M.S. Capstone Report: Oxidizing Decision Diagrams with RSDD

Matthew Wang

June 2023

Abstract

RSDD is a knowledge compilation library written in Rust. Among other features, it can compile and manipulate Binary Decision Diagrams (BDDs) and Sentential Decision Diagrams (SDDs). This paper highlights three core contributions to RSDD. First, we discuss a refactoring of the entire knowledge compilation library to properly use Rust's trait system; this unifies BDDs and SDDs, improves memory safety guarantees, and allows for more rapid experimentation via cleaner specialization. Then, we introduce *semantic hashing*, a probabilistic equivalence test for SDDs that provides an alternative to canonicalization. We show that even a naive implementation of probabilistic tests have promising results for certain classes of CNFs and SDDs. Finally, we outline three separate supporting software projects: a WebAssembly compilation target and online web demo, a head-to-head benchmark suite against other knowledge compilers, and a user-facing documentation site with interactive demos and tutorials.

1 Introduction

Knowledge representation and compilation are foundational areas of computer science with wide-ranging impacts on statistical machine learning, probabilistic programming, and explainable artificial intelligence. At its core, knowledge compilation is focused on converting human knowledge into a tractable representation that can then be frequently and efficiently queried. Significant strides have been made in the past two decades to bridge knowledge compilation theory and practice; libraries like the SDD library, CUDD, and Sylvan have wide-ranging applications in probabilistic inference and programming, cryptography, and explainable AI.

However, the majority of knowledge compilation libraries share a weakness: they are implemented in C. As a result, they have been unable to take advantage of recent advances in software engineering and programming languages. The language's reliance on manual memory management induces cognitive load and makes software more prone to bugs; the lack of true object-oriented and functional programming paradigms makes rapid research experimentation and code reuse challenging; the fracturing of C build systems has caused portability issues for newer targets (such as macOS on ARM or WebAssembly). More broadly, their relative age has made it harder for them to take advantage of newer developments in software engineering practices.

In 2017, Steven Holtzen developed the "Rust Decision Diagrams" (RSDD)[1] library as a supporting project for the Dice probabilistic programming language [5]. A core motivation for this new library was to address the shortcomings of older C-based libraries, while maintaining similar or better performance.

The rest of this capstone report is split up into five sections. First, we provide more context for decision diagrams, knowledge representation and compilation, and RSDD. Next, we discuss our largest code contribution: a complete re-architecture of the library to allow for modular and extensible "compilation backends". In turn, this allows us to iterate rapidly on research ideas. We then introduce the theory for "semantic" (or probabilistic equivalence) hashing for

[1]`https://github.com/neuppl/rsdd`

SDDs as an alternative for canonicalization via compression; we implement a naive approach in RSDD as a proof-of-concept and as a case study for the benefits of the refactor. Our final contributions are three pieces of new supporting software: a WebAssembly target and web interface; a reproducible head-to-head benchmark suite; and, public-facing user documentation and tutorials with a focus on interactivity. Finally, we discuss broader next steps building off of this work.

1.1 Decision Diagrams

These libraries primarily deal with a data structure called a Binary Decision Diagram (BDD). A BDD is a directed acyclic graph that corresponds with a Boolean function. Most nodes in the graph are *decision nodes*, which represent one Boolean variable in the formula. The outgoing edges from the node represent an assignment to that variable: one edge represents the rest of the Boolean formula if the variable is set to true, while the other is if the variable is set to false. There are also two special nodes in every BDD that represent the literals $true$ and $false$. Following a certain set of conditions, a BDD can be considered *reduced*; this is the most compact form for any BDD. This also provides a *canonicity property*, in which there is a one-to-one correspondence between Reduced Ordered BDDs (ROBDDs) and Boolean formulas.

BDDs have many useful properties, including polynomial-time conjunction, disjunction, and model counting — the latter being $\#P$-complete) [3]. However, compiling a Boolean formula into a BDD can be a non-trivial task, reducing a BDD is not a linear operation, and many other queries are worst-case exponential time and space. Nevertheless, they are frequently used in a wide range of applications.

There are many variations on BDDs — part of a larger class of data structures called *decision diagrams* — that have similar goals. In particular, they convert Boolean formulas (optionally, with more information) into data structures that enable tractable queries (such as model counting). Often times, these decision diagrams will trade off the canonicity property or efficiency in one type of query for improved performance in a specific task.

Key to this project is Sentential Decision Diagrams (SDDs), one such alternative decision diagram data structure. These were first introduced by Adnan Darwiche in his 2011 paper, "SDD: A New Canonical Representation of Propositional Knowledge Bases" [2]. They significantly relax the restrictions of a BDD — allowing formulas to be in nodes and adding AND and OR gates with arbitrary numbers of children — but also impose a different type of structure, including a stricter variable ordering regiment called a *vtree*, and a partitioning criteria for nodes splitting formulas into pairs of *primes* and *subs*. SDDs are more succinct than BDDs, in that any BDD can be represented as an SDD; however, SDDs have a significantly tighter bound on their size. SDDs also support all the operations a BDD supports.

1.2 RSDD

RSDD is a knowledge compilation library written in Rust. Among other features, it fully supports knowledge compilation with both BDDs and SDDs; that is, it can take an arbitrary Boolean formula and convert it into a canonical BDD or SDD. In addition, it can perform operations on BDDs and SDDs, including conjunction, disjunction, negation, existential quantification, and compound operations like *if-then-else*.

Understanding the terminology of RSDD is not crucial for this project. However, we briefly sketch out some important details that underpin our contributions to the library.

- **pointers**: BDDs and SDDs can become extraordinarily large. As a result, the library exposes pointers to BDDs and SDDs, which are fully operational data structures to the end user. Internally, the library allocates significantly fewer nodes, with the goal of converting redundancy within decision diagrams into multiple references on the same node. However, this presents a significant engineering challenge, as RSDD needs to properly manage all of its references and objects. In RSDD, pointer types are suffixed with `Ptr`, e.g. `BddPtr` and `SddPtr`.

- **managers/builders**: to create and modify BDDs and SDDs, RSDD provides managers (or builders): these create pointer objects and preform operations on them (ex AND, OR, XOR, ...). The manager is also responsible for managing all allocations, as well as caching any relevant operations. One commonly cached operation is `Apply`, which is described more in later sections.

Throughout this paper, we will primarily discuss inputs that are CNFs (Clausal Normal Form), which are essentially Boolean formulas that are a conjunction over disjunctions. This does not restrict the generality of the library (any Boolean formula can be converted to a CNF); but, CNFs are commonly used in decision diagram benchmarks, SAT solving, and other applications involving Boolean formulas.

Finally, some elements of this report require a cursory understanding of the programming language Rust. To briefly enumerate the core details:

- one of Rust's main selling points is its compile-time memory safety: using the *ownership system* and *borrow checker*, Rust can conservatively guarantee that all memory accesses are safe (primarily by imposing restrictions on mutable references). There is much more nuance to this topic, but that is out of scope of this report.

- Rust does not have classes or class-based inheritance. Instead, object-oriented programming is done via structs (similar to structs in other languages — just data) and methods defined on structs. Inheritance is done via traits, which behave similarly to interfaces.

- Rust supports generic programming through monomorphized functions, i.e. at compile-time parameterized types are replaced with concrete ones.

2 Large-scale Refactor

This section details the largest library software engineering work in this paper: a (mostly internal-facing) refactoring of the library's knowledge compilation primitives to better align with idiomatic Rust, utilize more of Rust's compile-time memory safety tooling, and provide a simple and extendable user interface. It spans over 10000 lines of code. Compared to the other work, this section leans more on knowledge about Rust and library internals; the novel contribution is on software design and engineering.

2.1 Motivation

In RSDD, various user-facing structs — such as the `BddManager` or `SddManager`, which build and operate on BDDs and SDDs respectively — have specialization features. For example, a user might want to use a different cache eviction strategy for different BDD and SDD operations, or change the definition of SDD equality to not rely on compression (the goal of the next section). The latter option changes the behaviour of the library on a semantic (and type) level; as a result, we felt that this needed to be reflected in the type system. Rust prefers (and only allows) trait-based inheritance, so the initial library design was *mixin*-based: relevant functions and structs are generic over their configurable parameters.

```
// specializing the BDD manager
// over its apply cache.
// requires nested generics!
let mut man = BddManager::<BddApplyTable<
↪ BddPtr>>::new_default_order_lru(cnf.
↪ num_vars());

// specializing the SDD manager
// over its compression mechanism.
// also requires mutability, verbosity
let mut compr_mgr = SddManager::<
↪ CompressionCanonicalizer>::new(vtree);
let mut sem_mgr = SddManager::<
↪ SemanticCanonicalizer<479001599>>::new(
↪ vtree);
```

However, this approach has several flaws. The largest comes from the mixin approach: it becomes infeasible to add significant specialization to each type of `BddManager` or `SddManager`, since it needs to be implemented over the entire generic struct (rather than subclassing, which Rust does not directly support). It also increases the bundle size of the code (since Rust monomorphizes generics). And, from an ergonomics perspective, the code becomes significantly clunkier: every struct and function has to become generic.

There were a handful of other areas of technical debt in the library around Rust's borrow checker, which manifested itself in a significant use of the `unsafe` keyword (to revert to compile-time memory checks) and unnecessary copying. Thus, this provided a great opportunity to resolve both of these issues through a large-scale refactor of the entire knowledge compilation portion of RSDD.

2.2 Implementation

The new approach to RSDD's knowledge compilers introduces two large changes:

1. The core builder functionality is refactored into a series of traits (`BottomUpBuilder`, `BddBuilder`, `SddBuilder`). Concrete managers will implement these traits, allowing internal specialization while maintaining the same public interface.

2. Each builder (and associated pointer type) is given an annotated lifetime. This is a feature from Rust that guarantees at compile-time that certain objects outlive other objects (in this case, that the builder will never have access to an invalid/dead decision diagram via a pointer). This requires some significant software engineering work — leaning on Rust's interior mutability features and associated types, both of which are only recently stable features of the language — but do more to ensure compile-time memory safety (a huge draw for using Rust).

The outcome of these changes is visible internally and externally. Internally, we have reduced the reliance on runtime memory checks and panics; annotated lifetimes provide compile-time guarantees on memory access. In areas that still require runtime memory checks (such as the validity of a cache that is accessible by multiple data structures), the guardrails are more explicitly noted for developers and downstream users. In particular, we have removed all interactions with raw pointers, which leads to less `unsafe` code and aligns with idiomatic Rust.

These changes are mostly opaque to users; but, externally, they benefit from a more unified and clean API. First and foremost, the code is more concise without losing its expressiveness.

```
// specializing the SDD manager.
// observe that a mutable reference
// is no longer required, and that
// the structs are simpler to use.
let m = CompressionSddManager::new(vtree);
let m = SemanticSddManager::<479001599>::
↪  new(vtree);
```

In addition, this change makes it significantly easier to define and use custom managers. This change unifies all builders under the `BottomUpBuilder` trait, which implements typical decision diagram operations such as **`and`**, `or`, and `xor`. Developers would then implement this trait with a custom struct (with arbitrary internal representations, extra operations, etc). It is then trivial to use this new builder in existing code that relies on the `BottomUpBuilder` interface.

One concern was whether or not this additional abstraction would incur a runtime cost. However, a core benefit of Rust's abstraction model (compared to other languages) is that abstractions are *zero-cost*. We confirmed with the benchmark suite discussed in later sections that previous performance gains from the library were still preserved; the head-to-head comparison with previous versions of the library had only negligible differences (in the sub-milliseconds).

This change was arduous (with a $\pm$ of over 10000 lines of code), but provided a more memory safe internal implementation of the library, a more ergonomic and extendable public interface, and enabled rapid research experimentation. In the next section, we provide one such case study benefiting from the new API.

2.3 Future Work: Stabilizing the API

As a brief note: there is still much to finalize in this refactoring. Our eventual goal is to completely finalize the new API for the library in the next few months, and expand this approach to other components of the library. Stabilizing the API would allow us to make bindings for other languages (such as Julia, OCaml, Python, or C), complete our documentation and tutorials, and overall encourage more users of the library.

3 Semantic Hashing

This next section describes a novel application of probabilistic equivalence checking to SDDs: providing an alternative canonicity metric for SDDs that relaxes the compression-based canonicity bound introduced in the original SDD paper [2]. It also serves as a case study for the ease-of-implementation following the large-scale refactor described in the previous section. Here, the novel contribution is primarily a new algorithm for operating on SDDs.

3.1 Motivation

In the 2011 paper introducing Sentential Decision Diagrams (SDDs) [2], Darwiche introduces the canonicity property for SDDs. Canonicity addresses a potential difference between syntactic and semantic equality. Two SDDs (or BDDs, or many other knowledge representation formats) could be syntactically different but be semantically equivalent. One trivial example is that the Boolean formulas $\alpha \vee \neg\alpha$ and $\neg(\alpha \wedge \neg\alpha)$ represent the same semantic meaning, i.e. they both reduce to *true*.

The original canonicity property guarantees a one-to-one mapping between (reduced) Boolean formulas and SDDs, which is a highly desirable property in theory and in practice. The original theorem is as follows:

Theorem 1 (SDD Canonicity) *Let α and β be compressed and trimmed SDDs respecting nodes in the same vtree. Then $\alpha \equiv \beta$ iff $\alpha = \beta$.*

In other words: given a fixed variable ordering (via vtree), two trimmed and compressed SDDs are only semantically equivalent if they are also syntactically equivalent. Trimmed refers to an SDD that does not have trivial decompositions that immediately evaluate to true (either a prime that is the true literal, or a pair of prime-sub pairs with constant subs that are negations of each other). Compressed refers to an SDD where no two subs are *semantically* equivalent.

Separately, the 2011 paper describes the `Apply` operation for SDDs: this provides a polytime algorithm for performing any binary operation for two SDDs (e.g. conjunction, disjunction, `xor`) [2]. Similar to the `Apply` function for (RO)BDDs, the SDD `Apply` provides a simple abstraction for a variety of operations: converting a CNF/DNF to an SDD, performed per-clause/term and then conjoined/disjoined; more complex operations, such as conditioning, if-then-else, or existential quantification; and, as a foundation for more complicated tasks on logical formulas, such as MPE (Most Probable Explanation) or Marginal MAP (Maximum A Priori hypothesis).

However, the SDD `Apply` operation has one core weakness: it does not guarantee that the output is compressed, even if the inputs are compressed. This means that frequent compression is needed to maintain the canonicity property across multiple `Apply` operations. The original paper notes that it is possible to augment `Apply` by functionally compressing after each iteration; however, they are not able to prove that this is a polytime operation in the size of the SDDs [2]. This is particularly important since the cost of *canonicalizing* (i.e. compressing and trimming) an SDD is non-trivial; its complexity scales more-than-linearly in the number of subs (as it is a pairwise comparison).

Active work has been done in the field to examine polytime `Apply` and its relationship to canonicity. In "On the Role of Canonicity in Bottom-up Knowledge Compilation", Van den Broeck and Darwiche conclude that no general polytime `Apply` exists for this form of canonical SDDs [6]. They provide a theoretical argument for a worst-case exponential `Apply`, but also show empirical results that generally favour compressing SDDs after each `Apply` as opposed to not using the canonicity property at all. The core benefit is that compression (and canonicalization) often reduces the size of SDDs, which reduces the complexity of the `Apply` operation.

However, this leads to a natural different question:

is there an alternative form of canonicalization that *does not require compression*? What other ways can we characterize SDD equality other than requiring the SDDs to be (fully) compressed, and can that lead to tangible improvements in wall-clock time, recursive calls, or allocated memory? In particular — as we'll see in the next section — can we use probabilistic tooling with slightly relaxed theoretical assumptions to provide large benefits in practice?

3.2 Probabilistic Equivalence

SDDs are one relaxation of the restrictions of OBDDs. Wachter and Haenni proposed another alternative called *Propositional DAGs* (PDAGs) in 2006 [8], which similarly allow an arbitrary number of children for series of AND and OR gates, as well as NOT gates. However, value nodes are still restricted to literals and variables, and there is no construct of primes and subs. In other words, they combine primitives from digital circuits with the framing for Boolean formulas. Similar to SDDs, PDAGs have some advantages in succinctness and compilation strategies opposed to BDDs; however, they are at least as expressive as BDDs (i.e., all BDDs are also both SDDs and PDAGs).

In a follow-up paper [7], Wachter and Haenni propose a form of canonicalization for a subset of PDAGs called *probabilistic equivalence checking*. This builds off of previous work from Blum, Chandra, and Wegman that provide an equivalence check for BDDs that is probabilistically decidable in polytime [1]. This work is interesting in that it provides an alternative form of characterizing canonicalization that is not based on compression or trimming, and scales to decision diagrams where nodes can have a large variable number of children (unlike BDDs).

The general approach for BDDs involves arithmetic over a prime finite field. Each propositional variable is assigned a unique integer within the field, denoted as its *hash code*. The hash code for a positive instantiation is the variable's hash code; for a negative instantiation, it is the additive inverse in the finite field (i.e. $1 - H(x)$ for a hash code $H(x)$). The hash code for a partial or total assignment of variables is simply the sum of each of the hash codes for each variable. AND and OR are represented as products and sums of the hash codes of their operands. The values 0 and 1 are reserved for false and true respectively (observe that this is consistent with the identities they hold for multiplication and addition, and thus AND and OR).

This construction provides a hash code for each Boolean formula. Any two Boolean formulas that have the same semantic meaning (i.e. reduce to the same formula) will have equivalent hash codes. In addition, the probability of a hash collision — that is, two Boolean formulas with different semantic meanings having identical hash codes — is probabilistically unlikely. The error rate can be scaled to any arbitrary $\varepsilon > 0$ by either increasing the size of the finite field, or performing the test multiple times (with different assignments) [1].

Given that PDAGs are structurally very similar to BDDs, extending this probabilistic equivalence test is straightforward: AND and OR retain their arithmetic properties, the NOT gate is done with the additive inverse, and all other properties remain the same. For decomposable (children of ANDs are disjoint) and deterministic (children of ORs are contradictory) PDAGs — denoted as cdPDAGs — Wachter and Haenni prove a similar error bound for probabilistic equivalence testing; an arbitrarily-low hash collision rate can be achieved by either scaling the prime or number of iterations for the algorithm [7].

The power of this comes from the polytime nature of the probabilistic test. Indeed, calculating the hash code for a cdPAG is functionally a model count, which for BDDs (and BDD-like structures) is a polytime operation. Thus, while this does not provide an exact test for equality, this provides a potential solution for a polytime canonicalization property that would enable a polytime `Apply`.

3.3 Application to SDDs

Thus, we are interested in applying probabilistic tests to SDDs, particularly since Van den Broeck and Darwiche's results show that an exact canonicalization scheme is likely incompatible with polytime `Apply`. This next subsection formalizes the notion of probabilistic testing for SDDs; we will interchangeably call this process *semantic hashing*, as we later use it *as* the hash of SDDs in allocators.

Definition 1 (SDD Hash Codes) *Given a prime p and an assignment of attributes $A(v) \in \mathbb{F}_p$ for each variable in an SDD, the* ***hash code*** *$H(x) \in \mathbb{F}_p$ is a value assigned to each SDD primitive x. They are defined as follows:*

- *the hash code for the trivial atoms are: $H(false) = 0$, $H(true) = 1$*
- *the hash code for an assignment to a variable v is* $\begin{cases} A(v) & v = true \\ 1 - A(v) & v = false \end{cases}$
- *the hash code of $\neg\alpha$ is $1 - H(\alpha)$*
- *the hash code of $\alpha \vee \beta$ is $H(\alpha) + H(\beta)$*
- *the hash code of $\alpha \wedge \beta$ is $H(\alpha)H(\beta)$*

We then observe (without proof):

- if α and β are semantically equivalent, we must have that $H(\alpha) = H(\beta)$
- if $H(\alpha) \neq H(\beta)$, then α and β are not semantically equivalent
- it is possible for α and β to be semantically different, but have $H(\alpha) = H(\beta)$

In particular, the latter *should* be unlikely, though this paper will not contain a proof outlining a rigorous bound on the error probability; however, the analysis would mirror Wachter and Haenni's. In particular, given that an SDD is a set of AND and OR gates with decomposability and determinism, the exact same proof should apply as it did for cdPDAGs, we should expect a similar bound that is inversely proportional to the desired error rate ε [7].

We can now define our own version of `ProbEquiv` that applies to SDDs, which is described in Algorithm 1.

This has a natural interpretation as a hash function. Similar to hashes, identical (semantic) SDDs have the same hash; two SDDs with different hashes are guaranteed to have a different semantic meaning; the probability of a hash collision is somewhat low.

Algorithm 1 Probabilistic Testing for SDDs

Inputs: SDDs α and β; a fixed hash code for literals H

```
ProbEquiv(\alpha, \beta)
  $r \leftarrow |vars(\alpha) \cup vars(\beta)|$
  $p \leftarrow$ a prime number $\geq \frac{r}{\varepsilon}$
  if $H_p(\alpha) = H_p(\beta)$ then
      return "probably semantically equivalent"
  else
      return "not equivalent"
  end if
```

This provides one immediate benefit in terms of allocating and retrieving SDDs. Because we have the property that $H(\alpha) = 1 - H(\neg\alpha)$ for any SDD α, we never need to allocate *both* an SDD and its negation. When dereferencing an `SddPtr`, we check our allocator by both the hash code for the pointer, and its negation. If an SDD already exists with either, we return a reference there — for the negation, which a complemented edge — and crucially, do not allocate a new SDD. This optimization is not possible with the original canonicity property, since we're given no guarantees on the relationship between an SDD and its negation.

In RSDD, we have implemented the above two algorithms in the `SemanticSddManager`, which uses `ProbEquiv` for all SDD equality operations and `ProbGetOrInsert` for dereferencing SDD pointers. In particular, this slightly changes the approach of `SddApply`, as demonstrated in Algorithm 2 (based on the original `SddApply` [2]).

The key difference is that the apply cache is indexed by the hash of the SDD, rather than the SDD itself (which is typically implemented via strict pointer equality or ID). In addition, observe that we do not compress after calling `ProbApply`.

We then use this to implement all relevant operations on SDD, such as conjunction, disjunction, XOR, etc.

Algorithm 2 Probabilistic SDD Apply

```
Inputs: SDDs $\alpha$ and $\beta$; Boolean operator $\circ$; a fixed hash code for literals $H$; maintain a $Cache$
ProbApply(\alpha, \beta, \circ)
  if $\alpha$ and $\beta$ are constants or literals then
      return $\alpha \circ \beta$
  else if $Cache(H(\alpha), H(\beta), \circ) \neq nil$ then
      return $Cache(H(\alpha), H(\beta), \circ)$
  else
      recursively call ProbApply (conjoin the primes, apply $\circ$ to the subs)
  end if
```

3.4 Naive Semantic Hashing in RSDD

As a proof of concept, I first implemented a naive approach to semantic hashing in RSDD. This naive approach has a few characteristics:

- the first SDD we see with a specific hash code is the canonical SDD for that hash code
- hash codes for variables are created once, when the manager is first initialized; we pick a prime that is orders of magnitude larger than the number of variables, i.e. shooting for $\varepsilon < 0.001$
- we lazily evaluate the hash code; the first time it's needed, we compute it with a weighted model count (where each node's weight is its hash code). Then, we store this on the node for use in subsequent calculations; `SddPtrs` in RSDD are immutable, so the cache will never have to be invalidated.
- we perform *no* compression!

This approach is naive in that there are many potential optimizations (ex, always keeping the smallest canonical SDD). However, the fundamental trade-off should still be observable: we should be able to make up for the time it takes to model count (amortizable, since it's only done once per SDD) and the slightly larger circuit size by not having to compress on every operation. For large CNFs, particularly those with significant logical repetition (and thus, identical hash codes for children), we should expect some observable benefit from our approach.

Implementing naive semantic hashing was relatively simple due to the large-scale refactoring discussed in the previous section. I first had to implement this in the prior RSDD architecture; doing so more than doubled the amount of code used to process SDDs, in part since I needed to conditionally implement all features for *both* the compression-based and semantic canonicalizers in the `SddManager`, rather than specializing each one. However, after the refactor, implementing the `SddBuilder` trait was much simpler; I simply needed to provide a different equality check (via hash code) and hashing functions for the cache; almost all other functionality could be shared across both builders, and generics would handle the rest!

3.5 Evaluation

I performed two types of empirical evaluation on semantic hashing. The first is centered on correctness: how empirically accurate is our estimation on the hash collision rate; is it acceptable for use in large CNFs?

I tested this in a handful of ways:

- comparing two SDDs from the same CNF by hash code / model count, one compiled with compression and one compiled with semantic hashing; since the SDDs should represent the same CNF, the hash code should *always* be the same.
- comparing two SDDs from different CNFs by hash code, both compiled with semantic hashing; since these SDDs represent different CNFs, the hash codes should be probabilistically different; the error rate is our hash collision rate.
- after compiling a large SDD with semantic hashing, examining our allocated SDD nodes; there should be no repeated hash codes (this indicates that our allocator logic is buggy)

I wrote some of these tests with traditional unit tests and known cases (either toy CNFs or known benchmarks, such as CNFs from the LGSynth89 benchmark [4] used in the original SDD paper [2]). However, I found more success using property-based testing tools — specifically, the quickcheck crate [2], which is inspired by the Haskell package of the same name [3]. After writing code to arbitrarily generate CNFs and vtrees, I encoded each of the above conditions as a property test on SDDs. Then, I used the crate to generate a random CNF and vtree and verify each property.

A benefit of a property-based testing approach is that testing scales flexibly. On continuous integration, we run 10000 random tests for each property. I ran a more rigorous test when eventually evaluating this approach by performing 1 million runs (with $p = 4391$, which is an intentionally small prime). In each run, we select a random CNF with at most 9 variables and 16 clauses, as well as a random vtree. Then, I validate each of the above properties. The results are as follows:

- 100% of the time, compiling the same CNF with either compilation strategy yields the same hash code.
- 186/1000000 = 0.0186% of the time, compiling two different distinct CNFs yielded the same semantic hash (i.e. a hash collision occurred)
- 100% of the time, there are no duplicate hashes in our allocation table.

This is a promising result; an error rate of $\sim 0.02\%$ is relatively successful for a probabilistic program. I also verified that scaling the prime reduces the error rate; in the case where $p = 100000049$, I discovered only one hash collision over 100000 runs ($\sim 0.001\%$ error rate). For $p = 18446744073709551591$, I did not encounter any hash collisions over 100000 runs. With that in mind, I am relatively confident in the empirical probabilistic correctness of our program.

The other type of evaluation is performance-based. Here, we have several metrics we can analyze: the traditional wall-clock time, the size of the final SDD, the number of `Apply` calls made, and the total number of allocated nodes.

To evaluate this, I wrote head-to-head comparisons of the semantic hash-based `SddManager` versus the traditional compression-based manager on a variety of CNFs. I elected to use the prime $p = 100000049$. I expected significantly worse wall-clock time (since modular arithmetic over large numbers has non-trivial cost, and our implementation is non-optimized), as well as slightly bigger final SDDs (as they are not compressed). However, our hope was that there would be some cases where the total allocated memory and number of nodes was smaller (since we did not need intermediate compression steps), and that the total number of calls to `Apply` were reduced.

The overall results are mixed. **This will be a table soon — am redoing an experiment using the latest commit!**

The methodology involved generating random CNFs with specific parameters (10 variables, 40 clauses; 20 variables, 50 clauses; etc.), fixing the same vtree (both right-linear and min-fill DTree), and then evaluating the wall clock time, number of nodes allocated, final circuit size, and number of recursive calls to `Apply`. We observed:

- as expected, the wall-clock time for semantic hashing was about an order of magnitude worse than compression in the 10 variables 40 clauses range; but, as the number of variables increased past 30, the difference was more than an order of magnitude.
- the number of allocated nodes is in the same ballpark; but, there would be causes were semantic hashing would allocate less nodes!
- most interestingly: semantic hashing would make up to 50% less recursive `Apply` calls when the number of clauses was much larger than the number of variables (ex: 4-CNFs with 10 variables and 50 clauses, or 5-CNFs with 10 variables and 60 clauses.

The last result is novel, in that I was originally expecting generally subpar performance from this naive

[2]`https://github.com/BurntSushi/quickcheck`

[3]`https://hackage.haskell.org/package/QuickCheck`

approach! However, a good property of the `Apply` is that it *only* measures recursive call numbers, and not performance. My unsubstantiated hypothesis is that semantic hashing works well in crowded CNFs because there is significant redundancy in subtrees; in contrast, semantic hashing performed the worst when there were few clauses compared to variables (ex: 3-CNFs with 30 variables and 40 clauses). However, more experimentation is needed!

3.6 Future Work

This work was mostly a preliminary exploration of the idea of probabilistic testing, and it had promising results. I plan on extending this idea over the next few months in conjunction with Steven Holtzen at NeuPPL.

Narrowly, the implementation for BDDs and SDDs could be better-optimized. Some ideas include:

- a *hybrid algorithm* that performs a bounded number of compressions; either with a fixed number of allowed compressions, or dynamically switching compilation strategies depending on the structure of the CNF
- smartly choosing which SDD to keep as the canonical one per hash code; e.g. the smallest one total, the smallest over the first 5 allocations, etc.
- interning/eagerly evaluating common SDDs for a specific CNF to speed up model counting

However, a more promising and novel application could be for top-down knowledge compilation with d-DNNFs. Unlike BDDs and SDDs, there is no inherent canonicity property to d-DNNFs, which makes optimizing top-down knowledge compilation challenging (particularly at the lower level). Given that d-DNNFs also satisfy the same properties used in Wachter and Haenni's construction of probabilistic equivalence checking for cd-PDAGs, it is possible that probabilistic tests for d-DNNFs can have a same probabilistic proxy for canonicity while also allowing significant speedups by pruning sub-diagrams with the same semantic meaning (and thus, hash code).

More broadly, I am interested in the application of probabilistically-correct methods for knowledge compilation, particularly in a hybrid algorithm setting (where the user can toggle a hyperparameter to get the desired level of accuracy); our initial experiments show promise in this field.

4 Supporting Software

This next section describes three independent software projects that are not directly part of RSDD. Each of these is aimed at maturing the library towards a fully-fledged open-source project. Each of these mini-projects is entirely my own work; I designed and developed them from front-to-back.

4.1 WebAssembly Target and Demo

WebAssembly[4] is a portable compilation target supported by all major web browsers. It allows for code from arbitrary programming languages to be executed performantly on the web, without requiring JavaScript as an intermediate compilation target. WebAssembly has enabled significant innovations in browser-based applications: billion-dollar startups like Figma have built their product around it[5][6], and it received the SIGPLAN Programming Languages Software Award in 2021[7].

Given their fundamental graphical nature, live online demonstrations would improve the onboarding experience and usability of knowledge compilation libraries; pedagogical benefits are also likely. Prior to WebAssembly, running C libraries like the SDD library, CNF2OBDD, Sylvan, or CUDD in the browser required either transpiling to JavaScript, re-implementing the library natively, or writing an application server. The first two options suffer from performance issues and are brittle, while the last incurs a non-trivial maintenance cost. In addition, compiling C libraries to WebAssembly typically requires some

[4] `https://webassembly.org/`

[5] `https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/`

[6] `https://news.adobe.com/news/news-details/2022/Adobe-to-Acquire-Figma/default.aspx`

[7] `https://www.sigplan.org/Awards/Software/`

special tooling and foresight; the older versions of C (and platform-dependent code) make this more challenging for earlier libraries.

In contrast, RSDD has two key advantages. First, Rust (as a programming language, and as an ecosystem) provides more seamless WebAssembly support, owing particularly to more robust macros and a unified build system. Secondly, as RSDD is still earlier in its software lifecycle, larger changes can be made to support different compilation targets.

With this in mind, I added WebAssembly as a compilation target for RSDD. Comparatively, this was an easier task: most of the work involved monomorphizing generic functions, converting and serializing Rust structs and types to WebAssembly types, and pruning library dependencies.

I then created a simple proof-of-concept demo application called *Inddecision*[8]. It primarily demonstrates RSDD's BDD and SDD knowledge compilers. In each of the use-cases, the user can provide a DIMACS CNF (and optionally a variable ordering scheme); RSDD then runs in the browser to compile the corresponding logical formula, generate the data structure, and perform a simple model count. The decision diagram is then rendered using a graphviz-like visualizer. When changing the variable ordering, the decision diagram is automatically re-rendered; this allows users to clearly see the effect of variable ordering on the resulting decision diagram. The demo also allows users to visualize various vtree selection heuristics.

This demo is intentionally simple. It did motivate further exploration into documentation and tutorials, which I outline in the last subsection. In the next steps section, I discuss other potential innovations using the WebAssembly compilation target.

4.2 Head-to-Head Benchmark Suite

Generally, there are few benchmarks for decision diagram software (and, more broadly, research code); this is particularly relevant for libraries like the SDD library and CUDD, which rely on platform-dependent C code. To better understand how RSDD compares to other libraries, I developed a benchmarking suite with an emphasis on portability and reproducibility. The entire suite is packaged up into a set of Docker containers, which make the suite portable for most consumer and server computers. To run a benchmark, the user simply needs to type in one command — neat!

[8]`https://inddecision.netlify.app/`

Currently, the suite compares RSDD to the SDD library and the CNF2OBDD library (for SDD and BDD compilation respectively). In each case, we compare the wall clock time for compiling the decision diagram from a DIMACS CNF (ignoring e.g. time to read the file from disk). We then vary the benchmark over various different CNFs (blending real-world and synthetic benchmarks) as well as different compilation strategies.

A subset of the results of these benchmarks are shown in Tables 1, 2, and 3. Each of these values is run over at 10 iterations; for the ratio, a higher number is better for RSDD. I ran these on my personal laptop (2022 ARM Macbook Pro, 16GB, M1).

CNF	SDD size	RSDD speedup
CM152A	138	0.25
S208.1	1422242	2.25
X2	2416	1.18
cht	6570	16.93

Table 1: RSDD vs SDD library — right-linear vtree (i.e. identical SDDs)

CNF	SDD circuit size	RSDD circuit size	RSDD speedup
CM152A	132	170	65.58
S208.1	2343	2262	68.24
X2	890	1666	22.56
cht	6127	2938	90.14

Table 2: RSDD vs SDD library — vtree heuristic (for RSDD: min-fill DTree). In this case, the vtrees are not the same.

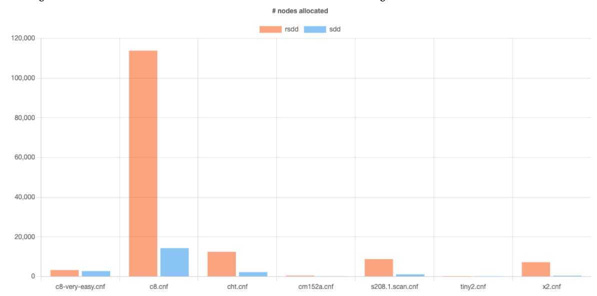
This is our first rigorous benchmark against other tools! It's nice to see that there are already cases where RSDD is very competitive; the heuristic for

CNF	BDD size	RSDD speedup
CM152A	138	0.13
S208.1	1422242	1.55
X2	2416	0.01
cht	6570	0.04

Table 3: RSDD vs CNF2OBDD library — linear varorder

finding the best VTree statically, the min-fill DTree, seems particularly effective. However, there is still nuance in this very small subset — in particular, both the SDD library and CNF2OBDD beat RSDD on select test cases.

The harness allows any CNF to be used as a benchmark (the repo contains more possible benchmark results; for CNFs larger than S208.1, we recommend a dedicated server). In addition, it has the option to output results as a JSON file; I've also included a small helper HTML page to render it on any computer. For example, here is one chart comparing the number of nodes allocated by RSDD versus the SDD library — observe that RSDD always allocates more.



The next natural step for the benchmark suite is to add more comparisons. CUDD is a good choice, though I'll need to implement a fair competitor manually (since, unlike the SDD library, there is no first-class CNF → BDD CLI tool).

In addition, I eventually plan on running the benchmarks on a set frequency, and automatically updating an online site (similar to Firefox's Are We Fast Yet benchmark [9]). This leads to reproducible claims about our performance that can be integrated into our docs.

4.3 Documentation and Tutorials

Prior to my work on RSDD, there were no user-facing tutorials or demos. This is a critical part of larger software adoption. Thus, I developed and published the first documentation site for the project at `https://neuppl.github.io/rsdd-docs/`.

The site is a work-in-progress project and is not yet completed. At the time of writing, I will highlight a handful of features:

- the site features live demos of RSDD using the WebAssembly demo infrastructure described in a previous subsection. As the user steps through tutorials and documentation, the goal is to have them be able to play with the corresponding featureset in RSDD.
- a core concept for BDDs and SDDs is the variable ordering (in the SDD case, the choice of vtree). To highlight the importance of this, I developed a mini-demo that compares side-by-side two SDDs compiled from the same CNF, but with different vtrees; the user can submit their own CNFs and select vtrees. This demo is currently visible on the homepage.
- there is a (slightly incomplete) tutorial that explains how to use the library from scratch.

The documentation site is implemented with Docusaurus[10], an open-source documentation framework that compiles both Markdown and React code into a static website. The primary documentation content is written in Markdown, which is easy to pick up and hopefully makes this documentation easy to maintain. However, the ability to step into React code allows me to implement various WebAssembly-based demos and provide interactivity to the documentation, which is not common among other research project documentation sites.

Out of all the contributions I've detailed, this has the most straightforward next steps. Over the next few weeks, I plan on:

[9]`https://arewefastyet.com/win10/benchmarks/overview?numDays=60`

[10]`https://docusaurus.io/`

- finishing the current basic tutorial
- writing more advanced tutorials: developing custom builders and pointer types, performing different queries (like marginal MAP)
- creating and designing more interactive demos alongside the documentation; e.g., looking at the importance of VarOrder for BDDs, annotating the graphs with semantic hash codes, comparing different caching strategies
- documenting other parts of the library
- documenting other platforms (ex WASM + TS, and eventually bindings for other languages)

Eventually, the goal is to treat RSDD like most mature open-source software projects, complete with versioned documentation and a comprehensive user guide.

5 Next Steps

In each section, I've highlighted future work related to my contributions to RSDD. During July and August, I will be a full-time research assistant at NeuPPL; my goal is to implement the vast majority of these goals, as well as additional items.

To summarize, I believe my next steps for this project broadly fall under three categories: novel compilation strategies, broadening the scope of RSDD, and maturing the project.

Novel compilation strategies encompasses the work around semantic hashing for SDDs, as well as extending it to other decision diagram data structures. More generally, this project demonstrates the potential for probabilistic tests to speed up computation and reduce allocated space. In addition, the completion of the large-scale refactoring makes it easier to experiment with orthogonal compilation strategies, such as varying different caching approaches or adding additional operation-specific optimization. Part of this work will be on the theory and algorithms side: applying probabilistic tests to new domains, potentially proving theoretical bounds, and implementing many algorithms in the library. The other part of this work will be in optimizing these compilation strategies to make them competitive with the state of the art.

There are also many other areas that I'm planning on contributing to. As we finalize the API for BDDs, SDDs, and the `BottomUpBuilder`, we will be able to provide bindings for other languages to use RSDD. There is also potential to develop an accompanying standalone program (similar to the SDD library or CNF2OBDD).

Finally, maturing the project often involves out-of-library improvements that are harder to dedicate research time towards. I would finish my current roadmap for the documentation website, implement more demos, and tightly integrate the benchmarking suite into the documentation platform (ex by auto-publishing benchmark results to the site). In addition, I plan on increasing the test coverage, particularly through doctests (which have a side effect of improving the reference API documentation). Similarly, I also plan on further fleshing out the benchmarks, including expanding the problem set and compared libraries (e.g. CUDD). There are also many other related RSDD projects, such as parallelizing compilation and garbage collection, that I'd be interested in working on.

I'd also be remiss if I didn't provide an overlap with my interest in CS education. I see three potential areas where my work in RSDD can provide better pedagogy around knowledge representation:

- demos can be used to build intuition and test hypotheses. Students could develop intuition on how CNFs or DNFs convert to various decision diagrams; why ordering heuristics matter; or see model counting happen live. This is similar to the use of SamIAm [11] in teaching Bayesian networks.
- an online RSDD code sandbox makes it easier for students to jump right in to knowledge compilation without dealing with installing various pieces of software. This is often a barrier with other BDD and SDD libraries, such as CUDD.

[11] `http://reasoning.cs.ucla.edu/samiam/`

RSDD's simple API — and lack of manual memory management — makes it easier for students to get started.

- the open-source nature of RSDD and its supporting software provide a more accessible platform for students to conduct their own research. This could be particularly helpful for graduate classes that are focused around class projects, but also more broadly relevant for any research that involves knowledge compilation. Unlike other libraries, all elements of RSDD — including datasets, benchmarks, documentation, and search heuristics — are completely open-source.

References

[1] Manuel Blum, Ashok K. Chandra, and Mark N. Wegman. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Inf. Process. Lett.*, 10:80–82, 1980.

[2] Adnan Darwiche. Sdd: A new canonical representation of propositional knowledge bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Two*, IJCAI'11, page 819–826. AAAI Press, 2011.

[3] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Int. Res.*, 17(1):229–264, sep 2002.

[4] Petr Fiser and Jan Schmidt. A comprehensive set of logic synthesis and optimization examples. 2016.

[5] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang. (OOPSLA)*, 2020.

[6] Guy Van den Broeck and Adnan Darwiche. On the role of canonicity in knowledge compilation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29(1), Feb. 2015.

[7] Michael Wachter and Rolf Haenni. Probabilistic equivalence checking with propositional dags. 2006.

[8] Michael Wachter and Rolf Haenni. Propositional dags: A new graph-based language for representing boolean functions. In *International Conference on Principles of Knowledge Representation and Reasoning*, 2006.