On the Relative Efficiency of Dynamic and Static Top-Down Compilation to Decision-DNNF

Alexis de Colnet
Algorithms and Complexity Group, TU Wien, Austria

Abstract

Top-down compilers of CNF formulas to circuits in decision-DNNF (Decomposable Negation Normal Form) have proved to be useful for model counting. These compilers rely on a common set of techniques including DPLL-style exploration of the set of models, caching of residual formulas, and connected components detection. Differences between compilers lie in the variable selection heuristics and in the additional processing techniques they may use. We investigate, from a theoretical perspective, the ability of top-down compilation algorithms to find small decision-DNNF circuits for two different variable selection strategies. Both strategies are guided by a graph of the CNF formula and are inspired by what is done in practice. The first uses a dynamic graph-partitioning approach while the second works with a static tree decomposition. We show that the dynamic approach performs significantly better than the static approach for some formulas, and that the opposite also holds for other formulas. Our lower bounds are proved despite loose settings where the compilation algorithm is only forced to follow its designed variable selection strategy and where everything else, including the many opportunities for tie-breaking, can be handled non-deterministically.

2012 ACM Subject Classification Theory of computation → Dynamic programming

Keywords and phrases Knowledge compilation, top-down compilation, decision-DNNF Circuits

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.11

Funding This work has been supported by the Austrian Science Fund (FWF), ESPRIT project FWF ESP 235.

1 Introduction

The foundation of *knowledge compilation* is the idea that different representations of a function facilitate solving different kind of problems. Classes of representations where specific problems become tractable are studied under the name of *(compilation) languages* [10, 11, 12]. The purpose of (knowledge) *compilers* is to transform (or compile) propositional formulas, circuits, or other, into a target language where some intractable problems become solvable in polynomial time. Compilers to languages where model counting is linear-time are, for obvious reasons, particularly investigated. In practice, one of the main language for doing model counting is *decision-DNNF*: the class of circuits in *decision decomposable negation normal form* [20], with compilers building decision-DNNF circuits from CNF formulas such as `c2d` [9], `dsharp` [19], `d4` [17] and the compiler version of `sharpsat-TD` [16, 15].

From a complexity theory point of view, compilation is often seen as a preprocessing task where only the size of the compiled form (that is, the output of the compiler) matters [4, 6]. In practice of course, running time matters, and perhaps is the priority since a compiler that ends rapidly is also guaranteed to construct a reasonably small compiled form (though the converse is not true). So practical compilers use strategies, heuristics and whatnot to try to terminate within a certain time window. But then we ask what are the consequences of these implementation choices on their ability to find small compiled forms. Are there formulas that admit small decision-DNNF circuits but for which our compilers always construct large circuits? In this paper, we answer this question positively for two, hopefully realistic, algorithms for top-down compilers of CNF to decision-DNNF. More precisely, we confront

© Alexis de Colnet;
licensed under Creative Commons License CC-BY 4.0
27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024).
Editors: Supratik Chakraborty and Jie-Hong Roland Jiang; Article No. 11; pp. 11:1–11:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the two algorithms by finding formulas that are hard to compile for one, in the sense that it returns super-polynomial-size compiled forms, but easy to compile for the other, and vice versa. The two algorithms share many similarities and are *top-down*.

Top-down compilation refers to a compilation based on an exhaustive DPLL search that uses caching and independent components identification. Here, "exhaustive" means that the DPLL procedure does not stop after finding one model of the formula but keeps searching for all of them. Caching allows to save time and memory by preventing the compiler to work twice on the same formula [1, 23, 9]. Independent components identification also speeds up compilation by determining when a formula can be split into independent subformulas that are then compiled separately [14, 8, 1]. Without this mechanism, the compiled form are FBDDs (free binary decision diagrams) [13], which are generally significantly less succinct than decision-DNNF circuits [2]. In practice, identifying independent subformulas means checking whether some graph of the formula has several connected components. Compilers work toward splitting the graph through heuristics and strategies that guide the choices of the branching variables [18]. The idea is to favor variables that belong to certain *cutsets*, or separators, of the graph. This is explicit in a compiler like `d4` that uses (hyper)graph partitioning tools to find such cutsets. In `c2d`, the cutsets are computed beforehand from recursive graph partitioning or from a tree decomposition and organized in a data structure called a dtree that is passed on to the compiler. In a model counter like `sharpsat-TD`, cutsets are hidden in a tree decomposition used to influence the scores of the variables to branch on. The two top-down compilation algorithms we study are inspired by these compilers. The first one always branch on variables from a cutset determined by a static tree decomposition of the primal graph of the original CNF and the second dynamically looks for cutsets that are balanced separators of the graph of the current formula.

Results on the efficiency of static decomposition-based model counters compared to dynamic model counters or compilers are not novel. For instance, more than twenty years ago, for model counting, the authors of [1] showed that decomposition-based techniques like *recursive conditioning* [7] do not poly-time simulate a dynamic variant of the counting algorithm #DPLLCache. But these are results on running time, and besides they are proved for unsatisfiable formulas. This is not very useful for us since we only care about the *size* of the compiler output and, even if it takes a long time, a good compiler always returns a constant-size output on unsatisfiable inputs. In addition, we want our algorithms to be realistic in their variable selection behavior, so they cannot be completely non-deterministic. It is typically not clear for which deterministic version of #DPLLCache the results of [1] still hold. Closer to our work, the authors of [5] show lower bounds on the size of decision-DNNF circuits that are *structured-decomposable* which they argue correspond the circuits constructed by `c2d`. This is true if certain features of compilers are disabled, in particular inprocessing techniques like unit propagation, which is not something we want for our results. Yet we believe it is true that the compilers based on static decompositions construct circuits "more structured" (though not structured-decomposable) than those relying on more dynamic approaches. Our lower bounds on the static decomposition-based compilation algorithm are in fact proved using some different notion of structuredness. These lower bounds are not surprising since it is known that structured-decomposability tends to dramatically increase the size of the circuits [22]. But then it is a bit more surprising that our decomposition-based algorithm can largely outperform our dynamic algorithm, whose output is barely structured, on specific instances.

The paper is organized has follows. We start with some preliminaries in Section 2. In Section 3 we describe our framework for top-down compilation and the loose settings in which we our results fit: basically the cutset selection mechanism is strict but every other

procedures (caching, inprocessing, variable selection inside the cutset) are mostly undefined, so that positive upper bound results hold even with naive procedures and negative lower bound results hold even with non-deterministic procedures. We then show in Section 4 that the static approach returns compiled forms that have some "structure". We use this in Section 5 to design functions that are easy to compile in the dynamic approach but hard for the static approach. Finally we prove that the opposite holds for other functions in Section 6.

2 Preliminaries

We use the notations $[n] = \{1, 2, \ldots, n\}$ and $[0, n] = \{0, 1, \ldots, n\}$.

The domain of a *Boolean variable* is $\{0, 1\}$ ($\{false, true\}$). An *assignment* to a set X of Boolean variables is a mapping $\alpha : X \to \{0, 1\}$. A *partial assignment* to X is an assignment to a subset of X. A *Boolean function* f over X maps the assignments to X to $\{0, 1\}$. α is a model of f when $\alpha \in f^{-1}(1)$. We sometimes write $f(X)$ to precise that f is a function over X. When X is not explicit we refer to it as $var(f)$. Given a partial assignment α to $var(f)$, we denote by $f|\alpha$ the Boolean function over $var(f) \setminus var(\alpha)$ whose models are the models of f consistent with α and projected onto $var(f) \setminus var(\alpha)$. Constants 0 and 1 are sometimes seen as functions over $\emptyset$.

A *literal* is a variable x or its negation $\neg x$, also written $\bar{x}$. The negated literal $\bar{\ell}$ equals x if $\ell = \bar{x}$, and equals $\bar{x}$ if $\ell = x$. A *clause* is a disjunction of literals and a CNF formula (Conjunctive Normal Form) F is a conjunction of clauses. CNF formulas are sometimes seen as sets of clauses, and we write $c \in F$ to mean $c \in clause(F)$. For us, $F|\alpha$ is the CNF formula obtained by removing all clauses of F containing literals satisfied by α, and all literals falsified by α from the remaining clauses. If $F|\alpha$ is empty, then it is replaced by 1. If $F|\alpha$ contains an empty clause, then it is replaced by 0. The *primal graph* G_F of F is the graph whose vertices are F's variables and such that there is an edge between x and y if and only if there is a clause $c \in F$ such that $x \in var(c)$ and $y \in var(c)$. The connected components of G_F correspond to the the largest subformulas of F that share no variable. We denote by *components*(F) the set of these subformulas.

2.1 Graphs Separators and Tree Decompositions

For T a tree and t and t' two nodes T, $t \leq_T t'$ means that t' is an ancestor of t or that $t' = t$, whereas $t <_T t'$ means that t' is an ancestor of t and $t \neq t'$. Let G be a graph with vertex set $V(G)$ and edge set $E(G)$. For $V \subseteq V(G)$, $G[S]$ is the graph with vertex set S and edge set $\{\{u, v\} \mid u \in S, v \in S, \{u, v\} \in E(G)\}$. We write $G - S = G[V(G) \setminus S]$. A *tree decomposition* $\mathcal{T}$ of G is a pair (T, b) where T is a rooted tree and b is a function $b : V(T) \to \mathcal{P}(V(G))$ such that

- for every $v \in V(G)$, there exists $t \in V(T)$ such that $v \in b(t)$;
- for every $\{u, v\} \in E(G)$, there exists $t \in V(T)$ such that $\{u, v\} \subseteq b(t)$;
- for every $v \in V(G)$, $T[\{t \mid v \in b(t)\}]$ is connected (so it is a tree).

The set $b(t)$ is called the *bag of* t. The width of $\mathcal{T}$ is the maximum size of a bag, so $\max_{t \in V(T)} |b(t)|$. The *treewidth* of G, noted $tw(G)$, is the minimum width for a tree decomposition of G minus 1. We denote by $b_\downarrow(t)$ the union of $b(t)$ and of the bags of all descendants of t, i.e., $b_\downarrow(t) = \bigcup_{t' \leq_T t} b(t')$.

Suppose G is connected. A *separator* of G, or a *cutset* of G, is a subset $S \subseteq V(G)$ such that $G - S$ has more than one connected component. For $\delta \in [0, 1]$, a δ-*balanced separator* of G is a separator S of G such that every connected component of $G - S$ has at most $\delta|V(G)|$ vertices. We denote by $s_\delta(G)$ the smallest size of a δ-balanced separator of G.

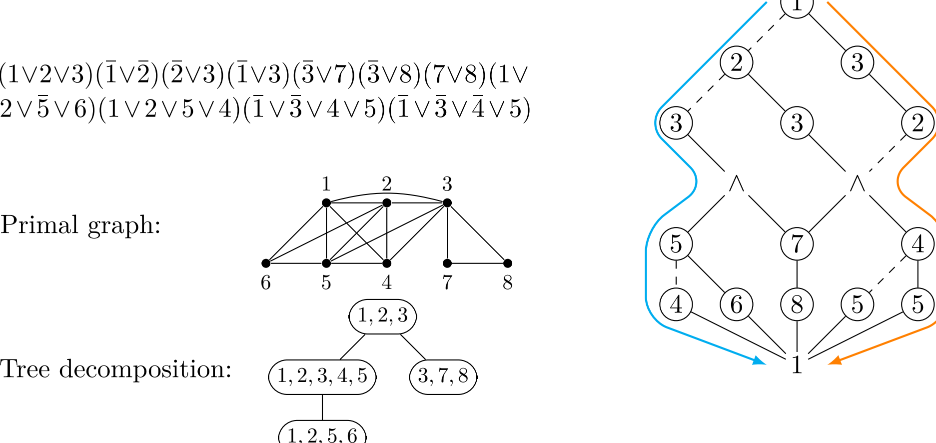


Figure 1 A CNF formula and a decision-DNNF circuit that represents it.

2.2 Decision-DNNF Circuits

A decision-DNNF (decision Decomposable Negation Normal Form) circuit is a directed acyclic graph with a single source and whose nodes are of three types: *sinks*, *decision nodes* and *decomposable* $\wedge$*-nodes*. Each node v computes a Boolean function $\langle v\rangle$ over $var(v)$.

- A sink has out-degree 0 and is labeled by a constant $c \in \{0, 1\}$. Here $\langle v\rangle = c$ and $var(v) = \emptyset$.
- A decision node v is labeled by a Boolean variable x and has two children: the 0-child v_0 and the 1-child v_1, with $x \notin var(v_0) \cup var(v_1)$. We write $v = ite(x, v_1, v_0)$ (if x then v_1 else v_0). Here $\langle v\rangle = (\bar{x} \wedge \langle v_0\rangle) \vee (x \wedge \langle v_1\rangle)$ and $var(v) = \{x\} \cup var(v_0) \cup var(v_1)$.
- A decomposable $\wedge$-node v is labeled by the conjunction symbol $\wedge$, it has children $v_1, \dots, v_k$ with $k \geq 1$ such that $var(v_i) \cap var(v_j) = \emptyset$ for every $i \neq j$. The node is interpreted as $\langle v\rangle = \langle v_1\rangle \wedge \dots \wedge \langle v_k\rangle$ and $var(v) = var(v_1) \cup \dots \cup var(v_k)$.

A decision-DNNF circuit C with source node v computes, or represents, the Boolean function $\langle v\rangle$ over $var(n)$. We directly see C as a Boolean function and write $var(C) = var(v)$. Graphically, for α an assignment to $var(C)$, the value $C(\alpha) = \langle v\rangle(\alpha)$ can be determined starting from v and by descending the circuit as follows: upon encountering the node u, if u is a decision node for x then continue from its $\alpha(x)$-child, if u is sink then stop, and if u is a decomposable $\wedge$-node then continue the from all its children $u_1, \dots, u_k$ in parallel. If at least one sink 0 is reached then $C(\alpha) = 0$, otherwise $C(\alpha) = 1$. Given a partial assignment β to $var(C)$, a decision-DNNF circuit for $C|\beta$ is constructed as follows: for every $x \in var(\beta)$ and every decision node $v = ite(x, v_1, v_0)$ in C, redirect all parents of v to $v_{\beta(x)}$ and delete v. Once this is done, remove all subcircuits not reachable from the source.

An example of decision-DNNF circuit is shown in Figure 1. Graphically, a decision node for x is represented with a circle labeled by x. It is connected to its 0-child by a dashed line and to its 1-child by a solid line. Only the 1-sink is represented in Figure 1. Missing outputs for decision nodes go straight to the 0-sink.

3 A Framework for Top-Down Compilation to decision-DNNF

Algorithm 1 encompasses the behavior of top-down compilers from CNF to decision-DNNF. It is largely inspired from [17]. The priority of the algorithm is to split the primal graph G_F of the input formula F. This is done by selecting a cutset of the graph, that is, a set of

vertices/variables whose removal leaves the graph disconnected. Assigning a variable to 0 or 1 removes at least this variable from the graph. The cutset variables are assigned in all possible ways until the graph is disconnected, which may happen before all are assigned. Every variable assignment adds a decision node to the circuit (line 11). Between two successive variable assignments, the algorithm checks whether the graph is disconnected and, if so, a decomposable $\wedge$-node is created and the subformulas for each components are dealt with independently (line 7). The cutset is reset to $\emptyset$ for these subformulas to notify that a new cutset must be computed. Due to its recursive nature, `Compile` requires two arguments: F and a subset S of F's variables that corresponds to what remains of the cutset. When S is empty (which is the case initially) we write `Compile`(F) instead of `Compile`$(F, \emptyset)$.

Cutset selection corresponds to the procedure `selectCutset`. It is the one procedure where the two algorithms studied in this paper behave differently. In the *dynamic approach*, `selectCutset`(F) returns a balanced separator of G_F whose size is minimal, or close to minimal. In the *static approach*, we have access to a tree decomposition $\mathcal{T}$ of G_F or of a supergraph of G_F and `selectCutset`(F) returns the bag of a highest node of $\mathcal{T}$ which has a non-empty intersection with $var(F)$. There will always be a unique such node in our settings. We give names to the two variants of `Compile`.

- `Compile`$_{d,\epsilon}(F)$ is the algorithm `Compile` where `selectCutset`(F) returns a 2/3-balanced separator of G_F of size at most $(1 + \epsilon)s_{2/3}(G_F)$.
- `Compile`$_s(F, \mathcal{T})$ is the algorithm `Compile` with an extra argument: a tree decomposition $\mathcal{T}$ that remains constant through the whole algorithm, and where `selectCutset`(F) returns a highest bag of $\mathcal{T}$ that has a non-empty intersection with $var(F)$.

The dynamic approach requires solving a hard problem (graph balanced-partitioning) several time during compilation while the static approach requires a hard preprocessing step, namely computing a tree decomposition whose width is close to minimal. We disregard running times in this paper, so we just ignore the complexity of these problems. However, to be a bit closer to reality we try not to put too much constrain on `selectCutset`, hence the requirement that the size of the cutset in the dynamic approach is minimal *up to a fixed constant factor*. One forces the dynamic compile algorithm to find a minimal-size cutset by setting $\epsilon = 0$. We will write `Compile`$_d$ instead of `Compile`$_{d,0}$. Similarly, for the static approach, we will allow tree decompositions of width minimal *up to a fixed constant factor*. We leave tie-breaking non-deterministic when several candidates exist for the output of `selectCutset`. We stress out that the two variants of `Compile` work on the *primal* graph. Practical compilers may choose other graphs, like `d4` which uses a *dual hypergraph* of F [17].

Variable selection corresponds to the procedure `selectVariable` for selecting the next variable to assign in the cutset. We leave this procedure undefined. One can select variables in a predefined order, or use frequency-based heuristic, or heuristics influenced by the outcome of the algorithm on previous branches like VSIDS, or a non-deterministic oracle, etc. Our results are agnostic to this procedure. The only rule is to *not select a variable outside of the cutset*. For the static approach, this is actually a deviation from practical tree-decomposition-based compilers and model counters. For instance, `sharpSAT-TD` [16] selects variables based on a score computed using the depth of the variables in the tree decomposition but also their frequency and VSIDS scores. On the one hand, the depth component of the score makes it more likely to select a variable appearing in the highest bag of the decomposition, which corresponds to our cutset in the static approach. On the other hand, the VSIDS and frequency components can force the selection of a variable outside of the cutset. It has been noticed that the depth component is dominant in practice and that the other components mostly serve as a tie-breaking mechanism [16, Section 5]. So we think our model is quite realistic. Though we do not know if it is for the functions that we use to prove our results.

Caching allows to avoid constructing a decision-DNNF for the same formula twice. By default, `cache`(F) equals `nil` for every formula F. We consider two caching variants: one realistic, the other idealized. The realistic caching is *syntactical*: in Line 5, `cache`$(F) \neq$ `nil` means that the formula F has already been seen in previous calls to the algorithm. The idealized caching is *semantical*: in Line 5, `cache`$(F) \neq$ `nil` means that some formula F' logically equivalent to F, with $var(F') \subseteq var(F)$, has been seen in previous calls (for instance $F = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3)$ and $F' = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3)$), and then `cache`$(F)$ is defined as `cache`(F'). The condition $var(F') \subseteq var(F)$ is here to avoid situations where decomposability of $\wedge$-nodes would be compromised because of caching. We write `Compile`$[smc](\dots)$ to precise that semantical caching is used. Otherwise, syntactical caching is used (no caching is not allowed).

Inprocessing simplifies the formula between every two variable selections. This is the procedure `process`(F) at Line 2. Here it returns a modified formula for F and a term (a conjunction of literals) τ. A classical procedure is *unit propagation*: while there is a *unit clause* $\ell \in F$, ℓ is added to the term τ and F is replaced by $F|\ell$. Unit propagation is linear-time, but we can also have more complex processing procedures like *satisfiability checking*: if F is unsatisfiable, then it is replaced by 0. The stronger procedure we consider is *backbone identification*: τ contains all literals ℓ such that $F \models \ell$ and F is replaced by $F|\tau$. Note that backbone identification subsumes both unit propagation and satisfiability checking. Indeed if F is unsatisfiable, then $F \models x$ and $F \models \bar{x}$ for every x and we just assume `process`(F) returns $(0, \{x, \bar{x} | x \in var(F)\})$. A compiler may implement several processing techniques at the same time. In practice NP-hard processing procedures require calling an external SAT solver. Again, we just disregard the running time of `process` and assume it is sound and complete. We write `Compile`$[bb](\dots)$ when backbone identification is enabled (and a fortiori unit propagation and satisfiability checking). Without $[bb]$ inprocessing is disabled (even unit propagation), then `process`(F) returns $(F, \emptyset)$. When τ is not empty, then it is added to the output with a $\wedge$-node at line 7 or 11. For instance, if $F = (x_1) \wedge (\bar{x}_1 \wedge \bar{x}_2) \wedge (x_2 \vee \bar{x}_1 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee \bar{x}_5 \vee x_6)$, and backbone identification is enabled, then `process`$(F) = ((x_4 \vee x_5 \vee x_6) \wedge (\bar{x}_4 \vee \bar{x}_5 \vee x_6), x_1 \wedge x_2 \wedge x_3)$ and the algorithm goes through lines 9, 10 and 11. Say x_4 is selected line 10, then line 11 returns the following:

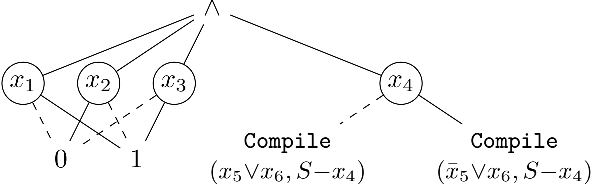


Figure 2

where each literal of $\tau = x_1 \wedge x_2 \wedge x_3$ is converted into a decision node to make sure that the resulting circuit is a decision-DNNF. Admittedly, in this paper we are far from exhaustive when it comes to inprocessing, for instance we do not consider literal equivalences detection [17], vivification [21], elimination of redundant clauses, etc.

4 Decision-DNNF Organized by Tree Decompositions

The output of `Compile`$_s(F, \mathcal{T})$ is a decision-DNNF circuit *organized by* $\mathcal{T}$. We introduce this notion in this section. In a decision-DNNF circuit, a *path* from the source to a sink is a sequence of nodes $(v_1, v_2, \dots, v_k)$ with v_1 the source, v_{i+1} a child of v_i, and v_k a sink.

Algorithm 1 The general `Compile` procedure.

1: **input:** a CNF formula F, a set of variable $S \subseteq var(F)$
2: $(F, \tau) \leftarrow \texttt{process}(F)$ // τ is a conjunction of literals
3: $S \leftarrow S \setminus var(\tau)$
4: **if** $F = 0$ or $F = 1$ **then** return F
5: **if** $\texttt{cache}(F) \neq \texttt{nil}$ **then** return $\texttt{cache}(F)$ // cache check
6: **if** F has more than one connected component **then**
7: $\quad C \leftarrow \tau \wedge \bigwedge_{H \in components(F)} \texttt{Compile}(H, \emptyset)$ // create a decomposable $\wedge$-node
8: **else**
9: $\quad$ **if** $S = \emptyset$ **then** $S \leftarrow \texttt{selectCutset}(F)$
10: $\quad x \leftarrow \texttt{selectVariable}(S, F)$
11: $\quad C \leftarrow \tau \wedge ite(x, \texttt{Compile}(F|x, S - x), \texttt{Compile}(F|\bar{x}, S - x))$ // create a decision node
12: **end if**
13: $\texttt{cache}(F) \leftarrow C$ // cache update
14: return C

Note that, when $i < k$, v_i may be a decision node or an $\wedge$-node. We use $\cdot$ to denote path concatenation. For instance $(v_1, v_2, v_3, v_4) = (v_1, v_2) \cdot (v_3, v_4)$. A variable x appear in a path p if p contains a decision node for x. For $\mathcal{T} = (T, b)$ and $x \in var(F)$ we denote by t_x the highest node of T whose bag contains x, that is, $x \in b(t_x)$, and there is no $t \in V(T)$ such that both $t_x <_T t$ and $x \in b(t)$.

▶ **Definition 1.** *Let F be a CNF formula and let $\mathcal{T} = (T, b)$ be a tree decomposition of G_F. A path p in a decision-DNNF circuit C is* organized *by $\mathcal{T}$ when, for every two decision nodes v_x and v_y on p for the variables x and y, respectively, v_x appears before v_y in p only if $t_y \leq_T t_x$. C is organized by $\mathcal{T}$ when all its paths are organized by $\mathcal{T}$.*

For example, the decision-DNNF circuit of Figure 1 is organized by the tree decomposition shown in the same figure. Readers well-versed in knowledge compilation may know the concept of *structured-DNNF circuits* [22] and may see similarities with our circuits organized by tree decompositions. But they should also note that the circuit shown in Figure 1 is not *structured-decomposable*. Indeed, a structured-DNNF circuit cannot have the two paths highlighted in the figure since they both contain the variables 2, 3, 4, 5 but in different order.

We are show that the output of $\texttt{Compile}_s[smc, bb](F, \mathcal{T})$ is organized by $\mathcal{T}$. We call *residual component of F* any $R \in components(F|\alpha)$ where α is a partial assignment to $var(F)$. Note that G_R is connected.

▶ **Lemma 2.** *Let F be a CNF formula, let $\mathcal{T} = (T, b)$ be a tree decomposition of G_F and let R be a residual component of F such that $var(R) \neq \emptyset$. Then there is a unique highest $t_R \in T$ such that $var(R) \cap b(t_R) \neq \emptyset$.*

Proof. Let $t_1, \ldots, t_k$ be the highest nodes in T whose bags intersect $var(R)$. Suppose, toward a contradiction, that $k > 1$. Then let t be the least common ancestor of $t_1, \ldots, t_k$. t has at least two children c_ℓ and c_r such that some t_i is a descendant of c_ℓ (or is c_ℓ itself) and some t_j, $j \neq i$, is a descendant of c_r (or is c_r itself). By assumption, $var(R) \cap b(t) = \emptyset$. As a general property of tree decompositions, $b(t)$ is a vertex separator of $G_F[b_\downarrow(t)]$ such that no component of $G_F[b_\downarrow(t)] - b(t)$ contains at the same time variables from $b_\downarrow(c_\ell) \setminus b(t)$ and variables from $b_\downarrow(c_r) \setminus b(t)$. So, since G_R is a subgraph of $G_F[b_\downarrow(t)] - b(t)$, we cannot have that G_R is connected and intersect both $b(t_i)$ and $b(t_j)$. This is a contradiction, so $k = 1$. ◀

▶ **Lemma 3.** *Let F be a CNF formula and $\mathcal{T}$ be a tree decomposition of G_F. $\texttt{Compile}_s(F, \mathcal{T})$ returns a decision-DNNF circuit organized by $\mathcal{T}$ and representing F.*

Proof. Here we have no inprocessing and only syntactical caching. Let $\mathcal{T} = (T, b)$ and let C be the output of $\texttt{Compile}_s(F, \mathcal{T})$. The statement follows from Lemma 2 and the fact that, given any residual component R of F, $\texttt{Compile}_s(R, \mathcal{T})$ constructs a decision-DNNF circuit over $var(R)$. Indeed, suppose C contains a path that violates Definition 1, that is, the path contains a decision node v_x for a variable x before the decision node v_y for the decision node y whereas $t_y \not\leq_T t_x$. Let R be the residual component of F for which $\texttt{Compile}_s(R, \mathcal{T})$ constructed the node v_x. We have $\{x, y\} \subseteq var(R)$. By Lemma 2, $\texttt{Compile}_s(R, \mathcal{T})$ must select a variable from $b(t_R)$ and, by uniqueness of t_R, t_R is either t_y or a strict ancestor of both t_x and t_y. In both cases $x \notin b(t_R)$ so $\texttt{Compile}_s(R, \mathcal{T})$ cannot select x and thus does not construct v_x. A contradiction. ◀

▶ Remark 4. Decision-DNNF circuits organized by tree decompositions lie between general decision-DNNF circuits and decision-structured-DNNF circuits (see for instance [5]). In the extreme case where $\mathcal{T}$ is a single bag containing all variables, every decision-DNNF circuit is organized by $\mathcal{T}$. Thus, the "one-bag" tree decomposition, while useless in practice, is the best for $\texttt{Compile}_s$ (more possibility for the output). But this is only because the only constraint for `variableSelect` is to select a variable from the bag. Using the "one-bag" tree decomposition essentially means lifting that constraint.

Lemma 3 still holds when non-deterministic caching is enabled.

▶ **Lemma 5.** *Let F be a CNF formula and $\mathcal{T}$ be a tree decomposition of G_F. Algorithm* $\texttt{Compile}_s[smc](F, \mathcal{T})$ *returns a decision-DNNF circuit organized by $\mathcal{T}$ that represents F.*

Proof sketch. We start with a trivial but key observation. Let $p = (v_1, \dots, v_k)$ and $p' = (v'_1, \dots, v'_h)$ be two paths organized by $\mathcal{T}$. For some $i \in [k-1]$ and some $j \in [h-1]$ let $p_1 = (v_1, \dots, v_i)$, $p_2 = (v_{i+1}, \dots, v_k)$, $p'_1 = (v'_1, \dots, v'_j)$, $p'_2 = (v'_{j+1}, \dots, v'_h)$. If for every variable x appearing in p_1 and every variable y appearing in p'_2 we have $t_y \leq_T t_x$, then the path $p_1 \cdot p'_2$ is organized by $\mathcal{T}$.

Now consider a run of $\texttt{Compile}_s[smc](F, \mathcal{T})$. Suppose every paths constructed up to the call $\texttt{Compile}_s[smc](R, S, \mathcal{T})$, with R a residual component of F, are organized by $\mathcal{T}$. Let p be the path corresponding to the branch that lead to $\texttt{Compile}_s[smc](R, S, \mathcal{T})$ and suppose that for all $y \in var(R)$ and all $x \in var(p)$ we have $t_y \leq_T t_x$. Assume $\texttt{Compile}_s[smc](R, S, \mathcal{T})$ identifies that there exists a CNF formula R' with $C' = \texttt{cache}(R') \neq \texttt{nil}$ such that R is logically equivalent to R' and such that $var(R') \subseteq var(R)$. Since $var(C') \subseteq var(R')$, for all $y \in var(C')$ and $x \in var(p)$, $t_y \leq_T t_x$ holds true. So, by the previous observation, since all paths in C' are organized by $\mathcal{T}$, concatenating a path of C' to p does not create any path not organized by $\mathcal{T}$. So, if $\texttt{Compile}_s(F, \mathcal{T})$ returns a circuit organized by $\mathcal{T}$, then so does $\texttt{Compile}_s[smc](F, \mathcal{T})$. ◀

Now let us further assume that `process` does backbone identification. Given F, $\texttt{process}(F)$ returns $\tau = ite(\ell_1, 1, 0) \wedge \dots \wedge ite(\ell_k, 1, 0)$ where $F \models \ell_i$ for every literal ℓ_i ($k = 0$ and $\tau = 1$ if no such literal exists), and replaces F by $F|\ell_1 \dots \ell_k$. When $k > 0$ the algorithm calls the decision nodes of τ are conjoined to the circuit. We say that these decision nodes have been *inferred from backbone identification.* Let $p = (v_1, v_2, \dots, v_k)$ be a source-to-sink path in C. There can be only one decision node inferred from backbone identification in this path: v_{k-1} (v_k is a sink). Every decision node v_i appearing before v_{k-1} on this path is for a variable y_i that has been selected from a bag of the tree decomposition. Let us call x the variable for v_{k-1}. For any given y_i we have $t_x \leq_T t_{y_i}$. Indeed, let R be the connected residual formula of F such that the call $\texttt{Compile}_s(R, \mathcal{T})$ constructed v_i. By Lemma 2, since $x \in var(R)$, we have that $t_x \leq_T t_R$, and since y_i was selected we also have

$t_R = t_{y_i}$. So we see that adding backbone identification, and a fortiori unit propagation or satisfiability testing, does not change the fact that $\texttt{Compile}_s$ returns a decision-DNNF organized by $\mathcal{T}$.

▶ **Lemma 6.** *Let F be a CNF formula and $\mathcal{T}$ be a tree decomposition of G_F. Algorithm $\texttt{Compile}_s[smc, bb](F, \mathcal{T})$ returns a decision-DNNF circuit organized by $\mathcal{T}$ that represents F.*

We finish this section with a result on the manipulation of decision-DNNF circuits organized by tree decomposition which will be needed in proofs to come. For $S \subseteq V(G)$ we denote by $\mathcal{T} - S$ the tree decomposition of $G - S$ obtained from $\mathcal{T}$ by removing every vertex of S from its bags. Formally, when $\mathcal{T} = (T, b)$, we have $\mathcal{T} - S = (T, b')$ with $b'(t) = b(t) \setminus S$.

▶ **Lemma 7.** *conditioning Let F be a CNF formula let C be a decision-DNNF circuit organized by the tree decomposition $\mathcal{T}$ of G_F. Let α be a partial assignment to F. Then $C|\alpha$ is a decision-DNNF circuit organized by the tree decomposition $\mathcal{T} - var(\alpha)$ of $G_F - var(\alpha)$.*

Proof. We prove the statement when α is an assignment to a single variable x. The lemma then follows by induction. Then $C|\alpha$ is obtained by replacing every decision node $ite(x, v_1, v_0)$ by $v_{\alpha(x)}$. It is readily verified that the paths in $C|\alpha$ are subpaths of paths of C: let $v = ite(x, v_1, v_0)$, then a path $p_0 \cdot (v, v_{\neg\alpha(x)}) \cdot p_1$ in C is not kept in $C|\alpha$ while a path $q = q_0 \cdot (v, v_{\alpha(x)}) \cdot q_1$ in C becomes $q' = q_0 \cdot (v_{\alpha(x)}) \cdot q_1$ in $C|\alpha$. For every variable different from x, the highest node whose bag contains it in $\mathcal{T} - \{x\}$ is the same as in $\mathcal{T}$, so if q is organized by $\mathcal{T}$ then q' is organized by $\mathcal{T} - \{x\}$. ◀

5 Hard Functions for Static Top-Down Compilation

In this section we show that there are CNF formulas that are hard for $\texttt{Compile}_s$ but easy for $\texttt{Compile}_d$.

▶ **Theorem 8.** *There is an infinite class $\mathcal{F}$ of CNF formulas and a constant $\delta \in (0, 1]$ such that, for every $F \in \mathcal{F}$ over n variables and every tree decomposition $\mathcal{T}$ of G_F of width $O(tw(G_F))$, the following holds:*
- *$\texttt{Compile}_d(F)$ returns a decision-DNNF circuit of size $n^{O(1)}$;*
- *$\texttt{Compile}_s[smc, bb](F, \mathcal{T})$ returns a decision-DNNF circuit of size $2^{\Omega(n^\delta)}$.*

One can see that, as n increases, $tw(G_F)$ has to become negligible compared n, because if $tw(G_F) = \Omega(n)$ then the "one-bag" tree decomposition becomes an option and $\texttt{Compile}_s$ essentially finds the smallest decision-DNNF (see Remark 4). One can also guess that $tw(G_F)$ cannot bounded by a constant for all $F \in \mathcal{F}$, because $\texttt{Compile}_s$ should be able to create circuits on size $2^{O(tw(G_F))}n^{O(1)}$. Grid graphs are convenient to get a treewidth that is large enough and yet vanishingly small compared to n. We start with some preliminaries on grid graphs. Then we describe our functions and explain why $\texttt{Compile}_d$ is effective on them. Finally, we prove the lower bound for $\texttt{Compile}_s$.

5.1 Grid Graphs and Spine Graphs

The grid graph $grid_{n,m}$ contains $n \times m$ vertices $\{x_{ij} \mid i \in [n], j \in [m]\}$ connected in n rows and m columns. Grid graphs have a nice well-known properties that we are going to use several time.

▶ **Lemma 9.** *For every fixed $\delta \in [0, 1)$, the $grid_{n,n}$ as no δ-balanced separator of size $o(\sqrt{n})$.*

(a) $G_{row}(2)$. (b) $G_{col}(4)$. (c)

Figure 3 Spine subgraphs of $grid_{4,6}$.

Proof. For $S \subseteq V(grid_{n,n})$, Let $\partial(S) = \{\{u,v\} \in E(grid_{n,n}) \mid u \in S, v \notin S\}$. By [3, Lemma 3], for every S of size $|S| \leq n^2/2$, $|\partial(S)| \geq \min(n, 2\sqrt{|S|})$ holds true. Suppose S is a δ-balanced separator of $grid_{n,n}$. Let $V_1, V_2, \dots$ be the components of $grid_{n,n} - S$. There is V', a union of some components of $grid_{n,n} - S$ such that $|V'| \leq n^2/2$ and $|V'| = \Omega(n^2 - |S|)$. Indeed suppose there is a component V_j of size $\delta n^2/2 \leq |V_j| \leq \delta n^2$. If $|V_j| \leq n^2/2$ we can choose $V' = V_j$ and we are done. If $|V_j| \geq n^2/2$ then we choose $V' = \bigcup_{i \neq j} V_i$ and obtain $|V'| \geq n^2 - |S| - |V_j| \geq (1-\delta)n^2 - |S|$. But when V_j does not exist, we let V' be $V_1 \cup \dots \cup V_i$ for the largest i such that $|V_1 \cup \dots V_i| \leq (n^2 - |S|)/2$, then we have that $|V'| + |V_{i+1}| \geq (n^2 - |S|)/2$ and $|V'| \geq ((1-\delta)n^2 - |S|)/2$. Now, $|V'| \leq n^2/2$ implies that $|\partial(V')| \geq \min(n, 2\sqrt{|V'|}) = \Omega(\sqrt{n^2 - |S|})$. It follows that if $|S| = o(\sqrt{n})$, then $|\partial(V')| \geq \Omega(n)$. So removing $\Omega(n)$ edges is needed to disconnect V' from the rest of the grid. But every vertex in $grid_{n,n}$ has degree at most 4, so removing the vertex set S cannot remove more than $4|S| = o(\sqrt{n})$ edges. So removing S is not enough to disconnect V' from the rest of the grid, a contradiction. ◀

For n and m fixed, we write $row_i = \{x_{i1}, \dots, x_{im}\}$ and $col_j = \{x_{1j}, \dots, x_{nj}\}$. An edge $\{x_{ij}, x_{i\,j+1}\}$ is called an edge of the ith row. An edge $\{x_{ij}, x_{i+1\,j}\}$ is called an edge of the jth column.

▶ **Definition 10** (Spine subgraphs). *A* spine subgraph *of $G = grid_{n,m}$ is a subgraph $G_{row}(i)$, $i \in [n]$ whose edge set comprises all column edges plus all edges of the ith row, or a subgraph $G_{col}(j)$, $j \in [m]$, whose edge set comprises all row edges plus all edges of the jth column. The unique row (resp. column) of $G_{row}(i)$ (resp. $G_{col}(j)$) is called its* spine.

Examples of spine subgraphs are shown in Figures (3a) and (3b). For an unspecified spine subgraph H, we refer to its spine as $spine(H)$. For every $i, k \in [n]$ and $j, \ell \in [m]$, the vertex x_{ij} is at distance $|i-k|$ from $G_{row}(k)$'s spine and at distance $|j-\ell|$ from $G_{col}(\ell)$'s spine.

▶ **Definition 11.** *For $S \subseteq V(grid_{n,m})$ and a spine subgraph H of $grid_{n,m}$, the vertex x_{ij}* sees H's spine despite S *if there is no vertex of $S \setminus \{x_{ij}\}$ on the shortest path connecting x_{ij} to H's spine.*

An example is shown in Figure (3c). The vertices of S are circled in red. The vertices that see the spine despite S are in black. The vertices that do not are in white. Note that not all vertices in S see the spine.

▶ **Lemma 12.** *Let $c > 1$ be a constant, $G = grid_{n,m}$ with $m \geq (3c+1)n$, and $S \subseteq V(G)$ with $n/3 \leq |S| \leq cn$. For n large enough, there is a spine subgraph H of G such that no vertex of S is at distance fewer than 2 from $spine(H)$ and such that at least $\Omega(\sqrt{n})$ vertices of S see $spine(H)$ despite S.*

Proof. Let $I = \{i \mid \exists j, x_{ij} \in S\}$ and let $J = \{j \mid \exists i, x_{ij} \in S\}$ be the indexes of the rows and columns, respectively, that intersect S.

If $|I| < \sqrt{n}/3$, then at least one row indexed in I contains $\sqrt{n}$ vertices of S. So $|J| \geq \sqrt{n}$. Moreover, $|I| < \sqrt{n}/3$ implies that at least $n - \sqrt{n}/3$ rows do not intersect S and thus at least $\frac{1}{3}(n - \sqrt{n})$ rows do not intersect S and are such that they neighboring rows do not intersect S (the neighboring rows of row_i being row_{i+1} and row_{i-1}). Let H be the spine subgraph $G_{row}(i)$ for any of these rows. Then all vertices of S are at distance at least 2 from $spine(H)$. Finally, for every $j \in J$, there is one vertex of $S \cap col_j$ that sees $spine(H)$ despite S.

Now suppose $|I| \geq \sqrt{n}/3$. Since $|S| \leq cn$, at least one of the m columns does not intersect S and is such that its neighboring columns do not intersect S either. Let H be the spine subgraph for that column. Then $spine(H) \cap S = \emptyset$ and all vertices of S are at distance at least 2 from $spine(H)$. Finally for every $i \in I$, there is one vertex of $S \cap row_i$ that sees $spine(H)$ despite S. ◀

5.2 The Hard Functions

We assume the tree decompositions given to $\texttt{Compile}_s$ have minimal width up to a constant factor $\rho \geq 1$ and we write $n' = \lceil 12\rho + 1 \rceil n$. There are $O(n)$ spine subgraphs of $grid_{n,n'}$. In this section, we call them $H_1, H_2, \ldots$ For each H_i we have a CNF formula

$$F_{H_i} = \bigwedge_{\{x,y\} \in E(H_i)} (x \vee y).$$

We introduce $s = \log(n) + O(1)$ variables $Z = \{z_0, \ldots, z_{s-1}\}$ such that 2^s is greater than the number of spine subgraphs. The Z-variables are used as selectors. Every assignment to Z is interpreted as a number between 1 and 2^s by $w(Z) = 1 + \sum_{k=0}^{s-1} z_k 2^k$. We denote by $(w(Z) \neq i)$ the clause satisfied exactly by the assignments α to Z verifying $w(\alpha) \neq i$. Let X be the variables/vertices of $grid_{n,n'}$. We define

$$SelectSpine_n(X, Z) = \bigwedge_{i \in [2^s]} \bigwedge_{c \in F_{H_i}} ((w(Z) \neq i) \vee c).$$

We then introduce a copy of $grid_{n,n'}$ over a new set of variables/vertices X', and we define

$$F_n(X, X', Z) = SelectSpine_n(X, Z) \wedge SelectSpine_n(X', Z). \tag{1}$$

The F_n are our hard functions. Notice that the same selectors are used for the two *SelectSpine* formulas. The reason for copying the variables is to help $\texttt{Compile}_d$ chooses Z as its first cutset. Indeed, the primal graph of F_n looks like Figure 4. Intuitively, Z is the smallest 2/3-balanced separator of this graph: only $O(\log(n))$ vertices, whereas cutting through one of the two grids requires $\Omega(n)$ vertices. So, $\texttt{Compile}_d$ starts by assigning all variables of Z in every possible way. This represents only $O(n)$ branches that each leads to a formula $F_H(X) \wedge F_{H'}(X')$ for two spine subgraphs H and H' over disjoint set of variables. The algorithm create a decomposable $\wedge$-node to deal with F_H and $F_{H'}$ independently and easily compile them into small decision-DNNF circuits.

▶ **Lemma 13.** *For every $n > 0$, $\texttt{Compile}_d(F_n)$ returns a decision-DNNF circuit of size $n^{O(1)}$, where $F_n(X, X', Z) = SelectSpine_n(X, Z) \wedge SelectSpine_n(X', Z)$ is defined as in (1).*

Proof sketch. The primal graph of $F_n = SelectSpine_n(X, Z) \wedge SelectSpine_n(X', Z)$ is connected. So $\texttt{Compile}_d(F_n)$ first computes a minimal-size 2/3-balanced separator of G_{F_n}. Z is a 2/3-balanced cutset of G_{F_n} so a minimal-size 2/3-balanced cutset S of G_{F_n} contains no more than $O(\log(n))$ variables.

Figure 4 The primal graph of $\mathit{SelectSpine}_n(X, Z) \wedge \mathit{SelectSpine}_n(X', Z)$.

Suppose S does not contain Z in its entirety. By Lemma 9, there are no 3/4-balanced separator of size $O(\log(n))$ of an $n \times n$ grid. Let G_X be the X-grid and $G_{X'}$ be the X'-grid. Both $G_X - S$ and $G_{X'} - S$ contain a connected component of size at least $3nn'/4$. Let Γ_X and $\Gamma_{X'}$ be these components. But then Γ_X and $\Gamma_{X'}$ are connected to a vertex $z \in Z \setminus S$ in $G_{F_n} - S$ and thus $G_{F_n} - S$ has a connected component of size at least $6nn'/4$. The number of variables of F_n is $2nn' + O(\log(n))$ and $6nn'/4 > 4nn'/3 + O(\log(n))$ for n large enough. So S is not a 2/3 balanced separator if it does not contain Z. And since Z is a 2/3-balanced cutset of G_{F_n}, we have $S = Z$.

As long as one Z-variable is left unassigned, the primal graph of the formula remains connected and, since there is no inprocessing mechanisms here, $\texttt{Compile}_d(F_n)$ keeps assigning all Z-variables in all possible way without creating decomposable $\wedge$-nodes. This amounts for $O(n)$ branches in the algorithms. When, at the end of a branch, all Z-variables are assigned the algorithm makes a call $\texttt{Compile}_d(F_H \wedge F_{H'})$ with H a spine subgraph of the X-grid and H' a spine subgraph of the X'-grid. A decomposable $\wedge$-node is created and the algorithm calls $\texttt{Compile}_d(F_H)$ and $\texttt{Compile}_d(F_{H'})$. H and its connected subgraphs all have 2/3-balanced separators of constant size, so $\texttt{Compile}_d(F_H)$ only need $O(\log(|H|)) = O(\log(n))$ recursive calls to finish. ◀

5.3 Lower Bounds for $\texttt{Compile}_s$

We now prove a $2^{\Omega(\sqrt{n})}$ lower bound on the size of the decision-DNNF circuits returned by $\texttt{Compile}_s$ for F_n when given a tree decomposition $\mathcal{T} = (T, b)$ of G_{F_n} of width at most $\rho \cdot tw(G_{F_n})$.

We give some intuition for why the lower bound holds true. Just like $\texttt{Compile}_d$, $\texttt{Compile}_s$ can assign the Z-variables first if they are in the highest bags of $\mathcal{T}$, then it would find $O(n)$ subformulas of the form $F_H \wedge F_{H'}$ with H a spine subgraph of the X-grid and H' a spine subgraph of the X'-grid. In such a situation, $\texttt{Compile}_s$ will create $\wedge$-nodes and compile F_H and $F_{H'}$ separately. Let $H_1, H_2, \dots, H_{2n}$ be the spine subgraphs of the X-grid. The problem is that $\texttt{Compile}_s$ uses the *same* tree decomposition to compile $F_{H_1}, F_{H_2}, \dots, F_{H_{2n}}$. For each $i \in [2n]$, there is indeed a tree decomposition $\mathcal{T}$ of the X-grid such that $\texttt{Compile}_s(F_{H_i}, \mathcal{T})$ constructs a small circuit, but there is no tree decomposition of the X-grid that simultaneously give a small circuit for all i. And $\texttt{Compile}_s$ is stuck with a unique tree decomposition of G_{F_n} (which contains a tree decomposition of the X-grid), so for some F_{H_i} the circuit constructed will be large.

Of course the lower bound has to be proved even in cases where the Z variables are not assigned first. For $\mathcal{T}$ fixed, we give a spine subgraph H such that F_H does not admit small decision-DNNF circuits organized by $\mathcal{T}$. Then we will just use Lemma 7 to extract in from a decision-DNNF circuit C representing F_n and organized by $\mathcal{T}$, a decision-DNNF circuit C' representing F_H and organized by $\mathcal{T}$. If C' is large, then C must be large and we will be done.

For a given path p of T we let $b(p) = \bigcup_{t \in p} b(t)$.

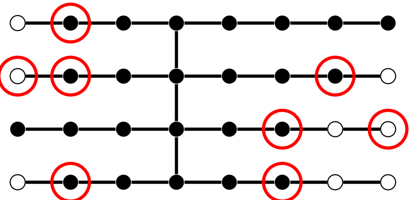
▶ **Lemma 14.** *Let $\mathcal{T}$ be a tree decomposition of width $w > 0$, there is a path p from its root to a node such that $w \leq |b(p)| \leq 2w$.*

Proof. Let t be one of the highest nodes of T with $|b(t)| = w$ and let q be the path from T's root to t. For every node $s \in q$, let q_s be the path from T's root to s. For two consecutive s, s' in q, s before s', we have $|b(q_{s'})| - |b(q_s)| \leq w$. So, since $|b(q_t)| = |b(q)| \geq |b(t)| = w$, there must be a node $s \in q$ such that $w \leq |b(q_s)| \leq 2w$. ◀

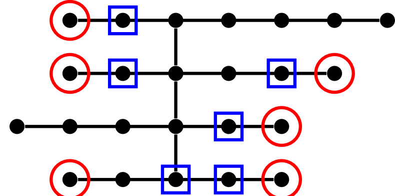
It is known that $tw(grid_{n,n}) = n$ and that there are tree decompositions of $grid_{n,n'}$ of width $n + 1$, so $tw(grid_{n,n'}) = n$. It is not hard to see that $tw(F_n)$ is at least n and at most $n + O(\log(n))$ and thus, the width of $\mathcal{T}$ is between n and $2\rho n$ for n large enough. The set $b(p)$ from Lemma 14 for $\mathcal{T}$ then contains between n and $4\rho n$ variables from X, X' and Z. There are only $O(\log(n))$ variables in Z so $|b(p) \cap X| \geq n/3$ or $|b(p) \cap X'| \geq n/3$ (for n large enough). Assuming $|b(p) \cap X| \geq n/3$ holds and setting $S = b(p) \cap X$, we claim that the spine subgraph H, given by Lemma 12 for S in the X-grid, is such that F_H has no small decision-DNNF circuit organized by $\mathcal{T}$.

▶ **Lemma 15.** *For every tree decomposition $\mathcal{T}$ of width at most $2\rho n$ of G_{F_n}, there is a spine subgraph H of the X-grid or of the X'-grid such that all decision-DNNF circuits that represent F_H and are organized by $\mathcal{T}$ have size $2^{\Omega(\sqrt{n})}$.*

The proof appears in the next section. We try to give a high-level idea, suppose H is the spine subgraph represented on the left of Figure 5a. The S-vertices/variables are circled in red. Since H is found using Lemma 12, many S-vertices see $spine(H)$ despite S and all S-vertices are at distance at least 2 from $spine(H)$. If we have a decision-DNNF circuit C organized by $\mathcal{T}$ for F_H, then we can assign all variables that do not see the spine to 1 (the white vertices on the figure) and we obtain another decision-DNNF circuit C' organized by $\mathcal{T}$ (by Lemma 7) for the formula $F_{H'}$ where H' is shown on the right. Only the S-variables that see the spine remain, call them S'. In C', these variables are assigned first. But one cannot disconnect H' by removing any subset of S'. Also, unit propagation or backbone identification can only infer the value for the direct neighbors of the S' variables, but these neighbors are not on the spine and so their removal along with S' will let the graph connected. So we are essentially condemn to assign all the S'-variables and sometimes their neighbors in H', and this create $2^{\Omega(|S'|)} = 2^{\Omega(\sqrt{n})}$ nodes in C'.



(a) S-vertices are circled. Black vertices see the spine despite S.



(b) The induced subgraph $H[V]$. S'-vertices are circled. $\eta(S')$-vertices are squared.

Figure 5

So to prove Theorem 8, the upper bound on $\texttt{Compile}_d$'s output is Lemma 13, and the lower bound on $\texttt{Compile}_s$'s output follows from Lemmas 6, 7 and 15 since, from the output of $\texttt{Compile}_s[smc, bb](F_n, \mathcal{T})$, one obtains a decision-DNNF organized by $\mathcal{T}$ for F_H by assigning the Z-variables to select only the clauses of F_H.

5.4 Proof of Lemma 15

Before starting, let us make a simple observation on the formulas defined over graphs like F_H. We say that a formula F can be *decomposed* if it can be written as $f \wedge g$ where f and g are Boolean functions with $var(f) \neq \emptyset$, $var(g) \neq \emptyset$, and $var(f) \cup var(g) = \emptyset$.

▷ Claim 16. Let G be a connected graph whose vertices are seen as variables. The formula $F_G = \bigwedge_{\{x,y\} \in G} x \vee y$ cannot be decomposed. In particular, its backbone is empty.

Proof. Consider a partition (X_1, X_2) of $var(F_G) = V(G)$ where X_1 and X_2 are non-empty. We prove that $F_G \not\equiv f(X_1) \wedge g(X_2)$. We have an edge (x_1, x_2) of G such that $x_1 \in X_1$ and $x_2 \in X_2$. If $F_G \equiv f \wedge g$ then we cannot have that f has a model where x_1 is set to 0 and that at the same time g has a model where x_2 is set to 0, for otherwise $f \wedge g$ would have a model that falsifies $x_1 \vee x_2$. Since F_G has models where x_1 is set to 0 and others where x_2 is set to 0, F_G is not equivalent to $f(X_1) \wedge g(X_2)$. ◁

As explained before, we use the path p from Lemma 14 and we assume, without loss of generality, that $|b(p) \cap X| \geq n/3$. Let H be the spine subgraph given by Lemma 12 for the X-grid and $S = b(p) \cap X$. We call S' the vertices of S that see $spine(H)$ despite S. By Lemma 12, we have $|S'| = \Omega(\sqrt{n})$. Let $V \subseteq X$ be the set of vertices of the whole grid that see $spine(H)$ despite S and let $\bar{V} = X \setminus V$. It holds that $spine(H) \cup S' \subseteq V$. For instance, if H is the spine subgraph represented in Figure (5a) with the S-vertices circled, then the black vertices are V. $H[V]$ is then the subgraph represented in Figure (5b).

Let C be a decision-DNNF circuit organized by $\mathcal{T}$ and let β be the assignment to $Z \cup X' \cup \bar{V}$ that sets all variables of $\bar{V} \cup X'$ to 1 and that sets Z to select the subgraph H. By Lemma 7, $C' = C|\beta$ is a decision-DNNF circuit computing $F_n|\beta = F_H|\beta = F_{H-\bar{V}} = F_{H[V]} = \bigwedge_{\{x,y\} \in E(H[V])} (x \vee y)$, and that is organized by $\mathcal{T}' = (T, V \cap b)$. We write $b' = V \cap b$. We prove several intermediate claims.

For t a node in p, p_t is the subpath of p from the root node of T to t. Recall that $b(p) = \bigcup_{t \in p} b(t)$.

▷ Claim 17. Let t be a node of p such that $b'(p_t) \subseteq S'$. If t has a child c not in p such that $b'_\downarrow(c) \nsubseteq S'$, then for all $x \in S'$, $t \leq_T t_x$.

Proof. $S' \subseteq b(p)$ so, for all $x \in S'$, $t_x \in p$. Suppose there exists $x \in S'$ and $t \in p$ such that $t_x <_T t$ and t has a child $c \notin p$ with $b'_\downarrow(c) \nsubseteq S'$. We have that $c \notin p$. Let $y \in b'_\downarrow(c) \setminus S'$. Then $x \notin b'(t)$ and, since $b'(t) \subseteq b'(p_t) \subseteq S'$, we have $y \notin b'(t)$. So there are variables not in $b'(t)$ appearing in bags under two distinct children of t and thus $b'(t)$ is a separator of $H[V]$ such that $H[V] - b'(t)$ has two non-empty components. But that cannot be, because $b'(t) \subseteq S'$ and neither S' nor any of its subset is a separator of $V[H]$. ◁

p contains the root r of T and $b(p) \cap X = S$, so $b'(r) \subseteq S'$ holds. So by Claim 17, on any path from C''s root to a sink, the variables of S' appear first: every path q from C''s source to the 1-sink, can be written $q' \cdot q''$ where only variables of S' appear in q' and where the q'' contains no variable of S'.

There can be $\wedge$-nodes in q'. But there are particular $\wedge$-nodes. A decomposable $\wedge$-node is called an $\wedge^*$-node if exactly one of its children does not compute a term, i.e., a conjunction of literals. The child in question is called the *non-term part* of the node, and the term obtained as the conjunction of all terms under all other children is called the *term part* of the node. For instance, Figure 2 represents a $\wedge^*$-node: the first three children all represent terms but the fourth one (on the right) does not. We are going to prove that every $\wedge$-node appearing in q' is $\wedge^*$-node

For $v \in S'$, we denote by $\eta(v)$ its unique neighbor in $H[V]$ and for any given $S'' \subseteq S'$ we write $\eta(S'') = \{\eta(v) \mid v \in S''\}$. See for instance Figure (5b).

▷ Claim 18. Let α be a partial assignment to S'. $F_{H[V]}|\alpha$ is equivalent to $\tau_\alpha \land F_{H_\alpha}$ where $\tau_\alpha = \bigwedge_{\alpha(v)=0} \eta(v)$ and $H_\alpha = H[V] - (var(\alpha) \cup var(\tau))$ is connected, and F_{H_α} cannot be decomposed.

Proof. The clause $v \lor \eta(v)$ forces that if $\alpha(v) = 0$, all models of $F_{H[V]}|\alpha$ must assign $\eta(v)$ to 1. For convienence, see τ_α as the assignment $\eta(v)$ that sets $\eta(v)$ tp 1 for all v such that $\alpha(v) = 0$. Then $F_{H[V]}|\alpha = \tau_\alpha \land F_{H[V]}|\tau_\alpha = \tau_\alpha \land F_{H[V]-var(\alpha)-var(\tau_\alpha)} = \tau_\alpha \land F_{H_\alpha}$. Since all vertices of S' are at distance 2 from $spine(H)$, we have that $H[V] - var(\alpha) - var(\tau_\alpha)$ is connected. So by Claim 16, F_{H_α} is not decomposable. ◁

▷ Claim 19. Let $q = (v_1, \dots, v_m) = q' \cdot q''$ be a path in C' with only variables of S' appearing in q' and no variables of S' appearing in q''. Let γ be the (partial) assignment to S' corresponding to q'. Then every $\land$-node v_l in q' is a $\land^*$-node whose term part is a subterm of $\tau_\gamma = \bigwedge_{\gamma(v)=0} \eta(v)$.

Proof. Let v_k be the first $\land$-node in q' and let α be the assignment corresponding to $(v_1, \dots, v_k)$. By Claim 18, v_k is a $\land^*$-node whose term part is $\tau_\alpha = \bigwedge_{\alpha(v)=0} \eta(v)$ or a subterm of τ_α, which is itself a subterm of τ_γ.

Now suppose v_ℓ is some $\land$-node in q', let again α be the assignment corresponding to $(v_1, \dots, v_\ell)$ and suppose all $\land$-nodes before v_ℓ on that path are $\land^*$-nodes whose term parts are subterms of τ_α. By Claim 18, $F_{V[H]}|\alpha$ is equivalent to $F_{H_\alpha} \land \tau_\alpha$ where F_{H_α} is not decomposable, so C'_{v_ℓ} computes $F_{H_\alpha} \land \tau'$ where τ' is τ_α minus all term parts of all $\land^*$-nodes before v_ℓ. So v_ℓ is a $\land^*$-node whose term part is τ' or a subterm of τ', and therefore a subterm of τ_γ. ◁

$F_{H[V]}|\gamma$ is satisfiable for every *complete* assignment γ to S' so, for every γ, we can construct a path q as follows: starting from v_1, if the current node v is a decision node for $x \in S'$, follows the $\gamma(x)$-child of v, if instead v is a $\land^*$-node then follow the non-term child, and otherwise adds v to q and stops. This path is unique. We call it q_γ. We claim that q_γ is not missing any variable of S'

▷ Claim 20. For every complete assignment γ to S', q_γ contains one decision node for every variable in S'.

Proof. Let v be the last node of q_γ. Let α be the variable assignment to S' corresponding to q_γ. Clearly α is consistent with γ, but we have to show that $var(\alpha) = var(\gamma)$. By Claim 18, $F_{H[V]}|\alpha$ is equivalent to $F_{H_\alpha} \land \tau_\alpha$ where $H_\alpha = H[V] - (var(\alpha) \cup var(\tau_\alpha))$. By Claim 19, the term part of the $\land^*$-nodes in q are subterms of τ_α. So C'_v represents a function $F_{H_\alpha} \land \tau'$ for τ' a subterm of τ_α (τ' is possibly empty). C'_v has models and counter models so v is not a sink, it also not a $\land^*$-node by construction of q_γ, nor is it a more general decomposable $\land$-node because F_{H_α} is not decomposable. So v is a decision node for a variable y, and $y \notin S'$ by construction of q_γ. If α is not a complete assignment of S', then H_α contains a variable/vertex x of S'. x is essential in F_{H_α} so it must appear in C'_{v_m}. Let y be the variable for v_m. Since $y \notin S'$, we deduce from Claims 17 that $t_y <_T t_x$. So x cannot exist for otherwise C' would not be organized by $\mathcal{T}'$. ◁

Now, the circuit C'_v rooted under the last node v of q_γ computes $F_{H_\gamma} \land \tau$, where τ is τ_γ or a subterm of τ_γ. We can then show that the functions computed are logically distinct for distinct γ.

▷ Claim 21. Let γ and γ' be distinct complete assignments to S' and let v and v' be the last nodes of q_γ and q'_γ, respectively, then C'_v and $C'_{v'}$ are not logically equivalent

Proof. Let $x \in S'$ be a variable where $\gamma(x) \neq \gamma'(x)$. Say $\gamma(x) = 1$. C'_v computes $F_{H'} \wedge \tau$ for τ a subterm of $\tau_\gamma = \bigwedge_{\gamma(v)=0} \eta(v)$ and $H' = H[V] - S' - var(\tau)$. Note that $\eta(x)$ is in H' so $F_{H'}$ and C'_v essentially depend on $\eta(x)$ and have models where $\eta(v)$ is set to 0. $C'_{v'}$ computes $F_{H''} \wedge \tau'$ for τ' a subterm of $\tau_{\gamma'} = \bigwedge_{\gamma'(v)=0} \eta(v)$ and $H'' = H[V] - S' - var(\tau')$. Either $\eta(x)$ is in τ' and then $C'_{v'} \models \eta(x)$, or $\eta(x)$ is not in τ' and then $C'_{v'}$ does not depend on $\eta(x)$. In both cases, C'_v and $C'_{v'}$ are not logically equivalent. ◁

So we have $2^{|S'|}$ paths (one per γ) each containing a unique node. So C' contains at least $2^{|S'|} = 2^{\Omega(\sqrt{n})}$ nodes and this finishes the proof of Lemma 15.

6 Hard Functions for Dynamic Top-Down Compilation

We show the opposite variant of Theorem 8.

▶ **Theorem 22.** *There is an infinite class $\mathcal{F}$ of CNF formulas and a constant $\delta \in (0, 1]$ such that, for every $F \in \mathcal{F}$ over n variables we have $tw(G_F) = o(n)$ and*

- $\texttt{Compile}_{d,\epsilon}[smc, bb](F)$ *returns a decision-DNNF circuit of size $2^{\Omega(n^\delta)}$;*
- *there exists a tree decomposition $\mathcal{T}$ of width $O(tw(G_F))$ such that $\texttt{Compile}_s(F, \mathcal{T})$ returns a decision-DNNF circuit of size $n^{O(1)}$.*

There is an asymmetry compared to Theorem 8 though: Theorem 22 is a positive result for the static compilation approach for *some* well-chosen tree decomposition of close-to-minimal width. The result does not hold for all tree decompositions of with the same width.

For the proof we will have $\delta = 1/4$. We have not looked to optimize this exponent and we could probably do better with formulas more cleverly crafted (but probably more complex). The point here is just to show that one algorithm has a polynomial size output while the other does not.

It is counter intuitive that, in our settings where many aspects of the algorithms are non-deterministic, the dynamic approach can be outperformed by its static counterpart. The idea is to design formulas where there is a clear optimal order to evaluate variables that can be hinted to the static approach via the tree decomposition, while the dynamic approach cannot take advantage of it. The hard formulas are of the form

$$F_n(X, X', X'') = L_n(X, X') \wedge R_n(X, X'')$$

L_n and R_n only share the X-variables. Roughly put, L_n's role is to ensure that the first cutset selected by $\texttt{Compile}_{d,\epsilon}$ contains X, and R_n is a formula that is hard to compile when the X-variables are the first selected to be assigned. There are many possibilities for L_n and R_n. Let us start with the formula chosen for R_n.

$$R_n(X, Y, Z) = \bigwedge_{i \in [0, n-1]} \bigwedge_{j \in [n]} (\bar{x}_{i+1} \vee y_{j+in} \vee z_{j+in}) \wedge (\bar{z}_1 \vee \cdots \vee \bar{z}_{n^2})$$

with $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_{n^2}\}$ and $Z = \{z_1, \ldots, z_{n^2}\}$ (so $X'' = Y \cup Z$).

▶ **Lemma 23.** *Let S such that $X \subseteq S$ and $|S| = O(n)$. The decision-DNNF circuit returned by $\texttt{Compile}_{d,\epsilon}[smc, bb](R_n, S)$ has size at least $2^{\Omega(n)}$.*

Proof. We call a clause $\bar{x}_{i+1} \vee y_{j+in} \vee z_{j+in}$ a small clause and $\bar{z}_1 \vee \cdots \vee \bar{z}_{n^2}$ the big clause. Since $|S| = O(n)$, we have that $|(Y \cup Z) \setminus S| = \Omega(n^2)$. Now let $I \subseteq [n]$ such that $i \in I$ if and only if $z_{j+in} \notin S$ and $y_{j+in} \notin S$ for some $j \in [n]$. Since $|S| = O(n)$, we have that $|I| = \Omega(n)$.

Let β be the assignment that maps all variables in $(Y \cup Z) \cap S$ to 1 and all x_i for $i \notin I$ to 0. For α a partial assignment to S consistent with β, the graph of $R_n|\alpha$ remains connected because α does not satisfies the big clause. The small clauses that remain in $R_n|\alpha$ are of the form $\bar{x}_{i+1} \vee y_{j+in} \vee z_{j+in}$ or $y_{j+in} \vee z_{j+in}$. We claim that the backbone of $R_n|\alpha$ is empty. To show this, it is sufficient to describe two families of assignments that satisfy $R_n|\alpha$:

- all assignments that set all Y-variables to 1 and one remaining Z-variable to 0 satisfy $R_n|\alpha$;
- for any $y_j \in var(R_n|\alpha)$, z_j must be in $var(R_n|\alpha)$ (because they appear together in a small clause), and there must be $z_{j'} \in var(R_n|\alpha) \setminus \{z_j\}$ (because S is too small to leave $R_n|\alpha$ with only one Z-variable) so assigning y_j to 0, z_j to 1, $z_{j'}$ to 0 and all remaining variables to 1 satisfies $R_n|\alpha$.

So, in each branch followed by $\texttt{Compile}_{d,\epsilon}[smc, bb](R_n, S)$ that leads to the residual formula $R_n|\alpha$ for some α consistent with β, no decomposable $\wedge$-nodes are created because the graph always stays connected, and `process` has no effect because the backbone is always empty. So for each *complete* assignment α to S consistent with β, the corresponding branch of the algorithm creates only decision nodes. For any two distinct *complete* assignments α and α' to S consistent with β, if α and α' disagree on a variable, then it is some X-variable x_i for $i \in I$ and we have that $R_n|\alpha \not\equiv R_n|\alpha'$ because one formula essentially depends on some $y_{j+in} \notin S$ while the other does not. So even with semantical caching, the circuit contains at least one distinct node per α, so at least $2^{|I|} = 2^{\Omega(n)}$ nodes. ◀

Now for L_n consider two disjoint $n^2 \times n^2$ grids G_1 and G_2 with vertices $V = \{v_{ij} \mid i,j \in [n^2]\}$ and $U = \{u_{ij} \mid i,j \in [n^2]\}$, and the two formulas $F_{G_1}(V) = \bigwedge_{(v,v') \in E(G_1)} (v \Leftrightarrow v')$ and $F_{G_1}(U) = \bigwedge_{(u,u') \in E(G_2)} (u \Leftrightarrow u')$. Then

$$L_n(X,U,V) = \bigwedge_{c \in F_{G_1} \wedge F_{G_2}} c \vee x_1 \vee \cdots \vee x_n \equiv (F_{G_1} \wedge F_{G_2}) \vee x_1 \vee \cdots \vee x_n$$

F_{G_1} and F_{G_2} are trivial. Their only models are the assignments where all V-variables and all U-variables are set to 0 or 1. Arguably we could have designed less trivial formulas. Most formulas with a grid-like primal graphs that are easy to compile would be acceptable substitutes. We are only interested in their primal graphs for the following lemma.

▶ **Lemma 24.** *For n large enough, every 2/3-balanced separator S of $G_{L_n \wedge R_n}$ of size $O(n)$ contains X.*

Proof. Suppose S does not contain X in its entirety. By Lemma 9, there are no 3/4-balanced separator of G_1 or G_2 of size $O(n)$. Thus both $G_{F_1} - S$ and $G_{F_2} - S$ contain a connected component of size at least $3n^4/4$. Let H_1 and H_2 be these components. But then H_1 and H_2 are connected to a vertex $x \in X \setminus S$ in $G_{L_n \wedge R_n} - S$ and thus $G_{L_n \wedge R_n} - S$ has a component of size at least $6n^4/4$. The number of variables of $L_n \wedge R_n$ is $2n^4 + O(n^2)$ and $6n^4/4 > 4n^4/3 + O(n^2)$ for n large enough. So S is not a 2/3 balanced separator of $L_n \wedge R_n$. ◀

It is readily verified that the backbone of $L_n \wedge R_n$ is empty and that its graph is connected, so the first task of $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n)$ is to find a balanced separator S of minimal size up to a factor $(1+\epsilon)$. Thus $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n)$ returns the same circuit as $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n, S)$. By Lemma 24, the chosen separator S contains X and has

size $O(n)$. If $S \subseteq X \cup Y \cup Z$ then by Lemma 23 we have that $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n)$ returns a circuit of size $2^{\Omega(n)}$. Even if S contains some U- and V-variables, we can show that the same lower bounds hold. This is because assigning U- and V-variables cannot disconnect the primal graph as long as some X-variables are left unassigned and the impact of assigning such variables on the backbone of the formula is quasi-null.

▶ **Lemma 25.** *$\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n)$ returns a circuit of size $2^{\Omega(n)}$.*

Proof sketch. By Lemma 24, $\texttt{Compile}_{d,\epsilon}(L_n \wedge R_n)$ selects a separator S of size $O(n)$ that contains X. S may contain some U- and V-variables but, once an X-variable is set to 1, all clauses of L_n disappear and all U- and V-variables with them. $\texttt{Compile}_{d,\epsilon}(L_n \wedge R_n)$ returns the same circuit as $\texttt{Compile}_{d,\epsilon}(L_n \wedge R_n, S)$.

We follow the proof of Lemma 23. The set I and the assignment β are defined the same way. For α a partial assignment to S consistent with β and that either assigns an X-variable to 1 or does not assign two X-variables, the graph of $(L_n \wedge R_n)|\alpha$ remains connected. Indeed α does not satisfies the big clause of R_n because it is consistent with β, and if it assigns an X-variable to 1 then the part of the graph for L_n vanishes, otherwise if some X-variable is left unassigned then L_n's graph and R_n's graph remains connected through that variable.

We claim that the backbone of $(L_n \wedge R_n)|\alpha$ is empty. If α sets an X-variable to 1 then $(L_n \wedge R_n)|\alpha = R_n|\alpha$ so we can just use the proof of Lemma 23. Otherwise, α sets all its X-variables to 0 but let at least 2 unassigned. In this case, clauses of $L_n|\alpha$ still exist and contain two X-variables x and x'. We can then extend the assignments to the Y-variables and the Z-variables described in the proof of Lemma 23 by setting one of x or x' to 0 and the other to 1. Such assignments satisfy $L_n \wedge R_n|\alpha$ and suffice to show that no X-, Y-, Z-, U- or V- literals can be in the backbone of $L_n \wedge R_n|\alpha$.

So, each branch followed by $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n, S)$ for one of the assignment α described above leads to the residual formula $(L_n \wedge R_n)|\alpha$ without creating a single decomposable $\wedge$-node nor benefiting from `process`. In particular, along each branch for each a *complete* assignment α to S' that is consistent with β and assigns at least one X-variable to 1, the algorithm creates only decision nodes. Since for any two such complete assignments α and α' we have that $(L_n \wedge R_n)|\alpha = R_n|\alpha$ and $(L_n \wedge R_n)|\alpha' = R_n|\alpha'$, and since we know by the proof of Lemma 23 that $R_n|\alpha \not\equiv R_n|\alpha'$, we deduce that each branch for these α contains a unique node. So there is are least $2^{|I|} - 1 = 2^{\Omega(n)}$ nodes (-1 because we remove the assignment where all X-variables are set to 0). ◀

It remains to study $\texttt{Compile}_s$ on $L_n \wedge R_n$. There is a tree decomposition we can use to force $\texttt{Compile}_s$ to assign the variables in the right order π, namely:

$$\begin{aligned}\pi : &x_1, y_1, z_1, y_2, z_2, y_3, z_3, \dots, y_n, z_n,\\ &x_2, y_{1+n}, z_{1+n}, y_{2+n}, z_{2+n}, y_{3+n}, z_{3+n}, \dots, y_{2n}, z_{2n}, \dots\\ &x_n, y_{1-n+n^2}, z_{1-n+n^2}, y_{2-n+n^2}, z_{2-n+n^2}, \dots, y_{n^2}, z_{n^2}, \dots\end{aligned}$$

where the U-variables and the V-variables are put at the end in any order. This is basically the reading order of the X, Y, Z-variables as they appear in R_n. To force $\texttt{Compile}_s$ to read the variable in that order, it suffices to use a *nice* path decomposition, that is, a tree decomposition $\mathcal{T} = (T, b)$ where T is a path and where, first the root bag is empty and, second, for every two consecutive nodes t and t' in T, $b(t) = b(t') \cup \{x\}$ or $b(t) = b(t') \setminus \{x\}$ for some variable x. In a nice path decomposition, the function $x \mapsto t_x$ is injective (recall that t_x is the highest node of T whose bag contains x): there is a total order σ in which the variables appear in the bags of $\mathcal{T}$. So a decision-DNNF circuit organized by a nice path decomposition is guaranteed that on all its paths, the ordering of the variables for the decision nodes is consistent with σ. We just have to make sure that $\sigma = \pi$.

We use a nice path decomposition where, in a nutshell, the X, Y, Z-variables are added to the bags in the order given by π until we obtain one big bag containing $X \cup Y \cup Z$, then $Y \cup Z$ are removed and the remaining is an $O(n^2)$-width path decomposition of G_1 with X added to all bags, followed by an $O(n^2)$-width path decomposition of G_2 with X is added to all bags. The treewidth of $L_n \wedge R_n$ is $\Theta(n^2)$ because of the $n^2 \times n^2$ grids in L_n, so the path decomposition has width $O(tw(G_{L_n \wedge R_n}))$.

▶ **Lemma 26.** *There exists a tree decomposition $\mathcal{T}$ of width $O(tw(G_{L_n \wedge R_n}))$ such that* $\texttt{Compile}_s(L_n \wedge R_n, \mathcal{T})$ *returns a circuit of size $n^{O(1)}$.*

Proof sketch. In the circuit returned by $\texttt{Compile}_s(L_n \wedge R_n, \mathcal{T})$, the decision nodes follow the order π. The clauses $(\bar{x}_i \vee y_{j+in} \vee z_{j+in}$ are then falsified or satisfied in increasing order of i and j. For every assignment α to $\{x_i, y_{j+in} \mid i < k \text{ and } j + in \leq h\}$, only a constant number of formulas $(L_n \wedge R_n)|\alpha$ are possible, so syntactical caching ensures that there are only a constant number of decision nodes labeled by the same variable in the circuit. ◀

Theorem 22 follows from the combination of Lemmas 25 and 26.

7 Conclusion

We have studied the relative efficiency of two top-down compilation algorithms. Both use variable selection mechanisms inspired from practical compilers. They both select batches of variables that correspond to separators of the primal graph, but one uses a dynamic graph balanced-partitioning approach while the other relies on a pre-computed tree decomposition. We have shown that the two algorithms construct large decision-DNNF circuits on instances that yet admit polynomial-size decision-DNNF circuits. Moreover we have shown that there are instances where only one of two approaches fails to construct a small circuit. Often in knowledge compilation, we compare languages saying that they offer small compiled forms for different kind of functions. What we show here is that even within the same language, similar comparisons are possible for the compilation algorithms: the classes of formulas for which two compilers to decision-DNNF are able to find polynomial-size compiled forms can be distinct and not included in one another. This calls for criterion for deciding to which compilers an instance should be sent.

References

1. Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #sat and bayesian inference. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 340–351. IEEE Computer Society, 2003. doi:10.1109/SFCS.2003.1238208.
2. Paul Beame, Jerry Li, Sudeepa Roy, and Dan Suciu. Lower bounds for exact model counting and applications in probabilistic databases. In Ann E. Nicholson and Padhraic Smyth, editors, *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013, Bellevue, WA, USA, August 11-15, 2013*. AUAI Press, 2013. URL: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2366&proceeding_id=29.
3. Béla Bollobás and Imre Leader. Edge-isoperimetric inequalities in the grid. *Comb.*, 11(4):299–314, 1991. doi:10.1007/BF01275667.
4. Marco Cadoli, Francesco M. Donini, Paolo Liberatore, and Marco Schaerf. Preprocessing of intractable problems. *Inf. Comput.*, 176(2):89–120, 2002. doi:10.1006/INCO.2001.3043.
5. Andrea Calì, Florent Capelli, and Igor Razgon. Non-fpt lower bounds for structural restrictions of decision DNNF. *CoRR*, abs/1708.07767, 2017. arXiv:1708.07767.

6 Hubie Chen. Parameterized compilability. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 412–417. Professional Book Center, 2005. URL: http://ijcai.org/Proceedings/05/Papers/0644.pdf.

7 Adnan Darwiche. Recursive conditioning. *Artif. Intell.*, 126(1-2):5–41, 2001. doi:10.1016/S0004-3702(00)00069-2.

8 Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In Rina Dechter, Michael J. Kearns, and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 627–634. AAAI Press / The MIT Press, 2002. URL: http://www.aaai.org/Library/AAAI/2002/aaai02-094.php.

9 Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 328–332. IOS Press, 2004.

10 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002. doi:10.1613/JAIR.989.

11 Hélène Fargier, Pierre Marquis, Alexandre Niveau, and Nicolas Schmidt. A knowledge compilation map for ordered real-valued decision diagrams. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 1049–1055. AAAI Press, 2014. doi:10.1609/AAAI.V28I1.8853.

12 Hélène Fargier and Jérôme Mengin. A knowledge compilation map for conditional preference statements-based languages. In Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé, editors, *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, pages 492–500. ACM, 2021. doi:10.5555/3463952.3464014.

13 Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 156–162. Professional Book Center, 2005. URL: http://ijcai.org/Proceedings/05/Papers/0876.pdf.

14 Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In Henry A. Kautz and Bruce W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, pages 157–162. AAAI Press / The MIT Press, 2000. URL: http://www.aaai.org/Library/AAAI/2000/aaai00-024.php.

15 Rafael Kiesel and Thomas Eiter. Knowledge compilation and more with sharpsat-td. In Pierre Marquis, Tran Cao Son, and Gabriele Kern-Isberner, editors, *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*, pages 406–416, 2023. doi:10.24963/KR.2023/40.

16 Tuukka Korhonen and Matti Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters (short paper). In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 8:1–8:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CP.2021.8.

17 Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017. doi:10.24963/IJCAI.2017/93.

18 Jean-Marie Lagniez, Pierre Marquis, and Anastasia Paparrizou. Defining and evaluating heuristics for the compilation of constraint networks. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 172–188. Springer, 2017. doi:10.1007/978-3-319-66158-2_12.

19 Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric Hsu. DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*, 2012.

20 Umut Oztok and Adnan Darwiche. On compiling CNF into decision-dnnf. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2014. doi:10.1007/978-3-319-10428-7_7.

21 Cédric Piette, Youssef Hamadi, and Lakhdar Sais. Vivifying propositional clausal formulae. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 525–529. IOS Press, 2008. doi:10.3233/978-1-58603-891-5-525.

22 Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 517–522. AAAI Press, 2008. URL: http://www.aaai.org/Library/AAAI/2008/aaai08-082.php.

23 Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. URL: http://www.satisfiability.org/SAT04/programme/21.pdf.