Separating Incremental and Non-Incremental Bottom-Up Compilation

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

Abstract

The aim of a compiler is, given a function represented in some language, to generate an equivalent representation in a target language L. In bottom-up (BU) compilation of functions given as CNF formulas, constructing the new representation requires compiling several subformulas in L. The compiler starts by compiling the clauses in L and iteratively constructs representations for new subformulas using an "Apply" operator that performs conjunction in L, until all clauses are combined into one representation. In principle, BU compilation can generate representations for any subformulas and conjoin them in any way. But an attractive strategy from a practical point of view is to augment one main representation – which we call the core – by conjoining to it the clauses one at a time. We refer to this strategy as incremental BU compilation. We prove that, for known relevant languages L for BU compilation, there is a class of CNF formulas that admit BU compilations to L that generate only polynomial-size intermediate representations, while their incremental BU compilations all generate an exponential-size core.

2012 ACM Subject Classification Theory of computation → Complexity theory and logic

Keywords and phrases Knowledge Compilation, Bottom-up Compilation, DNNF, OBDD

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.7

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

1 Introduction

Knowledge compilation (KC) is a domain of computer sciences that deals with different models for representing knowledge, or functions. Here, a *compilation* is a procedure that changes the representation of a function into something that allows for efficient reasoning. One aspect of KC is to find classes of representations, or *compilation languages*, that render interesting queries tractable [6, 15]. For Boolean functions, many such languages are subsets of the class of circuits in decomposable negation normal form (DNNF) [5].

When the function to compile into a sublanguage L of DNNF is given as a system of constraints, say a CNF formula (where constraints are clauses), different compilation paradigms are available, in particular top-down (TD) compilation and bottom-up (BU) compilation. TD compilers produce a compiled form as the trace of an algorithm that explores the space of solutions to the system, for instance the trace of a DPLL algorithm that does not stop after finding a solution but instead keep searching for more [11, 20]. A more general description is that TD compilers start from the whole system of constraints and recursively consider smaller systems. In contrast, BU compilers first compile each constraint independently, and then combine their compiled representations in pairs to construct a representation of the whole system. A key component in this paradigm is an "Apply" operator that, given two functions written in L, efficiently constructs a representation of their conjunction in L. Since binary conjunction is tractable, under some conditions, for ordered binary decision diagrams (OBDD) [2], for sentential decision diagrams (SDD) [4], and more generally for circuits in structured decomposable negation normal form (strDNNF) [19], in practice the target languages of BU compilers are restricted to sublanguages of strDNNF

© Alexis de Colnet;
licensed under Creative Commons License CC-BY 4.0

(including OBDD and SDD). A inconvenience of BU compilation is that, to compile a CNF formula in L, one must first compile several of its subformulas. Empirical observations show that intermediate subformulas sometimes require representations that are much larger than both the input and the output of the compiler [10, 16]. A good BU compiler tries to avoid such hard subformulas by combining the clauses in a smart way following heuristics. Unfortunately, for some formulas, hard subformulas simply are unavoidable. This has been proved for several models of BU compilation into OBDD [14, 23, 9, 12] and more generally into strDNNF [8, 13]. To be more precise, these results show exponential lower bounds on the size of intermediate results for L-based *refutations* of *unsatisfiable* CNF formulas [1], which can be seen as BU compilations to L when they do not use weakening or projection rules.

In this paper, we study how BU compilers differ space-wise for different strategies to combine the clauses. Our main result is that the variant of BU compilation often used in practice, which we call *incremental* BU compilation, is less powerful than general BU compilation in the sense that for some formulas, incremental BU compilation always generates intermediate representations of exponential size, while general BU compilation can avoid it. Let us now precise what we mean by *incremental* BU compilers for CNF formulas. These are compilers that work clause by clause. An incremental compiler keeps in memory a single representation for subformulas – that we call the *core* – and repeatedly combines the core with a clause until it represents the whole formula. So whenever an "Apply" is done, one of the inputs is the core and the other represents a clause. Importantly, the next clause to conjoin to the core may be selected just before the "Apply", so the order for combining the clauses is not necessarily decided ahead of compilation. Examples of incremental compiler can be found in [16] for the compilation of configuration problems into BDD; in the compiler SALADD[1] for compiling systems of pseudo-Boolean constraints into multi-valued decision diagrams (MDD); and we will later argue that the approach described in [3], which is the basis for the default compiler of the SDD package[2], can in fact be simulated by incremental compilation with only a polynomial size increase. The BU framework makes no assumption on how the constraints or clauses are combined and our objective is to show that fixing a strategy, while completely legitimate in practice, results in unavoidable exponential-size intermediate representations for some formulas that are otherwise "easy" to compile. Formally our result is the following

▶ **Theorem 1.** *There is an infinite class Φ of CNF formulas such that every $\phi \in \Phi$ over n variables admits polynomial-size compilations in OBDD($\wedge$,r) but all its incremental compilations in strDNNF($\wedge$,r) create intermediate circuits of size $2^{\Omega(\sqrt{n})}poly(1/n)$.*

An $L(\wedge, r)$ compilation refers to a BU compilation in L that uses the "Apply" to conjoin elements in L that are "similarly structured" and where arbitrary modifications preserving equivalence can be made to an intermediate result before it is fed to an "Apply" (this accounts for the so-called "restructuring" or "reordering" operation). To put our result into perspective, for $L \in \{$OBDD, SDD, strDNNF$\}$, the lower bounds shown in [14, 23, 12, 8, 13] hold regardless of the BU compilation strategy and, in the few cases where the clauses are chosen in a certain order, for instance in [9], it is not clear that the formulas for the lower bounds are easily compiled non-incrementally. Further, we are not aware of practical BU compilers for CNF formulas that are not incremental or that can not be simulated by incremental compilation. We do not claim that the development of non-incremental

[1] `https://www.irit.fr/~Helene.Fargier/BR4CP/CompilateurSALADD.html`
[2] `http://reasoning.cs.ucla.edu/sdd/`

compilers is the path to follow since we doubt of the practicability of that strategy. But we argue that while lower bounds in the general general framework are strong results, positive results on the other hand have no practical implication unless they can be proved for the incremental strategy. Things are different for OBDD-based refutations that rely on the weakening rule, so we can not make any claim in this context. Some refutation strategies improve upon the incremental approach using a clustering step [17, 18]. These approaches are non-incremental but are not very relevant to this work since the clusters are defined so that variables can be forgotten (i.e., existentially quantified out) after compilation of a cluster, which is forbidden in our setting as this modifies the function to compile. This is also the reason why the techniques used by Segerlind to separate tree-like OBDD-based refutations and general OBDD-based refutations [22] are not applicable in our setting, despite the apparent proximity with our topic.

The paper is organized as follows. We start in Section 2 with preliminaries where we describe the compilation languages considered and the general framework for BU compilation. Then in Section 3 we formalize incremental BU compilation and discuss some of its advantages compared to general BU compilation. The main part of the paper is Section 4 where we prove our main result on the separation between incremental and general BU compilation. Finally, Section 5 briefly discusses the implications of this result in practice and regarding the choice of a framework for modeling the behavior of algorithms.

2 Preliminaries

A Boolean variable is a variable x over $\{0,1\}$ (0 for *false*, 1 for *true*). An assignment to a set X of Boolean variables is a mapping from X to $\{0,1\}$. We call $\{0,1\}^X$ the set of assignments to X. A Boolean function f over X is a mapping from $\{0,1\}^X$ to $\{0,1\}$. When not specified, $var(f)$ denotes the set of variables of f. A literal is Boolean variable x or its negation $\overline{x}$. We use the usual symbols $\wedge$ and $\vee$ for conjunction and disjunction. A *clause* is a disjunction of literals and a *CNF formula* ϕ is a conjunction of clauses. We often see ϕ as the set of its clauses, so that we can write $c \in \phi$ to denote that c is a clause of ϕ, and $\phi \setminus c$ to denote the CNF formula whose clauses are all clauses of ϕ except c. Given a set of clauses S, we sometimes write $\bigwedge S$ to denote the CNF formula $\bigwedge_{c \in S} c$. CNF formulas are representations of Boolean functions. Two representations Σ and Σ' are *equivalent*, denoted by $\Sigma \equiv \Sigma'$ if they represent the same function, in particular they must be defined over the same set of variables. We say that Σ *entails* Σ', denoted by $\Sigma \models \Sigma'$ if $var(\Sigma') \subseteq var(\Sigma)$ and every assignment to $var(\Sigma)$ that satisfies Σ also satisfies Σ'.

Circuits in strDNNF

A circuit is in negation normal form (NNF) is a Boolean circuit whose gates are $\vee$-gates and $\wedge$-gates and whose inputs are literals or $\{0,1\}$ inputs. In particular there are no $\neg$-gates in a circuit in NNF. The set of variables below a gate g is denoted by $var(g)$. A gate g with inputs $g_1, \ldots, g_k$ is called *decomposable* when $var(g_i) \cap var(g_j) = \emptyset$ for every $i \neq j$. A circuit in NNF is in decomposable NNF (DNNF) when all its $\wedge$-gates are decomposable. Circuits in strDNNF respect a more constrained variant of decomposability called *structured decomposability*. It requires a *vtree* (variable tree), that is, a rooted binary tree T whose leaves are in bijection with the circuit's variables. For t a node of T, we denote by $var(t)$ the set of variables that label leaves under t. The circuit D in DNNF *respects* T when all $\wedge$-gates have fan-in 2 and when there is a mapping λ from D's gates to T's nodes such that for every gate g of D,

Figure 1 A circuit in strDNNF.

- $var(g) \subseteq var(\lambda(g))$,
- if g is a $\vee$-gate with inputs $g_1, \dots, g_k$, then $\lambda(g) = \lambda(g_1) = \dots = \lambda(g_k)$,
- if g is a $\wedge$-gate with inputs g_1, g_2 then $\lambda(g) = t$ is an internal node of T and there is a node t_1 under its first child and a node t_2 under its second child such that $\lambda(g_1) = t_1$ and $\lambda(g_2) = t_2$.

The size of D, denoted by $|D|$ is the number of connectors in the circuit. Not all circuits in DNNF are in strDNNF but the class of circuits in strDNNF, called the *strDNNF language*, is sufficiently expressive to represent all Boolean functions over finitely many variables [19]. An example of circuit in strDNNF is shown Figure 1: the mapping between internal gates and nodes of the vtree is represented with solid, dashed and dotted boxes. Important function representations in compilation admit linear-time translations into strDNNF, including SDDs whose definition we omit (see [4]) and OBDDs which we define now.

OBDDs

Given two Boolean functions c_0, c_1 and a Boolean variable x, the *decision node* [diagram: node x with arrows to c_0 and c_1] represents the function $(\overline{x} \wedge c_0) \vee (x \wedge c_1)$. Graphically, if x is set to 0 then follow the dashed arrow, otherwise if x is set to 1 then follow the solid arrow. A binary decision diagram (BDD) is a directed acyclic graph (DAG) with a single root and two sinks labelled by 0 and 1, and whose internal nodes are decision nodes. BDDs are interpreted as Boolean functions over the variables that label their decision nodes. The satisfying assignments are those whose unique corresponding path leads to the sink 1. An *ordered BDD* (OBDD) is such that the variables appear in the same order and at most once along every path from the root to a sink. Examples of OBDDs are shown in Example 2. The class of OBDDs is called the *OBDD language*. The size of an OBDD B, denoted by $|B|$, is its number of nodes. OBDDs can be transformed in linear time into strDNNF circuits respecting *linear vtrees*, that is, vtrees where every internal node has a child that is a leaf. So OBDDs can be seen as being structured by linear vtrees.

Bottom-up compilation

Given a language L whose elements are structured by vtrees and a system of finitely many constraints ϕ (for instance a CNF formula, where constraints are clauses), an $L(\wedge, r)$ *bottom-up compilation* of ϕ in L is a sequence

$$(\Sigma_1, I_1), (\Sigma_2, I_2), \dots, (\Sigma_N, I_N)$$

where $\Sigma_1, \dots, \Sigma_N$ are elements of L such that $\Sigma_N \equiv \phi$ and where, for every $i \in \{1, \dots, N\}$, I_i is an instruction telling how Σ_i is obtained. Three types of instructions are possible:

- $\Sigma_i = \mathsf{Compile}(c)$ for some constraint c of ϕ. Then $\Sigma_i \equiv c$.
- $\Sigma_i = \mathsf{Apply}(\Sigma_j, \Sigma_k, \wedge)$ for some $j, k < i$ such that $\Sigma_i, \Sigma_j, \Sigma_k$ respect the *same* vtree. Then $\Sigma_i \equiv \Sigma_j \wedge \Sigma_k$.
- $\Sigma_i = \mathsf{Restructure}(\Sigma_j)$ for some $j < i$. Then $\Sigma_i \equiv \Sigma_j$ and their vtrees may differ.

The *size* of the compilation is defined as $\max_{1 \leq i \leq N} |\Sigma_i|$. The Restructure instruction accounts for the r in the name "$L(\wedge, r)$ compilation". When this instruction is not used, we say that we have an $L(\wedge)$ compilation. Regarding the Apply instruction, the assumption that Σ_i, Σ_j and Σ_k respect the same vtree is essential since, for many languages L, polynomial-time conjunction of two elements in L into a another element in L is feasible only when the initial elements have compatible vtrees. Quadratic-time Apply procedures for conjunction are known for the languages considered in this paper, namely for OBDD [2], SDD [4] and strDNNF [19].

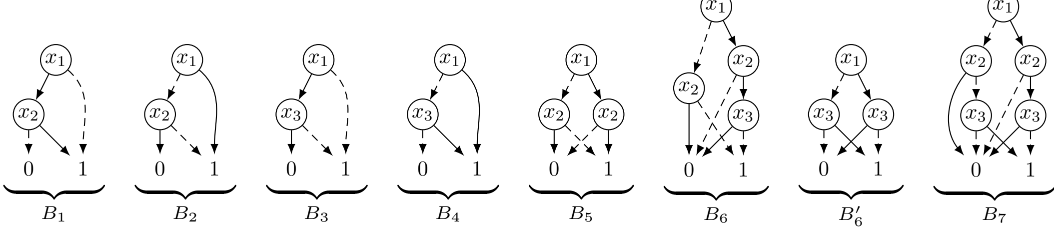
3 Incremental and Non-Incremental Bottom-Up Compilation

To compile a CNF formula in a bottom-up manner, some compilers work clause by clause. The idea is to keep one main representation, which we call the *core*, to which clauses are conjoined one after the other. We call this strategy *incremental* bottom-up compilation. Formally, an incremental $L(\wedge, r)$ compilation is an $L(\wedge, r)$ compilation where every Apply instruction is of the form $\mathsf{Apply}(\Sigma, \mathsf{Compile}(c), \wedge)$, where $\Sigma \in L$ has been computed previously in the compilation and c is a clause/constraint of the formula to compile. We used a simplified framework where, for all I_i, the core is Σ_{i-1} (by convention $\Sigma_0 = 1$).

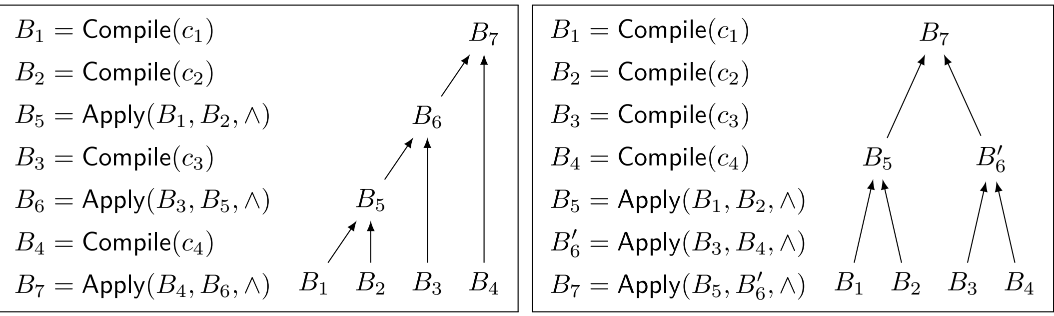
- $\Sigma_i = \mathsf{Apply}(\Sigma_{i-1}, \Sigma_c, \wedge)$ where $c \in \phi$ and $\Sigma_c \equiv c$ respects the *same* vtree as Σ_{i-1} and Σ_i.
- $\Sigma_i = \mathsf{Restructure}(\Sigma_{i-1})$. Then $\Sigma_i \equiv \Sigma_{i-1}$ and their vtrees may differ.

For convenience the compilation of Σ_c is implicit. We can visualize the compilation $(\Sigma_1, I_1), \dots, (\Sigma_N, I_N)$ as a directed acyclic graph whose vertices are in bijection with the Σ_is. If I_i is a Compile instruction, then Σ_i is a source of the DAG, else if I_i is $\Sigma_i = \mathsf{Restructure}(\Sigma_j)$ then there is an edge from Σ_j to Σ_i, and if I_i is $\Sigma_i = \mathsf{Apply}(\Sigma_j, \Sigma_k, \wedge)$, then there is an edge from Σ_j to Σ_i and another one from Σ_k to Σ_i. In this DAG representation, if we bypass the nodes of in-degree 1 created by the Restructure instructions, then an incremental BU compilation is shaped like a *linear* tree, that is, a tree such where every node is either a leaf or has a leaf child. In general BU compilation, the DAG can be shaped like a tree, in which case we have a *tree-like* compilation, or it can have a general DAG structure (some Σ_is may be inputs to more than one Apply).

▶ **Example 2.** Consider the CNF formula $\phi = (\overline{x}_1 \vee x_2) \wedge (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (x_1 \vee x_3) = c_1 \wedge c_2 \wedge c_3 \wedge c_4$, and the following OBDDs:



B_1, B_2, B_3 and B_4 represent the clauses c_1, c_2, c_3 and c_4, respectively. Now consider the following bottom-up compilations of ϕ:



The left compilation is incremental whereas the one on the right is tree-like but non-incremental since B_7 is obtained from B_5 and B'_6, none of which represents a clause from ϕ.

In incremental BU compilation, the constraints are conjoined in a certain order. The framework is general enough that, in practice, the order for the constraints is not necessarily fixed before the first Apply: an incremental BU compiler that has run a few steps and computed the core Σ can dynamically decide the next clause to conjoin based on Σ.

Some strategies for BU compilation are definitely not incremental but can be simulated by incremental BU compilation. For instance, we claim it is the case of the compiler scheme described in [3] for compiling a CNF formula into an SDD. Since we have omitted the definition of SDDs, we look at it as a compilation to strDNNF (of which SDD is a sublanguage). Say $\phi(X)$ be the CNF formula to compile and let T be a vtree over X. Let T_v denotes the subtree of T rooted under node v. Reusing some of the terminology in [4], we say that each clause c of ϕ is *hosted* at the unique deepest node v of T such that $var(c) \subseteq var(v)$. Let ϕ_v be the subformula of ϕ made uniquely of clauses over $var(v)$ and let ϕ_v^{host} be the subformula of ϕ made uniquely of clauses of ϕ hosted at v. Suppose v is not a leaf and let v_1 and v_2 be its children. The idea is, given T, to recursively compute a strDNNF circuit $D_v \equiv \phi_v$ for the node v as follows

(1) compute the strDNNF circuits $D_{v_1} \equiv \phi_{v_1}$ and $D_{v_2} \equiv \phi_{v_2}$ for v_1 and v_2 respecting the vtrees T_{v_1} and T_{v_2} respectively

(2) compute $D = \mathsf{Apply}(D_{v_1}, D_{v_2}, \wedge)$ that respects T_v

(3) incrementally conjoin all clauses of ϕ_v^{host} to D, perhaps restructuring D to change its vtree under v between two Apply (so T is modified but only under v).

The strDNNF circuit for the root node of T represents ϕ. Observe that, by definition of a vtree, in step (2) there is $var(\phi_{v_1}) \cap var(\phi_{v_2}) = var(v_1) \cap var(v_2) = \emptyset$, so we say that the Apply occurring there is *decomposable*. We claim that this approach can be simulated by an incremental strDNNF($\wedge$,r) compilation because any Apply that does not fit the incremental approach is a *decomposable* Apply from step (2) and such Apply are easy to simulate in the incremental approach. The proof of the following is deferred to the appendix.

▶ **Proposition 3.** *Every strDNNF($\wedge$,r) compilation of a CNF formula ϕ where every* Apply *that does not involve a clause of ϕ computes the conjunction of two strDNNF circuits representing subformulas of ϕ over disjoint set of variables, can be transformed into an incremental strDNNF($\wedge$,r) compilation with only a polynomial increase in size.*

Let us mention some advantages of incremental BU compilation. First, many guarantees on the size of the final output of a BU compilation also hold for incremental BU compilation. Especially, it is known that if the primal graph of a CNF formula has logarithmic treewidth, then the formula can be compiled into a polynomial-size SDD [4] and, since the primal treewidth cannot increase for subformulas it follows that, in the case of a logarithmic treewidth, polynomial-size compilations in SDD($\wedge$,r) exist even in the incremental case.

Another example is on the role of restructuring in BU compilation: it is shown in [12] that OBDD($\wedge$,r) compilations are exponentially more compact than OBDD($\wedge$) compilations and a careful observation of the proof reveals that the OBDD($\wedge$,r) compilations used there can be made incremental. So restructuring does not require stronger strategies than incremental compilation to have a positive effect. From a practical point of view, incremental compilers have the advantage that the purpose of restructuring is clear (at least when compiling CNF formulas): it is to reduce the size of the core. This is fairly intuitive and corresponds to what is done in practice, in particular by the default compiler of the SDD package. One can actually prove that there is not much interest space-wise in using restructuring for anything else as any incremental $L(\wedge,r)$ compilation for $L \in \{\text{OBDD}, \text{SDD}, \text{strDNNF}\}$ can be transformed in polynomial-time into another $L(\wedge,r)$ compilation where restructuring never increases the size of the core. The proof of the following appears in appendix.

▶ **Proposition 4.** *Let $L \in \{OBDD, SDD, strDNNF\}$, let $(\Sigma_1, I_1), \dots, (\Sigma_N, I_N)$ be an incremental $L(\wedge,r)$ compilation of the CNF formula ϕ over n variables. There exists an incremental $L(\wedge,r)$ compilation $(\Sigma'_1, I'_1), \dots, (\Sigma'_M, I'_M)$ of ϕ such that $M \leq 2N$ and, for every $j \in \{1, \dots, M\}$ there is an $i \in \{1, \dots, N\}$ such that $\Sigma'_j \equiv \Sigma_i$ and $|\Sigma'_j| \leq 2n|\Sigma_i|$. In addition, when Σ'_j is obtained by restructuring Σ'_{j-1}, we have $|\Sigma'_j| \leq |\Sigma'_{j-1}|$.*

We note that Proposition 4 holds because every clause has a linear-size representation in L for every vtree. The statement holds for other types of constraints that share this property, for instance parity constraints [20], but it is not clear whether it applies to general systems of constraints as well.

In non-incremental BU compilation, restructuring may not be used only for reducing the size. Indeed, given two circuits in L respecting different vtrees over the same variables, using an Apply to obtain their conjunction requires changing the vtree of at least one of them, so it requires a restructuring step. But it can be the case that there is no way to make both circuits respect the same vtree without a global size increase.

4 Separating Incremental and Non-Incremental Compilation

In this section, we prove the main result of the paper, which is the separation between incremental and non-incremental bottom-up compilation.

▶ **Theorem 1.** *There is an infinite class Φ of CNF formulas such that every $\phi \in \Phi$ over n variables admits polynomial-size compilations in OBDD($\wedge$,r) but all its incremental compilations in strDNNF($\wedge$,r) create intermediate circuits of size $2^{\Omega(\sqrt{n})}poly(1/n)$.*

4.1 General Idea for the Separation

First we sketch the general idea for proving the separation. We use two CNF formulas $\phi_1(X)$ and $\phi_2(X)$ and consider the formula $\phi(X) = \bigwedge_{c_1 \in \phi_1} \bigwedge_{c_2 \in \phi_2} (c_1 \vee c_2)$. This formula is in CNF and is equivalent to $\phi_1 \vee \phi_2$. Let us say that ϕ_1 is unsatisfiable, then ϕ is equivalent to ϕ_2. To compile ϕ in non-incremental OBDD($\wedge$,r), we can first compile independently the subformulas $\bigwedge_{c_1 \in \phi_1}(c_1 \vee c_2) \equiv \phi_1 \vee c_2 \equiv c_2$ for every $c_2 \in \phi_2$. This gives one OBDD B_{c_2} for every clause of ϕ_2. Any clause can be represented by a small OBDD with respect to any variable ordering so, using restructuring, we can freely choose the variable ordering for B_{c_2}. Then, starting from the OBDDs B_{c_2} for all $c_2 \in \phi_2$, we can compile an OBDD representing ϕ as if we were doing a BU compilation of ϕ_2. The idea is summarized in Figure 2

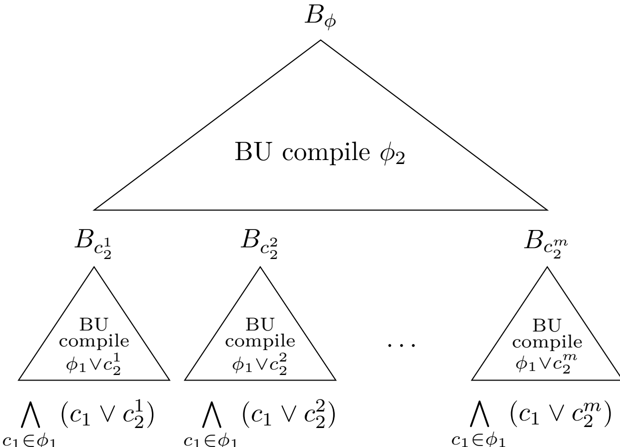


Figure 2 A bottom-up compilation of $\phi \equiv \phi_1 \vee \phi_2$ when ϕ_1 is unsatisfiable.

where $c_2^1, c_2^2, \ldots, c_2^m$ are the clauses of ϕ_2. The small triangles represent BU compilations of $\bigwedge_{c_1 \in \phi_1}(c_1 \vee c_2^i)$. Each costs roughly the same as a BU compilation of ϕ_1. So the cost of this compilation scheme is roughly m times the cost of a BU compilation of ϕ_1, plus the cost of a BU compilation of ϕ_2.

Now what happens when compiling ϕ incrementally in strDNNF($\wedge$,r) (which is more general than OBDD($\wedge$,r))? Well, let us look at the last instruction Apply of the compilation: suppose it is $D' = \mathsf{Apply}(D, D_{\gamma_1 \vee \gamma_2}, \wedge)$, that is, we conjoin the core circuit D with a circuit in strDNNF representing the clause $\gamma_1 \vee \gamma_2$ where $\gamma_1 \in \phi_1$ and $\gamma_2 \in \phi_2$. But then D computes the CNF formula $\phi \setminus (\gamma_1 \vee \gamma_2)$, which one can show is equivalent to $(\phi_2 \vee (\phi_1 \setminus \gamma_1)) \wedge (\phi_1 \vee (\phi_2 \setminus \gamma_2)) \equiv ((\phi_1 \setminus \gamma_1) \wedge (\phi_2 \setminus \gamma_2)) \vee \phi_2$ since ϕ_1 is unsatisfiable. For convenience let us assume that ϕ_2 is also unsatisfiable and that the minimal unsatisfiable subsets of ϕ_1 and ϕ_2 are themselves (so both $\phi_1 \setminus \gamma_1$ and $\phi_2 \setminus \gamma_2$ are satisfiable). Then the cost of compiling ϕ incrementally is at least the cost of compiling $(\phi_1 \setminus \gamma_1) \wedge (\phi_2 \setminus \gamma_2)$.

To separate general BU compilation from incremental BU compilation, we look for formulas ϕ_1 and ϕ_2 that are both easy to compile in OBDD($\wedge$,r), so that ϕ is easy to compile in non-incremental OBDD($\wedge$,r) using the scheme of Figure 2, but such that $(\phi_1 \setminus \gamma_1) \wedge (\phi_2 \setminus \gamma_2)$ can only be represented by exponential-size strDNNF circuits regardless of the vtree and regardless of the choice of γ_1 and γ_2. In the next sections, we describe our formulas and give the formal proof. For the sake of keeping the proof simple, we will choose a formula ϕ_2 that is satisfiable and we will not necessarily look at the very last Apply of the compilation, but the idea behind the proof is unchanged.

4.2 The Formula ϕ for the Separation

Let $X = \{x_{i,j} \mid 1 \leq i, j \leq n\}$ and consider the following functions:

- $\text{ROW}_n(X)$ (resp. $\text{COL}_n(X)$) is the function over X whose satisfying assignments are that for which each row (resp. column) of the $n \times n$ matrix $[x_{i,j}]_{i,j}$ contains exactly one 1.
- $\text{ODD}(X)$ (resp. $\text{EVEN}(X)$) is the function over X whose satisfying assignments are that for which there is an odd (resp. even) number of 1s in the $n \times n$ matrix $[x_{i,j}]_{i,j}$.

We will consider CNF formulas *representing* $\text{ROW}_n(X)$ and $\text{COL}_n(X)$, and CNF formulas *encoding* $\text{ODD}(X)$ and $\text{EVEN}(X)$. We distinguish a representation from an encoding in that the latter uses auxiliary variables while the former does not. The CNF formulas representing $\text{ROW}_n(X)$ and $\text{COL}_n(X)$ are:

$$row_n(X) = \bigwedge_{1 \leq i \leq n} \left((x_{i,1} \vee \cdots \vee x_{i,n}) \wedge \bigwedge_{1 \leq j < k \leq n} (\overline{x}_{i,j} \vee \overline{x}_{i,k}) \right)$$
$$col_n(X) = \bigwedge_{1 \leq j \leq n} \left((x_{1,j} \vee \cdots \vee x_{n,j}) \wedge \bigwedge_{1 \leq h < i \leq n} (\overline{x}_{h,j} \vee \overline{x}_{i,j}) \right)$$

ODD and EVEN do not have polynomial-size representations in CNF but they have linear-size encodings. We use the notation $(a + b + c = 0 \mod 2)$ to denote the clauses $(a \vee \overline{b} \vee c) \wedge (a \vee b \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \wedge (\overline{a} \vee \overline{b} \vee \overline{c})$. We use the standard encoding [21]:

$$odd_n(X, Y) = y_n \wedge \bigwedge_{1 \leq j \leq n} \bigwedge_{1 \leq i \leq n} (y_{i,j} + y_{i-1,j} + x_{i,j} = 0 \mod 2)$$

where $y_{0,j} = y_{n,j-1}$ if $j > 1$, and $y_{0,1} = 0$. In $odd_n(X, Y)$, the matrix is browsed row-wise and a satisfying assignment sets $y_{i,j}$ to 1 if and only if the number of 1s found before the cell i, j is odd.

We define $even_n(X, Y)$ analogously: we just replace the $(a + b + c = 0 \mod 2)$ by $(a + b + c = 1 \mod 2)$, which clauses are $(a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee \overline{c}) \wedge (\overline{a} \vee \overline{b} \vee c) \wedge (a \vee b \vee c)$. The following holds:

$$row_n(X) \equiv \mathrm{ROW}_n(X) \text{ and } col_n(X) \equiv \mathrm{COL}_n(X) \text{ and}$$
$$\exists Y . odd_n(X, Y) \equiv \mathrm{ODD}(X) \text{ and } \exists Y . even_n(X, Y) \equiv \mathrm{EVEN}(X).$$

Let n be an *even* integer, let $\phi_1(X) = row_n(X)$, and let

$$\phi_1^*(X, Y) = \phi_1(X) \wedge odd_n(X, Y) \quad \text{and} \quad \phi_2(X) = \bigwedge_{1 \leq j \leq n} \bigwedge_{1 \leq h < i \leq n} (\overline{x}_{h,j} \vee \overline{x}_{i,j}). \tag{1}$$

ϕ_2 represents the function that accepts exactly the matrices $[x_{i,j}]_{i,j}$ whose columns all contains at most one 1. We will later refer to this function as $\mathrm{AMOpCOL}_n$ (at most one per column). ϕ_1^* is unsatisfiable since n is even. We will use the following CNF formula for the separation:

$$\phi = \bigwedge_{c_1 \in \phi_1^*} \bigwedge_{c_2 \in \phi_2} (c_1 \vee c_2) \equiv \phi_1^* \vee \phi_2 \equiv \phi_2. \tag{2}$$

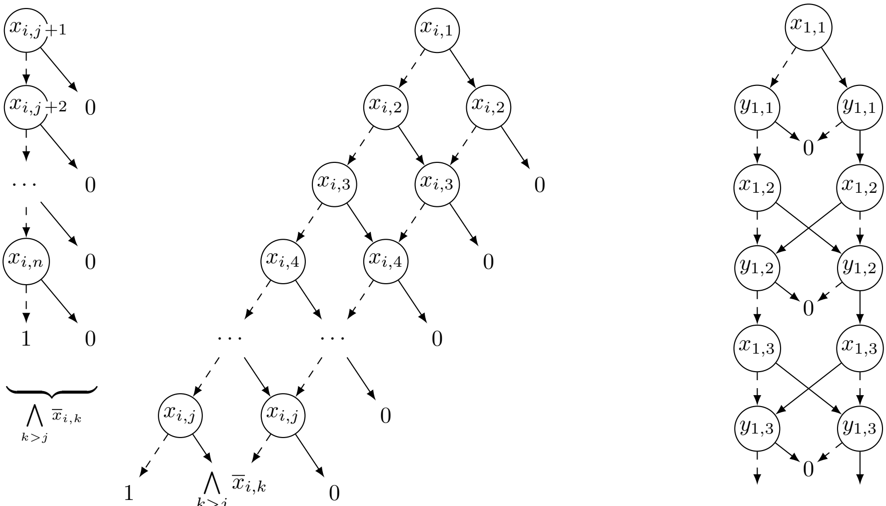
4.3 ϕ is Easy to Compile in OBDD($\wedge$,r)

Let us start with the proof that ϕ can be compiled in OBDD($\wedge$,r) using only polynomial-size OBDDs. The key element here is that both ϕ_1^* and ϕ_2 are easily compiled in OBDD($\wedge$,r).

▶ **Lemma 5.** *The formulas ϕ_1^* and ϕ_2 defined in (1) have polynomial-size OBDD($\wedge$,r) compilations.*

Proof. Let us start with ϕ_1^*. We fix an i between 1 and n. For every j, we separately compile $\bigwedge_{k>j}(\overline{x}_{i,j} \vee \overline{x}_{i,k}) \equiv \overline{x}_{i,j} \vee \bigwedge_{k>j} \overline{x}_{i,k}$ into an OBDD B_j that respects the variable ordering $x_{i,1}, x_{i,2}, \ldots, x_{i,n}$. It is easy to see that compiling each B_j is feasible in polynomial time in OBDD($\wedge$,r).

▷ Claim 6. $B_1 \wedge \cdots \wedge B_j$ is equivalent to the OBDD represented in Figure (3a).



(a) Intermediate step in the compilation of row_n. **(b)** Intermediate step in the compilation of odd_n.

Figure 3 Intermediate steps in the compilation of ϕ_1^*.

The proof of the Claim 6 is deferred to the appendix. We compile $B_1 \wedge \cdots \wedge B_n \equiv \bigwedge_{j \neq k}(\overline{x}_{i,j} \vee \overline{x}_{i,k})$ using only OBDDs of size $O(n)$. Let B^i be the resulting OBDD. Next, it is easy to compile the clause $x_{i,1} \wedge \cdots \wedge x_{i,n}$ into an OBDD of size $O(n)$ and that respects the variable ordering $x_{i,1}, x_{i,2}, \ldots, x_{i,n}$. We conjoin this OBDD to B^i and thus obtain an OBDD that represents $x_{i,1} + \cdots + x_{i,n} = 1$ whose size is $O(n)$ and that we call $B_{\text{ith } row}$. Now the OBDDs $B_{\text{ith } row}$ for every $1 \leq i \leq n$ have disjoint sets of variables so they can be incrementally conjoined into a single OBDD B_{rows} of size $O(n^2)$ that represents ROW_n and whose variable ordering is $x_{1,1}, \ldots, x_{1,n}, x_{2,1}, \ldots, x_{2,n}, \ldots, x_{n,n}$.

It remains to compile odd_n and to conjoin it to B_{rows}. We can compile odd_n incrementally in OBDD($\wedge$,r) using only the variable ordering $x_{1,1}, y_{1,1}, x_{1,2}, y_{1,2}, \ldots, x_{1,n}, y_{1,n}, x_{2,1}, y_{2,1}, \ldots, \ x_{n,n}, y_{n,n}$. First we conjoin the OBDDs for the clauses of the parity constraint $y_{1,1} + x_{1,1} = 0 \mod 2$ together, then we add the OBDDs for the clauses of the parity constraint $y_{1,1} + x_{1,2} + y_{1,2} = 0 \mod 2$, and so on. Once we have added all clauses of a parity constraint, we end up with an OBDD that looks like that represented Figure (3b). To add the next parity constraints, we only have four OBDDs to conjoin before we attain again an OBDD that looks like that represented Figure (3b). So the sizes of the OBDDs used to compile odd_n never grow bigger than $O(n^2)$. We call B_{odd} the final OBDD.

Finally, since B_{odd} and B_{rows} have compatible variable orderings, they can be conjoined in quadratic time, and the compilation of ϕ_1^* is finished.

For ϕ_2, the OBDD($\wedge$,r) compilation scheme used to compile the OBDD B^i computing $\bigwedge_{j \neq k}(\overline{x}_{i,j} \vee \overline{x}_{i,k})$, respecting the variable ordering $x_{i,1}, \ldots, x_{i,n}$, and whose is size is in $O(n)$, can be adapted to compile an OBDD Q^j computing $\bigwedge_{h \neq i}(\overline{x}_{h,j} \vee \overline{x}_{i,j})$, respecting the variable ordering $x_{1,j}, \ldots, x_{n,j}$ and whose size is in $O(n)$. Since the OBDDs $Q^1, \ldots, Q^n$ have pairwise disjoint sets of variables, they can be incrementally conjoined in polynomial time to obtain an OBDD computing $Q^1 \wedge \cdots \wedge Q^n \equiv \phi_2$. ◀

▶ **Lemma 7.** *The formula ϕ defined by (2) can be compiled in OBDD($\wedge$,r) using only OBDDs of size polynomial in n.*

Proof. For every clause $c_2 \in \phi_2$, we use Lemma 5 to construct a small OBDD($\wedge$,r) compilation of $\bigwedge_{c_1 \in \phi_1^*}(c_1 \vee c_2)$: we take the polynomial-size OBDD($\wedge$,r) compilation of ϕ_1^* described in the lemma and we apply a disjunction between every OBDD of the compilation and a small OBDD representing c_2. This gives a compilation of $\bigwedge_{c_1 \in \phi_1^*}(c_1 \vee c_2) \equiv \phi_1^* \vee c_2 \equiv c_2$ and increases the size of every OBDD by a constant factor only. We do this for each clause $c_2 \in \phi_2$ independently and obtain an OBDD B_{c_2} for every c_2. Then we use restructuring to change the variable ordering of the B_{c_2}s (clauses admit linear-size OBDDs under every variable ordering) so that we can use the polynomial-size compilation of ϕ_2 described in Lemma 5. ◀

4.4 ϕ is Hard to Compile Incrementally in strDNNF($\wedge$,r)

We move to the more difficult part that consists in proving that every incremental strDNNF($\wedge$,r) compilation of ϕ generates intermediate circuits of exponential size. To this end, we will need the following lemma (which we prove later):

▶ **Lemma 8.** *Let ϕ be the formula defined by (2). Every incremental strDNNF($\wedge$,r) compilation of ϕ generates an intermediate circuit that can be transformed in polynomial-time into a circuit in strDNNF computing $ROW_{n-\Delta} \wedge COL_{n-\Delta}$ for some integer $\Delta \leq 3$.*

The function $\mathrm{ROW}_n(X) \wedge \mathrm{COL}_n(X)$ is widely studied in computer sciences and is often called the *permanent*. Its satisfying assignments are the 0/1 matrices $[x_{i,j}]_{i,j}$ that contain exactly one 1 in every row and exactly one 1 in every column. In particular, these assignments are in bijection with the permutations of $\{1, \dots, n\}$ since every assignment a to X that satisfies $\mathrm{ROW}_n(X) \wedge \mathrm{COL}_n(X)$ uniquely corresponds to the permutation that maps every $i \in \{1, \dots, n\}$ to the unique $j \in \{1, \dots, n\}$ such that $a(x_{i,j}) = 1$. The permanent is known to be hard to represent as OBDDs, as read-once branching programs [25, Theorem 6.2.12.][3], as circuits in strDNNF [24, Proposition 7 and Lemma 27], and even as circuits in DNNF.

▶ **Lemma 9** ([7, Proof of Theorem 1])**.** *Every circuit in DNNF representing $ROW_n(X) \wedge COL_n(X)$ has size $2^{\Omega(n)}$.*

The desired lower bound on incremental bottom-up compilation of ϕ follows directly from Lemmas 8 and 9. Thus the proof of Theorem 1 is straightforward given Lemmas 7, 8 and 9. The rest of the section is dedicated to the proof of Lemma 8. Due to space constraint, the proof of several claims is deferred to appendix.

Consider an incremental strDNNF($\wedge$, r) compilation $(D_1, I_1), \dots, (D_N, I_N)$ of ϕ. The Apply operations are of the form

$$D_{i+1} = \mathsf{Apply}(D_i, D_{c_1 \vee c_2}, \wedge).$$

where $c_1 \in \phi_1 \wedge odd_n$ and $c_2 \in \phi_2$. Consider the *largest* integer $k < N$ such that

- we have that $D_{k+1} = \mathsf{Apply}(D_k, D_{\gamma_1 \vee \gamma_2}, \wedge)$
- and γ_1 is a clause of $\phi_1 = row_n$ (and not a clause of odd_n)
- and there are no clauses $c_1 \in \phi_1$ and $c_2 \in \phi_2$ such that $c_1 \vee c_2 \neq \gamma_1 \vee \gamma_2$ and $c_1 \vee c_2 \models \gamma_1 \vee \gamma_2$.

[3] The functions that we call ROW_n and COL_n do not equal the functions ROW_n and COL_n from [25]. Our $\mathrm{ROW}_n(X) \wedge \mathrm{COL}_n(X)$ corresponds to PERM_n in [25]

D → Conjoin missing clauses ($D = D_k$) → Forget variables Y ($D' \equiv \psi(X,Y) \wedge \bigwedge S \setminus \gamma_1 \vee \gamma_2$) → Conjoin with ODD($X$) ($D'' \equiv \exists Y.\psi(X,Y) \wedge \bigwedge S \setminus \gamma_1 \vee \gamma_2$) → Case disjunction on $\gamma_1 \vee \gamma_2$ ($D''' \equiv \mathrm{ODD}(X) \wedge \bigwedge S \setminus \gamma_1 \vee \gamma_2$)

Figure 4 Strategy for proving Lemma 8.

For this k we have that every D_h for $h > k+1$ is obtained either by a Restructure operation, or by an Apply with an strDNNF circuit $D_{c_1 \vee c_2}$ such that c_1 is a clause of odd_n, or by an Apply with an strDNNF circuit $D_{c_1 \vee c_2}$ that is entailed by D_{k+1}.

▷ Claim 10. For any $h > k+1$ such that $D_h = \mathsf{Apply}(D_{h-1}, D_{c_1 \vee c_2}, \wedge)$ and $c_1 \in row_n$, we have that $D_k \models c_1 \vee c_2$ or $\gamma_1 \vee \gamma_2 \models c_1 \vee c_2$.

In the rest of the proof, we write $D = D_k$ for convenience. The strategy for showing Lemma 8 is to turn D into a circuit in strDNNF representing $\mathrm{ROW}_{n-\Delta} \wedge \mathrm{COL}_{n-\Delta}$ by means of a few polynomial-time transformations, as schematized in Figure 4. This differs slightly from the strategy described in Section 4.1: we are not considering the very last Apply. The reason is that the formula ϕ contains some clauses c such that $\phi \setminus c \models c$. A few of these clauses are even tautological. In a sense, these clauses are "irrelevant" in ϕ. Clearly, if the last Apply of the compilation is conjoining the core to an irrelevant clause, then this Apply is in fact not modifying the core and we should look at a previous Apply. One could think that we could prove the statement for a formula ϕ where irrelevant clauses have been removed. This is certainly true if we only remove tautologuous clauses. However we do not want to remove the irrelevant clauses that are non-tautologous since such clauses are not necessarily irrelevant for subformulas of ϕ. So, rather than removing irrelevant clauses from ϕ, we choose to keep them and to consider another Apply than the very last one.

By definition, not all clauses of the form $c_1 \vee c_2$ with $c_1 \in \phi_1$ and $c_2 \in \phi_2$ have been conjoined to the core during the first k steps of the compilation. Let $S = \{c_1 \vee c_2 \mid c_1 \in \phi_1, c_2 \in \phi_2\}$ and let $S_{D\models} = \{c \in S \mid D \models c\}$ be the set of clauses of S that are entailed by D and let $S' = S_{D\models} \setminus \{\gamma_1 \vee \gamma_2\}$. Since $\gamma_1 \vee \gamma_2$ has not been conjoined to D we have that

$$D \equiv \psi \wedge \bigwedge S'$$

where ψ is a subformula of $\bigwedge_{c \in odd_n} \bigwedge_{c_2 \in \phi_2} (c \vee c_2)$. Ideally we would like S' to be $S \setminus \{\gamma_1 \vee \gamma_2\}$, but this is generally not the case. Given a clause c, let $S_{c\models} = \{c' \mid c' \in S, c' \neq c \text{ and } c \models c'\}$ be the set of clauses in S that are distinct from c and that are entailed by c. By Claim 10, each clause in S is either entailed by D or is entailed by $\gamma_1 \vee \gamma_2$, so $S' \cup S_{\gamma_1 \vee \gamma_2 \models} = S \setminus \{\gamma_1 \vee \gamma_2\}$.

▷ Claim 11. Let $\gamma = \gamma_1 \vee \gamma_2$. For every vtree over X, there is a strDNNF of size $O(n^2)$ respecting that vtree that represents $\bigwedge S_{\gamma\models}$ (by convention $\bigwedge \emptyset = 1$).

By Claim 11, there is an strDNNF circuit $D_{\gamma_1 \vee \gamma_2 \models}$ of size $O(n^2)$ that computes $\bigwedge S_{\gamma_1 \vee \gamma_2 \models}$, and that respects the same vtree as D, so let $D' = \mathsf{Apply}(D, D_{\gamma_1 \vee \gamma_2 \models}, \wedge)$ whose size is $O(n^2|D|)$. We have that:

$$\begin{aligned} D' &\equiv \psi \wedge \bigwedge (S' \cup S_{\gamma_1 \vee \gamma_2 \models}) = \psi \wedge \bigwedge (S \setminus \{\gamma_1 \vee \gamma_2\}) \\ &\equiv \psi \wedge \bigwedge_{c_1 \in \phi_1 \setminus \gamma_1} \bigwedge_{c_2 \in \phi_2} (c_1 \vee c_2) \wedge \bigwedge_{c_2 \in \phi_2 \setminus \gamma_2} \bigwedge_{c_1 \in \phi_1} (c_1 \vee c_2) \\ &\equiv \psi \wedge ((\phi_1 \setminus \gamma_1) \vee \phi_2) \wedge (\phi_1 \vee (\phi_2 \setminus \gamma_2)) \end{aligned}$$

The next step is to get rid of ψ, or rather to replace it somehow by the function $\mathrm{ODD}(X)$ that is more convenient to manipulate.

▷ Claim 12. We have that $\mathrm{ODD}(X) \models \exists Y.\psi(X,Y)$.

In knowledge compilation, the *forgetting* transformation is, given a Boolean function representation Σ in some fixed language L and a set $Z \subset var(\Sigma)$, to modify Σ into another function representation Σ' in L that is equivalent to $\exists Z.\Sigma$ [6]. Forgetting is feasible in linear time for the language of circuits in strDNNF [19]. So we can construct a circuit D'' in strDNNF representing $\exists Y.D'(X,Y)$ and such that $|D''| = O(|D'|) = O(n^2|D|)$. Since ϕ_1 and ϕ_2 are formulas over X, we have that

$$D''(X) \equiv \exists Y.D'(X,Y) \equiv ((\phi_1 \setminus \gamma_1) \vee \phi_2) \wedge (\phi_1 \vee (\phi_2 \setminus \gamma_2)) \wedge \exists Y.\psi(X,Y).$$

For every vtree over X, there exists a strDNNF circuit that represents $\mathrm{ODD}(X)$, that respects the vtree, and whose size is in $O(|X|)$ [20, Proposition 5]. Let D_{ODD} be such a circuit respecting the same vtree as D''. Then we have $D''' = \mathsf{Apply}(D'', D_{\mathrm{ODD}}, \wedge)$ whose size is $O(|X||D''|) = O(n^4|D|)$ and such that

$$D'''(X) \equiv D''(X) \wedge \mathrm{ODD}(X) \equiv \big((\phi_1 \setminus \gamma_1) \vee \phi_2\big) \wedge \big(\phi_1 \vee (\phi_2 \setminus \gamma_2)\big) \wedge \mathrm{ODD}(X),$$

where the last equivalence follows from Claim 12. Furthermore, since $\mathrm{ODD}(X) \wedge \phi_1(X) = \mathrm{ODD}(X) \wedge row_n(X)$ is unsatisfiable (because n is even), we have that

$$D'''(X) \equiv \Big(\big((\phi_1 \setminus \gamma_1) \wedge (\phi_2 \setminus \gamma_2)\big) \vee \phi_2\Big) \wedge \mathrm{ODD}(X).$$

We have gotten rid of ψ as promised. The rest of the proof requires a case disjunction over γ_1 and γ_2.

- If $\gamma_1 = (x_{i,1} \vee \cdots \vee x_{i,n})$ and $\gamma_2 = (\overline{x}_{i',j} \vee \overline{x}_{i'',j})$, we can assume, without loss of generality, that $i \neq i'$ and $i \neq i''$ for otherwise $\gamma_1 \vee \gamma_2$ would be tautological. We consider the partial assignment a to X that maps all variables of the ith row to 0, that maps $x_{i',j}$ and $x_{i'',j}$ to 1, and all other variable of the i'th and i''th row 0, and that maps two columns distinct from the jth column to 0. We denote by X' all other variables left unassigned. For instance when $i = 1$, $i' = 2$, $i'' = 3$ and $j = 1$ we may have:

$$a = \begin{pmatrix} 0 & 0 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & 0 & & & \\ \vdots & \vdots & \vdots & & X' & \\ 0 & 0 & 0 & & & \end{pmatrix}.$$

  Then we observe that $(\phi_1 \setminus \gamma_1)|a = (row_n \setminus \gamma_1)|a = row_{n-3}(X')$, that $\phi_2|a \equiv 0$, that $(\phi_2 \setminus \gamma_2)|a \equiv \mathrm{AMOpCOL}_{n-3}(X')$, and that $\mathrm{ODD}(X)|a = \mathrm{ODD}(X')$. Thus

$$D'''(X)|a \equiv row_{n-3}(X') \wedge \mathrm{AMOpCOL}_{n-3}(X') \wedge \mathrm{ODD}(X'),$$

  but since $n-3$ is odd, we have that $row_{n-3}(X') \models \mathrm{ODD}(X')$ and thus

$$D'''(X)|a \equiv row_{n-3}(X') \wedge \mathrm{AMOpCOL}_{n-3}(X') \equiv \mathrm{ROW}_{n-3}(X') \wedge \mathrm{COL}_{n-3}(X').$$

- If $\gamma_1 = (\overline{x}_{i,j} \vee \overline{x}_{i,j'})$ and $\gamma_2 = (\overline{x}_{i,j} \vee \overline{x}_{i',j})$, then we consider the partial assignment a to X that maps $x_{i,j}$, $x_{i,j'}$ and $x_{i',j}$ to 1, and all other variables of the ith and i'th rows to 0, and all other variables of the jth and j'th column to 0. We denote by X' all other variables left unassigned. For instance when $i = 1$, $i' = 2$, $j = 1$ and $j' = 2$ we have:

$$a = \left(\begin{array}{cc|ccc|}
1 & 1 & 0 & \cdots & 0 \\
1 & 0 & 0 & \cdots & 0 \\ \cline{3-5}
0 & 0 & & & \\
\vdots & \vdots & & X' & \\
0 & 0 & & & \\ \cline{3-5}
\end{array}\right).$$

Then we observe that $(\phi_1 \setminus \gamma_1)|a = (row_n \setminus \gamma_1)|a = row_{n-2}(X')$, that $\phi_2|a \equiv 0$, that $(\phi_2 \setminus \gamma_2)|a \equiv \mathrm{AMOpCOL}_{n-2}(X')$, and that $\mathrm{ODD}(X)|a = \mathrm{EVEN}(X')$. Thus

$$D'''(X)|a \equiv row_{n-2}(X') \wedge \mathrm{AMOpCOL}_{n-2}(X') \wedge \mathrm{EVEN}(X'),$$

but since $n-2$ is even, we have that $row_{n-2}(X') \models \mathrm{EVEN}(X')$ so

$$D'''(X)|a \equiv row_{n-2}(X') \wedge \mathrm{AMOpCOL}_{n-2}(X') \equiv \mathrm{ROW}_{n-2}(X') \wedge \mathrm{COL}_{n-2}(X').$$

- If $\gamma_1 = (\overline{x}_{i,j} \vee \overline{x}_{i,j'})$ and $\gamma_2 = (\overline{x}_{i',j''} \vee \overline{x}_{i'',j''})$, where j j' and j'' are pairwise distinct and where i, i' and i'' are pairwise distinct, then we consider the partial assignment a to X that maps $x_{i,j}$, $x_{i,j'}$, $x_{i',j''}$ and $x_{i'',j''}$ to 1, and that maps all other variables of the ith, i'th and i''th rows to 0, and that maps all other variables of the jth, j'th and j''th columns to 0. We denote by X' all other variables left unassigned. For instance when $i = 1$, $i' = 2$, $i'' = 3$ and $j = 2$, $j' = 3$, $j'' = 1$ we have:

$$a = \left(\begin{array}{ccc|ccc|}
0 & 1 & 1 & 0 & \cdots & 0 \\
1 & 0 & 0 & 0 & \cdots & 0 \\
1 & 0 & 0 & 0 & \cdots & 0 \\ \cline{4-6}
0 & 0 & 0 & & & \\
\vdots & \vdots & \vdots & & X' & \\
0 & 0 & 0 & & & \\ \cline{4-6}
\end{array}\right).$$

Then we observe that $(\phi_1 \setminus \gamma_1)|a = (row_n \setminus \gamma_1)|a = row_{n-3}(X')$, that $\phi_2|a \equiv 0$, that $(\phi_2 \setminus \gamma_2)|a \equiv \mathrm{AMOpCOL}_{n-3}(X')$, and that $\mathrm{ODD}(X)|a = \mathrm{ODD}(X')$. Thus

$$D'''(X)|a \equiv row_{n-3}(X') \wedge \mathrm{AMOpCOL}_{n-3}(X') \wedge \mathrm{ODD}(X'),$$

but since $n-3$ is odd, we have that $row_{n-3}(X') \models \mathrm{ODD}(X')$ and thus

$$D'''(X)|a \equiv row_{n-3}(X') \wedge \mathrm{AMOpCOL}_{n-3}(X') \equiv \mathrm{ROW}_{n-3}(X') \wedge \mathrm{COL}_{n-3}(X').$$

All other cases are either similar to the these three, or correspond to cases where $\gamma_1 \vee \gamma_2$ is tautological or entailed by some other clause of S, which is impossible due to the assumptions on k at the beginning of the proof. So, in all relevant cases, we have a partial assignment a to X such that

$$D'''(X)|a \equiv \mathrm{ROW}_{n-\Delta}(X') \wedge \mathrm{COL}_{n-\Delta}(X')$$

for some $\Delta \leq 3$. Then we just condition D''' on a in linear time and we obtain a strDNNF circuit over X' that computes $\mathrm{ROW}_{n-\Delta}(X') \wedge \mathrm{COL}_{n-\Delta}(X')$. This concludes the proof of Lemma 8. Given Lemmas 7 and 9, Theorem 1 follows.

5 Discussion and Conclusion

In this paper we have shown that non-incremental bottom-up (BU) compilation largely outperforms incremental BU compilation space-wise. This raises several questions which we address here. Perhaps the first that comes to mind is: what does it mean regarding potential improvements of practical BU compilers? We have mentioned a non-incremental BU compilation algorithm in Proposition 3 and the discussion before but only to give an example that could be simulated by incremental BU compilation. Looking back at Figure 2, it seems clustering is the key to the efficient compilation of ϕ into OBDD: first the clauses are partitioned into disjoint clusters, then each cluster is compiled, and finally the resulting circuits/diagrams are compiled incrementally together. Clustering is used in OBDD-based refutations where heuristics create each cluster so that a variable can be forgotten (i.e., existentially quantified out) after compilation of the cluster [17, 18]. Contrary to refutations, compilations do not allow forgetting variables so one should think of different clustering strategies. But it is reasonable to believe that some formula can actually be compiled quite efficiently incrementally while a systematic clustering step into a constant number of clusters would make the compilation much harder. Actually it is not clear this work will fuel any practical research on BU compilation. Indeed it is not hard to buy into the claim that hypothetical efficient non-incremental BU compilers ought to be much more complex that incremental ones. Aguably, finding a good ordering of the clauses for incremental compilers is not an easy task and it is reasonable to expect that finding a good “DAG ordering” or “tree ordering” is much harder as the space of possible orderings gets larger.

A discussion that we think is more interesting is about the strength of the frameworks used to analyze practical algorithms. The framework for BU compilation is very general and thus exponential lower bounds in this framework are strong results. On the other hand, positive results may not translate into practical observations as they are obtained in a framework that is too general. In our case, the positive results on the BU compilation of the formulas from Theorem 1 will not be observed for any of the practical BU compilers that we are aware of, as they all work (close to) incrementally. Going further, narrow frameworks can also help explaining the behavior of algorithms on certain instances. For instance in [8], specific CNF formulas are shown to be hard for BU compilation and the proof distinguishes two cases: one where the last Apply conjoins representations for two large subformulas, and another one where it conjoins at least one representation of a subformula with only a few clauses. To explain the behavior of incremental BU compilers on these formulas, the second case would be enough, and this case is arguably the easiest one in [8]. A future direction for us is the comparison of top-down (TD) compilers and BU compilers. Are there formulas that are provably hard for TD compilers (described in a suitable framework) but easy for BU compilers, and vice-versa? For the separation to be observed empirically, tractability results for BU compilation should use the incremental approach. In particular, should the formulas of Theorem 1 be hard for TD compilers, we would not be completely satisfied saying that these formulas answer one direction of the problem.

From a purely theoretical perspective, there are still open problems on BU compilation. One that is related to this work is the separation of tree-like and general BU compilation. A careful observation of the nice BU compilations of the formulas of Theorem 1 shows that they are tree-like, that is, every intermediate OBDD is an input to at most one Apply. We ask whether there exists a class of formulas that have polynomial-size BU compilation but such that all tree-like BU compilations generate intermediate results of exponential size.

References

1. Albert Atserias, Phokion G. Kolaitis, and Moshe Y. Vardi. Constraint Propagation as a Proof System. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 – October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2004. doi:10.1007/978-3-540-30201-8_9.
2. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.
3. Arthur Choi and Adnan Darwiche. Dynamic Minimization of Sentential Decision Diagrams. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. AAAI Press, 2013. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6470.
4. A. Darwiche. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *Proc. of IJCAI'11*, pages 819–826, 2011.
5. Adnan Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, 2001. doi:10.1145/502090.502091.
6. Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *J. Artif. Intell. Res.*, 17:229–264, 2002. doi:10.1613/jair.989.
7. Alexis de Colnet. A Lower Bound on DNNF Encodings of Pseudo-Boolean Constraints. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 312–321. Springer, 2020. doi:10.1007/978-3-030-51825-7_22.
8. Alexis de Colnet and Stefan Mengel. Lower Bounds on Intermediate Results in Bottom-Up Knowledge Compilation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 – March 1, 2022*, pages 5564–5572. AAAI Press, 2022. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20496.
9. Luke Friedman and Yixin Xu. Exponential Lower Bounds for Refuting Random Formulas Using Ordered Binary Decision Diagrams. In Andrei A. Bulatov and Arseny M. Shur, editors, *Computer Science – Theory and Applications – 8th International Computer Science Symposium in Russia, CSR 2013, Ekaterinburg, Russia, June 25-29, 2013. Proceedings*, volume 7913 of *Lecture Notes in Computer Science*, pages 127–138. Springer, 2013. doi:10.1007/978-3-642-38536-0_11.
10. Jinbo Huang and Adnan Darwiche. Using DPLL for Efficient OBDD Construction. 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/61.pdf.
11. 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.
12. Dmitry Itsykson, Alexander Knop, Andrei E. Romashchenko, and Dmitry Sokolov. On OBDD-based Algorithms and Proof Systems that Dynamically Change the order of Variables. *J. Symb. Log.*, 85(2):632–670, 2020. doi:10.1017/jsl.2019.53.
13. Dmitry Itsykson, Artur Riazanov, and Petr Smirnov. Tight Bounds for Tseitin Formulas. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 6:1–6:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.6.

14 Jan Krajícek. An exponential lower bound for a constraint propagation proof system based on ordered binary decision diagrams. *J. Symb. Log.*, 73(1):227–237, 2008. doi:10.2178/jsl/1208358751.

15 Pierre Marquis. Compile! In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 4112–4118. AAAI Press, 2015. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9596.

16 Nina Narodytska and Toby Walsh. Constraint and Variable Ordering Heuristics for Compiling Configuration Problems. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 149–154, 2007. URL: http://ijcai.org/Proceedings/07/Papers/022.pdf.

17 Guoqiang Pan and Moshe Y. Vardi. Search vs. Symbolic Techniques in Satisfiability Solving. 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/62.pdf.

18 Guoqiang Pan and Moshe Y. Vardi. Symbolic Techniques in Satisfiability Solving. *J. Autom. Reason.*, 35(1-3):25–50, 2005. doi:10.1007/s10817-005-9009-7.

19 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.

20 Knot Pipatsrisawat and Adnan Darwiche. Top-Down Algorithms for Constructing Structured DNNF: Theoretical and Practical Implications. In Helder Coelho, Rudi Studer, and Michael J. Wooldridge, editors, *ECAI 2010 – 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 3–8. IOS Press, 2010. URL: http://www.booksonline.iospress.nl/Content/View.aspx?piid=17704.

21 Steven D. Prestwich. CNF Encodings. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 75–100. IOS Press, 2021. doi:10.3233/FAIA200985.

22 Nathan Segerlind. On the Relative Efficiency of Resolution-Like Proofs and Ordered Binary Decision Diagram Proofs. In *Proceedings of the 23rd Annual IEEE Conference on Computational Complexity, CCC 2008, 23-26 June 2008, College Park, Maryland, USA*, pages 100–111. IEEE Computer Society, 2008. doi:10.1109/CCC.2008.34.

23 Olga Tveretina, Carsten Sinz, and Hans Zantema. Ordered Binary Decision Diagrams, Pigeonhole Formulas and Beyond. *J. Satisf. Boolean Model. Comput.*, 7(1):35–58, 2010. doi:10.3233/sat190074.

24 Romain Wallon and Stefan Mengel. Revisiting Graph Width Measures for CNF-Encodings. *J. Artif. Intell. Res.*, 67:409–436, 2020. doi:10.1613/jair.1.11750.

25 Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000. URL: http://ls2-www.cs.uni-dortmund.de/monographs/bdd/.

A Appendix

A.1 Missing Proofs of Section 3

We show the following lemma before proving Proposition 3.

▶ **Lemma 13.** *Let ϕ_1 and ϕ_2 be such that $var(\phi_1) \cap var(\phi_2) = \emptyset$. Given $(D^1_1, I^1_1), \dots, (D^1_s, I^1_s)$ an incremental strDNNF(∧,r) compilation of ϕ_1, and $(D^2_1, I^2_1), \dots, (D^2_t, I^2_t)$ an incremental strDNNF(∧,r) compilation of ϕ_2, there is an incremental strDNNF(∧,r) compilation of $\phi_1 \wedge \phi_2$ whose largest elements has size at most $\max_i |D^1_i| + \max_j |D^2_j| + 1$.*

Proof. Just take the compilation $(D_1, I_1), \ldots, (D_s, I_s), (D_{s+1}, I_{s+1}), \ldots, (D_{s+t}, I_{s+t})$ where $I_i = I_i^1$ and $D_i = D_i^1$ for $1 \leq i \leq s$ and I_{s+j} is I_j where every D_j^2 is replaced by $D_{s+j} = D_s \wedge D_j^2$ (that is, the root of D_{s+j} is one $\wedge$-node whose children are D_s and D_j^2). Since $var(D_s) \cap var(D_j^2) \subseteq var(\phi_1) \cap var(\phi_2) = \emptyset$, each D_{s+j} is a strDNNF circuit. It is a bottom-up compilation of $\phi_1 \wedge \phi_2$ whose size is at most $\max(\max_i |D_i^1|, 1 + |D_s^1| + \max_j |D_j^2|)$. ◀

▶ **Proposition 3.** *Every strDNNF($\wedge$,r) compilation of a CNF formula ϕ where every* Apply *that does not involve a clause of ϕ computes the conjunction of two strDNNF circuits representing subformulas of ϕ over disjoint set of variables, can be transformed into an incremental strDNNF($\wedge$,r) compilation with only a polynomial increase in size.*

Proof. We can restrict our focus to BU compilations where all circuits constructed, except the last one, are used as premises for some Restructure or Apply instructions. For D an intermediate circuit in the compilation, let $pred(D)$ be the set containing D plus all circuits constructed before D in the compilation and that are used in the construction of D. More formally, if $D = \mathsf{Compile(C)}$ then $pred(D) = \{D\}$, if $D = \mathsf{Apply}(D', D'', \wedge)$ then $pred(D) = \{D\} \cup pred(D') \cup pred(D'')$ and if $D = \mathsf{Restructure}(D')$ then $pred(D) = \{D\} \cup pred(D')$. In particular, calling $D^* \equiv \phi$ the final circuit of the compilation, $pred(D^*)$ is the set of all circuits generated during the compilation. Let further k_D be the number of decomposable Apply done to construct D. Consider the last decomposable Apply in the compilation: $D = \mathsf{Apply}(D', D'', \wedge)$ with $var(D') \cap var(D'') = \emptyset$. Suppose there is an incremental strDNNF($\wedge$,r) compilation of D' (resp. D'') whose largest circuit has size at most $k_{D'} + \sum_{K \in pred(D')} |K|$ (resp. $k_{D''} + \sum_{K \in pred(D'')} |K|$). By Lemma 13, there is an incremental BU compilation of D that generates only circuits of size at most $1 + k_{D'} + k_{D''} + \sum_{K \in pred(D')} |K| + \sum_{K \in pred(D'')} |K|$. And since $pred(D')$ and $pred(D'')$ are disjoint for a decomposable Apply, this upper bound equals $k_D + \sum_{K \in pred(D)} |K|$. Calling $D^* \equiv \phi$ the final circuit of the compilation, we obtain by induction that ϕ can be compiled by an incremental strDNNF($\wedge$,r) compilation that generates only circuits of size at most $k_{D^*} + \sum_{K \in pred(D^*)} |K|$ where $k_{D^*} = O(|\phi|)$, which is a polynomial size increase compared to the original compilation. ◀

We are now going to prove Proposition 4. Before that, we recall a seemingly insignificant property of OBDDs, SDDs and strDNNF circuits: given a set X of variables, for every vtree T over X, every clause c over X has a representation in OBDD, SDD and strDNNF that respects T and whose size is $O(|c|)$.

▶ **Proposition 4.** *Let $L \in \{OBDD, SDD, strDNNF\}$, let $(\Sigma_1, I_1), \ldots, (\Sigma_N, I_N)$ be an incremental L($\wedge$,r) compilation of the CNF formula ϕ over n variables. There exists an incremental L($\wedge$,r) compilation $(\Sigma'_1, I'_1), \ldots, (\Sigma'_M, I'_M)$ of ϕ such that $M \leq 2N$ and, for every $j \in \{1, \ldots, M\}$ there is an $i \in \{1, \ldots, N\}$ such that $\Sigma'_j \equiv \Sigma_i$ and $|\Sigma'_j| \leq 2n|\Sigma_i|$. In addition, when Σ'_j is obtained by restructuring Σ'_{j-1}, we have $|\Sigma'_j| \leq |\Sigma'_{j-1}|$.*

Proof. We follow the instructions $I_1, \ldots, I_N$ and construct $(\Sigma'_1, I'_1), \ldots, (\Sigma'_M, I'_M)$ along the way. We denote by S_i the subsequence of this sequence, that we have after obtaining Σ_i. We begin with $\Sigma'_1 = \Sigma_1$, $I'_1 = I_1$ and $S_1 = ((\Sigma'_1, I'_1))$. We will denote by $S_i \cdot x$ the sequence obtained adding x at the end of S_i. We say that invariant (I) holds at step i when, after Σ_i is constructed, denoting $S_i = ((\Sigma'_1, I'_1), \ldots, (\Sigma'_j, I'_j))$, we have that $\Sigma'_j \equiv \Sigma_i$ and $|\Sigma'_j| \leq |\Sigma_i|$. (I) clearly holds at step 1. Now let us assume that (I) holds at step i, let $j = |S_i|$, and consider the instruction I_{i+1}. There are three cases to consider.

- I_{i+1} is $\Sigma_{i+1} = \mathsf{Restructure}(\Sigma_i)$. If $|\Sigma_{i+1}| > |\Sigma'_j|$ then, since $\Sigma_{i+1} \equiv \Sigma_i \equiv \Sigma'_j$, we simply define $S_{i+1} = S_i$. Otherwise, if $|\Sigma_{i+1}| \leq |\Sigma'_j|$ then we define $\Sigma'_{j+1} = \Sigma_{i+1}$ and $I'_{j+1} : \Sigma'_{j+1} = \mathsf{Restructure}(\Sigma'_j)$ (recall that $\Sigma'_j \equiv \Sigma_i \equiv \Sigma_{i+1} = \Sigma'_{j+1}$ by the invariant) and $S_{i+1} = S_i \cdot (\Sigma'_{j+1}, I'_{j+1})$. In both cases, invariant (I) holds at step $i + 1$.

- I_{i+1} is $\Sigma_{i+1} = \mathsf{Apply}(\Sigma_i, \Sigma_c, \wedge)$ where $c \in \phi$. Let Σ'_c be the *linear-size* representation of c in L that respects the same vtree as Σ'_j. Now define I'_{j+1} as $\Sigma'_{j+1} = \mathsf{Apply}(\Sigma'_j, \Sigma'_c, \wedge)$. We have that $\Sigma'_j \equiv \Sigma_i$ so $\Sigma'_{j+1} \equiv \Sigma_{i+1}$. Moreover, since $|\Sigma'_c| \leq 2|var(c)| \leq 2n$ and since the Apply is a quadratic-time procedure, we have that $|\Sigma'_{j+1}| \leq 2n|\Sigma'_j| \leq 2n|\Sigma_i|$. If $|\Sigma'_{j+1}| \leq |\Sigma_{i+1}|$, then we define $S_{i+1} = S_i \cdot (\Sigma'_{j+1}, I'_{j+1})$. Otherwise if $|\Sigma'_{j+1}| > |\Sigma_{i+1}|$, then we define $\Sigma'_{j+2} = \Sigma_{i+1}$, $I'_{j+2} : \Sigma'_{j+2} = \mathsf{Restructure}(\Sigma'_{j+1})$ and $S_{i+1} = S_i \cdot (\Sigma'_{j+1}, I'_{j+1}) \cdot (\Sigma'_{j+2}, I'_{j+2})$. In both cases, invariant (I) holds at step $i+1$.

At the end of the construction we have $S_N = ((\Sigma'_1, I'_1), \dots, (\Sigma'_M, I'_M))$ and $\Sigma'_M \equiv \Sigma_N \equiv \phi$ since (I) holds at step N. At every step, we add at most two elements to S_i, so $M \leq 2N$. Furthermore, for every $j \geq 1$, Σ'_{j+1} is either the result of an Apply between Σ'_j and a clause of ϕ, or comes from restructuring Σ'_j. So S_N is an incremental $L(\wedge,r)$ compilation of ϕ. Looking back at our construction of S_i, we see that (1) restructuring is only used when it decreases the size of the OBDD and (2) that every Σ'_j is equivalent to some Σ_i and that its size is never greater than $2n|\Sigma_i|$. ◀

A.2 Missing Proofs of Section 4

▷ Claim 6. $B_1 \wedge \dots \wedge B_j$ is equivalent to the OBDD represented in Figure (3a).

Proof. Recall that $B_j \equiv \overline{x}_{i,j} \vee \bigwedge_{k>j} \overline{x}_{i,k}$ where i is fixed. So $B_1 \wedge B_2 \wedge \dots \wedge B_j \equiv \left(\overline{x}_{i,1} \vee \bigwedge_{k>1} \overline{x}_{i,k}\right) \wedge \left(\overline{x}_{i,2} \vee \bigwedge_{k>2} \overline{x}_{i,k}\right) \wedge \dots \wedge \left(\overline{x}_{i,j} \vee \bigwedge_{k>j} \overline{x}_{i,k}\right)$. We argue by case disjunction on $x_{i,1} + \dots + x_{i,j}$. First, any assignment to $(x_{i,k})_k$ that satisfies $x_{i,1} + \dots + x_{i,j} = 0$ clearly satisfies $B_1 \wedge B_2 \wedge \dots \wedge B_j$. Second, any assignment that satisfies $x_{i,1} + \dots + x_{i,j} = 1$ satisfies $B_1 \wedge B_2 \wedge \dots \wedge B_j$ if and only if it satisfies $\bigwedge_{k>j} \overline{x}_{i,k}$. Finally, any assignment that satisfies $x_{i,1} + \dots + x_{i,j} > 1$ falsifies $B_1 \wedge B_2 \wedge \dots \wedge B_j$. Thus $B_1 \wedge B_2 \wedge \dots \wedge B_j$ is equivalent to

$$(x_{i,1} + \dots + x_{i,j} = 0) \vee ((x_{i,1} + \dots + x_{i,j} = 1) \wedge (x_{i,j+1} + \dots + x_{i,n} = 0))$$

It is clear that the OBDD represented in Figure (3b) computes the function written above. ◁

▷ Claim 10. For any $h > k+1$ such that $D_h = \mathsf{Apply}(D_{h-1}, D_{c_1 \vee c_2}, \wedge)$ and $c_1 \in row_n$, we have that $D_k \models c_1 \vee c_2$ or $\gamma_1 \vee \gamma_2 \models c_1 \vee c_2$.

Proof. Suppose there is an $h > k+1$ such that $D_h = \mathsf{Apply}(D_{h-1}, D_{c_1 \vee c_2}, \wedge)$, $c_1 \in row_n$, and $D_k \not\models c_1 \vee c_2$ and $\gamma_1 \vee \gamma_2 \not\models c_1 \vee c_2$. If there is no other clause of ϕ that entails $c_1 \vee c_2$ then this contradicts the fact that k is maximal. Otherwise, if $c_1 \vee c_2$ is entailed by another clause $c'_1 \vee c'_2$ of ϕ, then $c'_1 \in row_n(X)$ since $c_1 \vee c_2$ uses only variables of X while every clause of $odd_n(X,Y)$ features an encoding variable of Y. In addition, the integer h' such that $D_{h'} = \mathsf{Apply}(D_{h'-1}, D_{c'_1 \vee c'_2}, \wedge)$ is greater than $k+1$, for otherwise we would have either $D_k \models D_{h'} \models D_{c'_1 \vee c'_2} \models D_{c_1 \vee c_2} \models c_1 \vee c_2$ (for $h' < k+1$) or $\gamma_1 \vee \gamma_2 = c'_1 \vee c'_2 \models c_1 \vee c_2$ (for $h' = k+1$). So we repeat the argument with the clause $c'_1 \vee c'_2$ instead of $c_1 \vee c_2$ until reaching a contradiction. ◁

▷ Claim 12. We have that $\mathrm{ODD}(X) \models \exists Y.\psi(X,Y)$.

Proof. First note that we have $odd_n(X,Y) \models odd_n(X,Y) \vee \phi_2(X) \models \psi(X,Y)$ since ψ is a subformula of a formula equivalent to $odd_n(X,Y) \vee \phi_2(X)$. Since $\exists Y.odd_n(X,Y) \equiv \mathrm{ODD}(X)$, it only remains to explain that $\exists Y.odd_n(X,Y) \models \exists Y.\psi(X,Y)$. Let a_X be an assignment to X satisfying $\exists Y.odd_n(X,Y)$. Then there exists an assignment a_Y to Y such that $a_X \cup a_Y$ satisfies $odd_n(X,Y)$. But then $a_X \cup a_Y$ satisfies $\psi(X,Y)$, and thus a_X satisfies $\exists Y.\psi(X,Y)$. So $\exists Y.odd_n(X,Y) \models \exists Y.\psi(X,Y)$. ◁

▷ Claim 11. Let $\gamma = \gamma_1 \vee \gamma_2$. For every vtree over X, there is a strDNNF of size $O(n^2)$ respecting that vtree that represents $\bigwedge S_{\gamma\models}$ (by convention $\bigwedge \emptyset = 1$).

Proof. Some clauses of S are tautological, that is, they contain a variable x and its negation $\overline{x}$ and are thus equivalent to 1, let $S_{taut} \subset S$ be the set of such clauses. We now consider the different possibilities for $\gamma_1 \vee \gamma_2$.

- If $\gamma_1 = (x_{i,1} \vee \cdots \vee x_{i,n})$ and $\gamma_2 = (\overline{x}_{i,j} \vee \overline{x}_{i',j})$ then γ is tautological so $S_{\gamma\models} \subseteq S_{taut}$.
- If $\gamma_1 = (x_{i,1} \vee \cdots \vee x_{i,n})$ and $\gamma_2 = (\overline{x}_{i',j} \vee \overline{x}_{i'',j})$ where i, i' and i'' are pairwise distinct, then $S_{\gamma\models} \subseteq S_{taut}$.
- If $\gamma_1 = (\overline{x}_{i,j} \vee \overline{x}_{i,j'})$ and $\gamma_2 = (\overline{x}_{i',j''} \vee \overline{x}_{i'',j''})$ where i, i', i'' are pairwise distinct and where j, j', j'' are pairwise distinct. Then $S_{\gamma\models} \subseteq S_{taut}$.
- If $\gamma_1 = (\overline{x}_{i,j} \vee \overline{x}_{i,j'})$ and $\gamma_2 = (\overline{x}_{i',j} \vee \overline{x}_{i'',j})$ where i, i', i'' are pairwise distinct. Then $S_{\gamma\models} \subseteq S_{taut}$.
- If $\gamma_1 = (\overline{x}_{i,j} \vee \overline{x}_{i,j'})$ and $\gamma_2 = (\overline{x}_{i,j''} \vee \overline{x}_{i',j''})$ where j, j', j'' are pairwise distinct. Then $S_{\gamma\models} \subseteq S_{taut}$.
- If $\gamma_1 = (\overline{x}_{i,j} \vee \overline{x}_{i,j'})$ and $\gamma_2 = (\overline{x}_{i,j} \vee \overline{x}_{i',j})$, then $S_{\gamma\models}$ is the set of all clauses of the form $\gamma_1 \vee (\overline{x}_{i',j} \vee \overline{x}_{i'',j})$ for $i'' \neq i$ and $i'' \neq i'$, plus all clauses of the form $(\overline{x}_{i,j'} \vee \overline{x}_{i,j''}) \vee \gamma_2$ for $j'' \neq j$ and $j'' \neq j'$, plus all the tautological clauses.

The claim is trivially true when $S_{\gamma\models} \subseteq S_{taut}$, so we only have the last case to consider. For any given vtree, we want an small strDNNF circuit that computes $\bigwedge_{i'' \notin \{i,i'\}}(\gamma_1 \vee \overline{x}_{i',j} \vee \overline{x}_{i'',j}) \wedge \bigwedge_{j'' \notin \{j,j'\}}(\overline{x}_{i,j'} \vee \overline{x}_{i,j''} \vee \gamma_2) \equiv \left(\gamma_1 \vee \overline{x}_{i',j} \vee \bigwedge_{i'' \notin \{i,i'\}} \overline{x}_{i'',j}\right) \wedge \left(\gamma_2 \vee \overline{x}_{i,j'} \vee \bigwedge_{j'' \notin \{j,j'\}} \overline{x}_{i,j''}\right)$. It is readily verified that, for any vtree, there are strDNNF circuits of size $O(n)$ that computes $\bigwedge_{i'' \notin \{i,i'\}} \overline{x}_{i'',j}$ and $\bigwedge_{j'' \notin \{j,j'\}} \overline{x}_{i,j''}$ and that there are strDNNF circuits of size of size $O(1)$ computing $\gamma_1 \vee \overline{x}_{i',j}$ and $\gamma_2 \vee \overline{x}_{i,j'}$. So, for any vtree, we start from these strDNNF circuits and apply two disjunctions and one conjunction to obtain a strDNNF circuit of size $O(n^2)$ computing the desired function. ◁