SDD Advanced-User Manual
Version 2.0

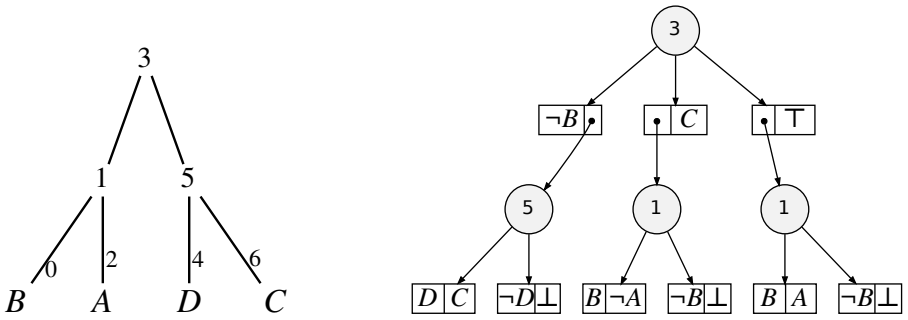
Arthur Choi and **Adnan Darwiche**
Automated Reasoning Group
Computer Science Department
University of California, Los Angeles

Email: sdd@cs.ucla.edu
Download: http://reasoning.cs.ucla.edu/sdd

January 8, 2018

The Sentential Decision Diagram (SDD) is a canonical representation of Boolean functions. The SDD package allows users to construct, manipulate and optimize SDDs. This manual describes the SDD package, which is distributed as C source code, and includes code for compiling CNFs and DNFs into SDDs. The license terms for the SDD package are given at the end of this manual.

Contents



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

Figure 1: A vtree and a corresponding SDD for the Boolean function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.

1 Introduction

The Sentential Decision Diagram (SDD) is a recently proposed representation of Boolean functions [4, 12, 2]. The SDD can be thought of as a "data structure" for representing Boolean functions since SDDs are canonical and support a number of efficient operations for constructing and manipulating Boolean functions.[1] Figure 1(b) depicts an example SDD and the Boolean function it represents.

This manual starts by going over some of the basic concepts underlying Sentential Decision Diagrams. These concepts are essential for getting started with the SDD library, which is discussed in Section 5. The code for compiling CNFs and DNFs into SDDs is then discussed in Section 7. Smaller code samples are also provided in Section 8 to illustrate some of the SDD library features.

1.1 Compressed Partitions

SDDs are based on a new type of Boolean function decomposition, which we start with. Consider a Boolean function f and suppose that we split its variables into two disjoint sets, $\mathbf{X}$ and $\mathbf{Y}$. We can always decompose function f as follows

$$f = \Big[p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})\Big] \vee \cdots \vee \Big[p_n(\mathbf{X}) \wedge s_n(\mathbf{Y})\Big],$$

while satisfying the following two conditions:

- *Partition:* The functions $p_i(\mathbf{X})$ are mutually exclusive, exhaustive, and non-false;
- *Compression:* The functions $s_i(\mathbf{Y})$ are distinct.

This kind of a decomposition always exists and is unique for a particular split of the function variables. This decomposition is called a *compressed* $(\mathbf{X}, \mathbf{Y})$-partition of the function f, where the functions $p_i(\mathbf{X})$ are called *primes* and the functions $s_i(\mathbf{Y})$ are called *subs* [4].

For an example, consider the Boolean function:

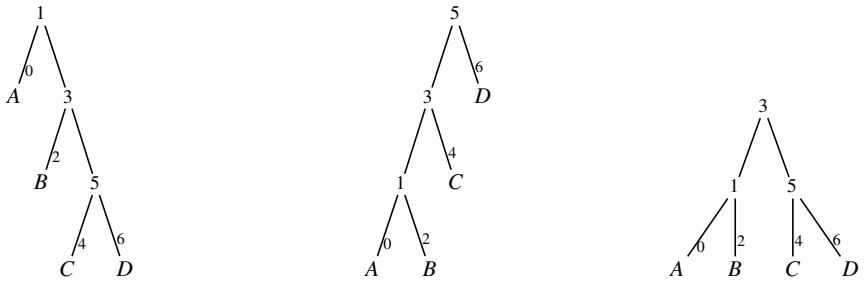
$$f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D). \tag{1}$$

By splitting the function variables into $\mathbf{X} = \{A, B\}$ and $\mathbf{Y} = \{C, D\}$, we get the following decomposition:

$$(\underbrace{A \wedge B}_{\text{prime}} \wedge \underbrace{\mathsf{true}}_{\text{sub}}) \vee (\underbrace{\neg A \wedge B}_{\text{prime}} \wedge \underbrace{C}_{\text{sub}}) \vee (\underbrace{\neg B}_{\text{prime}} \wedge \underbrace{C \wedge D}_{\text{sub}}). \tag{2}$$

The primes are mutually exclusive, exhaustive and non-false. Moreover, the subs are distinct.

[1] The SDD is a generalization of the Ordered Binary Decision Diagram (OBDD) [4]. SDDs have been shown to be exponentially more succinct than OBDDs [1].



(a) right-linear (b) left-linear (c) balanced

Figure 2: Different vtrees over the variables A, B, C, and D. A *right-linear* vtree is one in which each left child is a leaf. A *left-linear* vtree is one in which each right child is a leaf.

An SDD can be constructed for a Boolean function as follows. We first split the function variables into disjoint sets, $\mathbf{X}$ and $\mathbf{Y}$, and then compute its unique compressed $(\mathbf{X}, \mathbf{Y})$-partition. The process is repeated recursively for each of the resulting primes and subs, until reaching constants (true and false) or literals. These are called *terminal SDDs* and correspond to the base cases of this recursive process. Note that this process can yield different SDDs, depending on how we split the variables of each considered function. These splitting choices, however, can be fixed using the notion of a *vtree,* leading to canonical SDDs.

1.2 Vtrees

A *vtree* stands for a *variable vtree.* It is a full, binary tree, with its leaves labeled with the variables of a Boolean function. Figure 2 depicts a few vtrees over variables A, B, C, and D.

Suppose now that f is a non-trivial Boolean function and let v be the *lowest* vtree node that includes the variables that function f depends on. We will say in this case that function f is *normalized* for vtree node v. For an example, the function $f = B \wedge D$ is normalized for node $v = 3$ in Figure 2(a). It is also normalized for node $v = 5$ in Figure 2(b), and for node $v = 3$ in Figure 2(c).

If a Boolean function f is normalized for an internal vtree node v, its variables can be uniquely split into:

- $\mathbf{X}$: Variables of f appearing in the *left* subtree of v.
- $\mathbf{Y}$: Variables of f appearing in the *right* subtree of v.

We next show how this variable splitting strategy leads to a canonical SDD, once a vtree is fixed.

2 Sentential Decision Diagrams (SDDs)

Consider the Boolean function and vtree in Figure 1. This Boolean function is normalized for node $v = 3$, leading to the variable split $\mathbf{X} = \{A, B\}$, $\mathbf{Y} = \{C, D\}$ and the following decomposition:

$$(A \wedge B) \vee (B \wedge C) \vee (C \wedge D) = (\underbrace{A \wedge B}_{\text{prime}} \wedge \underbrace{\mathsf{true}}_{\text{sub}}) \vee (\underbrace{\neg A \wedge B}_{\text{prime}} \wedge \underbrace{C}_{\text{sub}}) \vee (\underbrace{\neg B}_{\text{prime}} \wedge \underbrace{C \wedge D}_{\text{sub}}). \tag{3}$$

This decomposition has three primes: $A \wedge B$, $\neg A \wedge B$ and $\neg B$. The last prime is a literal and is therefore a *terminal SDD.* The first two primes, however, will need to be decomposed further. They are both normalized for node $v = 1$, leading to the variable split $\mathbf{X} = \{B\}$, $\mathbf{Y} = \{A\}$ and the decompositions:

$$A \wedge B = (\underbrace{B}_{\text{prime}} \wedge \underbrace{A}_{\text{sub}}) \vee (\underbrace{\neg B}_{\text{prime}} \wedge \underbrace{\bot}_{\text{sub}}), \qquad \neg A \wedge B = (\underbrace{B}_{\text{prime}} \wedge \underbrace{\neg A}_{\text{sub}}) \vee (\underbrace{\neg B}_{\text{prime}} \wedge \underbrace{\bot}_{\text{sub}}).$$

Note that all resulting primes and subs are terminal SDDs.

The decomposition in (3) also has three subs: true, C, and $C \wedge D$. The first two are terminal SDDs. The last sub is not terminal. It is normalized for node $v = 5$, leading to the variable split $\mathbf{X} = \{D\}$, $\mathbf{Y} = \{C\}$ and the corresponding decomposition:

$$C \wedge D = (\underbrace{D}_{\text{prime}} \wedge \underbrace{C}_{\text{sub}}) \vee (\underbrace{\neg D}_{\text{prime}} \wedge \underbrace{\bot}_{\text{sub}}).$$

Note again that all resulting primes and subs are terminal SDDs.

2.1 Graphical Depiction

We have now fully decomposed the Boolean function of Figure 1, leading to the canonical SDD depicted graphically in Figure 1(b). This figure contains four decompositions, each of which is depicted by a circle ◯, which is called a *decision SDD node.* The children of a decision SDD node are depicted by paired boxes $\boxed{p}\boxed{s}$, called *elements.* The left box of an element corresponds to a prime p, while the right box corresponds to its sub s. In the graphical depiction of SDDs, a prime p (sub s) is either a constant, literal or pointer to a decision SDD node. Constants and literals are called *terminal SDD nodes.*

2.2 Size and Count

The *size* of a decision SDD node is the number of its elements. The *size* of a terminal SDD node is 0. The *size* of an SDD is the sum of the sizes attained by its nodes. The root decision node of Figure 1(b) has size 3. The remaining decision nodes have size 2 each. The size of this SDD is then $9 = 3 + 2 + 2 + 2$. The *node count* of an SDD is the number of its decision nodes. The node count of the SDD in Figure 1(b) is 4. The SDD library provides primitives for retrieving the size of an SDD and the count of its decision nodes.

2.3 Shared SDDs

An SDD is typically identified by its root SDD node. A *shared SDD* is a multi-rooted SDD, with each root representing its own Boolean function. Figure 3(a) depicts a shared SDD that represents three distinct Boolean functions, one for each root. The size of this shared SDD is 12. Its node count is 6.

2.4 Normalization

A literal or decision SDD node n is associated with the lowest vtree node v that contains the variables of n. We say in this case that n is *normalized* for v. A constant SDD node is not normalized for (associated with) any vtree node.[2] The SDD library provides primitives for saving SDDs into the *dot* format.[3] These primitives label each decision SDD node with the vtree node it is normalized for. All example SDDs used in this manual have been produced using these primitives.

2.5 SDD and Vtree Nodes

When an SDD node n is normalized for some node in vtree v, we will say that *node n is in vtree v*. We refer to the *SDDs of vtree v* when we want to refer to the collection of SDD nodes in vtree v. The SDD library provides primitives for accessing the size and count of SDD nodes in some vtree v.

[2] If f is the Boolean function represented by a literal or decision node n, then function f and node n are guaranteed to be normalized for the same vtree node (see Section 1.2).

[3] The *dot* format for describing graphs can be visualized by the GraphViz system, available at `http://www.graphviz.org/`

2.6 Bottom-up Construction of SDDs

Section 2 started by describing a top-down procedure for constructing SDDs, which decomposes a Boolean function and then recursively decomposes its primes and subs. Using the SDD library, however, SDDs are constructed in a bottom-up fashion. For example, to construct an SDD for the function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$, we first retrieve terminal SDDs for the literals A, B, C, and D. We then *conjoin* the terminal SDD for literal A with the one for literal B, to obtain an SDD for the term $A \wedge B$. The process is repeated to obtain SDDs for the terms $B \wedge C$ and $C \wedge D$. The resulting SDDs are then *disjoined* to obtain an SDD for the whole function. The SDD library provides primitives for retrieving terminal SDDs. It also provides primitives for conjoining, disjoining and negating SDDs. These primitives are explained in Section 5. A code sample is given in Section 8 that uses these primitives to construct an SDD for this particular example.

2.7 Canonicity

The SDDs we discussed are canonical and have been called *trimmed* and *compressed* SDDs in [4]; see also [11]. Another type of canonical SDD was identified in [4], called *normalized* SDDs. The SDD library supports only trimmed and compressed SDDs.

3 Garbage Collection

Suppose that two SDDs α and β are conjoined to yield a third SDD γ. Given how the conjoin and other SDD operations work, this process may yield additional SDD nodes that are not part of either α, β or γ. The SDD library keeps these auxiliary SDD nodes in memory as they correspond to the results of intermediate computations that can be reused when performing further conjoin and disjoin operations. While this strategy improves performance (i.e., saves time), it may eventually exhaust the available memory. To address this issue, the SDD library provides a garbage collector that can be invoked either by the user, or automatically by the SDD library. This is described next.

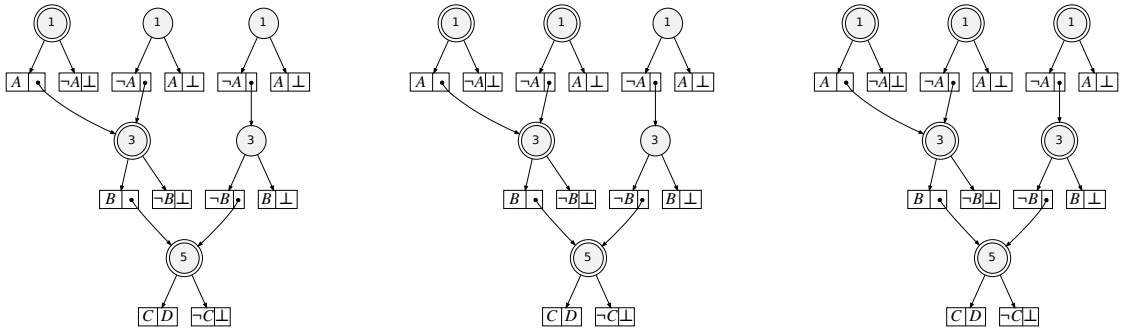
3.1 Live and Dead Nodes

A decision SDD node is either *live* or *dead*. When the garbage collector is invoked, dead SDD nodes are claimed (their memory is freed). The SDD library provides a primitive for *referencing* SDD nodes. When a node is referenced, it is guaranteed to be live. More generally, a decision SDD node is live if and only if (a) it has been referenced by the user or (b) it is referenced by a live SDD node (as a prime or sub). A reference to a decision SDD node can be canceled by *dereferencing* the node. The SDD library provides a primitive for dereferencing SDD nodes. Terminal SDDs are always live (never garbage collected). Terminal SDDs can be referenced and dereferenced but these operations have no effect in this case.

Consider Figure 3(a) in which the left root has been referenced by the user. This SDD has three live nodes: The one that has been referenced by the user (left root) and two other nodes that are referenced by live SDD nodes. A further reference by the user to the middle root leads to Figure 3(b). A final user reference to the right root leads to Figure 3(c) in which all nodes are now live. If we dereference the right root in Figure 3(c), we go back to Figure 3(b).

3.2 Reference Counts

The garbage collector is based on reference counts. In particular, every decision SDD node n has a reference count, whose value is the sum of two quantities: (a) the number of (uncanceled) references made by the user to node n, and (b) the number of live SDD nodes referencing node n (as a prime or sub). Hence, a decision SDD node is live if and only if its reference count is greater than 0. The SDD library provides a primitive for accessing this reference count of an SDD node.



(a) User referenced left root (b) User referenced left, middle roots (c) User referenced all roots

Figure 3: Double circles are live nodes and single circles are dead nodes.

3.3 Automatic and Manual Garbage Collection

The SDD library provides primitives for activating and deactivating automatic garbage collection.

The user can also invoke garbage collection manually at two levels: globally and locally. *Global* garbage collection will claim any dead SDD node. *Local* garbage collection is applied to a vtree v and it will claim dead SDD nodes in vtree v and above vtree v.[4] The SDD library provides primitives for both types of garbage collection, which are discussed in Section 5.

3.4 When to Reference Nodes

During automatic garbage collection, or before invoking the garbage collector manually, the user must make sure that all SDD nodes of interest are referenced.

If automatic garbage collection is not being used, and the user does not intend to invoke the garbage collector (a rare situation), then no referencing is ever needed. If automatic garbage collection is not being used, a more realistic situation, however, is for the user to call the garbage collector at specific places in their application, allowing them to selectively decide what SDD nodes to reference and when. Typically, one invokes the garbage collector manually when the number of dead nodes grows too large. The SDD library facilitates this process by providing primitives for conditional garbage collection, allowing the user to invoke garbage collection when the percentage of dead SDD nodes exceeds a certain threshold. During automatic garbage collection, the SDD library makes its own decisions on when to invoke the garbage collector, again based on the number of dead nodes.

4 More on Vtrees

We now provide further details on the support provided by the SDD library for vtrees.

4.1 Orders

Each vtree induces a total order on its variables, which is obtained by a left-to-right traversal of the vtree nodes. For example, all vtrees in Figure 2 induce the same total variable order: $\langle A, B, C, D \rangle$. On the other hand, the vtree in Figure 1(a) induces the total variable order $\langle B, A, D, C \rangle$. The SDD library provides primitives for retrieving the total variable order induced by a vtree. It also provides primitives for constructing

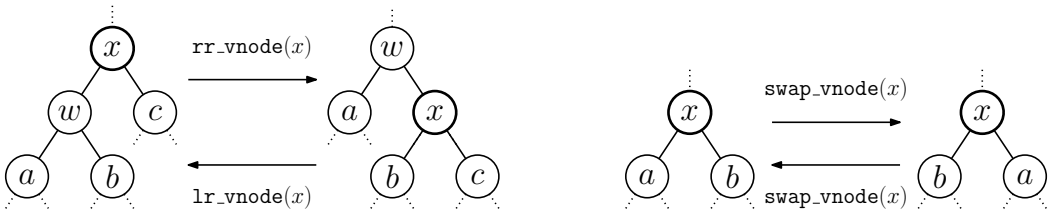
[4] If we view v as a vtree node, garbage collected SDD nodes will be those normalized for v, its descendants and its ancestors.

vtrees according to a given variable order. A code sample is provided in Section 8, which illustrates some methods for constructing vtrees.

More generally, a vtree induces a total order on all of its nodes, which is obtained by an inorder traversal of the vtree nodes (i.e., left subtree, node, right subtree). In Figure 2, we have labeled each vtree node with its *position* in the corresponding vtree inorder (the first position is 0). The SDD library provides primitives for retrieving the position of each vtree node. The library also provides primitives for saving vtrees into the *dot* format, while labeling each vtree node with its position in the corresponding inorder.[5] All example vtrees used in this manual have been produced using this primitive.

4.2 Operations

The SDD library provides primitives for changing the structure of a vtree and adjusting the structure of corresponding SDDs. Three primitives are provided for this purpose, which are explained in Figure 4: *right-rotating* a vtree node, *left-rotating* a vtree node, and *swapping* a vtree node.[6]



(a) Rotating a vtree node.

(b) Swapping the children of a vtree node.

Figure 4: Rotating and swapping vtree nodes.

For an example of these operations, consider the vtree and corresponding SDD in Figure 5(a). If we left-rotate vtree node $v = 3$, we get the vtree and corresponding SDD in Figure 5(b). Note how the rotation operation has adjusted the SDD structure in addition to changing the vtree structure; for details, see [2].

Consider Figure 4. Before swapping or right-rotating x, the user must ensure that there are no dead SDD nodes inside or above vtree x. Moreover, before left-rotating x, the user must ensure that are no dead SDD nodes inside or above vtree w. Swapping and rotation operations will not introduce any dead SDD nodes.

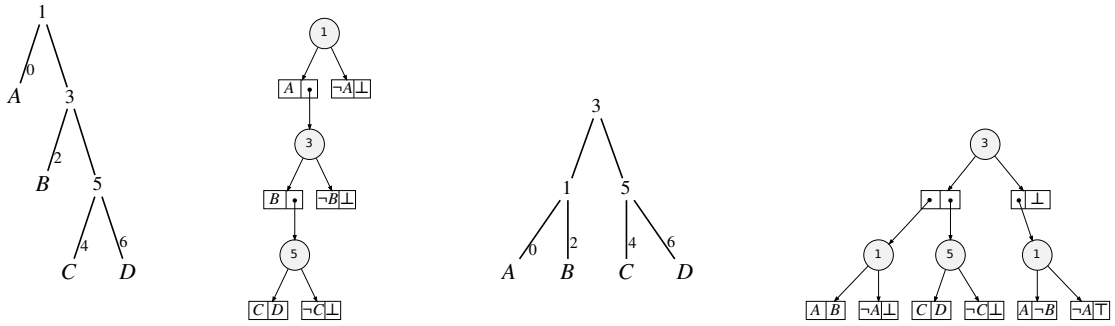
These operations are typically used to search for an alternative of some vtree v, which implies that all operations will be applied to nodes inside vtree v. In this case, one should start by calling local garbage collection on vtree v, to ensure that there are no dead nodes inside or above vtree v. This ensures the preconditions of these operations. Moreover, since these operations do not introduce dead SDD nodes, the preconditions continue to hold after applying each operation.

4.3 Roots

Consider the vtree on the left of Figure 4(a). The root of this vtree is x. Yet, when we right rotate x, the new root of this vtree becomes w. More generally, when applying rotation operations to a vtree fragment, the root of that fragment may change. One can in principle keep track of the various rotations applied and then compute the new root of the vtree fragment based on which rotations were applied. The SDD library facilitates this process by providing a primitive that recovers the new root of a vtree fragment. This primitive returns the memory location that stores the pointer to the vtree root. By recovering the content of this location, the user will then have access to the current root (which may have changed due to rotations). This primitive is discussed in Section 5. It is also illustrated in one of the code samples in Section 8.

[5] The *dot* format for describing graphs can be visualized by the GraphViz system, available at `http://www.graphviz.org/`

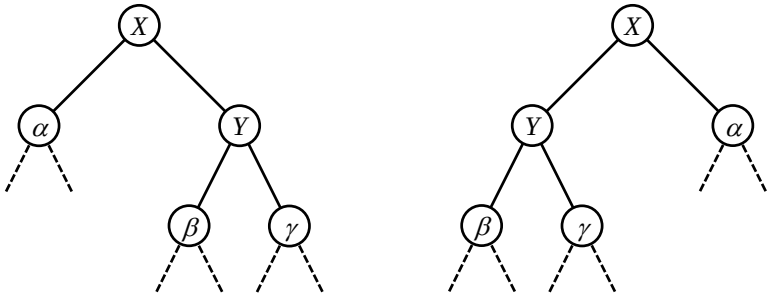
[6] For more on enumerating (unlabeled) trees, see [6]. For more on enumerating permutations, see [5].



(a) Before rotation.

(b) After rotation.

Figure 5: Left-rotating vtree node $v = 3$ in Figure 5(a) leads to the vtree and SDD in Figure 5(b).



(a) Right-linear fragment

(b) Left-linear fragment

Figure 6: Two vtree fragments where X is the root, Y is the child, and where α, β and γ are virtual leaves (which represent the roots of sub-vtrees).

4.4 SDD Minimization as Vtree Search

The SDD library provides a primitive for reducing the size of existing SDDs, which works by searching for a vtree that tries to minimize the SDD size (there is a unique SDD for a given vtree). Similar to garbage collection, SDD minimization can be invoked manually by the user, or automatically by the SDD library.

Automatic garbage collection and SDD minimization are activated or deactivated simultaneously. Manual SDD minimization will also invoke the garbage collector, so the SDD node referencing rules of manual garbage collection apply when invoking SDD minimization manually.

The SDD package is also distributed with source code for compiling CNFs and DNFs into SDDs. This compiler also makes use of SDD minimization, one version using manual minimization and another using automatic minimization. This compiler is discussed in Section 7.

The SDD minimization algorithm is based on *locally* searching the space of vtrees. The search algorithm employs the *vtree fragment* abstraction: a portion of a vtree containing five nodes; see Figure 6(a). Two of these nodes are internal and called the *root* and *child* of the fragment. The other three are treated as leaf nodes of the fragment and called *virtual leaves*. A vtree fragment can be in twelve possible states, depending on the total order of virtual leaves (6 orders) and which of the internal nodes is a root (2 possibilities). One can systematically navigate the twelve states of a fragment using rotation and swap operations; see Figure 7. The SDD library includes operations for navigating vtree fragments, which are employed by the vtree search algorithm (e.g., moving to the next/previous state of a fragment, rolling back to an initial state, etc).

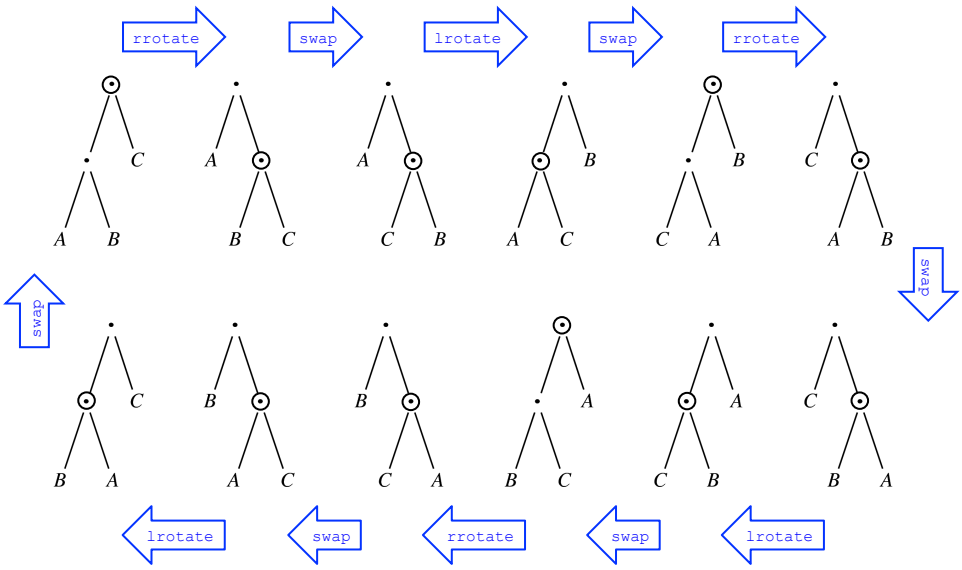


Figure 7: The twelve states of a vtree fragment, which can be enumerated using vtree rotations and swaps (operations apply to circled nodes). Starting from the upper-left vtree, each swap yields a new variable ordering: $ABC \rightarrow ACB \rightarrow CAB \rightarrow CBA \rightarrow BCA \rightarrow BAC \rightarrow ABC$.

The search algorithm performs bottom-up passes on the vtree. For each node n it visits, it identifies two possible fragments that may have n as a root: a left-linear fragment and a right-linear fragment; see Figure 6. The algorithm then tries to explore the twelve states of each fragment. The algorithm repeats its passes on the vtree until the reduction in size falls below a certain threshold, which can be set by the user. Moreover, the search algorithm employs a heuristic for identifying portions of the vtree that are unlikely to change during search, and then skips searching these parts.

4.5 Vtree Search Limits

Fragment operations are based on rotating and swapping vtree nodes, which in turn invoke many conjoin/disjoin operations on SDDs (called *apply* operations). Rotation, swap and apply operations can have a significant impact on the size of the corresponding SDD and may also take a substantial amount of time in some cases [12]. To protect against consuming too much time or memory, the SDD minimization algorithm can enforce *limits* on these operations. If an operation exceeds these limits, the operation fails and rolls back any changes it has made. There are four types of limits, which can all be controlled by the user and are described next.

The first type of limits are on *time* and apply at different levels of granularity during vtree search:

- **vtree**: a limit on the total time spent on vtree search.
- **fragment**: a limit on the total time spent on navigating the states of a vtree fragment.
- **rotate and swap**: a limit on the total time spent by a vtree operation.
- **apply**: a limit on the total time spent by an apply operation.

These limits are based on CPU time.

The second type of limits are on the growth in SDD *size* during a rotate or swap operation. These limits are relative to a reference size that must also be specified and can be updated. Suppose that a relative size limit of 1.2 is specified, along with a reference size s. In this case, the size of the SDD must remain at or

below $1.2 \times s$ during a rotate or swap. Subsequent operations are also limited based on the reference size s, until this size is updated. For example, the vtree search algorithm in the SDD library will update the reference size s when a better vtree is found, i.e, it lowers the size threshold as search proceeds.

The third type of limits are on the growth in SDD *memory* during a rotate or swap operation (that is, the memory needed to store the SDD nodes and edges). This limit is relative to the memory used before an operation is invoked.

The last type of limits concern right rotation and swap operations. These operations compute a cartesian product, which in some pathological cases may be exponentially large [12]. An absolute limit on the size of this cartesian product can be enforced.

Section 5.5 describes the interface for setting these limits, while also indicating their default values.

4.6 X-Constrained Vtrees

Consider a vtree v and let $\mathbf{X}$ be some of its variables. The vtree is called $\mathbf{X}$-constrained iff there exists a node w on the right-most path of vtree v, where $\mathbf{X}$ is precisely the set of variables outside w [8]. SDDs that are constructed with $\mathbf{X}$-constrained vtrees have various applications, including to NP^{PP}-complete and PP^{PP}-complete problems [8, 3]. $\mathbf{X}$-constrained vtrees have also been leveraged for modeling and learning with structured spaces, via Structured Bayesian Networks [9]. Section 5.3.1 describes a function for creating $\mathbf{X}$-constrained vtrees. When an SDD manager is created with an $\mathbf{X}$-constrained vtree, the $\mathbf{X}$-constrained property will be maintained, even during SDD minimization (vtree search).

5 SDD API

This section discusses the API of the SDD library. Section 8 provides a number of code samples, which provide concrete illustrations of using the API.

5.1 Managers

The creation and manipulation of SDDs are maintained by an SDD manager. The use of an SDD manager is analogous to the use of managers in OBDD packages such as CUDD [10]. For a discussion on the design and implementation of such systems, see [7].

To declare an SDD manager, one provides an initial vtree. By declaring an initial vtree, we also declare an initial number of variables. See the section on creating vtrees, for a few ways to specify an initial vtree.

Note that variables are referred to by an index. If there are n variables in an SDD manager, then the variables are referred to as $1, \ldots, n$. Literals are referred to by signed indices. For example, given the i-th variable, we refer to its positive literal by i and its negative literal by $-i$.

5.1.1 Creating Managers

`SddManager* sdd_manager_new(Vtree* vtree);`
Creates a new SDD manager, given a vtree. The manager copies the input vtree. Any manipulations performed by the manager are done on its own copy, and does not affect the input vtree.

`SddManager* sdd_manager_create(SddLiteral var_count, int auto_gc_and_minimize);`
Creates a new SDD manager using a balanced vtree over the given number of variables. Automatic garbage collection and automatic SDD minimization are activated in the new manager, when `auto_gc_and_minimize` is not 0.

`void sdd_manager_free(SddManager* manager);`
Frees the memory of an SDD manager created by `new_sdd_manager`, including all SDD nodes created using that manager.

The following four functions add a new variable with index $n+1$ to the manager, where n is the current number of variables in the manager.

```
void sdd_manager_add_var_before_first(SddManager* manager);
void sdd_manager_add_var_after_last(SddManager* manager);
```
Let v be the leftmost (rightmost) leaf node in the vtree. A new leaf node labeled with variable $n+1$ is created and made a left (right) sibling of leaf v.

```
void sdd_manager_add_var_before(SddLiteral target_var, SddManager* manager);
void sdd_manager_add_var_after(SddLiteral target_var, SddManager* manager);
```
Let v be the vtree leaf node labeled with variable `target_var`. A new leaf node labeled with variable $n+1$ is created and made a left (right) sibling of leaf v.

5.1.2 Terminal SDDs

`SddNode* sdd_manager_true(const SddManager* manager);`
Returns an SDD representing the function true.

`SddNode* sdd_manager_false(const SddManager* manager);`
Returns an SDD representing the function false.

`SddNode* sdd_manager_literal(const SddLiteral literal, SddManager* manager);`
Returns an SDD representing a literal. The variable `literal` is of the form $\pm i$, where i is an index of a variable, which ranges from 1 to the number of variables in the manager. If `literal` is positive, then the SDD representing the positive literal of the i-th variable is returned. If `literal` is negative, then the SDD representing the negative literal is returned.

5.1.3 Automatic Garbage Collection and SDD Minimization

Automatic garbage collection and SDD minimization will take place only when calling functions that construct SDDs. These functions are listed in Section 5.2.1.

```
void sdd_manager_auto_gc_and_minimize_on(SddManager* manager);
void sdd_manager_auto_gc_and_minimize_off(SddManager* manager);
```

Activates and deactivates automatic garbage collection and automatic SDD minimization.

```
int sdd_manager_is_auto_gc_and_minimize_on(SddManager* manager);
```

Returns 1 if automatic garbage collection and automatic SDD minimization is activated, and returns 0 otherwise.

```
void sdd_manager_set_minimize_function(SddVtreeSearchFunc func, SddManager* manager);
```

Instructs the SDD library to use the function `func` instead of the built-in function `sdd_vtree_minimize` during automatic SDD minimization. This allows the user to define their own vtree search algorithm to be used during automatic SDD minimization. The function `func` must behave similarly to `sdd_vtree_minimize`. The type `SddVtreeSearchFunc` is defined as `Vtree* func(Vtree* vtree, SddManager* manager)`.

```
void sdd_manager_unset_minimize_function(SddManager* manager);
```

Reverts back to `sdd_vtree_minimize` as the vtree search algorithm for automatic SDD minimization.

5.1.4 Size and Count

```
SddSize sdd_manager_size(const SddManager* manager);
SddSize sdd_manager_live_size(const SddManager* manager);
SddSize sdd_manager_dead_size(const SddManager* manager);
SddSize sdd_manager_count(const SddManager* manager);
SddSize sdd_manager_live_count(const SddManager* manager);
SddSize sdd_manager_dead_count(const SddManager* manager);
```

Returns the size or node count of all SDD nodes in the manager. For each size and count, three versions are provided: live, dead and total (live+dead).

5.1.5 Misc Functions

```
SddManager* sdd_manager_copy(SddSize size, SddNode** nodes, SddManager* from_manager);
```

Returns a new SDD manager, while copying the vtree and the specified SDD nodes of `from_manager`. The array `nodes` contains the list of SDD nodes to copy, whose length is specified by `size`.

```
void sdd_manager_print(SddManager* manager);
```

Prints various statistics that are collected by an SDD manager.

```
SddLiteral sdd_manager_var_count(SddManager* manager);
```

Returns the number of SDD variables currently associated with the manager.

```
Vtree* sdd_manager_vtree(const SddManager* manager);
```

Returns the root node of a manager's vtree.

```
Vtree* sdd_manager_vtree_copy(const SddManager* manager);
```

Returns a copy of a manager's vtree.

```
Vtree* sdd_manager_vtree_of_var(const SddLiteral var, const SddManager* manager);
```

Returns the leaf node of a manager's vtree, which is associated with `var`.

```
Vtree* sdd_manager_lca_of_literals(int count, SddLiteral* literals, SddManager* manager);
```

Returns the smallest vtree which contains the variables of `literals`, where `count` is the number of literals. If we view the variable of each literal as a leaf vtree node, the function will then return the lowest common ancestor (lca) of these leaf nodes.

```
int sdd_manager_is_var_used(SddLiteral var, SddManager *manager);
```

Returns 1 if `var` is referenced by a decision SDD node (dead or alive); returns 0 otherwise.

```
void sdd_manager_var_order(SddLiteral* var_order, SddManager *manager);
```

Fills the array `var_order` (whose length must equal the number of variables in the manager) with the left-to-right variable ordering of the manager's vtree.

```
void sdd_manager_set_options(void* options, SddManager* manager);
```

Sets the *options* field for an SDD manager. This is a void pointer, which is provided for user convenience. This field can be used to allocate auxiliary data to an SDD manager. The SDD library does not access this field directly (except to initialize it to `NULL`). The source code for the CNF/DNF-to-SDD compiler (see Section 7) provides an example of how this field could be used.

```
void* sdd_manager_options(SddManager* manager);
```

Returns the options field for an SDD manager.

5.2 SDDs

We now describe the functions for constructing and manipulating SDDs. This includes functions for accessing SDD properties, saving SDDs to files, and loading SDDs from files.

5.2.1 Queries and Transformations

```
SddNode* sdd_conjoin(SddNode* node1, SddNode* node2, SddManager* manager);
SddNode* sdd_disjoin(SddNode* node1, SddNode* node2, SddManager* manager);
SddNode* sdd_negate(SddNode* node, SddManager* manager);
```

Returns the result of applying the corresponding Boolean operation on the given SDDs.

```
SddNode* sdd_apply(SddNode* node1, SddNode* node2, BoolOp op, SddManager* manager);
```

Returns the result of combining two SDDs, where `op` can be `CONJOIN` (0) or `DISJOIN` (1).

```
SddNode* sdd_condition(SddLiteral lit, SddNode* node, SddManager* manager);
```

Returns the result of conditioning an SDD on a literal, where a literal is a positive or negative integer.

```
SddNode* sdd_exists(SddLiteral var, SddNode* node, SddManager* manager);
SddNode* sdd_forall(SddLiteral var, SddNode* node, SddManager* manager);
```

Returns the result of existentially (universally) quantifying out a variable from an SDD.

```
SddNode* sdd_exists_multiple(int* exists_map, SddNode* node, SddManager* manager);
```

Returns the result of existentially quantifying out a set of variables from an SDD. This function is expected to be more efficient than existentially quantifying out variables one at a time. The array `exists_map` specifies the variables to be existentially quantified out. The length of `exists_map` is expected to be $n + 1$, where n is the number of variables in the manager. `exists_map[i]` should be 1 if variable i is to be quantified out; otherwise, `exists_map[i]` should be 0. `exists_map[0]` is unused.

```
SddNode* sdd_exists_multiple_static(int* exists_map, SddNode* node, SddManager* manager);
```

This is the same as `sdd_exists_multiple`, except that SDD minimization is never performed when quantifying out variables. This can be more efficient than deactivating automatic SDD minimization and calling `sdd_exists_multiple`.

A *model* of an SDD is a truth assignment of the SDD variables which also satisfies the SDD. The *cardinality* of a truth assignment is the number of variables assigned the value *true*. An SDD may not mention all variables in the SDD manager. An SDD model can be converted into a *global* model by including the missing variables, while setting their values arbitrarily.

```
SddModelCount sdd_model_count(SddNode* node, SddManager* manager);
```
Returns the model count of an SDD (i.e., with respect to the SDD variables).

```
SddModelCount sdd_global_model_count(SddNode* node, SddManager* manager);
```
Returns the global model count of an SDD (i.e., with respect to the manager variables).

```
SddLiteral sdd_minimum_cardinality(SddNode* node);
```
Returns the minimum-cardinality of an SDD: the smallest cardinality attained by any of its models. The minimum-cardinality of an SDD is the same whether or not we consider the global models of the SDD.

```
SddNode* sdd_minimize_cardinality(SddNode* node, SddManager* manager);
```
Returns the SDD whose models are the minimum-cardinality models of the given SDD (i.e. with respect to the SDD variables).

```
SddNode* sdd_global_minimize_cardinality(SddNode* node, SddManager* manager);
```
Returns the SDD whose models are the minimum-cardinality global models of the given SDD (i.e., with respect to the manager variables).

5.2.2 Size and Count

```
SddSize sdd_size(SddNode* node);
SddSize sdd_count(SddNode* node);
```
Returns the size or node count of an SDD (rooted at `node`).

```
SddSize sdd_shared_size(SddNode** nodes, SddSize count);
```
Returns the size of a shared SDD: `nodes` contains the SDD roots and `count` is the number of roots.

5.2.3 File I/O

SDDs can be read from and written to file. When an SDD is saved to file, a description of its file format is printed as a header.

```
void sdd_save(const char* fname, SddNode *node);
```
Saves an SDD to file. Typically, one also saves the corresponding vtree to file. This allows one to read the SDD back using the same vtree.

```
SddNode* sdd_read(const char* filename, SddManager* manager);
```
Reads an SDD from file. The read SDD is guaranteed to be equivalent to the one in the file, but may have a different structure depending on the current vtree of the manager. SDD minimization will not be performed when reading an SDD from file, even if auto SDD minimization is active.

```
void sdd_save_as_dot(const char* fname, SddNode *node);
```
Saves an SDD to file, formatted for use with Graphviz `dot`, to produce graphs like the one in Figure 1(b).

```
void sdd_shared_save_as_dot(const char* fname, SddManager* manager);
```
Saves the SDD of the manager's vtree (a shared SDD), formatted for use with Graphviz `dot`, to produce graphs like the one in Figure 1(b).

5.2.4 Navigation

The following functions are useful for navigating the structure of an SDD (i.e., visiting all its nodes).

```
int sdd_node_is_true(SddNode* node);
int sdd_node_is_false(SddNode* node);
int sdd_node_is_literal(SddNode* node);
int sdd_node_is_decision(SddNode* node);
```
Returns 1 if `node` is of the mentioned type, 0 otherwise.

```
SddLiteral sdd_node_literal(SddNode* node);
```
Returns the signed integer (i.e., variable index or the negation of a variable index) of an SDD node representing a literal. Assumes that `sdd_node_is_literal(node)` returns 1.

```
SddNodeSize sdd_node_size(SddNode* node);
```
Returns the size of an SDD node (the number of its elements). Recall that the size is zero for terminal nodes (i.e., non-decision nodes).

```
SddNode** sdd_node_elements(SddNode* node);
```
Returns an array containing the elements of an SDD node. If the node has m elements, the array will be of size $2m$, with primes appearing at locations $0, 2, \ldots, 2m-2$ and their corresponding subs appearing at locations $1, 3, \ldots, 2m-1$. Assumes that `sdd_node_is_decision(node)` returns 1. Moreover, the returned array should not be freed.

```
void sdd_node_set_bit(int bit, SddNode* node);
```
Sets the bit flag for an SDD node. Bit flags are initialized to 0, and as a general rule, they should be reset to 0 when flags are not being used.

```
int sdd_node_bit(SddNode* node);
```
Returns the bit flag of an SDD node.

5.2.5 Misc Functions

```
Vtree* sdd_vtree_of(SddNode* node);
```
Returns the vtree that an SDD node is normalized for.

Every SDD node has an ID. When an SDD node is garbage collected, its structure is not freed but inserted into a *gc-list*. Moreover, this structure may be reused later by another, newly created SDD node, which will have its own new ID.

```
SddSize sdd_id(SddNode* node);
```
Returns the ID of SDD node. This function may be helpful for debugging.

```
int sdd_garbage_collected(SddNode* node, SddSize id);
```
Returns 1 if the SDD node with the given ID has been garbage collected; returns 0 otherwise. This function may be helpful for debugging.

```
SddNode* sdd_copy(SddNode* node, SddManager* dest_manager);
```
Returns a copy of an SDD, with respect to a new manager `dest_manager`. The destination manager, and the manager associated with the SDD to be copied, must have copies of the same vtree.

```
SddNode* sdd_rename_variables(SddNode* node, SddLiteral* variable_map,
                              SddManager* manager);
```
Returns an SDD which is obtained by renaming variables in the SDD `node`. The array `variable_map` has size $n+1$, where n is the number of variables in the manager. A variable i, $1 \leq i \leq n$, that appears in the given SDD is renamed into variable `variable_map[i]` (`variable_map[0]` is not used).

```
int* sdd_variables(SddNode* node, SddManager* manager);
```
Returns an `array` whose size is $n+1$, where n is the number of variables in the manager. For each variable i, $1 \leq i \leq n$, `array[i]` is 1 if variable i appears in the SDD `node;` otherwise, `array[i]` is 0. The array can be freed when it is no longer needed (`array[0]` is not used).

5.3 Vtrees

We now discuss the functions for constructing, manipulating and navigating vtrees. This includes functions for saving vtrees to files, and loading vtrees from files.

5.3.1 Creating Vtrees

There are a few default vtree structures (linear or balanced), and vtrees can also be read from and written to file (see next section).

```
Vtree* sdd_vtree_new(SddLiteral var_count, const char* type);
```

Returns a vtree over a given number of variables (`var_count`). The type of a vtree may be `"right"` (right linear), `"left"` (left linear), `"vertical"`, `"balanced"`, or `"random"`.

```
Vtree* sdd_vtree_new_with_var_order(SddLiteral var_count, SddLiteral* var_order,
                                    const char* type);
```

Returns a vtree over a given number of variables (`var_count`), whose left-to-right variable ordering is given in array `var_order`. The contents of array `var_order` must be a permutation of the integers from 1 to `var_count`. The type of a vtree may be `"right"` (right linear), `"left"` (left linear), `"vertical"`, `"balanced"`, or `"random"`.

```
Vtree* sdd_vtree_new_X_constrained(SddLiteral var_count, SddLiteral* is_X_var,
                                   const char* type);
```

Returns an $\mathbf{X}$-constrained vtree over a given number of variables (`var_count`). The input `is_X_var` is an array of size `var_count` $+ 1$ specifying variables $\mathbf{X}$. For variables i where $1 \leq i \leq$ `var_count`, if `is_X_var[i]` is 1 then $i \in \mathbf{X}$, and if it is 0 then $i \notin \mathbf{X}$. The type of a vtree may be `"right"` (right linear), `"left"` (left linear), `"vertical"`, `"balanced"`, or `"random"`.

```
void sdd_vtree_free(Vtree* vtree);
```

Frees the memory of a vtree.

5.3.2 Size and Count

```
SddSize sdd_vtree_size(const Vtree* vtree);
SddSize sdd_vtree_live_size(const Vtree* vtree);
SddSize sdd_vtree_dead_size(const Vtree* vtree);
SddSize sdd_vtree_count(const Vtree* vtree);
SddSize sdd_vtree_live_count(const Vtree* vtree);
SddSize sdd_vtree_dead_count(const Vtree* vtree);
```

Returns the size or node count of all SDD nodes in the vtree (see Section 2.5). For each size and count, three versions are provided: live, dead and total (live+dead).

```
SddSize sdd_vtree_size_at(const Vtree* vtree);
SddSize sdd_vtree_live_size_at(const Vtree* vtree);
SddSize sdd_vtree_dead_size_at(const Vtree* vtree);
SddSize sdd_vtree_count_at(const Vtree* vtree);
SddSize sdd_vtree_live_count_at(const Vtree* vtree);
SddSize sdd_vtree_dead_count_at(const Vtree* vtree);
```

Returns the size or node count of all SDD nodes normalized for a vtree node (see Section 2.5). For both size and count, there are three versions: live, dead and total (live+dead). For example, `sdd_vtree_size(vtree)` returns the sum of `sdd_vtree_size_at(v)` where `v` ranges over all nodes of `vtree`.

```
SddSize sdd_vtree_size_above(const Vtree* vtree);
SddSize sdd_vtree_live_size_above(const Vtree* vtree);
SddSize sdd_vtree_dead_size_above(const Vtree* vtree);
SddSize sdd_vtree_count_above(const Vtree* vtree);
SddSize sdd_vtree_live_count_above(const Vtree* vtree);
SddSize sdd_vtree_dead_count_above(const Vtree* vtree);
```

Returns the size or node count of all SDD nodes normalized for a vtree node that is an ancestor of the given vtree node (not including those of the vtree node itself). For each size and count, three versions are provided: live, dead and total (live+dead).

5.3.3 File I/O

Vtrees can be read from and written to file. When a vtree is saved to file, a description of its file format is printed as a header.

```
void sdd_vtree_save(const char* fname, Vtree* vtree);
```

Saves a vtree to file.

```
Vtree* sdd_vtree_read(const char* filename);
```

Reads a vtree from file.

```
void sdd_vtree_save_as_dot(const char* fname, Vtree* vtree);
```

Saves a vtree to file, formatted for use with Graphviz `dot`, to produce graphs like the one in Figure 1(a).

5.3.4 Navigation

```
int sdd_vtree_is_leaf(const Vtree* vtree);
```

Returns 1 if vtree is a leaf node, and 0 otherwise.

```
Vtree* sdd_vtree_left(const Vtree* vtree);
```

Returns the left child of a vtree node (returns `NULL` if the vtree is a leaf node).

```
Vtree* sdd_vtree_right(const Vtree* vtree);
```

Returns the right child of a vtree node (returns `NULL` if the vtree is a leaf node).

```
Vtree* sdd_vtree_parent(const Vtree* vtree);
```

Returns the parent of a vtree node (return `NULL` if the vtree is a root node).

5.3.5 Edit Operations

```
int sdd_vtree_rotate_left(Vtree* vtree, SddManager* manager, int limited);
int sdd_vtree_rotate_right(Vtree* vtree, SddManager* manager, int limited);
int sdd_vtree_swap(Vtree* vtree, SddManager* manager, int limited);
```

Operators for left-rotating a vtree node, right-rotating a vtree node, and swapping the children of a vtree node. Time and size limits, among others, are enforced when `limited` is set to 1; if `limited` is set to 0, limits are deactivated. This operation assumes no dead SDD nodes inside or above `vtree`. Moreover, this operation does not introduce any new dead SDD nodes. Section 5.5 describes the interface for setting limits.

5.3.6 State

```
void sdd_vtree_set_bit(int bit, Vtree* vtree);
```

Sets the bit flag for a vtree node. Bit flags are initialized to 0, and as a general rule, they should be reset to 0 when flags are not being used.

```
int sdd_vtree_bit(const Vtree* vtree);
```

Returns the bit flag of a vtree node.

```
void sdd_vtree_set_data(void* data, Vtree* vtree);
```

Sets the *data field* for a vtree node. This is a void pointer, which is provided for user convenience. This field can be used to allocate auxiliary data to individual vtree nodes. The SDD library does not access this field directly (except to initialize it to `NULL`). The source code for the CNF/DNF-to-SDD compiler (see Section 7) provides an example of how this field could be used.

```
void* sdd_vtree_data(Vtree* vtree);
```

Returns the data field for a vtree node.

```
void sdd_vtree_set_search_state(void* search_state, Vtree* vtree);
```

Sets the *search state* field for a vtree node. This is a void pointer, which is provided for user convenience. This field can be used to allocate additional auxiliary data to individual vtree nodes, which is dedicated to vtree search algorithms developed by the user. The SDD library does not access this field directly (except to initialize it to `NULL`). The source code for the vtree search algorithm (see Section 7) provides an example of how this field could be used.

```
void* sdd_vtree_search_state(const Vtree* vtree);
```

Returns the search state field for a vtree node.

5.3.7 Misc Functions

```
int sdd_vtree_is_sub(const Vtree* vtree1, const Vtree* vtree2);
```

Returns 1 if `vtree1` is a sub-vtree of `vtree2` and 0 otherwise.

```
Vtree* sdd_vtree_lca(Vtree* vtree1, Vtree* vtree2, Vtree* root);
```

Returns the lowest common ancestor (lca) of vtree nodes `vtree1` and `vtree2`, assuming that `root` is a common ancestor of these two nodes.

```
SddLiteral sdd_vtree_var_count(const Vtree* vtree);
```

Returns the number of variables contained in the vtree.

```
SddLiteral sdd_vtree_var(const Vtree* vtree);
```

Returns the variable associated with a vtree node, if the vtree node is a leaf, and returns 0 otherwise.

```
SddLiteral sdd_vtree_position(const Vtree* vtree);
```

Returns the position of a given vtree node in the vtree inorder. Position indices start at 0.

```
Vtree** sdd_vtree_location(Vtree* vtree, SddManager* manager);
```

Returns the location of the pointer to the vtree root. This location can be used to access the new root of the vtree, which may have changed due to rotating some of the vtree nodes.

5.4 Manual Garbage Collection

```
SddRefCount sdd_ref_count(SddNode* node);
```

Returns the reference count of an SDD node. The reference count of a terminal SDD is 0 (terminal SDDs are always live).

```
SddNode* sdd_ref(SddNode* node, SddManager* manager);
```

References an SDD node if it is not a terminal node. Returns the node.

```
SddNode* sdd_deref(SddNode* node, SddManager* manager);
```

Dereferences an SDD node if it is not a terminal node. Returns the node. The number of dereferences to a node cannot be larger than the number of its references (except for terminal SDD nodes, for which referencing and dereferencing has no effect).

```
void sdd_manager_garbage_collect(SddManager* manager);
```

Performs a global garbage collection: Claims all dead SDD nodes in the manager.

```
int sdd_manager_garbage_collect_if(float dead_node_threshold, SddManager* manager);
```

Performs a global garbage collection if the number of dead SDD nodes over the number of all SDD nodes in the manager exceeds `dead_node_threshold`. Returns 1 if garbage collection is performed; 0 otherwise.

```
void sdd_vtree_garbage_collect(Vtree* vtree, SddManager* manager);
```

Performs local garbage collection: Claims all dead SDD nodes inside or above `vtree`.

```
int sdd_vtree_garbage_collect_if(float dead_node_threshold, Vtree* vtree,
                                 SddManager* manager);
```

Performs local garbage collection if the number of dead SDD nodes over the number of all SDD nodes in the vtree exceeds `dead_node_threshold`. Returns 1 if garbage collection is performed; 0 otherwise.

5.5 Manual SDD Minimization

```
Vtree* sdd_vtree_minimize(Vtree* vtree, SddManager* manager);
```

Performs local garbage collection on `vtree` and then tries to minimize the size of the SDD of `vtree` by searching for a different vtree. Returns the root of the resulting vtree.

```
void sdd_manager_minimize(SddManager* manager);
```

Performs global garbage collection and then tries to minimize the size of a manager's SDD. This function calls `sdd_vtree_search` on the manager's vtree.

The following functions can be used to control the behavior of the vtree search algorithm as described in Section 4.5. They apply to both manual and automatic SDD minimization. The default values of limits set by these functions can be modified in the header file `include/parameters.h`.

```
void sdd_manager_set_vtree_search_convergence_threshold(float threshold,
                                                        SddManager* manager);
```

Sets the threshold for terminating the passes performed by the vtree search algorithm. Default value is 1.0, which means that search will terminate when a pass reduces the SDD size by less than 1.0%.

```
void sdd_manager_set_vtree_search_time_limit(float time_limit, SddManager* manager);
void sdd_manager_set_vtree_fragment_time_limit(float time_limit, SddManager* manager);
void sdd_manager_set_vtree_operation_time_limit(float time_limit, SddManager* manager);
void sdd_manager_set_vtree_apply_time_limit(float time_limit, SddManager* manager);
```

Set the time limits for the vtree search algorithm. A vtree operation is either a rotation or a swap. Times are in seconds and correspond to CPU time. Default values, in order, are 180, 60, 30, and 10 seconds.

```
void sdd_manager_set_vtree_operation_memory_limit(float mem_limit, SddManager* manager);
```

Sets the relative memory limit for rotation and swap operations. Default value is 3.0.

```
void sdd_manager_set_vtree_operation_size_limit(float size_limit, SddManager* manager);
```

Sets the relative size limit for rotation and swap operations. Default value is 1.2. A size limit l is relative to the size s of an SDD for a given vtree (s is called the reference size). Hence, using this size limit requires calling the following functions to set and update the reference size s, otherwise the behavior is not defined.

```
void sdd_manager_init_vtree_size_limit(Vtree* vtree, SddManager* manager);
```

Declares the size s of current SDD for `vtree` as the reference size for the relative size limit l. That is, a rotation or swap operation will fail if the SDD size grows to larger than $s \times l$.

```
void sdd_manager_update_vtree_size_limit(SddManager* manager);
```

Updates the reference size for relative size limits. It is preferrable to invoke this function, as it is more efficent than the function `sdd_manager_init_vtree_size_limit` as it does not directly recompute the size of `vtree`.

```
void sdd_manager_set_vtree_cartesian_product_limit(SddSize size_limit,
                                                   SddManager* manager);
```

Sets the absolute size limit on the size of a cartesian product for right rotation and swap operations. Default value is $8,192$.

5.6 Weighted Model Counting

Weighted model counting (WMC) is performed with respect to a given SDD and literal weights, and is based on the following definitions:

- The *weight* of a variable instantiation is the product of weights assigned to its literals.
- The *weighted model count of the SDD* is the sum of weights attained by its models. Here, a model is an instantiation (of *all* variables in the manager) that satisfies the SDD.
- The *weighted model count of a literal* is the sum of weights attained by its models that are also models of the given SDD.
- The *probability of a literal* is the ratio of its weighted model count over the one for the given SDD.

To facilitate the computation of weighted model counts with respect to changing literal weights, a WMC manager is created for the given SDD. This manager stores the weights of literals, allowing one to change them, and to recompute the corresponding weighted model count each time the weights change.

```
WmcManager* wmc_manager_new(SddNode* node, int log_mode, SddManager* manager);
```

Creates a WMC manager for the SDD rooted at `node` and initializes literal weights. When `log_mode` $\neq 0$, all computations done by the manager will be in natural log-space. Literal weights are initialized to 0 in log-mode and to 1 otherwise. A number of functions are given below for passing values to, or recovering values from, a WMC manager. In log-mode, all these values are in natural logs. Finally, a WMC manager may become invalid if garbage collection or SDD minimization takes place.[7]

```
void wmc_manager_free(WmcManager* wmc_manager);
```

[7] To avoid invalidating a WMC manager, the user should refrain from performing the SDD operations of Section 5.2.1 when auto garbage collection and SDD minimization is active.

Frees the memory of the WMC manager.

```
void wmc_set_literal_weight(const SddLiteral literal, const SddWmc weight,
                            WmcManager* wmc_manager);
```

Sets the weight of a literal in the given WMC manager (should pass the natural log of the weight in log-mode).

```
SddWmc wmc_propagate(WmcManager* wmc_manager);
```

Returns the weighted model count of the SDD underlying the WMC manager (using the current literal weights). This function should be called each time the weights of literals are changed.

```
SddWmc wmc_zero_weight(WmcManager* wmc_manager);
```

Returns $-\infty$ for log-mode and 0 otherwise.

```
SddWmc wmc_one_weight(WmcManager* wmc_manager);
```

Returns 0 for log-mode and 1 otherwise.

```
SddWmc wmc_literal_weight(const SddLiteral literal, const WmcManager* wmc_manager);
```

Returns the weight of a literal.

```
SddWmc wmc_literal_derivative(const SddLiteral literal, const WmcManager* wmc_manager);
```

Returns the partial derivative of the weighted model count with respect to the weight of `literal`. The result returned by this function is meaningful only after having called `wmc_propagate`.

```
SddWmc wmc_literal_pr(const SddLiteral literal, const WmcManager* wmc_manager);
```

Returns the probability of `literal`. The result returned by this function is meaningful only after having called `wmc_propagate`.

6 Compiling the SDD Library

The SDD library uses the `SCons` build tool for compiling the library.[8] If SCons is installed on the system, the library can be compiled by running the following on the command line:

```
scons
```

(run this command in the same directory that contains the file `SConstruct`). This command compiles a static library (named `libsdd.a`) and a shared library (named `libsdd.so` on Linux, and named `libsdd.dylib` on Mac). Both will be found under the `build` directory. The following command:

```
scons mode=debug
```

produces libraries with debugging information included, found under the `debug` directory. Adding a `-c` flag to either of the above commands will clean the respective build.

The debug build will enable assertions by default. To disable these assertions, run the command:

```
scons mode=debug --disable-assertions
```

A more expensive but more exhaustive debugging mode can be enabled by running the following command:

```
scons mode=debug --enable-full-debug
```

These options are ignored when compiling the library without the `mode=debug` option.

[8] See `http://scons.org`.

7 FNF-to-SDD Compiler

The SDD Package includes source code for the SDD library and an FNF-to-SDD compiler, which uses the SDD library. FNF stands for Flat Negation Normal Form, which includes CNF and DNF. The FNF-to-SDD compiler supports the compilation of CNFs and DNFs into SDDs.

The SDD Package also includes a pre-compiled binary for each supported architecture, along with some sample CNFs. To compile one of the sample CNFs into an SDD, using the pre-compiled Linux binary, we execute the following on the command line:

```
bin/sdd-linux -c cnf/s208.1.scan.cnf -R output.sdd
```

The `-c` flag is used to specify the filename of the input CNF. The `-R` flag is used to specify the filename of the output SDD. Otherwise, the default command line options correspond to the ones used in the experimental results reported in [2].[9] To see a list of all command line options, run the command:

```
bin/sdd-linux -h
```

There are a few particularly relevant functions, for individuals who are looking to modify the FNF-to-SDD compiler. First, the following functions are available for reading CNFs and DNFs from file.

```
Cnf* read_cnf(const char* filename);
Dnf* read_dnf(const char* filename);
```

The following functions compile an FNF into an SDD:

```
SddNode* fnf_to_sdd_auto(Fnf* fnf, SddManager* manager);
SddNode* fnf_to_sdd_manual(Fnf* fnf, SddManager* manager);
```

The `auto` version uses automatic SDD minimization, whereas `manual` performs SDD minimization manually.

Finally, the file format supported for reading CNF/DNF's is the widely used DIMACS format. This format includes a header "`p cnf n m`" where `n` is the number of CNF variables and `m` is the number of CNF clauses. Each subsequent line represents a clause, consisting of literals i or $-i$, where i is a variable from 1 to `n`, each line terminated by a `0`. Lines starting with `c` are treated as comments, and are ignored. For example, the following CNF file:

```
p cnf 6 3
1 4 0
-1 2 5 0
-1 -2 3 6 0
```

represents the CNF, $(X_1 \vee X_4) \wedge (\neg X_1 \vee X_2 \vee X_5) \wedge (\neg X_1 \vee \neg X_2 \vee X_3 \vee X_6)$. The same file format can be used to represent DNFs, except that we interpret a set of literals as a DNF term (a conjunction of literals). The file above will then specify the DNF, $(X_1 \wedge X_4) \vee (\neg X_1 \wedge X_2 \wedge X_5) \vee (\neg X_1 \wedge \neg X_2 \wedge X_3 \wedge X_6)$.

8 Code Samples

We provide here a few sample programs, and instructions for compiling each program with the SDD library. These code samples are also included with the SDD package.

[9] The choice of initial vtree and the use of dynamic vtree search can have a significant impact on the efficiency of compilation (or even whether an FNF can be successfully compiled or not).

8.1 Compiling

To compile the examples in Sections 8.2–8.6 under Linux, save any of the code samples to a file `test.c`, in a directory which contains (1) a subdirectory `include` containing the header file `sddapi.h`, and (2) a subdirectory `lib` including the library file `libsdd.a`. Compile the code using:

```
gcc -O2 -std=c99 test.c -Iinclude -Llib -lsdd -lm -o test
```

Executing the command `test` will then run the program.

Note that each sample program may output different `.dot` files in the directory they are run in. These files can be converted to graphs using the GraphViz `dot` program. Each sample program is also included with the SDD package, along with a Makefile for compiling them.

8.2 Example 1: Constructing a Simple SDD

```
#include <stdio.h>
#include <stdlib.h>
#include "sddapi.h"

int main(int argc, char** argv) {

  // set up vtree and manager
  SddLiteral var_count = 4;
  SddLiteral var_order[4] = {2,1,4,3};
  const char* type = "balanced";

  Vtree* vtree = sdd_vtree_new_with_var_order(var_count,var_order,type);
  SddManager* manager = sdd_manager_new(vtree);

  // construct a formula (A^B)v(B^C)v(C^D)
  printf("constructing SDD ... ");
  SddNode* f_a = sdd_manager_literal(1,manager);
  SddNode* f_b = sdd_manager_literal(2,manager);
  SddNode* f_c = sdd_manager_literal(3,manager);
  SddNode* f_d = sdd_manager_literal(4,manager);

  SddNode* alpha = sdd_manager_false(manager);
  SddNode* beta;

  beta  = sdd_conjoin(f_a,f_b,manager);
  alpha = sdd_disjoin(alpha,beta,manager);
  beta  = sdd_conjoin(f_b,f_c,manager);
  alpha = sdd_disjoin(alpha,beta,manager);
  beta  = sdd_conjoin(f_c,f_d,manager);
  alpha = sdd_disjoin(alpha,beta,manager);
  printf("done\n");

  printf("saving sdd and vtree ... ");
  sdd_save_as_dot("output/sdd.dot",alpha);
  sdd_vtree_save_as_dot("output/vtree.dot",vtree);
  printf("done\n");

  sdd_vtree_free(vtree);
```

```
  sdd_manager_free(manager);

  return 0;
}
```

8.3 Example 2: Garbage Collecting an SDD

```
#include <stdio.h>
#include <stdlib.h>
#include "sddapi.h"

int main(int argc, char** argv) {

  // set up vtree and manager
  SddLiteral var_count = 4;
  const char* type = "right";
  Vtree* vtree = sdd_vtree_new(var_count,type);
  SddManager* manager = sdd_manager_new(vtree);

  // construct the term X_1 ^ X_2 ^ X_3 ^ X_4
  SddNode* alpha = sdd_manager_literal(1,manager);
  alpha = sdd_conjoin(alpha,sdd_manager_literal(2,manager),manager);
  alpha = sdd_conjoin(alpha,sdd_manager_literal(3,manager),manager);
  alpha = sdd_conjoin(alpha,sdd_manager_literal(4,manager),manager);

  // construct the term ~X_1 ^ X_2 ^ X_3 ^ X_4
  SddNode* beta = sdd_manager_literal(-1,manager);
  beta = sdd_conjoin(beta,sdd_manager_literal(2,manager),manager);
  beta = sdd_conjoin(beta,sdd_manager_literal(3,manager),manager);
  beta = sdd_conjoin(beta,sdd_manager_literal(4,manager),manager);

  // construct the term ~X_1 ^ ~X_2 ^ X_3 ^ X_4
  SddNode* gamma = sdd_manager_literal(-1,manager);
  gamma = sdd_conjoin(gamma,sdd_manager_literal(-2,manager),manager);
  gamma = sdd_conjoin(gamma,sdd_manager_literal(3,manager),manager);
  gamma = sdd_conjoin(gamma,sdd_manager_literal(4,manager),manager);

  printf("== before referencing:\n");
  printf("  live sdd size = %zu\n", sdd_manager_live_size(manager));
  printf("  dead sdd size = %zu\n", sdd_manager_dead_size(manager));

  // ref SDDs so that they are not garbage collected
  sdd_ref(alpha,manager);
  sdd_ref(beta,manager);
  sdd_ref(gamma,manager);
  printf("== after referencing:\n");
  printf("  live sdd size = %zu\n", sdd_manager_live_size(manager));
  printf("  dead sdd size = %zu\n", sdd_manager_dead_size(manager));

  // garbage collect
  sdd_manager_garbage_collect(manager);
  printf("== after garbage collection:\n");
```

```
  printf("  live sdd size = %zu\n", sdd_manager_live_size(manager));
  printf("  dead sdd size = %zu\n", sdd_manager_dead_size(manager));

  sdd_deref(alpha,manager);
  sdd_deref(beta,manager);
  sdd_deref(gamma,manager);

  printf("saving vtree & shared sdd ...\n");
  sdd_vtree_save_as_dot("output/shared-vtree.dot",vtree);
  sdd_shared_save_as_dot("output/shared.dot",manager);

  sdd_vtree_free(vtree);
  sdd_manager_free(manager);

  return 0;
}
```

8.4 Example 3: Minimizing an SDD

```
#include <stdio.h>
#include <stdlib.h>
#include "sddapi.h"

int main(int argc, char** argv) {

  // set up vtree and manager
  Vtree* vtree = sdd_vtree_read("input/opt-swap.vtree");
  SddManager* manager = sdd_manager_new(vtree);

  printf("reading sdd from file ...\n");
  SddNode* alpha = sdd_read("input/opt-swap.sdd",manager);
  printf("  sdd size = %zu\n", sdd_size(alpha));

  // ref, perform the minimization, and then de-ref
  sdd_ref(alpha,manager);
  printf("minimizing sdd size ... ");
  sdd_manager_minimize(manager); // see also sdd_manager_minimize_limited()
  printf("done!\n");
  printf("  sdd size = %zu\n", sdd_size(alpha));
  sdd_deref(alpha,manager);

  // augment the SDD
  printf("augmenting sdd ...\n");
  SddNode* beta = sdd_disjoin(sdd_manager_literal(4,manager),
                              sdd_manager_literal(5,manager),manager);
  beta = sdd_conjoin(alpha,beta,manager);
  printf("  sdd size = %zu\n", sdd_size(beta));

  // ref, perform the minimization again on new SDD, and then de-ref
  sdd_ref(beta,manager);
  printf("minimizing sdd ... ");
  sdd_manager_minimize(manager);
```

```
  printf("done!\n");
  printf("  sdd size = %zu\n", sdd_size(beta));
  sdd_deref(beta,manager);

  sdd_manager_free(manager);
  sdd_vtree_free(vtree);

  return 0;
}
```

8.5 Example 4: Rotating a Vtree Node

```
#include <stdio.h>
#include <stdlib.h>
#include "sddapi.h"

int main(int argc, char** argv) {

  // set up vtree and manager
  Vtree* vtree = sdd_vtree_read("input/rotate-left.vtree");
  SddManager* manager = sdd_manager_new(vtree);

  // construct the term X_1 ^ X_2 ^ X_3 ^ X_4
  SddNode* alpha = sdd_manager_literal(1,manager);
  alpha = sdd_conjoin(alpha,sdd_manager_literal(2,manager),manager);
  alpha = sdd_conjoin(alpha,sdd_manager_literal(3,manager),manager);
  alpha = sdd_conjoin(alpha,sdd_manager_literal(4,manager),manager);

  // to perform a rotate, we need the manager's vtree
  Vtree* manager_vtree = sdd_manager_vtree(manager);
  Vtree* manager_vtree_right = sdd_vtree_right(manager_vtree);

  // obtain the manager's pointer to the root
  Vtree** root_location = sdd_vtree_location(manager_vtree,manager);

  printf("saving vtree & sdd ...\n");
  sdd_vtree_save_as_dot("output/before-rotate-vtree.dot",manager_vtree);
  sdd_save_as_dot("output/before-rotate-sdd.dot",alpha);

  // ref alpha (so it is not gc'd)
  sdd_ref(alpha,manager);

  // garbage collect (no dead nodes when performing vtree operations)
  printf("dead sdd nodes = %zu\n", sdd_manager_dead_count(manager));
  printf("garbage collection ...\n");
  sdd_manager_garbage_collect(manager);
  printf("dead sdd nodes = %zu\n", sdd_manager_dead_count(manager));

  printf("left rotating ... ");
  int succeeded = sdd_vtree_rotate_left(manager_vtree_right,manager,0); //not limited
  printf("%s!\n", succeeded?"succeeded":"did not succeed");
```

```
  // deref alpha, since ref's are no longer needed
  sdd_deref(alpha,manager);

  // the root changed after rotation, so get the manager's vtree again
  // this time using root_location
  manager_vtree = *root_location;
  //assert(manager_vtree==sdd_manager_vtree(manager));

  printf("saving vtree & sdd ...\n");
  sdd_vtree_save_as_dot("output/after-rotate-vtree.dot",manager_vtree);
  sdd_save_as_dot("output/after-rotate-sdd.dot",alpha);

  sdd_vtree_free(vtree);
  sdd_manager_free(manager);

  return 0;
}
```

8.6 Example 5: Swapping a Vtree Node, with Limits

```
#include <stdio.h>
#include <stdlib.h>
#include "sddapi.h"

int main(int argc, char** argv) {

  // set up vtree and manager
  Vtree* vtree = sdd_vtree_read("input/big-swap.vtree");
  SddManager* manager = sdd_manager_new(vtree);

  printf("reading sdd from file ...\n");
  SddNode* alpha = sdd_read("input/big-swap.sdd",manager);
  printf("  sdd size = %zu\n", sdd_size(alpha));

  // to perform a swap, we need the manager's vtree
  Vtree* manager_vtree = sdd_manager_vtree(manager);
  double limit;

  // ref alpha (no dead nodes when swapping)
  sdd_ref(alpha,manager);

  //using size of sdd normalized for manager_vtree as baseline for limit
  sdd_manager_init_vtree_size_limit(manager_vtree,manager);

  limit = 2.0;
  sdd_manager_set_vtree_operation_size_limit(limit,manager);

  printf("modifying vtree (swap node 7) (limit growth by %.1fx) ... ",limit);
  int succeeded;
  succeeded = sdd_vtree_swap(manager_vtree,manager,1); //limited
  printf("%s!\n", succeeded?"succeeded":"did not succeed");
  printf("  sdd size = %zu\n", sdd_size(alpha));
```

```
  printf("modifying vtree (swap node 7) (no limit) ... ");
  succeeded = sdd_vtree_swap(manager_vtree,manager,0); //not limited
  printf("%s!\n", succeeded?"succeeded":"did not succeed");
  printf("  sdd size = %zu\n", sdd_size(alpha));

  printf("updating baseline of size limit ...\n");
  sdd_manager_update_vtree_size_limit(manager);

  Vtree* left_vtree = sdd_vtree_left(manager_vtree);
  limit = 1.2;
  sdd_manager_set_vtree_operation_size_limit(limit,manager);
  printf("modifying vtree (swap node 5) (limit growth by %.1fx) ... ",limit);
  succeeded = sdd_vtree_swap(left_vtree,manager,1); //limited
  printf("%s!\n", succeeded?"succeeded":"did not succeed");
  printf("  sdd size = %zu\n", sdd_size(alpha));

  limit = 1.3;
  sdd_manager_set_vtree_operation_size_limit(limit,manager);
  printf("modifying vtree (swap node 5) (limit growth by %.1fx) ... ",limit);
  succeeded = sdd_vtree_swap(left_vtree,manager,1); //limited
  printf("%s!\n", succeeded?"succeeded":"did not succeed");
  printf("  sdd size = %zu\n", sdd_size(alpha));

  // deref alpha, since ref's are no longer needed
  sdd_deref(alpha,manager);

  sdd_vtree_free(vtree);
  sdd_manager_free(manager);

  return 0;
}
```


Acknowledgements

Special thanks to the members of the Automated Reasoning Group at UCLA, and the Statistical and Relational Artificial Intelligence (StarAI) lab at UCLA. In particular, thanks to Guy Van den Broeck for many helpful suggestions. We also thank members of the DTAI lab in KU Leuven, and in particular: Jessa Bekker, Anton Dries, Wannes Meert, Joris Renkens, and Jonas Vlasselaer.

References

[1] Simone Bova. SDDs are exponentially more succinct than OBDDs. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI)*, pages 929–935, 2016.

[2] Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*, pages 187–194, 2013.

[3] YooJung Choi, Adnan Darwiche, and Guy Van den Broeck. Optimal feature selection for decision robustness in Bayesian networks. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1554–1560, 2017.

[4] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 819–826, 2011.

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

[6] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation*. Addison-Wesley, 2006.

[7] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design: OBDD Foundations and Applications*. Springer, 1998.

[8] Umut Oztok, Arthur Choi, and Adnan Darwiche. Solving pp$^{\text{pp}}$-complete problems using knowledge compilation. In *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 94–103, 2016.

[9] Yujia Shen, Arthur Choi, and Adnan Darwiche. Conditional PSDDs: Modeling and learning with modular knowledge. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, 2018. To appear.

[10] Fabio Somenzi. CUDD: CU Decision Diagram Package. University of Colorado at Boulder, 2012.

[11] Guy Van den Broeck and Adnan Darwiche. On the role of canonicity in knowledge compilation. In *Proceedings of the 29th Conference on Artificial Intelligence (AAAI)*, pages 1641–1648, 2015.

[12] Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing decisions on sentences in decision diagrams. In *Proceedings of the 26th Conference on Artificial Intelligence (AAAI)*, pages 842–849, 2012.

A License

Copyright 2013-2018, Regents of the University of California

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

B SDD API

The `libsdd` library provides the following functions, which are described in Section 5. This section is intended more as a reference.

Managers

Creating Managers

```
SddManager* sdd_manager_new(Vtree* vtree);
SddManager* sdd_manager_create(SddLiteral var_count, int auto_gc_and_minimize);
void sdd_manager_free(SddManager* manager);
void sdd_manager_add_var_before_first(SddManager* manager);
void sdd_manager_add_var_after_last(SddManager* manager);
void sdd_manager_add_var_before(SddLiteral target_var, SddManager* manager);
void sdd_manager_add_var_after(SddLiteral target_var, SddManager* manager);
```

Terminal SDDs

```
SddNode* sdd_manager_true(const SddManager* manager);
SddNode* sdd_manager_false(const SddManager* manager);
SddNode* sdd_manager_literal(const SddLiteral literal, SddManager* manager);
```

Automatic Garbage Collection and SDD Minimization

```
void sdd_manager_auto_gc_and_minimize_on(SddManager* manager);
void sdd_manager_auto_gc_and_minimize_off(SddManager* manager);
int sdd_manager_is_auto_gc_and_minimize_on(SddManager* manager);
void sdd_manager_set_minimize_function(SddVtreeSearchFunc func, SddManager* manager);
void sdd_manager_unset_minimize_function(SddManager* manager);
```

Size and Count

```
SddSize sdd_manager_size(const SddManager* manager);
SddSize sdd_manager_live_size(const SddManager* manager);
SddSize sdd_manager_dead_size(const SddManager* manager);
SddSize sdd_manager_count(const SddManager* manager);
SddSize sdd_manager_live_count(const SddManager* manager);
SddSize sdd_manager_dead_count(const SddManager* manager);
```

Misc Functions

```
SddManager* sdd_manager_copy(SddSize size, SddNode** nodes, SddManager* from_manager);
void sdd_manager_print(SddManager* manager);
SddLiteral sdd_manager_var_count(SddManager* manager);
Vtree* sdd_manager_vtree(const SddManager* manager);
Vtree* sdd_manager_vtree_copy(const SddManager* manager);
Vtree* sdd_manager_vtree_of_var(const SddLiteral var, const SddManager* manager);
Vtree* sdd_manager_lca_of_literals(int count, SddLiteral* literals, SddManager* manager);
int sdd_manager_is_var_used(SddLiteral var, SddManager *manager);
void sdd_manager_var_order(SddLiteral* var_order, SddManager *manager);
void sdd_manager_set_options(void* options, SddManager* manager);
void* sdd_manager_options(SddManager* manager);
```

SDDs

Queries and Transformations

```
SddNode* sdd_conjoin(SddNode* node1, SddNode* node2, SddManager* manager);
SddNode* sdd_disjoin(SddNode* node1, SddNode* node2, SddManager* manager);
SddNode* sdd_negate(SddNode* node, SddManager* manager);
SddNode* sdd_condition(SddLiteral lit, SddNode* node, SddManager* manager);
SddNode* sdd_exists(SddLiteral var, SddNode* node, SddManager* manager);
SddNode* sdd_forall(SddLiteral var, SddNode* node, SddManager* manager);
SddNode* sdd_exists_multiple(int* exists_map, SddNode* node, SddManager* manager);
SddNode* sdd_exists_multiple_static(int* exists_map, SddNode* node, SddManager* manager);
SddModelCount sdd_model_count(SddNode* node, SddManager* manager);
SddModelCount sdd_global_model_count(SddNode* node, SddManager* manager);
SddLiteral sdd_minimum_cardinality(SddNode* node);
SddNode* sdd_minimize_cardinality(SddNode* node, SddManager* manager);
SddNode* sdd_global_minimize_cardinality(SddNode* node, SddManager* manager);
SddNode* sdd_apply(SddNode* node1, SddNode* node2, BoolOp op, SddManager* manager);
```

Size and Count

```
SddSize sdd_size(SddNode* node);
SddSize sdd_count(SddNode* node);
SddSize sdd_shared_size(SddNode** nodes, SddSize count);
```

File I/O

```
void sdd_save(const char* fname, SddNode *node);
SddNode* sdd_read(const char* filename, SddManager* manager);
void sdd_save_as_dot(const char* fname, SddNode *node);
void sdd_shared_save_as_dot(const char* fname, SddManager* manager);
```

Navigation

```
int sdd_node_is_true(SddNode* node);
int sdd_node_is_false(SddNode* node);
int sdd_node_is_literal(SddNode* node);
int sdd_node_is_decision(SddNode* node);
SddLiteral sdd_node_literal(SddNode* node);
SddNodeSize sdd_node_size(SddNode* node);
SddNode** sdd_node_elements(SddNode* node);
void sdd_node_set_bit(int bit, SddNode* node);
int sdd_node_bit(SddNode* node);
```

Misc Functions

```
Vtree* sdd_vtree_of(SddNode* node);
SddSize sdd_id(SddNode* node);
int sdd_garbage_collected(SddNode* node, SddSize id);
SddNode* sdd_copy(SddNode* node, SddManager* dest_manager);
SddNode* sdd_rename_variables(SddNode* node, SddLiteral* variable_map,
                              SddManager* manager);
int* sdd_variables(SddNode* node, SddManager* manager);
```

Vtrees

Creating Vtrees

```
Vtree* sdd_vtree_new(SddLiteral var_count, const char* type);
Vtree* sdd_vtree_new_with_var_order(SddLiteral var_count, SddLiteral* var_order,
                                    const char* type);
Vtree* sdd_vtree_new_X_constrained(SddLiteral var_count, SddLiteral* is_X_var,
                                   const char* type);
void sdd_vtree_free(Vtree* vtree);
```

Size and Count

```
SddSize sdd_vtree_size(const Vtree* vtree);
SddSize sdd_vtree_live_size(const Vtree* vtree);
SddSize sdd_vtree_dead_size(const Vtree* vtree);
SddSize sdd_vtree_count(const Vtree* vtree);
SddSize sdd_vtree_live_count(const Vtree* vtree);
SddSize sdd_vtree_dead_count(const Vtree* vtree);
SddSize sdd_vtree_size_at(const Vtree* vtree);
SddSize sdd_vtree_live_size_at(const Vtree* vtree);
SddSize sdd_vtree_dead_size_at(const Vtree* vtree);
SddSize sdd_vtree_count_at(const Vtree* vtree);
SddSize sdd_vtree_live_count_at(const Vtree* vtree);
SddSize sdd_vtree_dead_count_at(const Vtree* vtree);
SddSize sdd_vtree_size_above(const Vtree* vtree);
SddSize sdd_vtree_live_size_above(const Vtree* vtree);
SddSize sdd_vtree_dead_size_above(const Vtree* vtree);
SddSize sdd_vtree_count_above(const Vtree* vtree);
SddSize sdd_vtree_live_count_above(const Vtree* vtree);
SddSize sdd_vtree_dead_count_above(const Vtree* vtree);
```

File I/O

```
void sdd_vtree_save(const char* fname, Vtree* vtree);
Vtree* sdd_vtree_read(const char* filename);
void sdd_vtree_save_as_dot(const char* fname, Vtree* vtree);
```

Navigation

```
int sdd_vtree_is_leaf(const Vtree* vtree);
Vtree* sdd_vtree_left(const Vtree* vtree);
Vtree* sdd_vtree_right(const Vtree* vtree);
Vtree* sdd_vtree_parent(const Vtree* vtree);
```

Edit Operations

```
int sdd_vtree_rotate_left(Vtree* vtree, SddManager* manager, int limited);
int sdd_vtree_rotate_right(Vtree* vtree, SddManager* manager, int limited);
int sdd_vtree_swap(Vtree* vtree, SddManager* manager, int limited);
void sdd_manager_init_vtree_size_limit(Vtree* vtree, SddManager* manager);
void sdd_manager_update_vtree_size_limit(SddManager* manager);
```

State

```
int sdd_vtree_bit(const Vtree* vtree);
void sdd_vtree_set_bit(int bit, Vtree* vtree);
void* sdd_vtree_data(Vtree* vtree);
void sdd_vtree_set_data(void* data, Vtree* vtree);
void* sdd_vtree_search_state(const Vtree* vtree);
void sdd_vtree_set_search_state(void* search_state, Vtree* vtree);
```

Misc Functions

```
int sdd_vtree_is_sub(const Vtree* vtree1, const Vtree* vtree2);
Vtree* sdd_vtree_lca(Vtree* vtree1, Vtree* vtree2, Vtree* root);
SddLiteral sdd_vtree_var_count(const Vtree* vtree);
SddLiteral sdd_vtree_var(const Vtree* vtree);
SddLiteral sdd_vtree_position(const Vtree* vtree);
Vtree** sdd_vtree_location(Vtree* vtree, SddManager* manager);
```

Manual Garbage Collection

```
SddRefCount sdd_ref_count(SddNode* node);
SddNode* sdd_ref(SddNode* node, SddManager* manager);
SddNode* sdd_deref(SddNode* node, SddManager* manager);
void sdd_manager_garbage_collect(SddManager* manager);
int sdd_manager_garbage_collect_if(float dead_node_threshold, SddManager* manager);
void sdd_vtree_garbage_collect(Vtree* vtree, SddManager* manager);
int sdd_vtree_garbage_collect_if(float dead_node_threshold, Vtree* vtree,
                                 SddManager* manager);
```

Manual SDD Minimization

```
Vtree* sdd_vtree_minimize(Vtree* vtree, SddManager* manager);
void sdd_manager_minimize(SddManager* manager);
void sdd_manager_set_vtree_search_convergence_threshold(float threshold,
                                                        SddManager* manager);
void sdd_manager_set_vtree_search_time_limit(float time_limit, SddManager* manager);
void sdd_manager_set_vtree_fragment_time_limit(float time_limit, SddManager* manager);
void sdd_manager_set_vtree_operation_time_limit(float time_limit, SddManager* manager);
void sdd_manager_set_vtree_apply_time_limit(float time_limit, SddManager* manager);
void sdd_manager_set_vtree_operation_memory_limit(float memory_limit,
                                                  SddManager* manager);
void sdd_manager_set_vtree_operation_size_limit(float size_limit, SddManager* manager);
void sdd_manager_set_vtree_cartesian_product_limit(SddSize size_limit,
                                                   SddManager* manager);
```

Weighted Model Counting

```
WmcManager* wmc_manager_new(SddNode* node, int log_mode, SddManager* manager);
void wmc_manager_free(WmcManager* wmc_manager);
void wmc_set_literal_weight(const SddLiteral literal, const SddWmc weight,
                            WmcManager* wmc_manager);
SddWmc wmc_propagate(WmcManager* wmc_manager);
SddWmc wmc_zero_weight(WmcManager* wmc_manager);
SddWmc wmc_one_weight(WmcManager* wmc_manager);
```

```
SddWmc wmc_literal_weight(const SddLiteral literal, const WmcManager* wmc_manager);
SddWmc wmc_literal_derivative(const SddLiteral literal, const WmcManager* wmc_manager);
SddWmc wmc_literal_pr(const SddLiteral literal, const WmcManager* wmc_manager);
```