Variable Shift SDD: A More Succinct Sentential Decision Diagram

Kengo Nakamura
NTT Communication Science Laboratories, Kyoto, Japan
kengo.nakamura.dx@hco.ntt.co.jp

Shuhei Denzumi
Graduate School of Information Science and Technology, The University of Tokyo, Japan
denzumi@mist.i.u-tokyo.ac.jp

Masaaki Nishino
NTT Communication Science Laboratories, Kyoto, Japan
masaaki.nishino.uh@hco.ntt.co.jp

Abstract

The Sentential Decision Diagram (SDD) is a tractable representation of Boolean functions that subsumes the famous Ordered Binary Decision Diagram (OBDD) as a strict subset. SDDs are attracting much attention because they are more succinct than OBDDs, as well as having canonical forms and supporting many useful queries and transformations such as model counting and `Apply` operation. In this paper, we propose a more succinct variant of SDD named *Variable Shift SDD* (VS-SDD). The key idea is to create a unique representation for Boolean functions that are equivalent under a specific variable substitution. We show that VS-SDDs are never larger than SDDs and there are cases in which the size of a VS-SDD is exponentially smaller than that of an SDD. Moreover, despite such succinctness, we show that numerous basic operations that are supported in polytime with SDD are also supported in polytime with VS-SDD. Experiments confirm that VS-SDDs are significantly more succinct than SDDs when applied to classical planning instances, where inherent symmetry exists.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Computing methodologies → Knowledge representation and reasoning

Keywords and phrases Boolean function, Data structure, Sentential decision diagram

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.22

Related Version A full version of the paper is available at `https://arxiv.org/abs/2004.02502`.

1 Introduction

The succinct representations of a Boolean function have long been studied in the computer science community. Among them, the *Ordered Binary Decision Diagram* (OBDD) [5] has been used as a prominent tool in various applications. An OBDD represents a Boolean function as a directed acyclic graph (DAG). The reason for the popularity of OBDDs is that it can often represent a Boolean function very succinctly while supporting many useful queries and transformations in polytime with respect to the compilation size.

In the last few years, the *Sentential Decision Diagram* (SDD) [9], which is again a DAG representation, has also attracted attention [23, 20]. SDDs have a tighter bound on the compilation size than OBDDs [9], and there are cases in which the use of SDDs can make the size exponentially smaller than OBDDs [3]. In addition, SDDs also support a number of queries and transformations in polytime. Among them, the most important polytime operation is the `Apply` operation, which takes two SDDs representing two Boolean functions f, g and binary operator $\circ$, such as conjunction ($\wedge$) and disjunction ($\vee$), and returns the

SDD representing the Boolean function $f \circ g$. This operation is fundamental in compiling an arbitrary Boolean function into an SDD, as well as in proving the polytime solvability of various important and useful operations.

One of the reasons why OBDDs and SDDs, as well as many other such DAG representations, can express a Boolean function succinctly is that they share identical substructures that represent the equivalent Boolean function; they represent a Boolean function by recursively decomposing it into subfunctions that can also be represented as DAGs. If a decomposition generates equivalent subfunctions, we do not need to have multiple DAGs, and thus it can more succinctly represent the original Boolean function. Since the effectiveness of such representations depend on the DAG size, representations that are more succinct while still supporting useful operations are always in demand.

In this paper, we propose a new SDD-based structure named *Variable Shift SDD* (VS-SDD); it can even more succinctly represent Boolean functions, while supporting polytime `Apply` operations. The key idea is to extend the condition for sharing DAGs. While an SDD can share DAGs representing identical Boolean functions, a VS-SDD can share DAGs representing Boolean functions that are equivalent under a specific variable substitution. For example, consider two Boolean functions $f = A \wedge B$ and $g = C \wedge D$ defined over variables A, B, C, D. An SDD cannot share DAGs representing f and g since they are not equivalent. On the other hand, VS-SDD can share them since f and g are equivalent under the variable substitution that exchanges A with C and B with D. Such Boolean functions appear in a wide range of situations. One typical example is modeling time-evolving systems; such as, we want to find a sequence of assignments of variables $\mathbf{x}_1, \dots, \mathbf{x}_T$ over timestamps $t = 1, \dots, T$ such that every $\mathbf{x}_t$ satisfies the condition that $h(\mathbf{x}_t) = \mathit{true}$. Such a sequence is modeled as Boolean function $f(\mathbf{X}_1, \dots, \mathbf{X}_T) = h^{(1)}(\mathbf{X}_1) \wedge \cdots \wedge h^{(T)}(\mathbf{X}_T)$, where $h^{(t)}$ is $h(\mathbf{X})$ defined over $\mathbf{X}_t$. Since all $h^{(t)}(\mathbf{X}_t)$ are equivalent under variable substitutions, it is highly possible that VS-SDD can yield more succinct representations.

Technically, these advantages of VS-SDD are obtained by introducing the indirect specification of depending variables. Every SDD is associated with a set of variables that the corresponding Boolean function depends on. In SDD, such set of variables are represented by IDs, where each set of variables has a unique ID. On the other hand, VS-SDD represents such sets of variables by storing the *difference* of IDs. This allows the sharing of the Boolean functions that are equivalent under specific types of variable substitutions.

Our main results are as follows:

- VS-SDDs are never larger than their SDDs equivalents. Moreover, there is a class of Boolean functions for which VS-SDDs are exponentially smaller than SDDs.
- VS-SDD supports polytime `Apply`. Moreover, the queries and transformations listed in [10] that SDDs support in polytime are also supported in polytime by VS-SDDs.
- We experimentally confirm that when applied to classical planning instances, VS-SDDs are significantly smaller than SDDs.

To summarize, VS-SDDs incur no additional overhead over SDDs while being potentially much smaller than SDDs.

The rest of this paper is organized as follows. Sect. 2 reviews related works. Sect. 3 gives the preliminaries. Sect. 4 introduces SDD, on which our proposed structure is based. Sect. 5 describes the formal definition of the equivalence relation we want to share, the definition of VS-SDD, and the relation between them. Sect. 6 examines the properties of VS-SDDs. Sect. 7 deals with the operations on VS-SDDs, especially `Apply`. Sect. 8 mentions some implementation details that ensure that VS-SDDs suffer no overhead penalty relative to SDDs. Sect. 9 provides experiments and their results, and Sect. 10 gives concluding remarks.

2 Related Works

There have been studies that attempt to share the substructures that represent the “equivalent” Boolean functions up to a conversion. For OBDDs, the most famous among them are complement edges and attributed edges [16, 18]. For example, with complement edges, we can share the substructures representing the equivalent Boolean functions up to taking a negation. However, this study does not focus on the solvability of the operations in the compressed form. Actually, some `Apply` operations cannot be performed in a compressed form. After that, the differential BDD [1], especially $\uparrow\Delta$BDD, was proposed to share equivalent Boolean functions up to the shifting of variables, that is, given the total order of the variables, shift them uniformly to share isomorphic substructures. This structure supports operations like `Apply`, but its complexity depends on the number of variables, which means that this operation is not supported in polytime of the compilation size. With regard to other representations, Sym-DDG/FBDD [2], based on DDG [11] and FBDD [12], can share equivalent functions up to variable substitution. Since their method adopts a permutation of variables, it can, in principle, treat any variable substitution. However, Sym-DDG/FBDD fails to support some important operations such as conditioning and `Apply`. With regard to these previous works, VS-SDD differs in three points. First, to the best of our knowledge, VS-SDD is the first attempt to extend the equivalence relationships of an SDD. We should note that VS-SDD is not obtained by a straightforward application of the techniques invented for OBDDs. Second, VS-SDD has theoretical guarantees on its size. Last, it supports the flexible polytime `Apply` operation.

3 Preliminaries

We use an uppercase letter (e.g., X) to represent a variable and a lowercase letter (e.g., x) to denote its assignment (either *true* or *false*). A bold uppercase letter (e.g., $\mathbf{X}$) represents a set of variables and a bold lowercase letter (e.g., $\mathbf{x}$) denotes its assignment. *Boolean function* $f(\mathbf{X})$ is a function that maps each assignment of $\mathbf{X}$ to either *true* or *false*. The *conditioning* of f on instantiation $\mathbf{X}$, written $f|\mathbf{x}$, is the subfunction that results from setting variables $\mathbf{X}$ to their values in $\mathbf{x}$. We say f *essentially depends* on variable X iff $f|X \neq f|\neg X$. We take $f(\mathbf{Z})$ to mean that f can only essentially depend on variables in $\mathbf{Z}$. A *trivial* function maps all its inputs to *false* (denoted *false*) or maps all to *true* (denoted *true*).

Consider an ordered full binary tree. For two nodes u, w in it, we say w is a *left descendant* (resp. *right descendant*) of u if w is a (not necessarily proper) descendant of the left (resp. right) child of u.

4 Sentential Decision Diagrams

First, we introduce SDD. It is a data structure that can represent a Boolean function as a directed acyclic graph (DAG) like OBDD.

Let f be a Boolean function and $\mathbf{X}, \mathbf{Y}$ be non-intersecting sets of variables. The $(\mathbf{X}, \mathbf{Y})$-*decomposition* of f is $f = \bigvee_{i=1}^{n} [p_i(\mathbf{X}) \wedge s_i(\mathbf{Y})]$, where $p_i(\mathbf{X})$ and $s_i(\mathbf{Y})$ are Boolean functions. Here $p_1, \ldots, p_n$ are called *primes* and $s_1, \ldots, s_n$ are called *subs*. We denote $(\mathbf{X}, \mathbf{Y})$-decomposition as $\{(p_1, s_1), \ldots, (p_n, s_n)\}$, where n is the size of the decomposition. The pair (p_i, s_i) is called an *element*. An $(\mathbf{X}, \mathbf{Y})$-decomposition is called $\mathbf{X}$-*partition* iff $p_i \wedge p_j = \mathit{false}$ for all $i \neq j$, $\bigvee_{i=1}^{n} p_i = \mathit{true}$, and $p_i \neq \mathit{false}$ for all i. If $s_i \neq s_j$ for all $i \neq j$, the partition is called *compressed*. It is known that a function $f(\mathbf{X}, \mathbf{Y})$ has exactly one compressed $\mathbf{X}$-partition (see Theorem 3 of [9]).

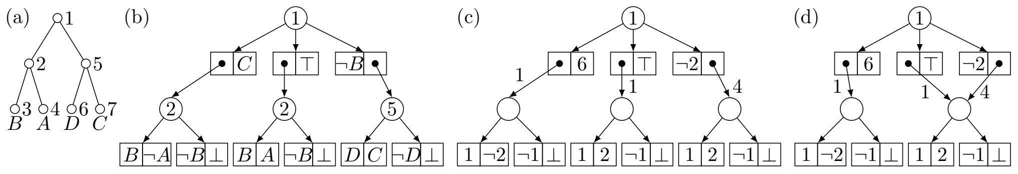

Figure 1 (a) An example of a vtree. (b) The SDD of $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ that respects the vtree of (a). (c)(d) The VS-SDD of $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ given the vtree (a) with offset 1. Here (d) is the more reduced form than (c).

An SDD decomposes a Boolean function by recursively applying $\mathbf{X}$-partitions. The structure of partitions is determined by an ordered full binary tree called the *vtree*; its leaves have a one-to-one correspondence with variables. Here, each internal node partitions the variables into those in the left subtree ($\mathbf{X}$) and those in the right subtree ($\mathbf{Y}$). For example, the vtree in Fig. 1(a) shows the recursive partition of variables A, B, C, D. The root node represents the partition of variables to $\{A, B\}, \{C, D\}$, while the left child of the root represents the partition $\{A\}, \{B\}$. SDD implements $\mathbf{X}$-partitions by following these recursive partitions of variables.

Let $\langle\cdot\rangle$ be a mapping from an SDD to a Boolean function (i.e., the semantics of SDD). The SDD is defined recursively as follows.

▶ **Definition 1.** *The following α is an SDD that respects vtree node v.*

- (constant) $\alpha = \top$ *or* $\alpha = \bot$*. Semantics:* $\langle\top\rangle =$ *true and* $\langle\bot\rangle =$ *false.*
- (literal) $\alpha = X$ *or* $\alpha = \neg X$*, and v is a leaf node with variable X. Semantics:* $\langle X\rangle = X$ *and* $\langle\neg X\rangle = \neg X$*.*
- (decomposition) $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$*, and v is an internal node. Here each p_i is an SDD respecting a left descendant node of v, each s_i is an SDD respecting a right descendant node of v, and $\langle p_1\rangle, \ldots, \langle p_n\rangle$ form a partition. Semantics:* $\langle\alpha\rangle = \bigvee_{i=1}^{n}[\langle p_i\rangle \wedge \langle s_i\rangle]$*.*

The size of α (denoted by $|\alpha|$) is defined as the sum of the sizes of all its decompositions.

Given the vtree of Fig. 1(a), Fig. 1(b) depicts an SDD that respects the vtree node labeled 1 and represents $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$. At the top level, f is decomposed as $[(\neg A \wedge B) \wedge C] \vee [(A \wedge B)] \vee [\neg B \wedge (C \wedge D)]$. This is the compressed $\{A, B\}$-partition since primes $\neg A \wedge B$, $A \wedge B$ and $\neg B$ satisfy the condition for $\{A, B\}$-partition and subs are all different. Here each circle represents a decomposition node, and the number inside each circle indicates the respecting vtree node ID. The size of the SDD is 9.

There are two classes of canonical SDDs. We say a class of SDDs is *canonical* iff, given a vtree, for any Boolean function f, there is exactly one SDD in this class that represents f. Here we consider only *reduced* SDDs, i.e. the SDDs such that the identical substructures are fully merged.

▶ **Definition 2.** *We say SDD α is* compressed *iff all partitions in α are compressed. We say α is* trimmed *iff it does not have decompositions of the form $\{(\top, \beta)\}$ and $\{(\beta, \top), (\neg\beta, \bot)\}$, and* lightly trimmed *iff it does not have decompositions of the form $\{(\top, \top)\}$ and $\{(\top, \bot)\}$. We say α is* normalized *iff for each decomposition that respects vtree node w, its primes respect the left child of w and its subs respect the right child of w.*

▶ **Theorem 3** ([9])**.** *Compressed and trimmed SDDs are canonical. Also, compressed, lightly trimmed, and normalized SDDs are canonical.*

The key property of SDDs is that they support the polytime `Apply` operation, which takes, given vtree v, two SDDs α, β and binary operation $\circ$, and computes a new SDD that represents $\langle\alpha\rangle \circ \langle\beta\rangle$ in $O(|\alpha||\beta|)$ time. Using `Apply`, we can compile an arbitrary Boolean function into an SDD.

5 Variable Shift Sentential Decision Diagrams

We now introduce our more succinct variant of SDD, named *variable shift SDD* (VS-SDD). As mentioned above, an SDD expresses a Boolean function succinctly by sharing equivalent substructures that represent the same Boolean subfunction. The motivation to introduce VS-SDD is, in addition to this, to share substructures that represent equivalent Boolean functions under a particular variable substitution.

First, we briefly describe the idea by using an intuitive example. Let us consider two Boolean functions $f = X_1 \wedge X_2$ and $g = X_3 \wedge X_4$ defined over variables $X_1, \ldots, X_4$. Apparently, f and g are not equivalent, but they are equivalent if we exchange X_1 with X_3 and X_2 with X_4. We formally define this equivalency of Boolean functions below.

▶ **Definition 4.** *We say two Boolean functions f, g defined over $\mathbf{X}$ are* substitution-equivalent *with permutation π if $f(X_1 = x_1, \ldots, X_M = x_M) = g(X_1 = x_{\pi(1)}, \ldots, X_M = x_{\pi(M)})$ for any assignment $\mathbf{x}$, where $M = |\mathbf{X}|$ and $\pi : \{1, \ldots, M\} \mapsto \{1, \ldots, M\}$ is a bijection.*

In the above example, f and g are substitution equivalent with π satisfying $\pi(3) = 1$ and $\pi(4) = 2$. For $i = 1, 2$, this permutation is defined by simply adding constant to an input, i.e., $\pi(i) = i + c$ $(i = 1, 2)$ where the constant $c = 2$. This result implies that a class of substitution-equivalent functions can be represented as the pair of a base representation and constant value c. VS-SDD exploits this idea.

5.1 Definition of the structure

Now we consider the structure and semantics of VS-SDD. VS-SDD shares many properties with SDDs; it is defined with a vtree and a DAG structure representing recursive $\mathbf{X}$-partitions following the vtree. VS-SDD has two main differences from SDD. First, it associates every vtree node with an integer ID and it considers some mathematical operations over them. We use $\mathtt{ID}(v)$ to represent the ID associated with vtree node v, and $\mathtt{ID}^{-1}(i)$ to represent the vtree node that corresponds to ID i. In the following, we assume that integer IDs of vtree nodes are assigned following a preorder traversal of the vtree. The IDs assigned to the vtree in Fig. 1(a) satisfy this condition. Second, while SDD represents a Boolean function as a node of a DAG, VS-SDD represents a Boolean function as a pair (α, k) of node α in a DAG and integer k. We say α is the VS-SDD *structure* and k is its *offset*. We use $\langle \alpha, k \rangle$ as a mapping from VS-SDD (α, k) to the corresponding Boolean function.

▶ **Definition 5.** *Given vtree v, the following (α, k) is a VS-SDD.*

- (constant) $\alpha = \top$ *or* $\alpha = \bot$. *Semantics:* $\langle \top, \cdot \rangle = $ *true and* $\langle \bot, \cdot \rangle = $ *false.*
- (literal) $\alpha = \mathbf{v}$ *or* $\alpha = \neg\mathbf{v}$*, and* $\mathtt{ID}^{-1}(k)$ *is a leaf vtree node. Semantics:* $\langle \mathbf{v}, k \rangle = l(\mathtt{ID}^{-1}(k))$ *and* $\langle \neg\mathbf{v}, k \rangle = \neg l(\mathtt{ID}^{-1}(k))$*, where $l(v)$ is a variable corresponding to vtree node v.*
- (decomposition) $\alpha = \{([p_1, d_1], [s_1, e_1]), \ldots, ([p_n, d_n], [s_n, e_n])\}$*, and* $\mathtt{ID}^{-1}(k)$ *is an internal node of v. Here each p_i is a VS-SDD structure and d_i is an integer such that $\mathtt{ID}^{-1}(d_i + k)$ is a left descendant vtree node of $\mathtt{ID}^{-1}(k)$. Similarly, each s_i is a VS-SDD structure and e_i is integer such that $\mathtt{ID}^{-1}(e_i + k)$ is a right descendant node of $\mathtt{ID}^{-1}(k)$ and Boolean functions $\langle p_1, d_1 + k \rangle, \ldots, \langle p_n, d_n + k \rangle$ form a partition. Semantics: $\langle \alpha, k \rangle = \bigvee_{i=1}^{n} (\langle p_i, d_i + k \rangle \wedge \langle s_i, e_i + k \rangle)$.*

The size of α (denoted by $|\alpha|$) is defined as the sum of the sizes of all decompositions.

Given the vtree of Fig. 1(a), Fig. 1(c)-(d) depict the VS-SDDs representing $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$, where Fig. 1(d) is a further reduced form created by sharing the identical substructures in Fig. 1(c). Here the offset is written in the circle of the root node.

Every prime $[p_i, d_i]$ is drawn as an arrow to structure p_i annotated with d_i, except for the following cases. If $p_i = \mathbf{v}$ (resp. $\neg\mathbf{v}$), it is drawn as simply d_i (resp. $\neg d_i$). If p_i is either of $\top$ or $\bot$, it is represented by p_i itself, since the value of d_i has no effect on the semantics. Subs $[s_i, e_i]$ are treated in the same way.

We first give an interpretation of VS-SDD. By comparing the SDD in Fig. 1(b) with the VS-SDD in Fig. 1(c) having the same structure, we find they differ only in the labels of nodes and edges. Actually, we can construct the SDD of Fig. 1(b) from the VS-SDD in Fig. 1(c) in the following way. Let P_α be a path from the root to VS-SDD structure α and D_{P_α} be the sum of the offset and edge values appearing along the path. Then, $\texttt{ID}^{-1}(D_{P_\alpha})$ is the vtree node that the corresponding SDD node respects. For example, the leftmost child of the root node in the VS-SDD in Fig. 1(c) has offset value 6. The sum of offset values for this node is $1 + 6 = 7$ and $\texttt{ID}^{-1}(D_{P_\alpha})$ corresponds to the leaf vtree node having variable C. In this way, VS-SDD can be seen as an SDD variant that employs an indirect way of representing the respecting vtree nodes.

5.2 Substitution-equivalency in VS-SDDs

Next we show how substitution-equivalent functions are shared in VS-SDD. In Fig. 1(d), we should observe that the bottom-right node (say β) represents two substitution-equivalent functions $A \wedge B$ and $C \wedge D$. There are two different paths (say P_1 and P_2) from the root to β, and they correspond to different offset values $D_{P_1} = 1 + 1 = 2$ and $D_{P_2} = 1 + 4 = 5$. Therefore, β is used in two VS-SDDs $(\beta, 2)$ and $(\beta, 5)$ and they correspond to $A \wedge B$ and $C \wedge D$, respectively. In this way, substitution-equivalent functions are represented by VS-SDDs with the same structure and different offsets.

Now we proceed to the formal description. Let u, w be isomorphic subtrees of vtree v, $\mathbf{X}$ be the set of variables corresponding to the leaves of v, and M be the number of variables. We consider permutation $\pi_{u,w} : \{1, \dots, M\} \mapsto \{1, \dots, M\}$ that preserves the relation between u and w. That is, let X_i and X_j be the variables associated with leaf nodes u' in u and w' in w, respectively. We assume that u' and w' are associated through the graph isomorphism between u and w. Then $\pi_{u,w}$ is the bijection satisfying $\pi_{u,w}(j) = i$ for every pair of X_i and X_j corresponding to the leaf nodes of u and w. If u and w are isomorphic and we employ preorder IDs, then the difference in IDs of corresponding nodes of u and w is unique. We call this the *shift* between u, w and denote it as δ. For example, in the vtree in Fig. 1(a), two child nodes of the root node represent isomorphic vtrees. In these vtrees $\delta = 3$ for every corresponding node pair.

▶ **Theorem 6.** *Let f, g be Boolean functions that essentially depend on isomorphic vtrees u and w (resp.), where u and w are nodes in the entire vtree v. If f and g are substitution-equivalent with $\pi_{u,w}$ then the compressed and trimmed VS-SDDs (α, k) and (β, ℓ) representing f and g satisfies $\alpha = \beta$ and $\ell = k + \delta$.*

Proof. The Boolean function $\langle \alpha, k + \delta \rangle$ is the one wherein every appearance of every variable $l(\texttt{ID}^{-1}(i))$ in $\langle \alpha, k \rangle$ is replaced with $l(\texttt{ID}^{-1}(i + \delta))$. It is equivalent to $\langle \beta, \ell \rangle$. ◀

It is possible that there exist two VS-SDDs (α, k) and (β, ℓ) where $\alpha = \beta$ but vtrees $\texttt{ID}^{-1}(k)$ and $\texttt{ID}^{-1}(\ell)$ are not isomorphic. In such case, we do not share their structure. In other words, we share the identical structures only when for the offsets k and ℓ, $\texttt{ID}^{-1}(k)$ and $\texttt{ID}^{-1}(\ell)$ are isomorphic. We call this the *identical vtree rule.* The VS-SDD in Fig. 1(d) satisfies this rule. Such rule is unique to VS-SDDs, since in SDDs all identical structures are fully merged (i.e. reduced). This rule is crucial for guaranteeing some attractive properties of VS-SDDs introduced in later sections.

6 Properties of VS-SDD

We show here some basic VS-SDD properties. First, we prove the canonicity of some classes of VS-SDD. Then we give proofs on VS-SDD size.

6.1 Canonicity

We say a class of VS-SDD is canonical iff, given a vtree, for any Boolean function f, there is exactly one VS-SDD in this class representing f. We first introduce two classes of VS-SDDs, both have counterparts in SDDs.

▶ **Definition 7.** *We say VS-SDD (α, k) is* compressed *iff for each VS-SDD (β, ℓ) appearing in (α, k) where β is a decomposition, it forms compressed X-partition. We say a VS-SDD (α, k) is* trimmed *if it contains no decompositions with form of $\{([\top, \cdot], [\beta, d])\}$ and $\{([\beta, d], [\top, \cdot]), ([\neg\beta, d], [\bot, \cdot])\}$. We also say VS-SDD is* lightly trimmed *if it contains no decompositions with form of $\{([\top, \cdot], [\top, \cdot])\}$ and $\{([\top, \cdot], [\bot, \cdot])\}$. We say VS-SDD (α, k) is* normalized *iff for each VS-SDD (β, ℓ) appearing in (α, k) where β is a decomposition, every prime $[p_i, d_i]$ ensures that $\mathtt{ID}^{-1}(d_i + \ell)$ is the left child of vtree node $\mathtt{ID}^{-1}(\ell)$ and every sub $[s_i, e_i]$ ensures that $\mathtt{ID}^{-1}(e_i + \ell)$ is the right child of vtree node $\mathtt{ID}^{-1}(\ell)$.*

The proof of canonicity is almost identical to that for SDDs. We first introduce some concepts and notations. We use $(\alpha, k) \equiv (\beta, \ell)$ to represent that the corresponding Boolean functions are identical.

▶ **Definition 8.** *A Boolean function f* essentially depends *on vtree node v if f is not trivial and f is a deepest node that includes all variables that f essentially depends on.*

▶ **Lemma 9** ([9])**.** *A non-trivial function essentially depends on exactly one vtree node.*

▶ **Lemma 10.** *Let (α, k) be a trimmed and compressed VS-SDD. If $(\alpha, k) \equiv$ false, then $\alpha = \bot$. If $(\alpha, k) \equiv$ true, then $\alpha = \top$. Otherwise, $\mathtt{ID}^{-1}(k)$ always equals to the vtree node v that $\langle \alpha, k \rangle$ essentially depends on.*

The above lemma suggests that compressed and trimmed VS-SDDs can be partitioned into groups depending on the offset. We can prove the canonicity by exploiting this fact.

▶ **Theorem 11.** *Compressed and trimmed VS-SDDs with the same vtree, v, are canonical. Also, compressed, lightly trimmed, and normalized VS-SDDs with the same vtree,v, are canonical.*

Proof. Here we give the proof for the case of compressed and trimmed VS-SDDs. The proof for compressed, lightly trimmed and normalized SDDs can be constructed in a similar way.

If two compressed SDDs (α, k) and (β, ℓ) satisfy $(\alpha, k) = (\beta, \ell)$, then $(\alpha, k) \equiv (\beta, \ell)$ from the definition. Suppose $\langle \alpha, k \rangle = \langle \beta, \ell \rangle$ and let $f = \langle \alpha, k \rangle = \langle \beta, \ell \rangle$. If $f =$ *true*, then $\alpha = \beta = \top$ and they are canonical. Similarly, if $f =$ *false*, then $\alpha = \beta = \bot$.

Next we consider the case of f being non-trivial. From Lemma 10, $\mathtt{ID}^{-1}(k) = w = \mathtt{ID}^{-1}(\ell)$ where w is the vtree node that f essentially depends on. Suppose w is a leaf, then VS-SDDs must be literals and hence $(\alpha, k) = (\beta, \ell)$. Suppose now that w is internal and that the theorem holds for VS-SDDs whose offsets correspond to descendant nodes of $\mathtt{ID}^{-1}(k)$. Let w^l and w^r be the left and the right subtree of w, respectively. Let $\mathbf{X}$ be variables in w^l, $\mathbf{Y}$ be variables in w^r, $\alpha = \{([p_1, d_1], [s_1, e_1]), \ldots, ([p_n, d_n], [s_n, e_n])\}$ and $\beta = \{([q_1, b_1], [r_1, c_1]), \ldots, ([q_m, b_m], [r_m, c_m])\}$. By the definition, offsets $d_i + k$ and

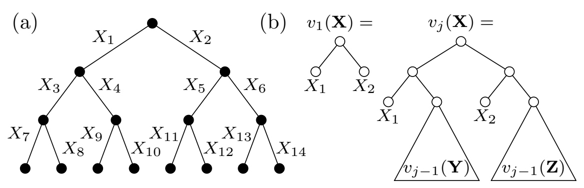

Figure 2 (a) A complete binary tree with variable-labeled edges. (b) The recursive structure of vtree $v_j(\mathbf{X})$. Here X_1 and X_2 indicate the first and second (resp.) variables of $\mathbf{X}$.

$b_j + \ell$ correspond to vtree nodes in w^l and offsets $e_i + k$ and $c_j + \ell$ correspond to vtree nodes in w^r. Since compressed $\mathbf{X}$-partitions $\{(\langle p_1, d_1\rangle, \langle s_1, e_1\rangle), \ldots, (\langle p_n, d_n\rangle, \langle s_n, e_n\rangle)\}$ and $\{(\langle q_1, b_1\rangle, \langle r_1, c_1\rangle), \ldots, (\langle q_m, b_m\rangle, \langle r_m, c_m\rangle)\}$ are identical (see Theorem 3 of [9]), $n = m$ and there is a one-to-one $\equiv$-correspondence between the primes and subs. From the inductive hypothesis, this means there is a one-to-one $=$-correspondence between the primes and subs. This implies $\alpha = \beta$ and thus $(\alpha, k) = (\beta, \ell)$. ◀

6.2 About the Size: Exponential Compression

We here compare VS-SDD size with SDD size. First of all, we observe that VS-SDD is always smaller than SDDs since it is made by sharing substitution-equivalent nodes in SDDs and no other size changes occur.

▶ **Proposition 12.** *For any SDD α defined with vtree v, there exists a VS-SDD whose size is not larger than $|\alpha|$.*

We turn our focus to the best compression ratio of the VS-SDD. Since a vtree has M leaves where M is the number of variables, a vtree might have at most M isomorphic subtrees. Thus the lower bound of VS-SDD size is $1/M$ of SDD when we employ the identical vtree rule. Here we prove that there is a series of functions that almost achieves this compression ratio asymptotically.

▶ **Theorem 13.** *There exists a sequence of Boolean functions $f_1, f_2, \ldots$ such that f_j uses $O(2^j)$ variables, the size of a compressed SDD representing f_j is $\Omega(2^j)$ with any vtree, and that of a compressed VS-SDD representing f_j is $O(j)$ with a particular vtree.*

The compression ratio is $O(j/2^j) = O(\log M/M)$. Theorem 13 makes a stronger statement, because "any" vtree can be considered for SDD.

One of the sequences satisfying Theorem 13 is as follows:

$$f_j(\mathbf{X}) = (\neg X_1 \vee \neg X_2) \wedge \bigwedge_{i=1}^{2^j-2} \big((\neg X_i \vee \neg X_{2i+1}) \wedge (\neg X_i \vee \neg X_{2i+2}) \wedge (\neg X_{2i+1} \vee \neg X_{2i+2})\big).$$

By considering a complete binary tree like Fig. 2(a), we observe that $f_j(\mathbf{x}) = \mathit{true}$ iff the edges whose corresponding variables are set to *true* constitute a matching.

We outline the proof here; details are given in the full version. Since the first part of Theorem 13 can easily be proved, we refer to the second part. We define vtree $v_j(\mathbf{X})$ in a recursive manner as shown in Fig. 2(b). Here $\mathbf{Y}$ includes the variables corresponding to the edges below X_1 when considering the complete binary tree as in Fig. 2(a) (namely $X_3, X_4, \ldots$) and $\mathbf{Z}$ includes those corresponding to the edges below X_2 ($X_5, X_6, \ldots$). Now we decompose $f_j(\mathbf{X})$ with respect to $v_j(\mathbf{X})$ by using $f_{j-1}(\mathbf{Y})$, $f_{j-1}(\mathbf{Z})$ and some other subfunctions. We then use the fact that $f_{j-1}(\mathbf{Y})$ and $f_{j-1}(\mathbf{Z})$ are substitution-equivalent with $\pi_{v_{j-1}(\mathbf{Y}), v_{j-1}(\mathbf{Z})}$. By repetitively applying this argument, we observe that by decomposing f_j with respect to v_j, the SDD of f_j has 2^i nodes that represent $f_{j-i}(\cdot)$, which all represent substitution-equivalent functions, and thus the VS-SDD reduces the size exponentially.

Algorithm 1 $\texttt{Apply}(\alpha, \beta, k, \circ)$, which computes a VS-SDD representing $\langle \alpha, k\rangle \circ \langle \beta, k\rangle$ for two normalized VS-SDDs $(\alpha, k), (\beta, k)$ and a binary operator $\circ$.

$\texttt{Cache}(\cdot,\cdot,\cdot) = \texttt{nil}$ initially. $\texttt{Expand}(\alpha)$ returns $\{([\top,\cdot],[\top,\cdot])\}$ if $\alpha = \top$; $\{([\top,\cdot],[\bot,\cdot])\}$ if $\alpha = \bot$; else α. $\texttt{UniqueD}(\gamma)$ returns $\top$ if $\gamma = \{([\top,\cdot],[\top,\cdot])\}$; $\bot$ if $\gamma = \{([\top,\cdot],[\bot,\cdot])\}$; else the unique VS-SDD with elements γ.

```
1:  if $\alpha$ and $\beta$ are either of $\top, \bot, \mathbf{v}, \neg\mathbf{v}$ then
2:      return the pair of corresponding value and offset.
3:  else if $\texttt{Cache}(\alpha, \beta, \circ) \neq \texttt{nil}$ then
4:      $\lambda \leftarrow \texttt{Cache}(\alpha, \beta, \circ)$
5:      return $(\lambda, k)$
6:  else
7:      $\gamma \leftarrow \{\}$
8:      for all elements $([p_i, d], [s_i, e])$ in $\texttt{Expand}(\alpha)$ do
9:          for all elements $([q_j, d], [r_j, e])$ in $\texttt{Expand}(\beta)$ do
10:             $(p, \ell_p) \leftarrow \texttt{Apply}(p_i, q_j, d + k, \circ)$
11:             if $(p, \ell_p)$ is consistent then
12:                 $(s, \ell_s) \leftarrow \texttt{Apply}(s_i, r_j, e + k, \circ)$
13:                 add element $([p, \ell_p - k], [s, \ell_s - k])$ to $\gamma$
14: $\lambda \leftarrow \texttt{UniqueD}(\gamma)$, $\texttt{Cache}(\alpha, \beta, \circ) \leftarrow \lambda$
15: return $(\lambda, k)$
```

7 Operations of VS-SDD

The most important property of VS-SDDs is that they support numerous key operations in polytime. We focus here on the important queries and transformations shown in [10]. Most of these operations are based on Apply. Apply takes, given a vtree, two VS-SDDs (α, k), (β, ℓ) and binary operation $\circ$ such as $\vee$ (disjunction), $\wedge$ (conjunction) and $\oplus$ (exclusive-or), and returns a VS-SDD of $\langle \alpha, k\rangle \circ \langle \beta, \ell\rangle$. By repeating Apply operations, we can flexibly construct VS-SDDs representing various Boolean functions.

To simplify the explanation of Apply, we assume that VS-SDDs are normalized and thus have the same offset value k. Given two normalized VS-SDDs (α, k), (β, k), Alg. 1 provides pseudocode for the function $\texttt{Apply}(\alpha, \beta, k, \circ)$. The mechanism behind the Apply computation of VS-SDDs is as follows. Let f, g be Boolean functions with the same variable set, and suppose that f is $\mathbf{X}$-partitioned as $f = \bigvee_{i=1}^{n}[p_i(\mathbf{X}) \wedge s_i(\mathbf{Y})]$ and g is also $\mathbf{X}$-partitioned (with the same $\mathbf{X}$) as $g = \bigvee_{j=1}^{m}[q_j(\mathbf{X}) \wedge r_j(\mathbf{Y})]$. Then, $f \circ g$ can be expressed as $\bigvee_{i=1}^{n}\bigvee_{j=1}^{m}[(p_i(\mathbf{X}) \wedge q_j(\mathbf{X})) \wedge (s_i(\mathbf{Y}) \circ r_j(\mathbf{Y}))]$, where $(p_i(\mathbf{X}) \wedge q_j(\mathbf{X})) \wedge (p_{i'}(\mathbf{X}) \wedge q_{j'}(\mathbf{X})) = \mathit{false}$ for $(i, j) \neq (i', j')$ and $\bigvee_{i=1}^{n}\bigvee_{j=1}^{m}(p_i(\mathbf{X}) \wedge q_j(\mathbf{X})) = \mathit{true}$. Thus, computing $p_i \wedge q_j$ and $s_i \circ r_j$ for each (i, j) pair and ignoring the pairs such that $p_i \wedge q_j = \mathit{false}$ yields the $\mathbf{X}$-partition of $f \circ g$. Alg. 1 follows this recursive definition.

▶ **Proposition 14.** $\texttt{Apply}(\alpha, \beta, k, \circ)$ *runs in* $O(|\alpha||\beta|)$ *time.*

The above result is the same as in the case of Apply for SDDs. The key to achieving this result is that we use Cache without using offset k as a key. We use the fact that if a pair of functions $f(\mathbf{X})$, $f'(\mathbf{Y})$ and $g(\mathbf{X})$, $g'(\mathbf{Y})$, where $\mathbf{X}$ and $\mathbf{Y}$ are non-overlapping, are substitution-equivalent with permutation π, then the composed functions $f \circ g$ and $f' \circ g'$ are also substitution-equivalent with the same permutation π. For example, let $f = A \wedge B$, $f' = C \wedge D$, $g = \neg A$ and $g' = \neg C$, in which f and f', and g and g' (resp.) are substitution-equivalent with permutation $\pi_{\texttt{ID}^{-1}(2),\texttt{ID}^{-1}(5)}$ defined with the vtree in Fig. 1(a). Then $f \vee g = \neg A \vee B$ and $f' \vee g' = \neg C \vee D$ are also substitution-equivalent with $\pi_{\texttt{ID}^{-1}(2),\texttt{ID}^{-1}(5)}$. This means the results of $\texttt{Apply}(\alpha, \beta, k, \circ)$ with different k are all substitution-equivalent. Therefore, we can reuse the result obtained with different offsets.

Table 1 List of supported (a) queries and (b) transformations for SDDs (S), VS-SDDs (V), compressed SDDs (S(C)) and compressed VS-SDDs (V(C)). ✓ indicates the existence of a polytime algorithm, while • indicates such polytime algorithm is shown to be impossible.

	(a) Query	S	**V**	S(C)	**V(C)**
CO	consistency	✓	✓	✓	✓
VA	validity	✓	✓	✓	✓
CE	clausal entailment	✓	✓	✓	✓
IM	implicant check	✓	✓	✓	✓
EQ	equivalence check	✓	✓	✓	✓
CT	model counting	✓	✓	✓	✓
SE	sentential entailment	✓	✓	✓	✓
ME	model enumeration	✓	✓	✓	✓

	(b) Transformation	S	**V**	S(C)	**V(C)**
$\wedge$**C**	conjunction	•	•	•	•
$\wedge$**BC**	bounded conjunction	✓	✓	•	•
$\vee$**C**	disjunction	•	•	•	•
$\vee$**BC**	bounded disjunction	✓	✓	•	•
$\neg$**C**	negation	✓	✓	✓	✓
CD	conditioning	✓	✓	•	•
FO	forgetting	•	•	•	•
SFO	singleton forgetting	✓	✓	•	•

If VS-SDDs are trimmed, we can also define `Apply` operations for them. While similar to the case for trimmed SDDs, the `Apply` for trimmed VS-SDDs are more complicated than that of normalized VS-SDDs since we have to take different operations depending on the combination of offset values of input VS-SDDs. However, the complexity of `Apply` for trimmed VS-SDDs is also $O(|\alpha||\beta|)$. We detail the `Apply` for trimmed VS-SDDs in the full version.

Note that even if two VS-SDDs are compressed, the resulting VS-SDD cannot be assumed to be compressed since the same sub may appear. It is said in [4] that there is a case in which compression makes an SDD exponentially larger, and thus a similar statement holds for VS-SDDs. Therefore, if we oblige the output to be compressed, Prop. 14 does not hold. Note that during `Apply`, compression can be performed by taking the disjunction of primes when the same subs emerge.

By extensively using Prop. 14 with some other algorithms, it can be shown that the various important queries and transformations in [10] can be performed in polytime. The proof is in the full version.

▶ **Proposition 15.** *The results in Table 1 hold.*

Note that some applications, e.g. probabilistic inference [22, 10], need *weighted* model counting, where each variable has a weight. Though this cannot be performed in $O(|\alpha|)$ time for VS-SDD α, it can be performed at least as fast as is possible by using the corresponding SDD, by preparing, for each node, as many counters as the number of unified nodes in the original SDD. Moreover, if the weights of variables are the same for the same vtree structures, we can share counters, which speeds up the computation.

8 Implementation

We should address implementation in order to ensure space-efficiency. One suspects that even if VS-SDD size is never larger than SDD size, the memory usage may increase because we store the information of respecting vtree node ids in the edges of a diagram (differentially) instead of in the nodes. This is true if VS-SDDs are implemented as is.

However, a small modification avoids this problem. First, for a normalized VS-SDD, we simply ignore the differences of vtree node ids attached to the edges. Even so, we can recover the respecting vtree node because if an SDD node respects vtree v, its primes respect the left child of v and its subs respect the right child of v. Second, for a general VS-SDD, we just reuse the structure of the original SDD. Among the SDD nodes that are merged into one in the VS-SDD structure, we just leave one representative (e.g. the one with the smallest respecting vtree node id). Then each of the other nodes has a pointer to the representative node instead of storing the prime-sub pairs. An example of such a structure is drawn in

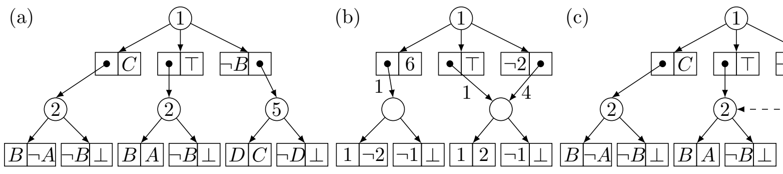

Figure 3 (a)(b) An SDD and a VS-SDD that are the same as Fig. 1. (c) The example of the representation of VS-SDD (b) using original SDD structure (a). The dashed arrow indicates a pointer to the representative node.

Fig. 3(c). Here the dashed arrow indicates a pointer to the representative node described above. Since each decomposition node has at least one prime-sub pair that typically uses two pointers, replacing it by single pointer will never increase memory usage. Working with such a structure does not violate any properties about VS-SDDs, including the operations described above.

9 Evaluation

We use some benchmarks of Boolean functions to evaluate how our approach reduces the size of an SDD. we compile a CNF into an SDD with the dynamic vtree search [7] and then compare the sizes yielded by the SDD and its VS form (VS-SDD). To compile a CNF, we use the SDD package version 2.0 [8] with a balanced initial vtree. Here note that we use for both SDD and VS-SDD the same vtree, which is searched to suit for SDD. All the experiments are conducted on a 64-bit macOS (High Sierra) machine with 2.5 GHz Intel Core i7 CPU (1 thread) and 16 GB RAM.

Here we focus on the planning CNF dataset that was used in the experiment of Sym-DDG [2]. The planning problem naturally exhibits symmetries, e.g. see [21]. Given time horizon T, this data represents a deterministic planning problem with varying initial and goal states. Here we can choose an action from a fixed action set for each time point, and a plan for this problem is a time series of actions for $t = 0, \dots, T-1$ that leads from the initial state to the goal state. For more details, see [2]. We use the planning problems that were also used in the experiment of Sym-DDG: “blocks-2”, “bomb-5-1”, “comm-5-2”, “emptyroom-4/8”, and “safe-5/30”, with varying time horizons $T = 3, 5, 7, 10$.

The next focus is on the benchmarks with apparent symmetries. The first one is the N-queens problem, that is, given an $N \times N$ chessboard, place N queens such that no two queens attack each other. We assign a variable to each square in the chessboard, and consider a Boolean function that evaluates *true* iff the *true* variables constitute one answer for this problem. This problem is used as a benchmark in Zero-suppressed BDD and other DD studies [17, 6]. The second one is enumerating matchings of grid graphs. Subgraph enumeration with decision diagrams has several applications; see [14] and [19]. Here the grid graph is often used as a benchmark [13], because it is closely related to self-avoiding walk [15], and subgraph enumeration becomes much harder for larger grids despite their simplicity. We can observe that both the chessboard and the grid have line symmetries and point symmetry. Again we exploit dynamic vtree search implemented in the SDD package.

Table 2 shows the results of our experiments. The “S” column represents SDD size, “V” represents VS-SDD size, and “ratio” indicates the ratio of VS-SDD size compared to SDD size. Here the problems in which the SDD compilation took more than 10 minutes are omitted. For planning problems, the suffix “_tn” stands for $T = n$, and for matching problems, the suffix indicates the grid size. It is observed that for many planning problems, the VS-SDD

Table 2 Results for experiments. The “S” column represents SDD size, “V” represents VS-SDD size, and “ratio” indicates the ratio of VS-SDD size compared to SDD size.

Problem	#vars	S	V	ratio
blocks-2_t3	248	8811	7057	80.1%
blocks-2_t5	406	31861	28858	90.6%
bomb-5-1_t3	348	3798	2278	60.0%
bomb-5-1_t5	564	6327	3960	62.6%
bomb-5-1_t7	780	11212	7287	65.0%
bomb-5-1_t10	1104	16514	10426	63.1%
comm-5-2_t3	488	20584	18033	87.6%
emptyroom-4_t3	116	1822	1146	62.9%
emptyroom-4_t5	188	3090	1885	61.0%
emptyroom-4_t7	260	5073	3001	59.2%
emptyroom-4_t10	368	106737	103417	96.8%
emptyroom-8_t3	244	10511	8549	81.3%
safe-5_t3	54	567	441	77.8%
safe-5_t5	86	898	640	71.2%
safe-5_t7	118	1710	1314	76.8%
safe-5_t10	166	2506	1756	70.1%
safe-30_t3	304	5476	4067	74.3%
safe-30_t5	486	8710	6328	72.7%
safe-30_t7	668	14449	10371	71.8%
safe-30_t10	941	23469	17421	74.2%
8-Queens	64	2222	1624	73.1%
9-Queens	81	5559	4767	85.8%
10-Queens	100	10351	9159	88.5%
11-Queens	121	30611	28876	94.3%
Matching-6x6	60	13091	12671	96.8%
Matching-8x8	112	98200	97103	98.8%
Matching-6x18	192	36228	34241	94.5%

reduces the size to around 60% to 80% of the original SDD. We observe that for these cases, many nodes representing substitution-equivalent functions are found among the bottom nodes of the original SDD, which yields the substantial size decrease. These compression ratios are competitive to, and for some cases better than, that of the Sym-DDG [2] compared to the DDG. For the N-queens problems, still better compression ratios are achieved except for $N = 11$. However, for matching enumeration problems, the effect of variable shift is relatively small. One reason is the asymmetry of primes and subs, that is, primes must form a partition while subs do not have such a limitation. The success in planning datasets may be explained as follows. The dynamic vtree search typically gathers variables with strong dependence locally to achieve succinctness. For planning problems, the variables with near time points are gathered, which captures the symmetric nature of the problem.

10 Conclusion

We proposed a variable shift SDD (VS-SDD), a more succinct variant of SDD that is obtained by changing the way in which respecting vtree nodes are indicated. VS-SDD keeps the two important properties of SDDs, the canonicity and the support of many useful operations. The size of a VS-SDD is always smaller than or equal to that of an SDD, and there are cases where the VS-SDD is exponentially smaller than the SDD. Experiments show that our idea effectively captures the symmetries of Boolean functions, which leads to succinct compilation.

References

1 Anuchit Anuchitanukul, Zohar Manna, and Tomás E. Uribe. Differential BDDs. In *Computer Science Today*, pages 218–233, 1995. doi:10.1007/BFb0015246.

2 Anicet Bart, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. Symmetry-driven decision diagrams for knowledge compilation. In *ECAI*, pages 51–56, 2014. doi:10.3233/978-1-61499-419-0-51.

3 Simone Bova. SDDs are exponentially more succinct than OBDDs. In *AAAI*, pages 929–935, 2016.

4 Guy Van den Broeck and Adnan Darwiche. On the role of canonicity in knowledge compilation. In *AAAI*, pages 1641–1648, 2015.

5 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35:677–691, 1986. doi:10.1109/TC.1986.1676819.

6 Randal E. Bryant. Chain reduction for binary and zero-suppressed decision diagrams. In *TACAS*, pages 81–98, 2018. doi:10.1007/978-3-319-89960-2_5.

7 Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *AAAI*, pages 187–194, 2013.

8 Arthur Choi and Adnan Darwiche. The SDD package: version 2.0. http://reasoning.cs.ucla.edu/sdd/, 2018.

9 Adnan Darwiche. SDD: a new canonical representation of propositional knowledge bases. In *IJCAI*, pages 819–826, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-143.

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

11 Héiène Fragier and Pierre Marquis. On the use of partially ordered decision graphs for knowledge compilation and quantified Boolean formulae. In *AAAI*, pages 42–47, 2006.

12 Jordan Gergov and Christoph Meinel. Efficient Boolean manipulation with OBDD's can be extended to FBDD's. *IEEE Trans. Comput.*, 43:1197–1209, 1994. doi:10.1109/12.324545.

13 Hiroaki Iwashita, Yoshio Nakazawa, Jun Kawahara, Takeaki Uno, and Shin-ichi Minato. Efficient computation of the number of paths in a grid graph with minimal perfect hash functions. Technical Report TCS-TR-A-13-64, Division of Computer Science, Hokkaido University, 2013.

14 Donald E. Knuth. *The Art of Computer Programming*, volume 4A: Combinatorial Algorithms, Part I. Addison-Wesley, 2011.

15 Neal Madras and Gordon Slade. *The Self-Avoiding Walk*. Birkhäuser Basel, 2011.

16 Jean-Christophe Madre and Jean-Paul Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *DAC*, pages 205–210, 1988. doi:10.1109/DAC.1988.14759.

17 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC*, pages 272–277, 1993. doi:10.1145/157485.164890.

18 Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *DAC*, pages 52–57, 1990. doi:10.1145/123186.123225.

19 Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Compiling graph substructures into sentential decision diagrams. In *AAAI*, pages 1213–1221, 2017.

20 Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *IJCAI*, pages 3141–3148, 2015.

21 Héctor Palacios, Blai Bonet, Adnan Darwiche, and Héctor Geffner. Pruning conformant plans by counting models on compiled d-DNNF representations. In *ICAPS*, pages 141–150, 2005.

22 Tian Sang, Paul Beame, and Henry Kautz. Performing Bayesian inference by weighted model counting. In *AAAI*, pages 475–481, 2005.

23 Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. Compiling probabilistic logic programs into sentential decision diagrams. In *PLP*, 2014.