SDD Beginning-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 open-source SDD package allows users to construct, manipulate and optimize SDDs. This manual describes the basics of the SDD package, meant for users who are interested in immediately constructing SDDs and performing queries on them. An advanced user's manual is also provided for those users interested in the much broader set of features supported by the library. The license terms for the SDD package are given at the end of this manual.

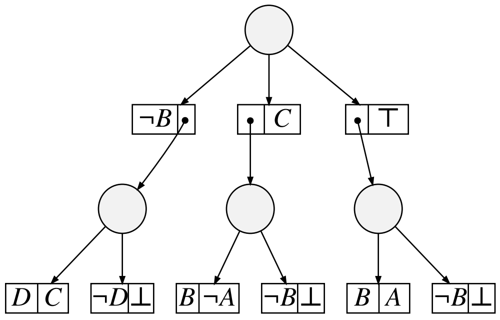


Figure 1: A graphical depiction of an SDD, for the Boolean function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$.

1 Introduction

The Sentential Decision Diagram (SDD) is a recently proposed representation of Boolean functions [1, 2, 3]. 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 depicts an example SDD and the Boolean function it represents.

This manual provides a brief introduction to the SDD library, using a small set of library functions, which are sufficient to do the following:

- Construct complex SDDs from elementary SDDs, corresponding to constants or literals.
- Manipulate and query SDDs.
- Read/write SDDs from/to a file.
- Visualize SDDs, as in Figure 1.

An advanced manual is available for those users who want a richer interface, which provides more fine-grained access to the SDD library's features. Related lectures and videos may also be found on the YouTube channel `https://www.youtube.com/channel/UCA05XHgEZVeVlkLHvpz48VQ`.

2 SDDs

Consider again Figure 1, which contains a graphical depiction of a Sentential Decision Diagram (SDD), along with the Boolean function that the SDD represents. There are two types of nodes here: circle nodes $\bigcirc$ are called *decision* nodes, and paired boxes $\boxed{p \mid s}$ are called *elements*. A decision node represents a disjunction of its child elements, and a paired box $\boxed{p \mid s}$ represents a conjunction $p \wedge s$, where a box can contain either a pointer to a decision node, a constant ($\top$ for true, and $\bot$ for false), or a literal (X or $\neg X$).

Each node and paired box then corresponds to some sub-function of the SDD, with the root node representing the function itself. For example, in the bottom row of paired boxes, from left-to-right, we have the corresponding functions: $D \wedge C$, $\neg D \wedge \bot = \bot$, $B \wedge \neg A$, etc. In the lower row of circular nodes, from left-to-right, we have the corresponding functions:

$$\begin{aligned} (D \wedge C) \vee (\neg D \wedge \bot) &= C \wedge D \\ (B \wedge \neg A) \vee (\neg B \wedge \bot) &= \neg A \wedge B \\ (B \wedge A) \vee (\neg B \wedge \bot) &= A \wedge B. \end{aligned}$$

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

The root node of the SDD represents the Boolean function:

$$\begin{aligned} f &= [\neg B \wedge (C \wedge D)] \vee [(\neg A \wedge B) \wedge C] \vee [(A \wedge B) \wedge \top] \\ &= (A \wedge B) \vee (B \wedge C) \vee (C \wedge D). \end{aligned}$$

SDDs are structured representations of Boolean functions, which allow certain operations to be performed efficiently (in practice, if not also in theory). For example, one can conjoin and disjoin two SDDs efficiently. One can also, for example, perform model counting in time that is linear in the size of the SDD. For more on SDDs, including theoretical and practical comparisons with OBDDs, see [1, 2, 3, 5].

3 Preliminaries

In the next section, we shall highlight the core components of the SDD library, using a running example based on a diagnostic application. The source code for this example is included in the SDD package, and also in Appendix A. We will start though by showing how to compile a program with the SDD library.

3.1 The Header File

The functions and types used in the SDD library are declared in the header file `sddapi.h`. This manual, however, is based on a small subset of the functions declared in this file. We call this the core API and summarize it in Appendix C.

3.2 Compiling

To compile the examples in this manual, under Linux, one can save a code sample (say from Section A) to a file `circuit.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 the command:

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

Executing the command `circuit` will then run the program. The sample program outputs `.dot` files in an `output` sub-directory (which must also exist in the current directory). These files can be converted to graphs using the GraphViz `dot` program.

4 Diagnosing Failures in Digital Circuits

Consider Figure 2, which depicts a simple digital circuit, consisting of two inverters, labeled 1 and 2, and three wires, labeled A, B and C.

We can specify a knowledge base modeling this digital circuit, using propositional logic:

$$\Delta = \left\{ \begin{array}{rcl} \neg\text{faulty}_1 & \Rightarrow & A \Leftrightarrow \neg B \\ \text{faulty}_1 & \Rightarrow & (A \Leftrightarrow B) \vee \neg B \\ \\ \neg\text{faulty}_2 & \Rightarrow & B \Leftrightarrow \neg C \\ \text{faulty}_2 & \Rightarrow & (B \Leftrightarrow C) \vee \neg C \end{array} \right.$$

The knowledge base Δ is composed of four sentences, two sentences for each inverter. The first sentence specifies the normal operating behavior of an inverter: the output of an inverter is the complement of its input. The second sentence specifies the behavior of an inverter when it is faulty: either it behaves as a buffer (the output value is the same as the input), i.e., $A \Leftrightarrow B$, or it behaves as if it is fixed to low, i.e., $\neg B$.

Variables faulty_1 and faulty_2 are called *health variables*, and a term over all health variables is called a *health state*. For example, $\text{faulty}_1 \wedge \neg\text{faulty}_2$ is a health state where inverter 1 is faulty and inverter 2 is healthy. Given an (abnormal) observation about the system, e.g., $A \wedge \neg C$, we would like to reason about the health states that are consistent with that observation. Any such health state is further called a *diagnosis*.

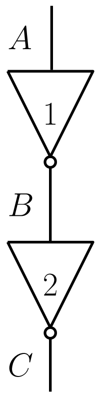


Figure 2: A simple circuit consisting of two inverters, labeled 1 and 2, and three wires, labeled A, B and C.

5 The SDD Library

In this section, we will highlight the core components of the SDD library, using a running example for the diagnostic application we just discussed. In particular, we will show how to construct an SDD that represents the knowledge base Δ. We will then show how to manipulate this SDD to obtain answers to diagnostic queries.

5.1 The SDD Manager

The construction and manipulation of SDDs is done via an *SDD manager.* Hence, a typical program using the SDD library will have the following structure, where the first thing to do is *create* an SDD manager, and the last thing to do is *free* the manager:

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

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

  // initialize manager
  SddLiteral var_count = 5;     // initial number of variables
  int auto_gc_and_minimize = 0; // disable (0) or enable (1) auto-gc & auto-min
  SddManager* manager = sdd_manager_create(var_count,auto_gc_and_minimize);

  // CONSTRUCT, MANIPULATE AND QUERY SDDS
  // ...

  // free manager
  sdd_manager_free(manager);

  return 0;
}
```

An SDD manager keeps track of all its associated SDDs. Hence, by freeing the SDD manager, one also frees the memory of all SDDs created using that manager.

When we create an SDD manager, we declare an (initial) number of variables, `var_count`. We also disable or enable automatic garbage collection and SDD minimization. In our example, we have created a new SDD manager, declaring that we want to have 5 variables. We have further disabled automatic garbage collection and SDD minimization.

Some degree of memory management is integral for the construction of SDDs in practical applications. In the SDD library, memory management is based on garbage collection with reference counting. We will, however, delay this discussion until Section 6 to simplify our introduction to the SDD library. This is why we have disabled automatic garbage collection and SDD minimization for now.

5.2 Variables, Literals and Elementary SDDs

In the SDD library, variables are referred to by indices. Since our manager has five variables, their indices will be 1, 2, 3, 4 and 5. Literals are referred to by *signed* indices. For example, the third variable has a positive literal 3 and a negative literal −3.

The variables of our diagnosis example are A, B, C, faulty_1 and faulty_2. In the code snippet below, we associate each one of these variables with an index from 1 to 5:

```
  SddLiteral A = 1, B = 2, C = 3, faulty1 = 4, faulty2 = 5;
```

Here, `SddLiteral` is an integer type that is defined by the SDD library, in the header file `sddapi.h`. This type is used to specify literals: variables or their negation.

The SDD manager maintains a unique SDD for each constant and literal. To access these elementary SDDs, we have the three functions:

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

We next illustrate how we can use the SDDs returned by these functions to create more complex SDDs.

5.3 Constructing SDDs from Elementary SDDs

Our goal here is to construct an SDD representing the following knowledge base:

$$\Delta = \begin{cases} \neg\text{faulty}_1 & \Rightarrow & A \Leftrightarrow \neg B \\ \text{faulty}_1 & \Rightarrow & (A \Leftrightarrow B) \vee \neg B \\ \neg\text{faulty}_2 & \Rightarrow & B \Leftrightarrow \neg C \\ \text{faulty}_2 & \Rightarrow & (B \Leftrightarrow C) \vee \neg C \end{cases}$$

We can construct each of the four sentences from elementary SDDs, one at a time. We can then compose these four sentences together, giving us an SDD for the knowledge base Δ.

We start with a vacuous knowledge base, represented by a true SDD:

```
  SddNode* delta = sdd_manager_true(manager);
```

Using the following functions, we can conjoin two SDDs, disjoin two SDDs, or negate an SDD:

```
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);
```

Using these functions as primitives, we can create our own functions that perform other operations on SDDs. For example, given an SDD representing a function α and an SDD representing a function β, we can construct an SDD representing the sentence $\alpha \Rightarrow \beta$, which is logically equivalent to $\neg\alpha \vee \beta$:

```
SddNode* sdd_imply(SddNode* node1, SddNode* node2, SddManager* manager) {
  return sdd_disjoin(sdd_negate(node1,manager),node2,manager);
}
```

Using the above function as a primitive, we can construct an SDD representing the sentence $\alpha \Leftrightarrow \beta$, which is logically equivalent to $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$:

```
SddNode* sdd_equiv(SddNode* node1, SddNode* node2, SddManager* manager) {
  return sdd_conjoin(sdd_imply(node1,node2,manager),
                     sdd_imply(node2,node1,manager),manager);
}
```

We are now prepared to construct an SDD for our knowledge base Δ.

Let us first construct an SDD for the sentence: $\neg\text{faulty}_1 \Rightarrow (A \Leftrightarrow \neg B)$. We can first construct the SDD for the sub-sentence $A \Leftrightarrow \neg B$, using the SDDs for literals A and $\neg B$. We then take the resulting SDD, and combine it with the SDD for literal $\neg\text{faulty}_1$, as we do in the following code snippet:

```
  SddNode* alpha;
  alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(-B,m),m);
  alpha = sdd_imply(sdd_manager_literal(-faulty1,m),alpha,m);
```

where we have used the abbreviated name `m` for the SDD manager. We then conjoin this sentence with the SDD for our knowledge base (which we initialized to `true`):

```
  delta = sdd_conjoin(delta,alpha,m);
```

We can construct SDDs for each sentence similarly, conjoining each with our knowledge base. For example, the following code snippet constructs an SDD for the sentence $\text{faulty}_1 \Rightarrow [(A \Leftrightarrow B) \vee \neg B]$, and conjoins it with the SDD for our knowledge base:

```
  alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(B,m),m);
  alpha = sdd_disjoin(alpha,sdd_manager_literal(-B,m),m);
  alpha = sdd_imply(sdd_manager_literal(faulty1,m),alpha,m);
  delta = sdd_conjoin(delta,alpha,m);
```

Once we have incorporated all sentences, we can start performing queries on our model, using the final SDD that we constructed.

5.4 Querying SDDs

Given an SDD representation of our knowledge base Δ, we can begin to assert observations in our model, and make inferences about them. In our diagnosis example, we would like to observe some system behavior, and then diagnose the possible faults.

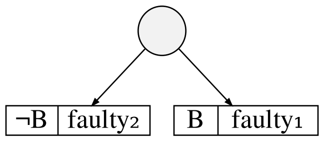
First, we can condition an SDD on a literal, using the following function:

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

For example, we can assert the observation that input A is high, and output C is low:

```
  delta = sdd_condition(A,delta,m);
  delta = sdd_condition(-C,delta,m);
```

Conditioning a knowledge base Δ on a literal X, denoted by $\Delta|X$, replaces each instance of literal X in Δ by true, and each instance of literal $\neg X$ in Δ by false. Analogously, conditioning Δ on a literal $\neg X$, denoted by $\Delta|\neg X$, replaces X by false and $\neg X$ by true. In our example, the SDD for $(\Delta|A)|\neg C$ is then:



First, we want to test whether our observation $A \wedge \neg C$ is *normal* (all components can be healthy) or *abnormal* (there exists a fault). In particular, we want to test if the knowledge base Δ and the evidence $A \wedge \neg C$ are consistent with normal system behavior, i.e., there are no faults: $\neg\text{faulty}_1 \wedge \neg\text{faulty}_2$.

```
  SddNode* gamma;
  gamma = sdd_condition(-faulty1,delta,m);
  gamma = sdd_condition(-faulty2,gamma,m);
  int is_abnormal = sdd_node_is_false(gamma);
```

If the resulting SDD `gamma` is false,[2] then our observations are abnormal; otherwise, they are normal. We can test whether a given SDD is false or true, using the following functions:

```
int sdd_node_is_false(SddNode* node);
int sdd_node_is_true(SddNode* node);
```

These functions return 1 if the given SDD node corresponds to the one being queried, and 0 otherwise.

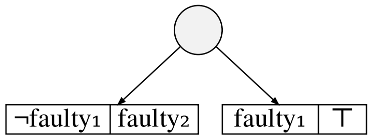
Next, to compute our diagnoses, we project our SDD onto the health variables faulty_1 and faulty_2. In particular, we existentially quantify out all non-health variables, which is just variable B in this case:

```
  SddNode* diagnosis = sdd_exists(B,delta,m);
```

Given a knowledge base Δ and a variable X, existential quantification is more formally defined as $\exists X\Delta = \Delta|X \vee \Delta|\neg X$. Similarly, universal quantification is defined as $\forall X\Delta = \Delta|X \wedge \Delta|\neg X$. The corresponding functions are used to existentially and universally quantify out a variable from an SDD:

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

In our diagnosis example, we obtain the following SDD, representing our diagnoses:



This SDD corresponds to the sentence $(\neg\text{faulty}_1 \wedge \text{faulty}_2) \vee \text{faulty}_1$, which has three models:

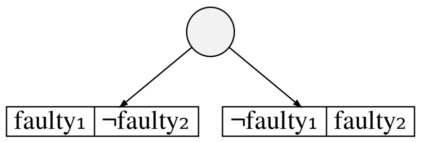
$$\begin{array}{rcr} \text{faulty}_1 & \wedge & \neg\text{faulty}_2 \\ \neg\text{faulty}_1 & \wedge & \text{faulty}_2 \\ \text{faulty}_1 & \wedge & \text{faulty}_2. \end{array}$$

Often, we are interested in the *minimum cardinality* diagnoses: ones with a minimal number of faults. The SDD package supports this type of query as well:

[2]Testing if `sdd_node_is_false(gamma)` is also equivalent to testing if `gamma == sdd_manager_false(m)`.

```
SddNode* min_diagnosis = sdd_minimize_cardinality(diagnosis,m);
```

More generally, the cardinality of a model is the number of positive literals that appear in the model. The models of the SDD returned by `sdd_minimize_cardinality` are precisely the minimum cardinality models of the input SDD. In our example, this function gives the following SDD:



which represents the sentence $(\text{faulty}_1 \wedge \neg\text{faulty}_2) \vee (\neg\text{faulty}_1 \wedge \text{faulty}_2)$.

The SDD library provides a function for counting the models of an SDD. For example, the following code snippet counts the number of diagnoses and minimum-cardinality diagnoses in our example.

```
SddModelCount count = sdd_model_count(diagnosis,m);
SddModelCount min_count = sdd_model_count(min_diagnosis,m);
```

Here, we have 3 diagnoses, and 2 minimum-cardinality diagnoses.

The functions `sdd_model_count`, `sdd_minimum_cardinality`, and `sdd_minimize_cardinality` are with respect to the variables that a given SDD essentially depends on. More formally, we say that a Boolean function f (and analogously, an SDD) essentially depends on a variable X iff $f|X \neq f|\neg X$. We can see what variables an SDD depends on using the function:

```
int* sdd_variables(SddNode* node, SddManager* manager);
```

If an SDD manager is associated with n variables, then function `sdd_variables` returns an array of length $n+1$, where the i-th entry is 1 if the SDD essentially depends on the i-th variable and 0 otherwise. The 0-th entry is unused.

Suppose that we had observed wires A and B to be high. If we condition our knowledge base on this observation, and then existentially quantify out variable C, we get faulty_1. Calling `sdd_minimize_cardinality` on this result also returns faulty_1. However, the minimum-cardinality diagnoses (with respect to variables faulty_1 and faulty_2) is $\text{faulty}_1 \wedge \neg\text{faulty}_2$. As the function representing our diagnoses does not essentially depend on variable faulty_2, we should compensate for the absence of faulty_2 in terms of the model count and minimum cardinality models. For model counting, we can multiply the model count obtained by 2 for each missing variable. For minimum cardinality models, we can conjoin the corresponding negative literal, for each missing variable. We perform such a check in our example in Appendix A.

5.5 File I/O

We can save any SDD that we create to a file. For example, the following code snippet saves the SDD representing our minimum-cardinality diagnoses to a file called `diagnosis-min.sdd`:

```
sdd_save("diagnosis-min.sdd",min_diagnosis);
```

The SDD file format is a simple text format, which is described in the file itself.

The file can be read in again using the code snippet:

```
SddLiteral var_count = 5;
SddManager* manager = sdd_manager_create(var_count);
SddNode* sdd = sdd_read("diagnosis-min.sdd",manager);
```

To use the function sdd_read, we need an SDD manager with a variable count that is large enough to cover all variables mentioned by the saved SDD.

The SDD library also allows us to visualize SDDs via the graph drawing system GraphViz (available at http://www.graphviz.org/). The SDDs illustrated in this manual were indeed generated by the dot program. The following code snippet saves an SDD to dot format:

```
sdd_save_as_dot("diagnosis-min.dot",min_diagnosis);
```

We can then use the dot program to generate a graphical depiction of the SDD, as a PNG image file, using the following command:

```
dot -Tpng -odiagnosis-min.png diagnosis-min.dot
```

Note that, in the generated graphs, variables with indices from 1 to 26 are mapped to letters from A to Z. Higher indices simply have their index printed.

6 Garbage Collection

While constructing and manipulating SDDs, one may construct many intermediate SDDs that are used once, but can subsequently be discarded. The SDD library itself produces auxiliary SDDs to perform certain operations. For sufficiently complex programs, these auxiliary SDDs will eventually exhaust a system's memory unless garbage collected.

The SDD library includes a garbage collector which is based on reference counts. SDD nodes can be referenced and dereferenced using the following functions:

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

For convenience, these functions return the same pointer to the SDD node being referenced.

Any SDD node that is not currently referenced is subject to garbage collection, which can be invoked manually, or automatically by the SDD library as needed. We will restrict our discussion here to automatic garbage collection, leaving manual garbage collection to the advanced-user manual.

Automatic garbage collection can be activated by creating an SDD manager as follows:

```
int auto_gc_and_minimize = 1; // disable (0) or enable (1) auto-gc & auto-min
SddManager* manager = sdd_manager_create(var_count,auto_gc_and_minimize);
```

When automatic garbage collection is activated, the SDD library will occasionally try to minimize the size of existing SDDs (an operation that is distinct from cardinality minimization, which we discussed in the previous section). Hence, the structure of SDDs (which can be visualized using dot) may change over time, although the logical content of such SDDs will remain the same.

In the example from Section 5, we can reference any SDD node as soon as soon as it is created. We can also dereference an SDD node as soon as we know that we will no longer use it. We can do this in a systematic way, line-by-line.

Consider the following code snippet, where we construct the first sentence of our knowledge base Δ:

```
SddNode* delta = sdd_manager_true(m);
SddNode* alpha;
alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(-B,m),m);
alpha = sdd_imply(sdd_manager_literal(-faulty1,m),alpha,m);
delta = sdd_conjoin(delta,alpha,m);
```

For each function that returns an SDD, we will immediately reference the returned SDD. Moreover, after we obtain that SDD, we will dereference any SDD used to construct it, if we know that it will be the last time we need to use that SDD. It is not necessary, however, to reference or dereference elementary SDDs, i.e., the SDDs for true, false and for literals. These SDDs will never be garbage collected. Moreover, referencing and dereferencing them has no effect.

First, we reference the SDD returned by the function `sdd_equiv`:

```
alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(-B,m),m);
sdd_ref(alpha,m);
```

We do not need to reference or dereference any of the input SDDs here, as they are SDDs for literals. Next, we want to reference the SDD returned by the function `sdd_imply`. However, we will want to dereference the input SDD that was returned by `sdd_equiv`, as we will no longer need it after calling `sdd_imply`.

```
SddNode* tmp;
alpha = sdd_imply(sdd_manager_literal(-faulty1,m),tmp = alpha,m);
sdd_ref(alpha,m);
sdd_deref(tmp,m);
```

Here, we have used a temporary pointer `tmp` to save the address of the input SDD `alpha`. Finally, we reference the SDD returned by function `sdd_conjoin`.

```
delta = sdd_conjoin(delta,alpha,m);
sdd_ref(delta,m);
sdd_deref(alpha,m);
```

Here, we have dereferenced the input `alpha`, which we will again no longer need. Note that we do not need to dereference the input `delta`, since it corresponds to the SDD for true, which never needs to be dereferenced.

Consider the following code snippet from the previous section, where we constructed the second sentence of our knowledge base Δ:

```
alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(B,m),m);
alpha = sdd_disjoin(alpha,sdd_manager_literal(-B,m),m);
alpha = sdd_imply(sdd_manager_literal(faulty1,m),alpha,m);
delta = sdd_conjoin(delta,alpha,m);
```

we can also systematically augment this snippet, line-by-line, in a similar way.

```
alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(B,m),m);
sdd_ref(alpha,m);
alpha = sdd_disjoin(tmp = alpha,sdd_manager_literal(-B,m),m);
sdd_ref(alpha,m); sdd_deref(tmp,m);
alpha = sdd_imply(sdd_manager_literal(faulty1,m),tmp = alpha,m);
sdd_ref(alpha,m); sdd_deref(tmp,m);
delta = sdd_conjoin(tmp = delta,alpha,m);
sdd_ref(delta,m); sdd_deref(tmp,m); sdd_deref(alpha,m);
```

Appendix A contains the complete circuit diagnosis example, where we have systematically incorporated referencing and dereferencing in the same way: reference any SDD returned by a call to the library (or a call to a function that calls the library), and dereference any input SDDs that are no longer needed.

One can check the reference count of an SDD node using the following function:

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

This can be useful for debugging purposes. To aid further with debugging, the following functions are provided by the SDD library.

```
SddSize sdd_id(SddNode* node);
int sdd_garbage_collected(SddNode* node, SddSize id);
```

Every SDD node is assigned a unique ID when created. Moreover, when an SDD node is garbage collected, its structure is not freed. Instead, it is added to a special list, called the gc-list, allowing this structure to be reused for newly created SDD nodes. When this structure is reused though, it will be assigned a new ID. Hence, to test whether a node has been garbage collected, one must supply both a pointer to the node and its ID. The following code snippet shows how to test whether a node has been garbage collected.

```
SddSize id = sdd_id(alpha);
// ...
// CONSTRUCT, MANIPULATE AND QUERY SDDS
// ...
if(sdd_garbage_collected(alpha,id)) {
  //alpha has been garbage collected
}
```

7 Beyond the Manual

This manual covered only the core concepts for getting started with the SDD library. The advanced-user manual covers a number of additional topics that would become important for applications that require finer-grained control of the machinery underlying the SDD library. These include topics such as:

- **manual garbage collection and minimization:** The function `sdd_manager_create` initializes an SDD manager that can perform automatic garbage collection and SDD minimization, by default. For certain applications, however, the user may want to have control on when garbage collection and minimization are invoked. The SDD library includes support for this, by providing primitives for manual garbage collection and SDD minimization.

- **vtrees and vtree search:** A *vtree* is a full, binary tree with its leaves labeled with variables. Every SDD manager has a vtree associated with it, which also determines the size and structure of its SDDs. The SDD library includes functions for specifying these vtrees. It also includes functions which allow the user to search for good vtrees (i.e., ones that try to minimize the size of corresponding SDDs) [3].

- **X-constrained vtrees:** SDDs that are constructed with **X**-constrained vtrees have various applications, including to $\mathrm{NP}^{\mathrm{PP}}$-complete and $\mathrm{PP}^{\mathrm{PP}}$-complete problems [6, 7]. **X**-constrained vtrees have also been leveraged for modeling and learning with structured spaces, via Structured Bayesian Networks [8]. The SDD library provides support for **X**-constrained vtrees. In particular, when an SDD manager is created with an **X**-constrained vtree, the **X**-constrained property will be maintained, even during SDD minimization (vtree search).

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

[2] Y. Xue, A. Choi, and A. Darwiche, "Basing decisions on sentences in decision diagrams," in *Proceedings of the 26th Conference on Artificial Intelligence (AAAI)*, pp. 842–849, 2012.

[3] A. Choi and A. Darwiche, "Dynamic minimization of sentential decision diagrams," in *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*, pp. 187–194, 2013.

[4] S. Bova, "SDDs are exponentially more succinct than OBDDs," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI)*, pp. 929–935, 2016.

[5] G. Van den Broeck and A. Darwiche, "On the role of canonicity in knowledge compilation," in *Proceedings of the 29th Conference on Artificial Intelligence (AAAI)*, pp. 1641–1648, 2015.

[6] U. Oztok, A. Choi, and A. Darwiche, "Solving $\mathrm{pp}^{\mathrm{pp}}$-complete problems using knowledge compilation," in *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 94–103, 2016.

[7] Y. Choi, A. Darwiche, and G. Van den Broeck, "Optimal feature selection for decision robustness in Bayesian networks," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1554–1560, 2017.

[8] Y. Shen, A. Choi, and A. Darwiche, "Conditional PSDDs: Modeling and learning with modular knowledge," in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, 2018. To appear.

A Code Sample (With Garbage Collection Disabled)

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

// returns an SDD node representing ( node1 => node2 )
SddNode* sdd_imply(SddNode* node1, SddNode* node2, SddManager* manager) {
  return sdd_disjoin(sdd_negate(node1,manager),node2,manager);
}

// returns an SDD node representing ( node1 <=> node2 )
SddNode* sdd_equiv(SddNode* node1, SddNode* node2, SddManager* manager) {
  return sdd_conjoin(sdd_imply(node1,node2,manager),
                     sdd_imply(node2,node1,manager),manager);
}


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

  // set up vtree and manager
  SddLiteral var_count = 5;
  int auto_gc_and_minimize = 0;
  SddManager* m = sdd_manager_create(var_count,auto_gc_and_minimize);

  SddLiteral A = 1, B = 2, C = 3, faulty1 = 4, faulty2 = 5;

  SddNode* delta = sdd_manager_true(m);
  SddNode* alpha;

  ////////// CONSTRUCT KNOWLEDGE BASE //////////

  // ~faulty1 => ( A <=> ~B )
  alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(-B,m),m);
  alpha = sdd_imply(sdd_manager_literal(-faulty1,m),alpha,m);
  delta = sdd_conjoin(delta,alpha,m);

  // faulty1 => ( ( A <=> B ) v ~B )
  alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(B,m),m);
  alpha = sdd_disjoin(alpha,sdd_manager_literal(-B,m),m);
  alpha = sdd_imply(sdd_manager_literal(faulty1,m),alpha,m);
  delta = sdd_conjoin(delta,alpha,m);

  // ~faulty2 => ( B <=> ~C )
  alpha = sdd_equiv(sdd_manager_literal(B,m),sdd_manager_literal(-C,m),m);
  alpha = sdd_imply(sdd_manager_literal(-faulty2,m),alpha,m);
  delta = sdd_conjoin(delta,alpha,m);

  // faulty2 => ( ( B <=> C ) v ~C )
  alpha = sdd_equiv(sdd_manager_literal(B,m),sdd_manager_literal(C,m),m);
  alpha = sdd_disjoin(alpha,sdd_manager_literal(-C,m),m);
  alpha = sdd_imply(sdd_manager_literal(faulty2,m),alpha,m);
  delta = sdd_conjoin(delta,alpha,m);

  ////////// PERFORM QUERY //////////
```

```
int* variables;
SddLiteral health_vars = 2, health_vars_count, missing_health_vars;

// make observations
delta = sdd_condition(A,delta,m);
delta = sdd_condition(-C,delta,m);

// check if observations are normal
SddNode* gamma;
gamma = sdd_condition(-faulty1,delta,m);
gamma = sdd_condition(-faulty2,gamma,m);
int is_abnormal = gamma == sdd_manager_false(m); // sdd_node_is_false(gamma);
printf("observations normal?  : %s\n", is_abnormal ? "abnormal":"normal");

// project onto faults
SddNode* diagnosis = sdd_exists(B,delta,m);
// diagnosis no longer depends on variables A,B or C

// count the number of diagnoses
SddModelCount count = sdd_model_count(diagnosis,m);
// adjust for missing faults
variables = sdd_variables(diagnosis,m);
health_vars_count = variables[faulty1] + variables[faulty2];
missing_health_vars = health_vars - health_vars_count;
count <<= missing_health_vars; // multiply by 2^missing_health_vars
free(variables);

// find minimum cardinality diagnoses
SddNode* min_diagnosis = sdd_minimize_cardinality(diagnosis,m);
variables = sdd_variables(min_diagnosis,m);
// adjust for missing faults
if ( variables[faulty1] == 0 )
  min_diagnosis = sdd_conjoin(min_diagnosis,sdd_manager_literal(-faulty1,m),m);
if ( variables[faulty2] == 0 )
  min_diagnosis = sdd_conjoin(min_diagnosis,sdd_manager_literal(-faulty2,m),m) ;
free(variables);

// count the number of minimum cardinality diagnoses, and minimum cardinality
SddModelCount min_count = sdd_model_count(min_diagnosis,m);
SddLiteral min_card =  sdd_minimum_cardinality(min_diagnosis);

printf("sdd model count       : %"PRImcS"\n",count);
printf("sdd model count (min) : %"PRImcS"\n",min_count);
printf("sdd cardinality       : %"PRIlitS"\n",min_card);

////////// SAVE SDDS //////////

printf("saving sdd and dot ...\n");

sdd_save("output/circuit-kb.sdd",delta);
sdd_save("output/diagnosis.sdd",diagnosis);
sdd_save("output/diagnosis-min.sdd",min_diagnosis);

sdd_save_as_dot("output/circuit-kb.dot",delta);
sdd_save_as_dot("output/diagnosis.dot",diagnosis);
sdd_save_as_dot("output/diagnosis-min.dot",min_diagnosis);
```

```
  ////////// CLEAN UP //////////

  sdd_manager_free(m);

  return 0;
}
```

B Code Sample (With Garbage Collection Enabled)

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

// returns an SDD node representing ( node1 => node2 )
SddNode* sdd_imply(SddNode* node1, SddNode* node2, SddManager* manager) {
  SddNode* neg_node1 = sdd_negate(node1,manager);
  sdd_ref(neg_node1,manager);
  SddNode* alpha = sdd_disjoin(neg_node1,node2,manager);
  sdd_deref(neg_node1,manager);
  return alpha;
}

// returns an SDD node representing ( node1 <=> node2 )
SddNode* sdd_equiv(SddNode* node1, SddNode* node2, SddManager* manager) {
  SddNode* imply1 = sdd_imply(node1,node2,manager);
  sdd_ref(imply1,manager);
  SddNode* imply2 = sdd_imply(node2,node1,manager);
  sdd_ref(imply2,manager);
  SddNode* alpha = sdd_conjoin(imply1,imply2,manager);
  sdd_deref(imply1,manager); sdd_deref(imply2,manager);
  return alpha;
}


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

  // set up vtree and manager
  SddLiteral var_count = 5;
  int auto_gc_and_minimize = 1;
  SddManager* m = sdd_manager_create(var_count,auto_gc_and_minimize);

  SddLiteral A = 1, B = 2, C = 3, faulty1 = 4, faulty2 = 5;

  SddNode* delta = sdd_manager_true(m);
  SddNode* alpha;
  SddNode* tmp;

  ////////// CONSTRUCT KNOWLEDGE BASE //////////

  // ~faulty1 => ( A <=> ~B )
  alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(-B,m),m);
  sdd_ref(alpha,m);
  alpha = sdd_imply(sdd_manager_literal(-faulty1,m),tmp = alpha,m);
  sdd_ref(alpha,m); sdd_deref(tmp,m);
  delta = sdd_conjoin(tmp = delta,alpha,m);
  sdd_ref(delta,m); sdd_deref(tmp,m); sdd_deref(alpha,m);

  // faulty1 => ( ( A <=> B ) v ~B )
  alpha = sdd_equiv(sdd_manager_literal(A,m),sdd_manager_literal(B,m),m);
  sdd_ref(alpha,m);
  alpha = sdd_disjoin(tmp = alpha,sdd_manager_literal(-B,m),m);
  sdd_ref(alpha,m); sdd_deref(tmp,m);
  alpha = sdd_imply(sdd_manager_literal(faulty1,m),tmp = alpha,m);
```

```
sdd_ref(alpha,m); sdd_deref(tmp,m);
delta = sdd_conjoin(tmp = delta,alpha,m);
sdd_ref(delta,m); sdd_deref(tmp,m); sdd_deref(alpha,m);

// ~faulty2 => ( B <=> ~C )
alpha = sdd_equiv(sdd_manager_literal(B,m),sdd_manager_literal(-C,m),m);
sdd_ref(alpha,m);
alpha = sdd_imply(sdd_manager_literal(-faulty2,m),tmp = alpha,m);
sdd_ref(alpha,m); sdd_deref(tmp,m);
delta = sdd_conjoin(tmp = delta,alpha,m);
sdd_ref(delta,m); sdd_deref(tmp,m); sdd_deref(alpha,m);

// faulty2 => ( ( B <=> C ) v ~C )
alpha = sdd_equiv(sdd_manager_literal(B,m),sdd_manager_literal(C,m),m);
sdd_ref(alpha,m);
alpha = sdd_disjoin(tmp = alpha,sdd_manager_literal(-C,m),m);
sdd_ref(alpha,m); sdd_deref(tmp,m);
alpha = sdd_imply(sdd_manager_literal(faulty2,m),tmp = alpha,m);
sdd_ref(alpha,m); sdd_deref(tmp,m);
delta = sdd_conjoin(tmp = delta,alpha,m);
sdd_ref(delta,m); sdd_deref(tmp,m); sdd_deref(alpha,m);

////////// PERFORM QUERY //////////

int* variables;
SddLiteral health_vars = 2, health_vars_count, missing_health_vars;

// make observations and project onto faults
delta = sdd_condition(A,tmp = delta,m);
sdd_ref(delta,m); sdd_deref(tmp,m);
delta = sdd_condition(-C,tmp = delta,m);
sdd_ref(delta,m); sdd_deref(tmp,m);

// check if observations are normal
SddNode* gamma;
gamma = sdd_condition(-faulty1,delta,m);
sdd_ref(gamma,m);
gamma = sdd_condition(-faulty2,tmp = gamma,m);
sdd_ref(gamma,m); sdd_deref(tmp,m);
int is_abnormal = gamma == sdd_manager_false(m); // sdd_node_is_false(gamma);
printf("observations normal?  : %s\n", is_abnormal ? "abnormal":"normal");
sdd_deref(gamma,m);

// project onto faults
SddNode* diagnosis = sdd_exists(B,delta,m);
sdd_ref(diagnosis,m);
// diagnosis no longer depends on variables A,B or C

// count the number of diagnoses
SddModelCount count = sdd_model_count(diagnosis,m);
// adjust for missing faults
variables = sdd_variables(diagnosis,m);
health_vars_count = variables[faulty1] + variables[faulty2];
missing_health_vars = health_vars - health_vars_count;
count <<= missing_health_vars; // multiply by 2^missing_health_vars
free(variables);
```

```
  // find minimum cardinality diagnoses
  SddNode* min_diagnosis = sdd_minimize_cardinality(diagnosis,m);
  sdd_ref(min_diagnosis,m);
  variables = sdd_variables(min_diagnosis,m);
  // adjust for missing faults
  if ( variables[faulty1] == 0 ) {
    min_diagnosis = sdd_conjoin(tmp = min_diagnosis,sdd_manager_literal(-faulty1,m),m);
    sdd_ref(min_diagnosis,m); sdd_deref(tmp,m);
  }
  if ( variables[faulty2] == 0 ) {
    min_diagnosis = sdd_conjoin(tmp = min_diagnosis,sdd_manager_literal(-faulty2,m),m);
    sdd_ref(min_diagnosis,m); sdd_deref(tmp,m);
  }
  free(variables);

  // count the number of minimum cardinality diagnoses, and minimum cardinality
  SddModelCount min_count = sdd_model_count(min_diagnosis,m);
  SddLiteral min_card =  sdd_minimum_cardinality(min_diagnosis);

  printf("sdd model count       : %"PRImcS"\n",count);
  printf("sdd model count (min) : %"PRImcS"\n",min_count);
  printf("sdd cardinality       : %"PRIlitS"\n",min_card);

  ////////// SAVE SDDS //////////

  printf("saving sdd and dot ...\n");

  sdd_save("output/circuit-kb.sdd",delta);
  sdd_save("output/diagnosis.sdd",diagnosis);
  sdd_save("output/diagnosis-min.sdd",min_diagnosis);

  sdd_save_as_dot("output/circuit-kb.dot",delta);
  sdd_save_as_dot("output/diagnosis.dot",diagnosis);
  sdd_save_as_dot("output/diagnosis-min.dot",min_diagnosis);

  ////////// CLEAN UP //////////

  sdd_manager_free(m);

  return 0;
}
```

C SDD API (Core)

Following are the core functions of the SDD library. The advanced-user manual describes the complete API.

SDD Manager

```
SddManager* sdd_manager_create(SddLiteral var_count, int auto_gc_and_minimize);
void sdd_manager_free(SddManager* manager);
```

Elementary SDDs

```
SddNode* sdd_manager_true(const SddManager* manager);
SddNode* sdd_manager_false(const SddManager* manager);
SddNode* sdd_manager_literal(const SddLiteral literal, SddManager* manager);
int sdd_node_is_true(SddNode* node);
int sdd_node_is_false(SddNode* node);
```

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);
SddModelCount sdd_model_count(SddNode* node, SddManager* manager);
SddLiteral sdd_minimum_cardinality(SddNode* node);
SddNode* sdd_minimize_cardinality(SddNode* node, SddManager* manager);
int* sdd_variables(SddNode* node, SddManager* manager);
```

File I/O

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

Memory Management

```
SddNode* sdd_ref(SddNode* node, SddManager* manager);
SddNode* sdd_deref(SddNode* node, SddManager* manager);
SddRefCount sdd_ref_count(SddNode* node);
SddSize sdd_id(SddNode* node);
int sdd_garbage_collected(SddNode* node, SddSize id);
```

D License