF and a set of *functional dependencies* that are known to hold in the CNF. A functional dependency is a statement indicating that a variable, say, c, is functionally determined by a set of other variables, say, $\{a, b\}$. The functional dependency is denoted by $\{a, b\} \rightarrow c$ in this case. We will also find it useful to define the *closure* of a set of variables $\mathbf{V}$ under some functional dependencies $\mathbf{FD}$, denoted $\mathbf{V}+$ [11]. This set includes $\mathbf{V}$ and other variable that can be derived using the dependencies $\mathbf{FD}$. Consider the following dependencies for example:

$$\begin{array}{l} \{a, b\} \rightarrow c, \\ \{b, c\} \rightarrow d, \\ \{d\} \quad \rightarrow e \end{array}$$

We then have $\{a, b\}+ = \{a, b, c, d, e\}$, $\{a\}+ = \{a\}$ and $\{d\}+ = \{d, e\}$.

The basic intuition behind functional treewidth is that not all instantiations of a cluster in a dtree are indeed consistent with the given CNF, and that complexity can be linear in the number of consistent instantiations instead of all instantiations. Moreover, by reasoning about the functional dependencies that are known to hold in the CNF, one can bound the number of consistent instantiations for a given cluster. To provide such a bound, we need the notion of a (functional) implicant.

Definition 1. *Let* $\mathbf{FD}$ *be a set of functional dependencies over variables* $\mathbf{V}$*, and let* $\mathbf{X}$ *be a subset of* $\mathbf{V}$*. We will say that variables* $\mathbf{I}$ *are a minimal implicant for*

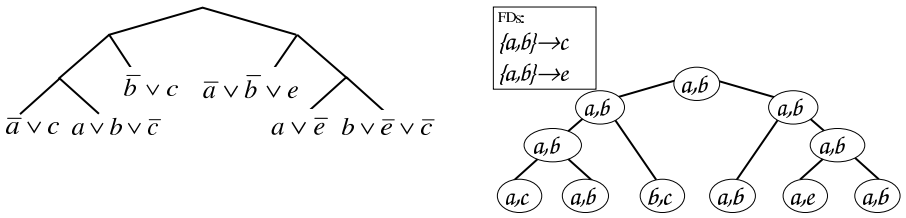

Fig. 2. Dtree (left) and functional clusters (right).

variables $\mathbf{X}$ *under* **FD** *iff* $\mathbf{X} \subseteq \mathbf{I}+$ *and for any other set of variables* $\mathbf{J}$ *where* $\mathbf{X} \subseteq \mathbf{J}+$, *we have* $|\mathbf{I}+| \leq |\mathbf{J}+|$.

The importance of minimal implicants is this: if a cluster has a minimal implicant of size m, then it has no more than 2^m instantiations which are consistent with the given CNF. Note that the identity of the minimal implicant is not essential for this bound, only its size is.[2] Note also that this notion is different from the notion of a *key* as employed in database theory in the context of functional dependencies, where a key for $\mathbf{X}$ is an implicant for $\mathbf{X}$ that is also a subset of $\mathbf{X}$.

We are now ready to define functional treewidth.

Definition 2. *Let T be a node in a dtree for CNF Δ with functional dependencies* **FD**. *A functional cluster for node T, denoted cluster$^f(T)$, is a minimal implicant for cluster(T) under dependencies* **FD**.

Figure 2 depicts functional clusters for the dtree introduced in Figure 1.

Definition 3. *Let T be a dtree for CNF Δ with functional dependencies* **FD**. *The functional width of dtree T is the size of its maximal functional cluster minus 1. The functional treewidth of CNF Δ is the functional width of its best dtree (the one with the smallest functional cluster).*

It should be clear that functional treewidth can be no greater than treewidth, with equality in case the set of functional dependencies is empty. We will show constructively in the following section that if a CNF Δ has functional treewidth w^f, it must then have a d-DNNF compilation exponential only in w^f.

3 The Compilability of CNFs

We will consider in this section the compilability of CNFs into d-DNNFs, which is a tractable form that supports in polytime queries such as clausal entailment, model counting, and existential quantification [7]. This tractable form is also closed under conditioning (the setting of variable values), allowing an exponential number of queries to be answered each in polytime.

[2] If an algorithm is to take advantage of functional dependencies to improve its running time, the identity of the minimal implicant may matter then.

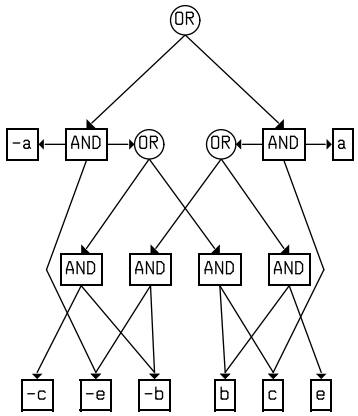

Fig. 3. A d-DNNF.

Algorithm 1 cnf2ddnnf(T: dtree, α: instantiation): returns a d-DNNF.

1: $\gamma = project(\alpha, context(T))$
2: $result = CACHE_T(\gamma)$
3: **if** $result \neq NIL$ **then**
4: return $result$
5: **if** T is a leaf **then**
6: $result = clause2ddnnf(cnf(T), \alpha)$
7: **else**
8: $result = \vee_\beta cnf2ddnnf(T^l, \alpha \wedge \beta) \wedge cnf2ddnnf(T^r, \alpha \wedge \beta) \wedge \beta$
9: where β ranges over all instantiations of $cutset(T)$
10: $insert\_CACHE_T(\gamma, result)$
11: return $result$

A d-DNNF is a rooted directed acyclic graph in which each leaf node is labeled with a literal, *true* or *false*, and each internal node is labeled with a conjunction or disjunction; see Figure 3. For any node N in a d-DNNF graph, $vars(N)$ denotes all propositional variables that appear in the subgraph rooted at N, and $\Delta(N)$ denotes the formula represented by N and its descendants. The nodes in a d-DNNF have the following two properties:

- **Decomposability:** $vars(N_i) \cap vars(N_j) = \emptyset$ for any two children N_i and N_j of an and-node N.
- **Determinism:** $\Delta(N_i)$ is inconsistent with $\Delta(N_j)$ for any two children N_i and N_j of an or-node N.

Algorithm 1 provides a procedure for compiling a CNF into a d-DNNF, adapted from [4]. We will explain the intuition behind the algorithm shortly, but we first point out its complexity. If the algorithm is passed a dtree with n nodes and width w, it will generate a d-DNNF for the corresponding CNF in

Algorithm 2 clause2ddnnf($l_1 \vee \ldots \vee l_m$: clause, α: instantiation): returns a d-DNNF for clause $(l_1 \vee \ldots \vee l_m)|\alpha$

```
1: if m = 1 then
2:     return l_1|α
3: else if α ⊨ ¬l_1 then
4:     return clause2ddnnf(l_2 ∨ ... ∨ l_m, α)
5: else if α ⊨ l_1 then
6:     return true
7: else
8:     return l_1 ∨ (¬l_1 ∧ clause2ddnnf(l_2 ∨ ... ∨ l_m, α))
```

$O(nw2^w)$ time. The algorithm must initially be called with α being the empty instantiation.

The main technique in Algorithm 1 is that of recursive decomposition. Specifically, given a CNF Δ, we partition its clauses into Δ^l and Δ^r. If Δ^l and Δ^r share no variables, we can then compile them independently and simply conjoin the results. Suppose, however, that the two sets turn out to share a variable v. We will then use what is known as Boole's or Shannon's expansion:

$$\Delta = (v \wedge \Delta^l|v \wedge \Delta^r|v) \vee (\neg v \wedge \Delta^l|\neg v \wedge \Delta^r|\neg v),$$

where $\Delta|v$ ($\Delta|\neg v$) denotes the process of *conditioning,* which consists of replacing the occurrences of variable v by $true$ ($false$) in Δ. This recursive decomposition process is then governed by the given dtree, since each dtree node can be viewed as inducting a binary partition on the clauses below that node.

The algorithm makes use of two sets of variables at each node T. First, is $cutset(T)$ which is the set of variables that we must condition on so we can decompose the clauses below node T, $\Delta(T)$, into those below child T^l, $\Delta(T^l)$, and those below child T^r, $\Delta(T^r)$. Note that by the time we reach node T in the recursive decomposition process, the cutsets of all ancestors of node T must be instantiated. These variables are called $acutset(T)$, for ancestoral cutset of node T. Hence, $cutset(T)$ is defined as $vars(T^l) \cap vars(T^r) - acutset(T)$.

The second set of variables used at node T is $context(T) = acutset(T) \cap vars(T)$. These are variables that are guaranteed to be set when we recurse on node T, and that also appear in clauses below T. Any two recursive calls to node T which agree on the value of variables $context(T)$ must return equivalent answers. Hence, the algorithm maintains a cache at each node indexed by the instantiations of $context(T)$ to avoid recursing multiple times on the same subproblem. We note here that for an internal dtree node T, $cutset(T) \cup context(T)$ is actually the cluster of node T as defined in the previous section.

Before we present the central result in this section, we point out the following about Algorithm 1. Figure 4 depicts the d-DNNF substructure that a call $cnnf2dnnf(T, \_)$ will contribute to the final d-DNNF. In particular, for every instantiation γ of $context(T)$, Algorithm 1 will produce an OR–node, and for each instantiation β of $cutset(T)$ (under a given context instantiation γ), it will

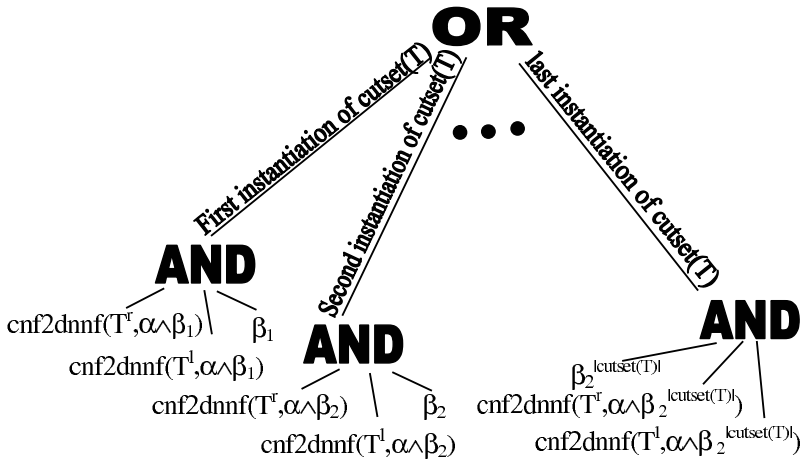

Fig. 4. The d-DNNF substructure constructed by $cnf2ddnnf(T, \alpha)$ at node T.

produce an AND–node. In fact, the contributed substructures can be exponential in the size of $cluster(T)$. In particular, the call will contribute $2^{|context(T)|}$ OR–nodes, and for each such node, it will contribute $2^{|cutset(T)|}$ AND–nodes as children. Since $cutset(T)$ and $context(T)$ share no variables by definition, the size of the contributed structure is then

$$2^{|cutset(T) \cup context(T)|} = 2^{|cluster(T)|}.$$

Our central result is then as follows.

Theorem 1. *If Algorithm 1 is passed a dtree with n nodes, m variables and functional treewidth w^f, it will return a d-DNNF of size $O((n+m)2^{w^f})$.*[3]

This basically shows that if a CNF Δ has a functional treewidth of w^f, then it must have a d-DNNF compilation of size $O((n+m)2^{w^f})$. In fact, for $cluster(T)$, only instantiations which are consistent with the CNF will contribute structures to the final d-DNNF compilations, as shown in Figure 4. The proof of Theorem 1 is given in the Appendix.

4 Approximating Functional Treewidth

Determining the functional width of a dtree requires the computation of minimal implicants, which includes the computation of minimal keys as a special case (a problem known to be NP-complete [10]). We consider in this section a method for approximating functional width of a given dtree and present a number of empirical results, showing its effectiveness.

[3] Alternatively, we can bound the size of produced d-DNNF by $O(nw2^{w^f})$, where w is the width of a dtree.

Our basic method for computing minimal implicants for $cluster(T)$ is based on an exhaustive procedure which searches for implicants of increasing sizes, up to size $k = |cluster(T)|$. This procedure is not practical though given the number of candidate implicants we need to consider, which is a function of both k and the number of CNF variables (from which we need to compute an implicant). We improve this procedure by not involving all CNF variables in the analysis, but only those that can be reached by traversing the functional dependencies backward, starting from variables in $cluster(T)$. We also approximate the procedure if this is not sufficient by restricting the set of variables from which the implicant is computed. In particular, we restrict our implicants to the following sets, with decreasing size: $(acutset(T) \cup vars(T))+$, $(acutset(T) \cup cluster(T))+$, and $cluster(T)$. Our method will switch from one approximation to the next if it examines more than a certain number of candidates, finally giving up and returning $cluster(T)$ as the approximation if none of the tried approximations yield a smaller set. Another approximation technique is to try to find implicants for $cutset(T)$ and $context(T)$ separately, instead of $cluster(T)$, as that provides more specific choices for reducing the set of variables from which to draw an implicant from.

To evaluate the effectiveness of proposed approximations, we experimented with many benchmarks with abundance of (easily recognizable) functional dependencies. This included various digital circuits from the LGSynth93 suite (*http://www.bdd-portal.org/benchmarks.html*), and grid CNFs from [13] which come with varying degrees of functional dependencies. Table 1 depicts results for the LGSynth93 suite, showing exponential improvements in the bounds based on (approximated) functional treewidth compared to those based on (approximated) treewidth. This basically allows us to prove the compilability of corresponding CNFs using (approximate) functional treewidth, even though the CNFs have very large (approximate) treewidths.

Grid networks were defined in [13]. They are $N \times N$ Bayesian networks with variables denoted $X_{i,j}$ for $1 \leq i, j \leq N$, where each variable $X_{i,j}$ has parents $X_{i-1,j}$ and $X_{i,j-1}$ when the corresponding indices are greater than zero. A fraction of the nodes, equal to *d ratio,* is determined by their parents (half of those nodes determined by both parents and another half by a single parent). Table 2 depicts results on CNF encodings of grid networks, showing how the (approximated) functional treewidth gets smaller as we increase the amount of determinism.

5 Bounding the Size of CNF Compilations

We have evaluated in the previous section the quality of our approximations for functional treewidth by comparing them to treewidth. In this section, we do another comparison with the actual size of d-DNNFs computed by cnf2ddnnf. To be able to perform this comparison, we had to restrict ourselves to problems whose (approximate) treewidth is manageable (≤ 20) since the time complexity of cnf2ddnnf is exponential in this treewidth. Hence, our results in this section

cnf	w	w^f	dtree nodes	m	n	a	time (min)
5xp1	27	8	1563	296	782	35	0.59
5xp1_ok	19	7	1253	255	627	9	0.23
9sym	51	14	2403	442	1202	155	1.89
9sym.scan_ok	35	9	2195	439	1098	85	1.44
9symml	31	13	1901	376	951	67	1.03
alu2	55	17	4551	833	2276	686	15.32
apex2.scan_ok	47	39	3869	784	1935	322	6.00
C1355	32	26	4775	979	2388	702	11.28
C1908	38	34	3769	770	1885	915	5.80
C2670	29	26	5985	1407	2993	415	11.19
C880	24	24	2949	633	1475	192	1.71
clip	58	12	3881	716	1941	219	6.09
clip_ok	29	14	1577	316	789	49	0.49
duke2	66	41	3743	689	1872	375	5.64
e64	83	67	4875	899	2438	633	10.89
ex4p	22	17	4497	1014	2249	197	3.74
f51m	28	12	1853	347	927	49	0.66
frg1	47	25	4541	834	2271	383	11.46
inc	20	7	1483	299	742	14	0.27
rd53	19	10	717	138	359	19	0.15
rd73	37	7	2871	536	1436	168	4.47
rd84	42	22	5331	985	2666	605	12.02
sao2	34	17	1871	346	936	155	1.18
sct	18	11	1293	266	647	45	0.32
sqrt8ml	16	10	1781	363	891	174	1.80
squar5	25	12	885	167	443	62	0.64
term1	45	25	5471	1064	2736	342	8.88
ttt2	22	15	4017	774	2009	192	4.49
vda	101	51	6691	1180	3346	772	31.86
vg2	58	33	4063	748	2032	235	4.72
x4	25	23	5919	1199	2960	281	8.71
z4ml	22	11	1581	294	791	29	0.48

Table 1. LGsynth93 suite: n is number of clauses, m is number of variables, and a is the number of clusters for which the functional cluster was not necessarily minimal (approximated).

$N\times$ N	*d-ratio*	w	w^f	dtree nodes	m	n	a	time (min)
10×10	50	15	13	455	100	228	27	0.01
10×10	75	15	9	575	100	288	49	0.01
10×10	100	15	1	627	100	314	0	0.02
14×14	50	21	18	835	196	418	37	0.02
14×14	75	21	17	1091	196	546	252	0.06
14×14	100	21	1	1295	196	648	0	0.13
18×18	50	27	26	1293	324	647	62	0.07
18×18	75	27	23	1767	324	884	408	0.19
18×18	100	27	1	2155	324	1078	0	0.59
22×22	50	33	32	2073	484	1037	138	0.25
22×22	75	33	30	2709	484	1355	885	0.63
22×22	100	33	1	3269	484	1635	0	1.97
26×26	50	39	36	2927	676	1464	229	0.60
26×26	75	39	34	3675	676	1838	848	1.74
26×26	100	40	1	4569	676	2285	89	6.73
30×30	50	46	42	3940	900	1970	319	2.20
30×30	75	43	38	5043	900	2522	1682	4.07
30×30	100	47	3	6125	900	3063	451	13.10
34×34	50	52	49	5083	1156	2542	361	3.23
34×34	75	51	47	6536	1156	3268	1950	4.68
34×34	100	52	27	7855	1156	3928	1447	48.93

Table 2. Grid networks.

are to some extent biased towards problems that are somewhat easy due to the relatively small clusters involved.

For a set of circuit benchmarks from the suite LGsynth93, we extracted CNFs together with functional dependencies: for every gate constructing a functional dependency from its inputs to its output. For every CNF we constructed a dtree using hypergraph partitioning method [6], approximated its treewidth w and approximated its functional width w^f. Next, we bounded the number of edges that every node T will contribute to the d-DNNF based on on the size of $cluster^f(T)$. Using this procedure we got two bounds on the number of d-DNNF edges: a bound based on structural clusters *s-bound*, and a bound based on functional clusters *f-bound*. We then ran Algorithm 1 and calculated the true number of edges in every d-DNNF *e-count*. Tables 3 and 4 depict the results of our experiments. To make the assessment of the quality of approximations easier, we also report *s/f = s-bound/f-bound* and *f/e =f-bound/e-count*. As *f/e* approaches 1, our functional treewidth bounds get closer to the true size of compilation. The point that is worth noting is that even in the cases where w and w^f are quite close, the bound provided by functional treewidth can still be much smaller than the one based on treewidth since our bounds are a function of all clusters in the dtree, not just the largest ones (captured by width).

cnf	A	n	m	s-bound	f-bound	e-count	s/f	f/e	w	w^f	T_c(s)	T_w(s)
5xp1	A	627	255	7066536	26246	21427	269.24	1.22	19	7	4.92	12.30
apex7	A	1147	493	2246904	148576	35994	15.12	4.13	16	11	0.72	48.66
b1	E	66	29	5316	1266	919	4.20	1.38	8	3	0.00	0.05
b12	A	402	174	715024	62318	23887	11.47	2.61	15	11	0.84	2.52
b9	A	551	256	449264	69156	36807	6.50	1.88	15	11	0.20	6.17
bw	A	844	337	20659544	25464	19858	811.32	1.28	20	5	37.25	74.20
C17	E	30	17	900	556	355	1.62	1.57	4	3	0.01	0.02
C432	A	946	403	4446852	829182	53043	5.36	15.63	17	15	3.09	51.02
c8	A	1091	444	11843932	431312	98070	27.46	4.40	19	13	16.77	87.19
cc	A	302	140	19876	8042	5532	2.47	1.45	8	5	0.05	1.11
cht	A	1243	515	310124	44348	32945	6.99	1.35	12	7	0.20	42.72
cm138a	A	74	35	3156	1738	1268	1.82	1.37	6	4	0.00	0.06
cm150a	A	364	165	28672	8464	6458	3.39	1.31	8	6	0.03	2.55
cm151a	A	176	82	12224	4028	3074	3.03	1.31	8	5	0.01	0.48
cm152a	A	117	54	14276	5134	2287	2.78	2.24	8	7	0.01	0.19
cm162a	A	238	107	15028	6306	4034	2.38	1.56	7	7	0.01	0.80
cm163a	A	230	106	13708	5644	3873	2.43	1.46	7	6	0.00	0.75
cm42a	E	84	37	4044	1796	1381	2.25	1.30	6	4	0.00	0.08
cm82a	A	116	52	5416	2012	1528	2.69	1.32	5	3	0.00	0.17
cm85a	A	208	95	16776	6700	3395	2.50	1.97	8	8	0.01	0.64
cmb	A	214	97	65912	34124	8020	1.93	4.25	11	11	0.06	0.73
comp	A	577	258	50528	21478	10900	2.35	1.97	9	7	0.03	7.45
con1	A	93	45	6812	2678	1608	2.54	1.67	8	6	0.01	0.11
cordic	A	533	235	135768	54876	11444	2.47	4.80	12	12	0.11	6.47
count	A	641	292	39324	14702	10237	2.67	1.44	7	5	0.03	7.03
cu	A	259	110	82200	24388	10918	3.37	2.23	11	10	0.08	0.78
decod	E	98	39	8416	2580	1886	3.26	1.37	8	5	0.01	0.09
i1	A	177	95	7289	3733	2559	1.95	1.46	6	5	0.01	0.36
i2	A	1144	657	523620	440606	101623	1.19	4.34	15	15	0.45	38.05
i3	A	774	456	22565	12981	8955	1.74	1.45	5	4	0.02	14.08
i4	A	922	530	42863	25627	16000	1.67	1.60	6	6	0.03	16.81
i5	A	1538	734	171886	69408	45664	2.48	1.52	10	9	0.16	72.16
i6	A	1844	866	285256	97294	25765	2.93	3.78	14	11	0.30	112.70
i7	A	2346	1115	369436	140848	32221	2.62	4.37	13	10	0.30	267.94
inc	A	742	299	17985228	37284	27866	482.38	1.34	20	7	15.30	12.98
lal	A	643	275	7525122	2263668	68058	3.32	33.26	19	17	3.80	8.98
ldd	A	418	162	16931664	1312920	9504	12.90	138.14	20	17	0.63	3.70
majority	A	73	33	6300	2280	1709	2.76	1.33	7	5	0.00	0.11
misex1	A	320	135	136812	22928	7852	5.97	2.92	12	10	0.11	1.31
misex2	A	457	194	1153460	294684	36024	3.91	8.18	15	13	0.89	2.95

Table 3. LGsynth93 suite: A stands for approximate and E for exact value of w^f (for given dtree), n is number of clauses, m is number of variables, *s-bound* is edge bound based on structural clusters, *f-bound* is edge bound based on functional clusters, *e-count* is true number of edges in d-DNNF compiled by cnf2ddnnf, *s/f = s-bound/f-bound* and *f/e =f-bound/e-count*, T_c is time to run cnf2ddnnf and T_w is time to calculate w^f.

cnf	A	n	m	s-bound	f-bound	e-count	s/f	f/e	w	w^f	T_c(s)	T_w(s)
mux	A	501	210	464996	24372	16759	19.08	1.45	13	7	0.31	6.31
o64	A	519	325	20624	10670	7745	1.93	1.38	6	5	0.02	4.89
parity	A	257	122	10924	4000	2954	2.73	1.35	5	3	0.00	1.06
pcle	A	280	128	16100	6352	4142	2.53	1.53	6	5	0.01	1.06
pcler8	A	344	160	24560	12090	5575	2.03	2.17	7	6	0.03	1.69
pm1	A	270	121	38507	15577	13147	2.47	1.18	10	8	0.03	0.84
rd53	A	279	117	1064492	17360	5431	61.32	3.20	16	10	0.38	1.08
rd73	A	738	300	74425360	34784	26292	2139.64	1.32	22	7	88.98	27.42
sct	A	647	266	7146520	364242	60214	19.62	6.05	19	14	4.33	10.19
sqrt8	A	310	131	437540	26908	8566	16.26	3.14	15	10	0.13	1.31
sqrt8ml	A	891	363	1389624	62652	14777	22.18	4.24	16	11	1.42	40.03
squar5	A	298	122	275848	8838	6342	31.21	1.39	14	6	0.31	1.31
t481	A	409	172	45598204	7203748	98915	6.33	72.83	21	19	30.67	6.25
tcon	E	202	98	9472	3878	2969	2.44	1.31	5	3	0.00	0.41
unreg	A	568	264	56284	18402	13415	3.06	1.37	8	6	0.06	4.74
x2	A	223	93	602572	37384	8102	16.12	4.61	15	11	0.41	0.66
xor5	A	115	52	7660	2158	1332	3.55	1.62	6	5	0.00	0.19
z4ml	A	791	294	53209456	107650	38650	494.28	2.79	22	11	100.74	31.59

Table 4. LGsynth93 suite. Continuation of Table 3.

6 Discussion

We proposed in this paper the notion of functional treewidth, as a complexity parameter for bounding the size of certain CNF compilations, which permit polytime queries, such as clausal entailment, model counting and existential quantification. We have also presented a method for approximating functional treewidth and applied it to a number of benchmarks, showing its ability to provide bounds that are exponentially better than those based on treewidth.

Our current and future work on this subject centers around three directions. First, the development of better approximation algorithms for functional treewidth (there is a long tradition of such approximations for treewidth (see, e.g., [2]). Next, the development of dtree construction methods which are sensitive to functional dependencies. Note that a dtree with larger width may have a smaller functional width. This means that a method for constructing dtrees that minimizes width may miss dtrees which are optimal from a functional width viewpoint. Finally, the use of functional dependencies in improving the time and space complexity of compilation algorithms, therefore, allowing bounds on the running time based on functional treewidth.

Proof of Theorem 1

Without loss of generality, we will assume that the original CNF Δ is consistent, and that $cnf2ddnnf(T, \alpha)$ is initially called with $\alpha = true$.

The proof is based on a number of lemmas, which concern a recursive call $cnf2ddnnf(T, \alpha)$ to dtree node T, with $\Delta(T)$ denoting the clauses below node T:

1. Lemma 1: $cnf2ddnnf(T, \alpha)$ returns $false$ if $\Delta(T)|\alpha$ is inconsistent.
2. Lemma 2: If $\Delta \wedge \alpha$ is consistent, and $\Delta \wedge \alpha \wedge \beta$ is inconsistent for some instantiation β of $cutset(T)$, then $\Delta(T)|\alpha\beta$ is inconsistent and, moreover, either $\Delta(T^l)|\alpha\beta$ or $\Delta(T^r)|\alpha\beta$ is inconsistent.
3. Lemma 3: If $\Delta \wedge \alpha$ is inconsistent, then there is an ancestor T^a of node T for which $\Delta \wedge \alpha^a$ is consistent, where $\alpha^a = project(\alpha, acutset(T^a))$. Moreover, $\Delta \wedge \alpha^a \wedge \beta^a$ is inconsistent for instantiation $\beta^a = project(\alpha, cutset(T^a))$.

The proof is based on the observation that each disjunct on Line 8 of Algorithm 1 corresponds to an instantiation $\gamma \wedge \beta$ of $cluster(T)$ (γ is instantiation of $context(T)$ and β is instantiation of $cutset(T)$). We will prove that the disjuncts corresponding to instantiations $\gamma \wedge \beta$ of $cluster(T)$ will not be part of the returned d-DNNF if $\gamma \wedge \beta$ is inconsistent with the CNF Δ.

Consider now the call $cnf2ddnnf(T, \alpha)$, and let $\gamma = project(\alpha, context(T))$. If $\gamma \wedge \beta$ is inconsistent with Δ then $\alpha \wedge \beta$ is inconsistent with Δ as well. We have two cases:

1. $\Delta \wedge \alpha$ is consistent, but $\Delta \wedge \alpha \wedge \beta$ is inconsistent.
   Note that $context(T)$ contains all variables shared between clauses in $\Delta(T)$ and other clauses. Hence, if $\Delta \wedge \alpha \wedge \beta$ is inconsistent, then $\Delta(T) \wedge \alpha \wedge \beta$ is inconsistent. By Lemma 2, $\Delta(T)|\alpha\beta$ is inconsistent and, moreover, either $\Delta(T^l)|\alpha\beta$ or $\Delta(T^r)|\alpha\beta$ is inconsistent. Hence, by Lemma 1, the corresponding disjunct on Line 8 of Algorithm 1 will evaluate to $false$ and will not be included in the computed d-DNNF.
2. $\Delta \wedge \alpha$ is inconsistent (and, hence,$\Delta \wedge \alpha \wedge \beta$ is inconsistent).
   By Lemma 3, there is an ancestor T^a of node T for which $\Delta \wedge \alpha^a$ is consistent, where $\alpha^a = project(\alpha, acutset(T^a))$. Moreover, $\Delta \wedge \alpha^a \wedge \beta^a$ is inconsistent for instantiation $\beta^a = project(\alpha, cutset(T^a))$. By Lemma (1), the call $cnf2ddnnf(T^a, \alpha^a)$ will return $false$ and, hence, the disjunct on Line 8 of Algorithm 1 constructed during the call $cnf2ddnnf(T, \alpha)$ will not be part of the returned d-DNNF.

We will now bound the size of returned d-DNNF, measured by the number of edges in the d-DNNF. Let cnf Δ have n_Δ clauses and m variables, then dtree T_Δ will have n_Δ leaf nodes and $n_\Delta - 1$ internal nodes. Each internal node T will contribute $\leq 2^{|cluster^f(T)|}$ disjuncts. Each disjunct will have the following edges: an edge to the parent OR–node; two edges to the solutions produced by left and right child; $|cutset(T)|$ edges to the literal nodes corresponding to the instantiation of the $cutset(T)$. Thus the total amount of edges contributed by internal nodes is (an additional factor of 2 appears when we switch from $|cluster^f(T)|$ to w^f due to -1 in the definition of w^f):

$$\sum_{\text{internal node } T} (3 + |cutset(T)|) \cdot 2^{|cluster^f(T)|} \leq 2 \cdot 2^{w^f} \cdot (3n_\Delta + m).$$

Note that $\sum_{\text{internal node } T} |cutset(T)|$ can be bounded either by the number of variables m (because all the cutsets are disjoint) or by $w \cdot n_\Delta$ (because none of the cutsets have the size greater than w).

Each consistent instantiation of the $cluster(T)$ for leaf T contributes $4 \cdot (|var(T) - acutset(T)| - 1)$ edges: each uninstantiated variable contributes at most one AND-node, one OR-node and two literal nodes (except for the last one contributing one literal node). Thus the total number of edges contributed by leaf nodes is

$$\sum_{\text{leaf node T}} 4 \cdot (|var(T) - acutset(T)| - 1) \cdot 2^{|cluster^f(T)|} \leq 4m \cdot 2 \cdot 2^{w^f}$$

The result is due to the fact that $\sum_{\text{leaf node T}}(|var(T) - acutset(T)| - 1)$ can be bounded analogous to $\sum_{\text{internal node T}} |cutset(T)|$.

Noting that the total number of nodes n in dtree T_Δ is equal to $2n_\Delta - 1$, we conclude that the number of edges in the final compilation produced by Algorithm 1 is bounded by $O((n + m)2^{w^f})$.

References

1. Stefan Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic and Discrete Methods*, 8:277–284, 1987.
2. H. L. Bodlaender. A tourist guide through treewidth. *ACTA CYBERNETICA*, 11(1-2):1–22, 1993.
3. Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.
4. 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.
5. Adnan Darwiche. A logical approach to factoring belief networks. In *Proceedings of KR*, pages 409–420, 2002.
6. Adnan Darwiche and Mark Hopkins. Using recursive decomposition to construct elimination orders, jointrees and dtrees. In *Trends in Artificial Intelligence, Lecture notes in AI, 2143*, pages 180–191. Springer-Verlag, 2001.
7. Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.
8. Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 2003.
9. F. V. Jensen, S.L. Lauritzen, and K.G. Olesen. Bayesian updating in recursive graphical models by local computation. *Computational Statistics Quarterly*, 4:269–282, 1990.
10. Claudio L. Lucchesi and Sylvia L. Osborn. Candidate keys for relations. *Journal of Computer and System Sciences*, 17:270–279, 1978.
11. David Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, MD, 1983.
12. N. Robertson and P. D. Seymour. Graph minors II: Algorithmic aspects of treewidth. *J. Algorithms*, 7:309–322, 1986.
13. T. Sang, P. Beam, and H. Kautz. Solving bayesian networks by weighted model counting. In *Proceedings of AAAI*. AAAI, 2005.