Dynamic Minimization of Sentential Decision Diagrams

Arthur Choi and **Adnan Darwiche**
Computer Science Department
University of California, Los Angeles
{*aychoi,darwiche*}@*cs.ucla.edu*

Abstract

The Sentential Decision Diagram (SDD) is a recently proposed representation of Boolean functions, containing Ordered Binary Decision Diagrams (OBDDs) as a distinguished subclass. While OBDDs are characterized by total variable orders, SDDs are characterized more generally by vtrees. As both OBDDs and SDDs have canonical representations, searching for OBDDs and SDDs of minimal size simplifies to searching for variable orders and vtrees, respectively. For OBDDs, there are effective heuristics for dynamic reordering, based on locally swapping variables. In this paper, we propose an analogous approach for SDDs which navigates the space of vtrees via two operations: one based on tree rotations and a second based on swapping children in a vtree. We propose a particular heuristic for dynamically searching the space of vtrees, showing that it can find SDDs that are an order-of-magnitude more succinct than OBDDs found by dynamic reordering.

Introduction

A new representation of Boolean functions was recently proposed, called the Sentential Decision Diagram (SDD), which generalizes the Ordered Binary Decision Diagram (OBDD), and has a number of interesting properties (Darwiche 2011; Xue, Choi, and Darwiche 2012). First, while decisions are performed on the state of a single variable in OBDDs (i.e., literals), such decisions are performed on the state of multiple variables in SDDs (i.e., sentences). Second, while an OBDD is characterized by a total variable order (Bryant 1986), an SDD is characterized by a dissection of a total variable order, known as a vtree. Despite this generality, SDDs are still able to maintain a number of properties that have been critical to the success of OBDDs in practice. For example, SDDs are canonical (under certain conditions) and support an efficient `apply` operation which allows one to combine SDDs using Boolean operators. On the theoretical side, an upper bound was identified on the size of SDDs (based on treewidth) (Darwiche 2011) that is tighter than the corresponding upper bound on the size of OBDDs (based on pathwidth) (Prasad, Chong, and Keutzer 1999; Huang and Darwiche 2004; Ferrara, Pan, and Vardi 2005).

From a practical perspective, OBDDs benefit greatly from a variety of heuristic algorithms for finding variable orders that yield compact OBDD representations. For example, sifting algorithms, based on swapping neighboring variables in a total variable order, have been particularly effective at navigating the space of total variable orders (Rudell 1993). As an OBDD with a particular variable order corresponds to an SDD with a particular vtree, which dissects the order, we can potentially find even more compact representations if we develop effective heuristics for navigating the space of all vtrees. In fact, (Xue, Choi, and Darwiche 2012) identified a class of Boolean functions where certain variable orders lead to exponentially large OBDDs, but where particular dissections of these orders lead to SDDs of only linear size.

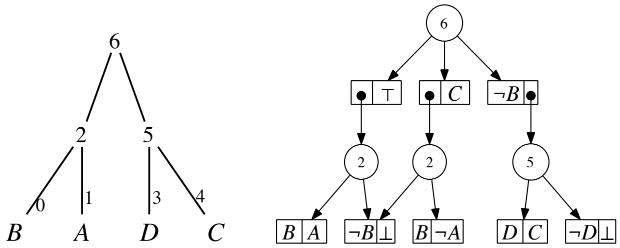
In this paper, we propose a new greedy search algorithm for optimizing vtrees. We introduce two operations that are sufficient for navigating the full space of vtrees: one based on tree rotations, and another based on swapping the children of a vtree node. Using the rotation and swapping operations as primitives, we propose a greedy search algorithm that can be used to find good vtrees, in a manner analogous to methods that swap neighboring variables to search for good variable orders. We evaluate our dynamic vtree search algorithm empirically, finding that it can identify SDDs that are an order of magnitude more succinct than OBDDs found by the CUDD package (Somenzi 2004), using dynamic variable reordering. The dynamic vtree search algorithm that we evaluate is further implemented in a publicly available SDD library.[1]

Technical Preliminaries

Upper case letters (e.g., X) will be used to denote variables and lower case letters to denote their instantiations (e.g., x). Bold upper case letters (e.g., $\mathbf{X}$) will be used to denote sets of variables and bold lower case letters to denote their instantiations (e.g., $\mathbf{x}$).

A *Boolean function* f over variables $\mathbf{Z}$, denoted $f(\mathbf{Z})$, maps each instantiation $\mathbf{z}$ of variables $\mathbf{Z}$ to 0 or 1. A *trivial* function maps all its inputs to 0 (denoted false) or maps all its inputs to 1 (denoted true).

[1] The SDD package is available at http://reasoning.cs.ucla.edu/sdd/



(a) vtree (b) graphical depiction of an SDD

Figure 1: Function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.

Consider a Boolean function $f(\mathbf{X}, \mathbf{Y})$ with disjoint sets of variables $\mathbf{X}$ and $\mathbf{Y}$. If

$$f(\mathbf{X}, \mathbf{Y}) = (p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})) \vee \ldots \vee (p_n(\mathbf{X}) \wedge s_n(\mathbf{Y}))$$

then the set $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ is called an $(\mathbf{X}, \mathbf{Y})$-decomposition of the function f and each pair (p_i, s_i) is called an *element* of the decomposition (Pipatsrisawat and Darwiche 2010). The decomposition is further called an $(\mathbf{X}, \mathbf{Y})$-*partition* iff the p_i's form a partition (Darwiche 2011). That is, $p_i \neq \mathsf{false}$ for all i; and $p_i \wedge p_j = \mathsf{false}$ for $i \neq j$; and $\bigvee_i p_i = \mathsf{true}$. In this case, each p_i is called a *prime* and each s_i is called a *sub*. An $(\mathbf{X}, \mathbf{Y})$-partition is *compressed* iff its subs are distinct, i.e., $s_i \neq s_j$ for $i \neq j$ (Darwiche 2011). Compression can always be ensured by repeatedly disjoining the primes of equal subs. Moreover, a function $f(\mathbf{X}, \mathbf{Y})$ has a *unique,* compressed $(\mathbf{X}, \mathbf{Y})$-partition. Finally, the *size* of a decomposition, or partition, is the number of its elements.

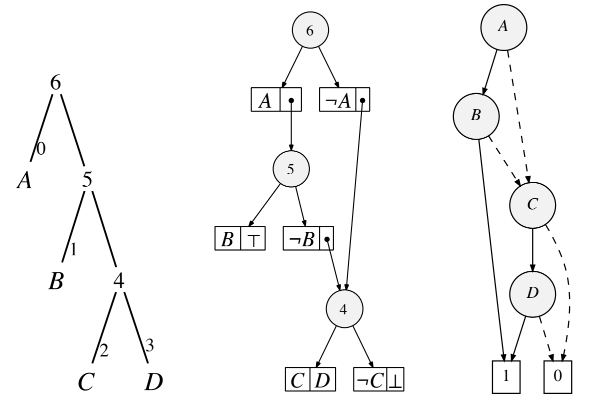
Note that $(\mathbf{X}, \mathbf{Y})$-partitions generalize Shannon decompositions, which fall as a special case when $\mathbf{X}$ contains a single variable. OBDDs result from the recursive application of Shannon decompositions, leading to decision nodes that branch on the states of a single variable (i.e., literals). As we show next, SDDs result from the recursive application of $(\mathbf{X}, \mathbf{Y})$-partitions, leading to decision nodes that branch on the state of multiple variables (i.e., arbitrary sentences).

Sentential Decision Diagrams (SDDs)

Consider the *full* binary tree in Figure 1(a), which is known as a *vtree* (v^l and v^r will be used to denote the left and right children of a vtree node). Consider also the Boolean function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ over the same variables. Node $v = 6$ is the vtree root. Its *left* subtree contains variables $\mathbf{X} = \{A, B\}$ and its *right* subtree contains $\mathbf{Y} = \{C, D\}$. Decomposing function f at node $v = 6$ amounts to generating an $(\mathbf{X}, \mathbf{Y})$-partition of function f. The unique compressed $(\mathbf{X}, \mathbf{Y})$-partition here is

$$\{(\underbrace{A \wedge B}_{\text{prime}}, \underbrace{\mathsf{true}}_{\text{sub}}), (\underbrace{\neg A \wedge B}_{\text{prime}}, \underbrace{C}_{\text{sub}}), (\underbrace{\neg B}_{\text{prime}}, \underbrace{D \wedge C}_{\text{sub}})\}$$

This partition is represented by the root node of Figure 1(b). This node, which is a circle, represents a *decision node* with three branches. Each branch corresponds to one element $\boxed{p \mid s}$ of the above partition. Here, the left box contains a prime when the prime is a literal or a constant; otherwise, it contains a pointer to a prime. Similarly, the right box contains a sub or a pointer to a sub. The three primes are decomposed recursively, but using the vtree rooted at $v = 2$. Similarly, the subs are decomposed recursively, using the vtree rooted at $v = 5$. This recursive decomposition process moves down one level in the vtree with each recursion, terminating when it reaches leaf vtree nodes. The full SDD for this example is depicted in Figure 1(b).



(a) right-linear (b) SDD (c) OBDD

Figure 2: A vtree, SDD and OBDD for $(A \wedge B) \vee (C \wedge D)$.

A decision node is said to be *normalized* for a vtree node v iff it represents an $(\mathbf{X}, \mathbf{Y})$-partition where $\mathbf{X}$ are the variables of v^l and $\mathbf{Y}$ are the variables of v^r. In Figure 1(b), each decision node is labeled with the vtree node it is normalized for. The *size* of an SDD is the sum of sizes attained by its decision nodes. The SDD in Figure 1(b) has size 9.

SDDs obtained from the above process are called *compressed* iff the $(\mathbf{X}, \mathbf{Y})$-partition computed at each step is compressed. These SDDs may contain trivial decision nodes which correspond to $(\mathbf{X}, \mathbf{Y})$-partitions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$. When these decision nodes are removed (by directing their parents to α), the resulting SDD is called *trimmed*. Compressed and trimmed SDDs are canonical for a given vtree (Darwiche 2011)[2] and we shall restrict our attention to them in this paper.[3] SDDs support a polytime `apply` operation, allowing one to combine two SDDs using any Boolean operator (Darwiche 2011).[4]

OBDDs correspond to SDDs that are constructed using right-linear vtrees (Darwiche 2011). A *right-linear* vtree is one in which each left-child is a leaf; see Figure 2(a). When using such vtrees, each constructed $(\mathbf{X}, \mathbf{Y})$-partition is such that $\mathbf{X}$ contains a single variable, therefore, corresponding

[2]Given the usual assumption that the SDD has no isomorphic subgraphs, which can be easily ensured in practice using the unique-node technique from the OBDD literature.

[3]We will also assume *reduced* OBDDs, which are canonical for a given variable order (Bryant 1986).

[4]The polytime `apply` does not guarantee that the resulting SDD is compressed, yet the `apply` we utilize in this work ensures such compression as this has proved critical in practice.

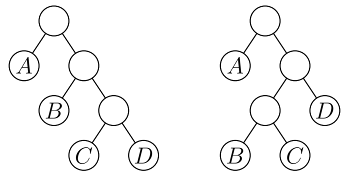


Figure 3: Two vtrees that dissect order $\langle A, B, C, D\rangle$.

to a Shannon decomposition. In this case, primes are guaranteed to always be literals (i.e., a variable or its negation). Moreover, decision nodes are guaranteed to be binary, leading to OBDDs (but with a different syntax); see Figure 2.

Vtrees and Variable Orders

OBDDs are characterized by total variable orders, so searching for a compact OBDD is done by navigating the space of variable orders. Similarly, SDDs are characterized by vtrees, so searching for a compact SDD will be done by navigating the space of vtrees. In fact, the space of vtrees can be induced by considering the dissections of all variable orders.

Definition 1 (Dissection) *A vtree* <u>*dissects*</u> *a total variable order π iff a left-right traversal of the vtree visits leaves (variables) in the same order as π.*

Figure 3 depicts two dissections of the same variable order.

The search space over vtrees can then be characterized by two dimensions: total variable orders and their dissections. We have $n!$ total variable orders over n variables. We also have $C_{n-1} = \frac{(2n-2)!}{n!(n-1)!}$ dissections of a total variable order over n variables.[5] Hence, there are $n! \times C_{n-1} = \frac{(2n-2)!}{(n-1)!}$ total vtrees over n variables. The following table gives a sense of these search spaces in terms of the number of variables n.

n	1	2	3	4	5	6
# of orderings	1	2	6	24	120	720
# of dissections	1	1	2	5	14	42
# of vtrees	1	2	12	120	1680	30240

It is well known that the choice of a total variable order can lead to exponential changes in the size of a corresponding OBDD and, hence, SDD. Moreover, it is known that different dissections of the same variable order can lead to exponential changes in the SDD size (Xue, Choi, and Darwiche 2012). In fact, this last result is more specific: a right-linear dissection of certain orders leads to an SDD/OBDD of exponential size, yet some other dissection of the same order leads to an SDD of only linear size. This only emphasizes the importance of searching for good vtrees.

Navigating the Space of Vtrees

When searching for a compact OBDD, the space of total variable orders is usually navigated via swaps of neighboring variables since repeated application of this operation is guaranteed to induce all total variable orders (Knuth 2005).

[5] C_{n-1} is the $(n-1)$-st Catalan number, which is the number of full binary trees with n leaves (Campbell 1984).

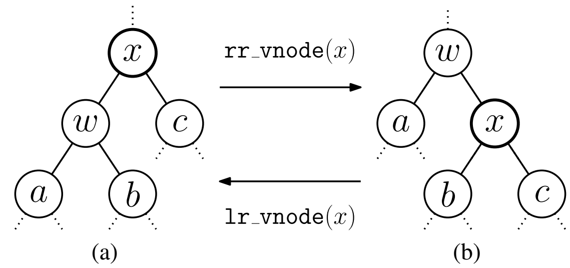


Figure 4: Rotating a vtree node x right and left. Nodes a, b, and c may represent leaves or subtrees.

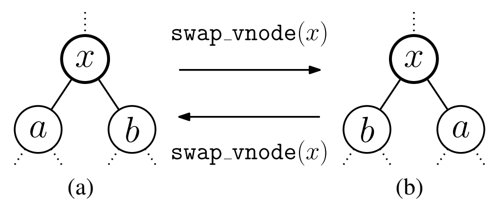


Figure 6: Swapping the children of a vtree node x, back and forth. Nodes a and b may represent leaves or subtrees.

As we shall see next, two vtree operations, called *rotate* and *swap,* allow one to navigate the space of all vtrees.

Figure 4 illustrates two *rotate* operations on binary trees. The first is right rotation, denoted by `rr_vnode`(x), and the second is left rotation, denoted by `lr_vnode`(x). These are inverse operations that cancel each other.

Rotations are known to be sufficient for enumerating all binary trees over n nodes (Lucas, van Baronaigien, and Ruskey 1993). Moreover, rotations are known to keep the in-ordering of nodes in a binary tree invariant. Hence, using rotations, one can enumerate all dissections of a given variable order. Figure 5 illustrates how to enumerate all dissections of variable orders over 4 variables, using rotations.

Figure 6 illustrates the *swap* operation on binary trees, denoted by `swap_vnode`(x). This operation switches the left and right children a and b. Upon performing a swap operation, a second swap will undo the first.

Importantly, rotations and swap allow one to explore all vtrees. The proof rests on showing first how to swap two neighboring variables, A and B, in the variable order of a right-linear vtree. Suppose that we have such a vtree which dissects the order $\langle \pi_1, A, B, \pi_2\rangle$, where π_1 and π_2 are suborders (possibly empty). We have two cases. First case: A and B are children of the same parent x. In this case, π_2 must be empty and `swap_vnode`(x) will generate a vtree that dissects the order $\langle \pi_1, B, A\rangle$. Second case: A has parent w and B has parent x. In this case, w must also be a parent of x. Moreover, the operations `lr_vnode`(x), `swap_vnode`(w), and `rr_vnode`(x) will generate a right-linear vtree that dissects the order $\langle \pi_1, B, A, \pi_2\rangle$.

Suppose now that we have an arbitrary vtree dissecting an order π_1 and we wish to navigate to another vtree that dissects a different order π_2. Using rotations only, we can navigate to a right-linear vtree that dissects order π_1. By re-

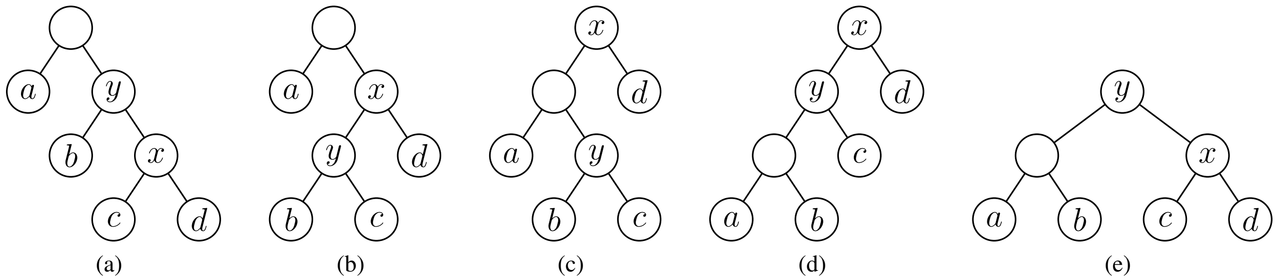


Figure 5: All 5 dissections of variable orders over 4 variables. Starting from vtree 5(a), one obtains vtrees 5(b), 5(c), 5(d), 5(e), and then 5(a) again via the operations $\texttt{lr\_vnode}(x)$, $\texttt{lr\_vnode}(x)$, $\texttt{lr\_vnode}(y)$, $\texttt{rr\_vnode}(x)$ and $\texttt{rr\_vnode}(y)$.

peated application of the technique discussed above, we can navigate to another right-linear vtree that dissects order π_2. We can now navigate to any other vtree that dissects order π_2, using rotations only. Hence, the rotate and swap operations are complete for navigating the space of all vtrees.

The SDD Package

The rest of our discussion will need to make reference to our publicly available implementation, the *SDD Package*. The high-level interface of this package and some of its architecture is highly influenced by the CUDD package for OBDDs. In particular, it provides the following primitive operations: `apply` for conjoining or disjoining two SDDs,[6] `negate` for negating an SDD,[7] $\texttt{lr\_vnode}(x)$, $\texttt{swap\_vnode}(x)$, and $\texttt{rr\_vnode}(x)$ for performing the corresponding operations on vtree nodes and adjusting any corresponding SDDs accordingly (more on this next). The package employs constructs that are similar to those of the CUDD package, such as managers, unique-tables, computation caches, and a garbage collector based on reference counts. Our SDD package exposes all these primitives together with source code for two algorithms that we discuss later: a vtree search algorithm, and a CNF-to-SDD compiler that makes dynamic calls to our vtree search algorithm. First, however, we discuss the process of adjusting an SDD after the underlying vtree has been changed by rotation or swapping.

Adjusting SDDs under Rotate and Swap

Consider the SDD in Figure 1 and its corresponding vtree. Consider in particular the decision node normalized for vtree node 6, which corresponds to a compressed (AB, CD)-partition. If we swap vtree node 6, this decision node must be *adjusted* so it corresponds to an equivalent and compressed (CD, AB)-partition.[8]

A similar adjustment is needed when rotating vtree nodes. Consider for example Figure 4 and suppose that $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are the variables appearing in the vtrees rooted at a, b and c, respectively. Upon right rotation, all decision nodes normalized for vtree node x in Figure 4(a) must be adjusted so they become normalized for vtree node w in Figure 4(b). That is, these decision nodes, which correspond to $(\mathbf{AB}, \mathbf{C})$-partitions, must be adjusted so they correspond to equivalent $(\mathbf{A}, \mathbf{BC})$-partitions. Left rotation calls for a similar adjustment, requiring one to convert $(\mathbf{A}, \mathbf{BC})$-partitions into equivalent $(\mathbf{AB}, \mathbf{C})$-partitions.[9]

Adjusting an SDD in response to a vtree change is then a matter of converting between equivalent $(\mathbf{X}, \mathbf{Y})$-partitions. We will discuss these conversions next and show how they can be implemented using `apply` and `negate`.

The simplest conversion is from an $(\mathbf{A}, \mathbf{BC})$-partition to an $(\mathbf{AB}, \mathbf{C})$-partition (left rotation). Consider then a compressed $(\mathbf{A}, \mathbf{BC})$-partition $\{(a_1, bc_1), \ldots, (a_n, bc_n)\}$, where each sub bc_i has the compressed $(\mathbf{B}, \mathbf{C})$-partition $\{(b_{i1}, c_{i1}), \ldots, (b_{im_i}, c_{im_i})\}$. One can then show that $\{(a_i \wedge b_{ij}, c_{ij}) \mid i = 1, \ldots, n \text{ and } j = 1, \ldots, m_i\}$ is an equivalent $(\mathbf{AB}, \mathbf{C})$-partition, which may not be compressed. This partition can be computed using `apply` to conjoin existing SDD nodes a_i and b_{ij}. Moreover, it can be compressed by disjoining the primes of equal subs, again, using `apply`.

The next two conversions require one to compute the Cartesian product of formula-partitions. In particular, suppose that $\{\alpha_1, \ldots, \alpha_n\}$ is a formula-partition (i.e., $\alpha_i \wedge \alpha_j = \mathsf{false}$ for $i \neq j$, and $\alpha_1 \vee \ldots \vee \alpha_n = \mathsf{true}$). Suppose further that $\{\beta_1, \ldots, \beta_m\}$ is another formula-partition. The Cartesian product is defined as $\{\alpha_i \wedge \beta_j \mid \alpha_i \wedge \beta_j \neq \mathsf{false}, i = 1, \ldots, n, j = 1, \ldots, m\}$. This product is also a formula-partition and can be computed using `apply`.

To see how to adjust an SDD for a swap, consider a compressed $(\mathbf{X}, \mathbf{Y})$-partition $\{(p_1, s_1), \ldots, (p_n, s_n)\}$. We can construct the equivalent, compressed $(\mathbf{Y}, \mathbf{X})$-partition as

[6] The `apply` operation is based on the following result. If $\circ$ is a Boolean operator, and we have two compressed $(\mathbf{X}, \mathbf{Y})$-partitions $\{(p_i, s_i)\}_i$ and $\{(q_j, r_j)\}_j$ for functions f and g, then $\{(p_i \wedge q_j, s_i \circ r_j) \mid p_i \wedge q_j \neq \mathsf{false}\}$ is an $(\mathbf{X}, \mathbf{Y})$-partition for function $f \circ g$, although it may not be compressed. Successively disjoining the primes of equal subs yields a compressed partition.

[7] The negate operator is based on the following result. If $\{(p_i, s_i)\}_i$ is the compressed $(\mathbf{X}, \mathbf{Y})$-partition for function f, then $\{(p_i, \neg s_i)\}_i$ is the compressed $(\mathbf{X}, \mathbf{Y})$-partition for $\neg f$.

[8] This adjustment may lead to the creation of new decision nodes normalized for the descendants of vtree node 6. It may also lead to removing references to existing decisions nodes.

[9] Note that decision nodes normalized for vtree node w in Figure 4(a) continue to be normalized for w after right rotation. Similarly, decision nodes normalized for vtree node x in Figure 4(b) continue to be normalized for x after left rotation.

follows. We first compute the Cartesian product of formula-partitions $\{s_1, \neg s_1\}, \ldots, \{s_n, \neg s_n\}$, which is guaranteed to contain the primes of our sought $(\mathbf{Y}, \mathbf{X})$-partition. Each prime in this product must correspond to a conjunction of the form $c_1 \wedge \ldots \wedge c_n$ where each c_i equals s_i or $\neg s_i$. Let I be the indices i of all $c_i = s_i$. The corresponding sub is then $\bigvee_{i \in I} p_i$. One can show that the described $(\mathbf{Y}, \mathbf{X})$-decomposition is a compressed $(\mathbf{Y}, \mathbf{X})$-partition and equivalent to the original $(\mathbf{X}, \mathbf{Y})$-partition. Moreover, it can be directly computed using `apply` and `negate`.

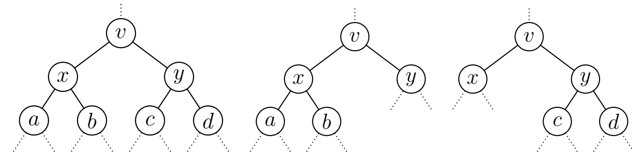
Finally, we consider the adjustment of an SDD due to a right rotation. Consider a compressed $(\mathbf{AB}, \mathbf{C})$-partition $\{(ab_1, c_1), \ldots, (ab_n, c_n)\}$, where each prime ab_i has the compressed $(\mathbf{A}, \mathbf{B})$-partition $\{(a_{i1}, b_{i1}), \ldots, (a_{im_i}, b_{im_i})\}$. We first compute the Cartesian product of formula-partitions $\{a_{11}, \ldots, a_{1m_1}\}, \ldots, \{a_{n1}, \ldots, a_{nm_n}\}$, which is guaranteed to contain the primes of our sought $(\mathbf{A}, \mathbf{BC})$-partition. Each prime in this product corresponds to a conjunction of the form $a_{1j_1} \wedge \ldots \wedge a_{nj_n}$. The corresponding sub is then $\bigvee_{i=1}^{n} b_{ij_i} \wedge c_i$. One can show that the described $(\mathbf{A}, \mathbf{BC})$-decomposition is an $(\mathbf{A}, \mathbf{BC})$-partition and equivalent to the original $(\mathbf{AB}, \mathbf{C})$-partition, but may not be compressed. It can also be computed and compressed using `apply`.

We close this section by a comparison to the process of adjusting OBDDs after swapping two neighboring variables in a total variable order. Such adjustments are known to have a bounded impact on the OBDD size as it only involves a local adjustment to the OBDD structure (Rudell 1993). Hence, when searching for a total variable order using variable swaps, each move in the search space is guaranteed to be efficient. The situation is different for vtrees. In particular, each move in this space (rotation or swap) involves non-trivial changes to the SDD structure. In fact, (Xue, Choi, and Darwiche 2012) has shown that swapping the children of a vtree node can lead to an exponential change in the SDD size. Hence, while swapping two variables in a total variable order can have a predictable, but conservative, effect on the OBDD size, swapping a single pair of children in a vtree can potentially obtain a significantly more succinct SDD in one operation. From this perspective, the potentially expensive swap operation for vtrees allows one to make large jumps in the search space, whereas the relatively inexpensive swap operation in total variable orders may need to be applied many times before achieving the same effect.

Searching for a Good Vtree

Our search algorithm assumes an existing vtree and a corresponding SDD. The algorithm can be called on any vtree node v and it will try to minimize the SDD size by searching for a new sub-vtree to replace the one currently rooted at v.

The algorithm first makes recursive calls on the children of vtree node v. It then considers v and the two levels below it (when applicable) as shown in Figure 7(a). One can isolate two vtree fragments in this figure: the l-vtree in Figure 7(b) and the r-vtree in Figure 7(c). The l-vtree is just one of 12 vtrees over leaves a, b and y. Similarly, the r-vtree is just one of 12 vtrees over leaves x, c and d. Each one of these 24 vtrees leads to a variation on the original sub-vtree



(a) vtree fragment (b) l-vtree (c) r-vtree

Figure 7: A vtree fragment.

rooted at v. The proposed search algorithm *attempts* to navigate through all 24 variations using a pre-stored sequence of rotations and swaps which is guaranteed to cycle through them, returning back to the original l-vtree or r-vtree that we start with. The algorithm then chooses the one variation with smallest SDD size[10] and navigates back to that variation. At this point, the algorithm is said to have completed a single pass on the sub-vtree rooted at v.

If a pass changes the SDD size by more than a given threshold (set to 1% in our experiments), another pass is made. That is, another call is made on vtree node v with corresponding recursive calls on its children. Otherwise, the algorithm terminates.

We will now explain the term "attempt" used earlier. The SDD package allows the user to specify time or size limits for the rotate and swap operations. If these limits are exceeded while the operation is taking place, the operation fails and the state of the vtree and corresponding SDD are rolled back to how they existed before the operation started. In our experiments, we use size limits but not time limits. Thus, the algorithm may not navigate through all 24 variations described above if the size limit is exceeded. We use a size limit of 25% for swap, causing the operation to fail if swapping a vtree node leads to increasing the SDD size by more than 25%. The size limits for rotations are set to 75%.

Dynamic Compilation of CNFs into SDDs

One typically generates an SDD incrementally and tries to minimize it if its size starts growing too much during the generation process. Consider for example the process of compiling a CNF into an SDD. One typically starts with an initial vtree, compiles each clause of the CNF into a corresponding SDD, and then conjoins these SDDs (using `apply`). Since these conjoin operations take place in sequence, the final SDD is then said to be constructed incrementally. Typically, if a conjoin operation grows the SDD size by a certain factor, one tries to search for a better vtree before proceeding with the remaining conjoin operations. This would be the typical usage of the search algorithm developed in the previous section. This would also be the proper context for evaluating its effectiveness — which is also the context usually used for evaluating dynamic ordering heuristics for OBDDs. The experiments of the next section are thus conducted in the context of compiling CNFs to SDDs while using dynamic vtree search as discussed above.

[10]We break ties by preferring smaller decision node counts and better vtree balance.

Suppose we are given a CNF Δ as a set of clauses and a vtree for the variables of Δ. Suppose further that each clause c is assigned to the lowest vtree node v which contains the variables of clause c (node v is unique). Our algorithm for compiling CNFs takes this labeled vtree as input.

The initial vtree structure provides a recursive partitioning of the CNF clauses, with each node v in the vtree hosting a set of clauses Δ_v. The algorithm recursively compiles the clauses hosted by nodes in the sub-vtrees rooted at the children of v. This leads to two SDDs corresponding to these children. The algorithm conjoins these two SDDs using `apply`. It then iterates over the clauses hosted at node v, compiling each into an SDD,[11] and conjoining the result with the existing SDD. If this last conjoin operation grows the SDD size by more than a certain factor (since the last call to vtree search), the vtree search algorithm is called again on node v.[12] In our experiments, we set this factor to 20%. We also visit the clauses hosted by node v according to their length, with shorter clauses visited first.

Experimental Results

We evaluate our algorithm on CNFs of combinational circuits used in the CAD community, and in particular, from the `LGSynth89`, `iscas85` and `iscas89` benchmark sets. We use the CUDD package to compile CNFs to OBDDs, using dynamic variable reordering and default parameters.[13] We further conjoin clauses according to the procedure from the previous section, assuming a right-linear vtree induced by the natural variable order of the CNF.[14] For our SDD compiler, we initially used a balanced vtree dissecting the natural variable order of the CNF. Experiments on the `LGSynth89` suite were performed on a 2.67GHz Intel Xeon x5650 CPU with access to 12GB RAM. Experiments on the `iscas85` and `iscas89` suites were performed on a 2.83GHz Intel Xeon x5440 CPU with access to 8GB RAM. We excluded benchmarks from these suites if (1) both OBDD and SDD compilations failed given a two hour time limit, or (2) if the resulting SDD has a size of less than 2,000, which we consider too trivial.

Our experimental results in Table 1 call for a number of observations. First, the SDD turns out to be a more compact representation in all benchmarks successfully compiled by both the SDD and OBDD compilers. This is perhaps not too surprising as it has been previously observed that even a random dissection of total variable orders tend to lead to SDDs that are more compact than the corresponding OBDDs (Darwiche 2011). We also note that in 11 of the benchmarks we evaluated, we observed at least an order-of-magnitude improvement in size. This suggests that the theoretical properties of SDDs discussed by (Xue, Choi, and Darwiche 2012) can also be realized in practice. Moreover, in 4 instances, SDD compilation succeeded and OBDD compilation failed. In 2 instances, OBDD compilation succeeded and SDD compilation failed.

Next, our second observation is that in many benchmarks, our dynamic compilation algorithm was faster than CUDD's, and in 7 cases, orders-of-magnitude faster. This is more evident in the more challenging benchmarks with larger compilations. This is particularly interesting since dynamic vtree search uses more expensive, yet more powerful, operations for navigating its search space, in comparison to the efficient, but less powerful, operation used for navigating total variable orders.

We see similar results in Table 2, where as a preprocessing step, we applied MINCE to our CNFs, to find an initial static variable ordering for CUDD (Aloul, Markov, and Sakallah 2004). For SDDs, we initially used a balanced vtree dissecting the same order. We observe, in general, that using MINCE orderings improves the resulting OBDD and SDD compilations, further allowing both to successfully compile cases where they failed to before. We find that SDD compilations, in 4 cases, can still be an order-of-magnitude more succinct. Moreover, there were 5 cases where SDD compilation succeeded and OBDD compilation failed. In total, across all experiments, there were 15 cases where the SDD compilation was at least an order-of-magnitude more succinct than the OBDD compilation. Moreover, there were 9 cases where SDD compilation succeeded and OBDD compilation failed, and there were 3 cases where OBDD compilation succeeded and SDD compilation failed.

We finally ask: Is it the total variable orders discovered by our vtree search algorithm, or the particular dissection of these orders, which is responsible for these favorable results? To help answer this question, we extracted the variable order embedded in each discovered vtree and then constructed an OBDD using that order. The sizes of these OBDDs are reported in the column titled "r. linear SDD." In a number of cases, the resulting OBDDs are much worse than the OBDDs found by CUDD. These cases show emphatically that dissection is what explains the improvements (i.e., making decisions on arbitrary sentences instead of literals). That is, by simply dissecting these OBDDs, we can obtain even more succinct SDDs, even if we dissect an OBDD with a poor variable order. In other cases, the resulting OBDD is comparable or more succinct than the OBDD found by CUDD. Interestingly, this suggests that the vtree search algorithm that we proposed can also find effective variable orderings, in comparison to more specialized reordering heuristics.

[11]The SDD package provides a primitive that returns an SDD for a given literal. Hence, one can easily compile a clause into an SDD by disjoining the SDDs corresponding to its literals, using `apply`.

[12]In principle, a clause hosted at node v, that has not yet been conjoined, could possibly be re-assigned to a lower node after a vtree search is invoked. However, we would have already visited these nodes during the compilation algorithm, so we just finish conjoining those clauses at node v.

[13]We used heuristic `CUDD_REORDER_SYMM_SIFT`, which is symmetric sifting (Panda, Somenzi, and Plessier 1994). We also invoked sifting in a post-processing step, after compilation.

[14]In an earlier version of this paper, we assumed a natural ordering of the clauses, which produced worse results for OBDD compilations. Moreover, these earlier evaluations always pre-processed the CNF using MINCE. Here, we consider compilation with and without MINCE, leading to a more revealing comparison.

CNF	size OBDD	size SDD	OBDD/SDD	time OBDD	time SDD	OBDD/SDD	r. linear SDD	OBDD/SDD
9symml	54,230	—	—	10.64	—	—	—	—
C432	—	9,469	—	—	24.63	—	88,512	—
C432_out	228,036	5,768	**39.53**	178.07	6.54	**27.23**	36,370	6.27
apex7	148,876	8,191	**18.18**	164.16	46.07	3.56	120,414	1.24
b9	136,440	8,813	**15.48**	4,619.65	27.31	**169.16**	186,722	0.73
c8	2,456,186	22,179	**110.74**	5,047.04	1,289.80	3.91	307,402	7.99
cht	12,296	4,924	2.50	1.39	9.56	0.15	38,492	0.32
comp	2,774	2,211	1.25	3.30	5.65	0.58	4,664	0.59
count	55,312	3,359	**16.47**	1,719.99	5.37	**320.30**	7,902	7.00
example2	34,456	8,361	4.12	247.38	127.27	1.94	58,806	0.59
f51m	10,186	3,090	3.30	0.47	2.65	0.18	10,612	0.96
frg1	358,216	81,133	4.42	60.16	169.42	0.36	750,776	0.48
lal	31,082	6,139	5.06	19.23	29.40	0.65	107,424	0.29
mux	3,052	2,058	1.48	0.33	1.04	0.32	5,924	0.52
my_adder	19,632	2,959	6.63	75.14	4.68	**16.06**	8,416	2.33
pm1	2,650	2,107	1.26	0.45	1.53	0.29	8,688	0.31
sct	9,954	8,094	1.23	3.60	332.45	0.01	53,206	0.19
ttt2	244,598	14,872	**16.45**	703.63	139.61	5.04	79,768	3.07
unreg	20,338	3,092	6.58	4.97	2.78	1.79	5,814	3.50
vda	43,626	—	—	4,557.45	—	—	—	—
z4ml	3,112	2,311	1.35	0.12	1.21	0.10	6,160	0.51
s298	10,122	3,758	2.69	2.31	3.43	0.67	28,370	0.36
s344	39,646	5,769	6.87	9.61	5.67	1.69	39,778	1.00
s349	10,048	3,319	3.03	3.99	4.10	0.97	18,346	0.55
s382	17,974	3,520	5.11	14.25	4.13	3.45	12,412	1.45
s386	28,560	7,162	3.99	10.46	11.39	0.92	33,938	0.84
s400	6,088	3,520	1.73	8.55	4.43	1.93	13,454	0.45
s420	17,744	4,180	4.24	16.25	5.56	2.92	11,302	1.57
s444	10,472	4,024	2.60	51.51	6.21	8.29	74,816	0.14
s510	26,892	7,136	3.77	157.95	96.13	1.64	30,196	0.89
s526	129,662	9,446	**13.73**	49.73	53.14	0.94	92,394	1.40
s526N	107,334	6,649	**16.14**	136.64	20.61	6.63	45,230	2.37
s641	517,688	10,164	**50.93**	1,434.14	18.09	**79.28**	229,650	2.25
s713	—	11,386	—	—	26.46	—	291,816	—
s832	125,652	23,028	5.46	659.46	252.79	2.61	255,916	0.49
s838.1	52,934	9,387	5.64	442.04	41.16	**10.74**	30,926	1.71
s838	312,470	14,458	**21.61**	1,536.27	776.06	1.98	119,962	2.60
c432	689,092	9,922	**69.45**	1,416.69	71.93	**19.70**	96,380	7.15
c499	—	387,400	—	—	1,476.44	—	—	—
c1355	—	325,193	—	—	2,550.90	—	—	—

Table 1: OBDD and SDD compilations over `LGSynth89`, `iscas85` and `iscas89` suites. Missing entries indicate a failed compilation, either an out-of-memory, or a timeout of 2 hours. OBDD/SDD columns report relative improvement (in size or time). Bolded text indicate cases where time/size improvements were an order-of-magnitude or more, or if SDD compilation succeeded and OBDD compilation failed. Reported sizes are based on SDD notation. Reported times are in seconds.

	size		$\frac{\text{OBDD}}{\text{SDD}}$	time		$\frac{\text{OBDD}}{\text{SDD}}$	r. linear	$\frac{\text{OBDD}}{\text{SDD}}$
CNF	OBDD	SDD		OBDD	SDD		SDD	
9symml	49,576	14,057	3.53	6.05	35.17	0.17	102,596	0.48
C432	169,984	9,063	**18.76**	69.71	52.54	1.33	110,148	1.54
C499	—	240,253	—	—	1,033.67	—	18,369,492	—
C1355	—	723,549	—	—	1,494.54	—	—	—
C1908	—	3,216,974	—	—	5,676.72	—	—	—
alu2	38,298	11,513	3.33	8.89	90.11	0.10	108,250	0.35
alu4	267,562	—	—	5,982.86	—	—	—	—
apex6	—	283,174	—	—	1,746.71	—	—	—
apex7	33,124	6,509	5.09	2.38	5.51	0.43	61,878	0.54
b9	50,270	10,362	4.85	8.95	5.16	1.73	486,484	0.10
c8	54,674	13,257	4.12	8.07	10.09	0.80	142,138	0.38
cht	8,764	4,012	2.18	0.31	1.92	0.16	11,262	0.78
count	13,938	3,157	4.41	1.89	1.94	0.97	8,262	1.69
example2	15,044	7,742	1.94	2.48	6.80	0.36	43,618	0.34
f51m	8,030	3,149	2.55	0.32	1.34	0.24	9,674	0.83
frg1	374,988	107,447	3.49	51.84	164.10	0.32	772,108	0.49
frg2	—	255,136	—	—	6,591.69	—	—	—
lal	7,216	5,229	1.38	1.01	3.09	0.33	19,062	0.38
mux	3,366	2,061	1.63	0.06	0.57	0.11	10,090	0.33
my_adder	1,728	2,341	0.74	0.06	1.15	0.05	2,822	0.61
sct	7,154	6,948	1.03	1.63	7.84	0.21	41,564	0.17
term1	1,484,518	128,152	**11.58**	2,209.67	873.59	2.53	3,930,362	0.38
ttt2	36,428	13,331	2.73	6.61	14.83	0.45	72,268	0.50
unreg	39,590	3,168	**12.50**	4.22	18.06	0.23	8,232	4.81
vda	44,568	13,659	3.26	599.67	125.72	4.77	—	—
x4	595,160	27,720	**21.47**	961.90	47.46	**20.27**	2,091,326	0.28

Table 2: OBDD and SDD compilations over the `LGSynth89` suite, using MINCE variable orders. See also Table 1.

Conclusion

This paper provides the elements necessary for making the sentential decision diagram (SDD) a viable tool in practice — at least in comparison to the influential OBDD. In particular, the paper shows how one may dynamically search for vtrees that attempt to minimize the size of constructed SDDs. Our contributions include: (1) a characterization of the vtree search space as dissections of total variable orders; (2) an identification of two vtree operations that allow one to navigate this search space; (3) corresponding operations for adjusting an SDD due to vtree changes; (4) a CNF to SDD compiler based on dynamic vtree search; and (5) a publicly available SDD package that embodies all of the previous elements. Empirically, our results show that our proposed approach for constructing SDDs can lead to order-of-magnitude improvements in time and space over similar approaches for constructing OBDDs. By releasing our SDD package to the public, we hope that the community will have the necessary infrastructure for building even more effective search algorithms in the future.

Acknowledgments

This work has been partially supported by ONR grant #N00014-12-1-0423, NSF grant #IIS-1118122, and NSF grant #IIS-0916161.

References

Aloul, F. A.; Markov, I. L.; and Sakallah, K. A. 2004. Mince: A static global variable-ordering heuristic for SAT search and BDD manipulation. *J. UCS* 10(12):1562–1596.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35:677–691.

Campbell, D. M. 1984. The computation of Catalan numbers. *Mathematics Magazine* 57(4):195–208.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, 819–826.

Ferrara, A.; Pan, G.; and Vardi, M. Y. 2005. Treewidth in verification: Local vs. global. In *LPAR*, 489–503.

Huang, J., and Darwiche, A. 2004. Using DPLL for efficient OBDD construction. In *SAT*.

Knuth, D. E. 2005. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations*. Addison-Wesley Professional.

Lucas, J. M.; van Baronaigien, D. R.; and Ruskey, F. 1993. On rotations and the generation of binary trees. *J. Algorithms* 15(3):343–366.

Panda, S.; Somenzi, F.; and Plessier, B. 1994. Symmetry detection and dynamic variable ordering of decision diagrams. In *ICCAD*, 628–631.

Pipatsrisawat, K., and Darwiche, A. 2010. A lower bound on the size of decomposable negation normal form. In *AAAI*.

Prasad, M. R.; Chong, P.; and Keutzer, K. 1999. Why is ATPG easy? In *DAC*, 22–28.

Rudell, R. 1993. Dynamic variable ordering for Ordered Binary Decision Diagrams. In *ICCAD*, 42–47.

Somenzi, F. 2004. CUDD: CU decision diagram package. http://vlsi.colorado.edu/~fabio/CUDD/.

Xue, Y.; Choi, A.; and Darwiche, A. 2012. Basing decisions on sentences in decision diagrams. In *AAAI*, 842–849.