Binary Decision Diagrams

Theory, Implementation, and Applications

The bdd-rs contributors

December 19, 2025

Preface

0.1 Why This Guide Exists

In 1994, a single floating-point division bug cost Intel $475 million. The Pentium FDIV error slipped past traditional testing because exhaustive checking was impossible — billions of input combinations existed. After that expensive lesson, Intel turned to **formal verification**, and Binary Decision Diagrams became essential infrastructure.

BDDs are one of computer science's quiet success stories. They power hardware verification at Intel, AMD, and every major chip manufacturer. They enable SAT solvers to prune search spaces. They verify protocols, validate configurations, and solve combinatorial puzzles. Yet despite their importance, good learning resources are scarce.

This guide fills that gap. Whether you are a student encountering formal methods for the first time, an engineer building verification tools, or a researcher pushing the boundaries of symbolic computation, you will find what you need here.

0.2 What You Will Learn

By working through this guide, you will understand:

- **Theory**: The mathematical foundations — Boolean algebra, Shannon expansion, the canonicity theorem that makes BDDs magical.
- **Implementation**: How to build an efficient BDD library from scratch, including data structures, algorithms, and the non-obvious engineering decisions.
- **Applications**: Where BDDs shine — model checking, configuration management, combinatorial optimization — and where they struggle.
- **Trade-offs**: When to reach for BDDs versus SAT solvers, and how design choices (variable ordering, complement edges, caching strategies) affect performance by orders of magnitude.

0.3 How to Read This Guide

There is no single "right" path. Choose based on your goals:

Path 1: Theory First

Read Part I thoroughly, then Part II. Build understanding from first principles. Best for students and those who want the complete picture.

Path 2: Implementation Focus

Skim Part I for notation, dive into Part II. Reference Part III when you hit advanced techniques. Best for engineers building BDD-based systems.

Path 3: Application-Driven

Start with Part IV to see BDDs solving real problems. Backtrack to earlier parts when curiosity strikes. Best for practitioners with specific use cases in mind.

Path 4: Comparative Analysis

Jump to Part V for the lay of the land. Use earlier chapters as reference material. Best for those evaluating BDD libraries or architectural approaches.

0.4 The bdd-rs Library

This guide accompanies `bdd-rs`, a BDD library written in Rust. The code examples use Rust syntax, but the concepts are universal — they apply to CUDD, BuDDy, or any BDD implementation.

```rust
use bdd_rs::bdd::Bdd;

let bdd = Bdd::default();
let x = bdd.mk_var(1);
let y = bdd.mk_var(2);
let f = bdd.apply_and(x, -y);  // f = x ∧ ¬y

// O(1) checks after construction
assert!(!bdd.is_zero(f));  // Satisfiable
assert!(!bdd.is_one(f));   // Not a tautology
```

0.5 A Note on Style

This guide prioritizes **understanding** over encyclopedic coverage. When a choice exists between being comprehensive and being clear, clarity wins.

You will find:

- **Diagrams** that visualize concepts
- **Code** that shows how ideas become implementations
- **Examples** that ground abstractions in concrete problems
- **Insights** that explain **why**, not just **how**

0.6 Acknowledgments

BDDs emerged from decades of research, starting with Lee’s 1959 work on decision programs and crystallizing in Bryant’s landmark 1986 paper. This guide builds on that foundation and on the practical wisdom embodied in libraries like CUDD, BuDDy, and Sylvan.

We thank the formal methods community for creating such a rich field to explore.

Contents

PART I

Foundations

Chapter 1

Introduction

Imagine you could represent any Boolean formula — no matter how complex — as a compact diagram where checking if two formulas are equivalent takes a single pointer comparison. Where determining satisfiability requires examining just one node. Where counting all solutions is a single traversal.

This is the promise of Binary Decision Diagrams.

Since their refinement in the 1980s, BDDs have become one of computer science's most elegant success stories. They enabled Intel to verify microprocessors before fabrication, catching bugs that would have cost billions. They power configuration tools that validate millions of product combinations in milliseconds. They form the foundation of symbolic model checking — a technique so impactful that it earned its inventors the Turing Award.

This chapter takes you on a journey from the fundamental challenge of Boolean reasoning to the elegant solution that BDDs provide.

1.1 The Challenge of Boolean Reasoning

Boolean functions hide in plain sight. Every `if` statement in your code. Every logic gate in a processor. Every constraint in a configuration system. Every rule in a firewall policy.

The ubiquity of Boolean logic makes reasoning about it essential — and surprisingly difficult.

1.1.1 A Deceptively Simple Question

Consider this innocent-looking question:

✏️ **Example — The Equivalence Puzzle**

Are these two formulas the same function?

$$f = (a \land b) \lor (a \land c) \lor (b \land c) \tag{1}$$

$$g = (a \lor b) \land (a \lor c) \land (b \lor c) \tag{2}$$

Take a moment to think about it.

With just three variables, you could check all $2^3 = 8$ input combinations. But what if there were 100 variables? A million?

The brute-force approach hits a wall. With n variables, you face 2^n possible inputs — more than atoms in the universe for $n > 260$.

This exponential blowup is not a failure of imagination. Boolean satisfiability (SAT) is NP-complete. Equivalence checking is co-NP-complete. Unless P = NP, no shortcut exists for the general case.

And yet, engineers verify circuits with thousands of variables every day. How?

1.1.2 The Power of Representation

The secret lies in choosing the right **representation**.

Think of Roman numerals versus decimal notation. Both can represent any number, but try multiplying MCMXCIV by CDXLVII. The representation matters enormously.

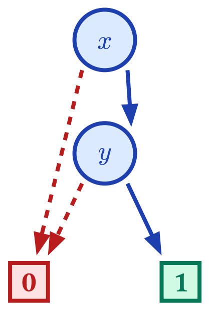
For Boolean functions, most representations have crippling weaknesses:

- **Truth tables** are canonical (unique) but exponentially large
- **CNF formulas** are compact but checking equivalence is co-NP-complete
- **Circuits** are efficient to build but hard to analyze

BDDs hit a sweet spot: they are often compact **and** canonical **and** support efficient operations. When they work, they work spectacularly well.

1.2 What is a BDD?

A **Binary Decision Diagram** is a way of drawing a Boolean function as a flowchart.



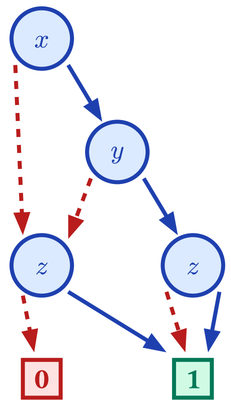
Reading the BDD:
Start at x (root)
If $x = 0$: follow dashed line $\to$ **0**
If $x = 1$: go to y
 If $y = 0$: follow dashed $\to$ **0**
 If $y = 1$: follow solid $\to$ **1**

Figure 1: BDD for $x \wedge y$: the function outputs 1 only when both inputs are 1.

Here is how to read a BDD:

1. **Start at the root** — the topmost circle
2. **Check the variable** — is it true (1) or false (0)?
3. **Follow the edge** — solid for true, dashed for false
4. **Repeat** until you reach a square terminal
5. **The terminal's value** is your answer

The magic happens when you have **structure sharing**. Consider the function $(x \wedge y) \vee z$:



Notice: both z nodes *share* the same terminals.

This sharing is the key to BDD efficiency.

Figure 2: BDD for $(x \wedge y) \vee z$ showing structure sharing.

Without sharing, a decision tree for n variables needs up to 2^n leaves. With sharing, a BDD often needs far fewer nodes — sometimes polynomially many, sometimes even constant.

1.3 A Brief History

The story of BDDs is one of incremental insight building to breakthrough.

1.3.1 The Early Days (1959–1978)

In 1959, **C. Y. Lee** described "binary decision programs" for representing switching circuits — essentially, BDDs before the name existed. His insight was that any Boolean function could be represented as a binary tree of if-then-else decisions.

In 1978, **S. B. Akers** formalized the structure and coined the term "Binary Decision Diagram." But these early BDDs had a problem: the same function could be drawn in many different ways. Checking if two BDDs represented the same function required expensive graph isomorphism tests.

1.3.2 The Bryant Revolution (1986)

The transformation came from **Randal Bryant**, then at Carnegie Mellon. His 1986 paper introduced two deceptively simple restrictions:

1. **Order the variables** — every path from root to terminal encounters variables in the same sequence
2. **Reduce the diagram** — merge identical subgraphs and eliminate redundant nodes

These constraints create **Reduced Ordered Binary Decision Diagrams** (ROBDDs), with a stunning property:

💡 **Key Insight**

For a fixed variable ordering, every Boolean function has **exactly one** ROBDD.

This means: two functions are identical if and only if their BDDs are pointer-equal. Equivalence checking becomes a single comparison.

Bryant also provided efficient algorithms for combining BDDs. Computing $f \wedge g$ or $f \vee g$ takes time proportional to $|f| \times |g|$ — the product of their sizes, not exponential in variables.

1.3.3 The Verification Revolution (1987–1995)

The impact was immediate and profound.

In 1987, a team including **Edmund Clarke** (later a Turing Award recipient) demonstrated **symbolic model checking**. They verified systems with 10^{20} states — astronomically beyond what explicit enumeration could handle.

Hardware companies took notice. Intel began using BDD-based tools to verify processor designs. The infamous Pentium FDIV bug of 1994 — which cost Intel $475 million — accelerated adoption of formal verification. BDDs became essential infrastructure.

1.3.4 Maturity and Beyond (1995–Present)

By the mid-1990s, BDDs were a standard tool, but their limitations were better understood:

- Some functions (like integer multiplication) have exponentially large BDDs **regardless** of variable ordering
- Finding the best variable ordering is itself NP-hard
- Memory usage can be unpredictable

These limitations spurred alternatives:

- **SAT solvers** excel at finding single solutions quickly
- **BDD variants** like ZDDs handle sparse sets efficiently
- **Hybrid methods** combine the strengths of multiple approaches

Today, BDDs remain essential for problems requiring canonicity, counting, or symbolic state-space exploration.

1.4 What Makes BDDs Special?

Three properties distinguish BDDs from other Boolean function representations:

1.4.1 Canonicity

For a fixed variable ordering, every Boolean function has exactly one reduced ordered BDD. This property is powerful:

- **Equivalence checking**: Two BDDs represent the same function if and only if they are identical. With hash consing, this reduces to pointer comparison: $O(1)$.
- **Satisfiability**: A function is unsatisfiable if and only if its BDD is the 0-terminal. This is also $O(1)$ after construction.
- **Tautology checking**: A function is a tautology if and only if its BDD is the 1-terminal.

No other compact representation offers these constant-time queries. Truth tables are canonical but exponentially large. CNF and DNF are compact but non-canonical.

1.4.2 Efficient Operations

BDD operations have polynomial complexity in the sizes of the input BDDs:

Operation	Complexity
Negation (with complement edges)	$O(1)$

Operation	Complexity
AND, OR, XOR, etc.	$O(\|f\| \cdot \|g\|)$
Equivalence check	$O(1)$
Satisfiability check	$O(1)$
Model counting	$O(\|f\|)$

The $O(|f| \cdot |g|)$ bound for binary operations comes from memoization: each pair of nodes from the two BDDs is processed at most once.

1.4.3 Sharing

BDDs naturally share common subfunctions. When building $f \land g$ and $f \lor g$, the subgraph for f is constructed once and reused. This sharing arises automatically from **hash consing**: before creating a node, we check if an identical node already exists.

In a manager-centric implementation like `bdd-rs`, all BDDs share a single node pool. Memory is proportional to the total number of distinct subfunctions, not the total number of BDDs.

1.5 When BDDs Work Well

BDDs excel when the Boolean function has structure that permits compact representation:

Sequential circuits and finite-state machines. Transition relations of digital circuits often have small BDDs because related bits are tested together. State reachability can be computed symbolically, avoiding enumeration of individual states.

Configuration constraints. Feature models and product line constraints typically yield manageable BDDs. The hierarchical structure of features often suggests good variable orderings.

Symmetric and threshold functions. Functions like "at least k of n variables are true" have polynomial-size BDDs. Many constraints arising in combinatorial problems have this form.

Problems requiring counting or enumeration. When you need to count satisfying assignments or enumerate all solutions, BDDs shine. SAT solvers can find **one** solution quickly but struggle with **all** solutions.

1.6 When BDDs Struggle

BDDs have well-known limitations:

Integer multiplication. The function "output bits of n-bit multiplier" requires exponential BDD size regardless of variable ordering. This is not a limitation of the algorithm but a fundamental property of the function.

Large unstructured problems. Random Boolean functions or problems without exploitable structure tend to produce large BDDs.

Dynamic problems. If the optimal variable ordering changes as constraints are added, maintaining good BDD size requires expensive reordering operations.

Memory consumption. BDD operations can create many intermediate nodes. Without garbage collection, memory can grow rapidly.

> ⚠️ **No Silver Bullet**
>
> BDDs are not universally superior to SAT solvers or other techniques. The choice depends on the problem structure and the queries needed. For single satisfiability queries on large formulas, modern SAT solvers often win. For repeated queries, counting, or symbolic state-space exploration, BDDs often win.

1.7 The bdd-rs Library

This guide accompanies `bdd-rs`, a BDD library written in Rust. Its design reflects lessons from decades of BDD research:

```rust
use bdd_rs::bdd::Bdd;

// Create a BDD manager
let bdd = Bdd::default();

// Variables are 1-indexed
let x = bdd.mk_var(1);
let y = bdd.mk_var(2);
let z = bdd.mk_var(3);

// Build a formula: (x ∧ y) ∨ z
let f = bdd.apply_or(bdd.apply_and(x, y), z);

// Constant-time queries
assert!(!bdd.is_zero(f));  // satisfiable?
assert!(!bdd.is_one(f));   // tautology?

// Count solutions (8 total assignments, how many satisfy f?)
let count = bdd.sat_count(f, 3);
println!("Solutions: {}", count);  // 5
```

Key design choices in `bdd-rs`:

- **Manager-centric architecture**: All operations go through the `Bdd` manager, ensuring hash consing and canonical form.
- **Complement edges**: Negation is $O(1)$, implemented as a single bit flip.
- **Type-safe handles**: `Ref` is a 32-bit handle; accidental misuse is caught at compile time.
- **Rust's safety guarantees**: Memory safety without garbage collector overhead.

1.8 Guide Overview

The remainder of this guide is organized as follows:

Part I establishes the theoretical foundations: Boolean functions, Shannon expansion, the formal BDD definition, the canonicity theorem, and core algorithms.

Part II covers implementation: manager architecture, node representation, unique tables, the Apply algorithm, caching, and complement edges.

Part III explores advanced topics: variable ordering, garbage collection, quantification, and BDD variants.

Part IV demonstrates applications: model checking, combinatorial problems, symbolic execution, and configuration management.

Part V surveys the ecosystem: library comparisons, design trade-offs, and future directions.

Each chapter builds on previous ones, but readers with specific interests can skip ahead using the cross-references provided.

Chapter 2

Boolean Functions

Before diving into BDDs, we need to establish the mathematical ground we're standing on. This chapter reviews Boolean algebra and introduces the key concepts — cofactors, Shannon expansion, and the representation problem — that BDDs are designed to solve.

If you're comfortable with Boolean algebra, skim to Section 2.3; that's where the BDD story really begins.

2.1 Boolean Algebra Foundations

2.1.1 The Boolean Domain

The Boolean domain $\mathbb{B} = \{0, 1\}$ contains exactly two values: **false** (0) and **true** (1). Everything in Boolean computation reduces to these two primitives.

📘 **Definition (Boolean Function)**

A **Boolean function** of n variables is a mapping $f : \mathbb{B}^n \to \mathbb{B}$. Given n Boolean variables $x_1, x_2, \dots, x_n$, the function f assigns a truth value to each of the 2^n possible input combinations.

How many Boolean functions of n variables exist? Each function is determined by its output on 2^n inputs, and each output can be 0 or 1. Thus, there are exactly 2^{2^n} distinct Boolean functions of n variables.

Variables	Inputs	Functions
1	2	4
2	4	16
3	8	256
4	16	$65,536$
5	32	4.3×10^9
10	$1,024$	$\approx 10^{308}$

The explosive growth explains why naive enumeration is impractical. Even with 10 variables, there are more Boolean functions than atoms in the observable universe.

2.1.2 Basic Operations

The fundamental Boolean operations are:

Negation (NOT): $\neg x$ flips the value.

$$\neg 0 = 1, \quad \neg 1 = 0 \tag{3}$$

Conjunction (AND): $x \wedge y$ is true when both operands are true.

$$x \wedge y = 1 \text{ iff } x = 1 \text{ and } y = 1 \tag{4}$$

Disjunction (OR): $x \vee y$ is true when at least one operand is true.

$$x \vee y = 1 \text{ iff } x = 1 \text{ or } y = 1 \tag{5}$$

Exclusive OR (XOR): $x \oplus y$ is true when exactly one operand is true.

$$x \oplus y = 1 \text{ iff } x \neq y \tag{6}$$

Implication: $x \to y$ is false only when x is true and y is false.

$$x \to y = \neg x \vee y \tag{7}$$

Equivalence (XNOR): $x \leftrightarrow y$ is true when both operands have the same value.

$$x \leftrightarrow y = \neg(x \oplus y) \tag{8}$$

2.1.3 Algebraic Laws

Boolean algebra satisfies many useful identities:

Commutativity:

$$x \wedge y = y \wedge x, \quad x \vee y = y \vee x \tag{9}$$

Associativity:

$$(x \wedge y) \wedge z = x \wedge (y \wedge z), \quad (x \vee y) \vee z = x \vee (y \vee z) \tag{10}$$

Distributivity:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \tag{11}$$

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \tag{12}$$

De Morgan's Laws:

$$\neg(x \wedge y) = \neg x \vee \neg y \tag{13}$$

$$\neg(x \vee y) = \neg x \wedge \neg y \tag{14}$$

Absorption:

$$x \wedge (x \vee y) = x, \quad x \vee (x \wedge y) = x \tag{15}$$

Complement:

$$x \wedge \neg x = 0, \quad x \vee \neg x = 1 \tag{16}$$

These laws enable algebraic manipulation of Boolean expressions, but they do not directly solve the representation problem.

2.2 Representations of Boolean Functions

A Boolean function can be represented in many ways. Each representation has trade-offs in space, canonicity, and operation efficiency.

2.2.1 Truth Tables

The most explicit representation lists the function's output for every input:

✏️ **Example — Truth Table for XOR**

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Truth tables are **canonical** — each function has exactly one truth table. Equivalence checking is straightforward: compare tables entry by entry.

However, truth tables require 2^n entries for n variables. This exponential size makes them impractical for functions with more than about 20 variables.

2.2.2 Boolean Formulas

A Boolean formula is a syntactic expression using variables and operations:

$$f = (x_1 \land x_2) \lor (\neg x_1 \land x_3) \tag{17}$$

Formulas can be compact, but they are **non-canonical** — many different formulas represent the same function. For example, these all represent the same function:

$$x \land y = y \land x = \neg(\neg x \lor \neg y) = (x \lor 0) \land (y \lor 0) \tag{18}$$

Checking formula equivalence requires reasoning about all possible simplifications, which is co-NP-complete in general.

2.2.3 Normal Forms

Normal forms impose structure on Boolean formulas.

Disjunctive Normal Form (DNF): An OR of ANDs (sum of products).

$$f = (x_1 \land x_2) \lor (\neg x_1 \land x_3) \lor (x_2 \land x_3) \tag{19}$$

Conjunctive Normal Form (CNF): An AND of ORs (product of sums).

$$f = (x_1 \lor x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor \neg x_3) \tag{20}$$

Normal forms are useful for specific algorithms (SAT solvers work on CNF), but they are still non-canonical. The same function can have multiple DNF or CNF representations.

💡 **Key Insight**

The representation dilemma:

- **Truth tables** are canonical but exponentially large.
- **Formulas** are compact but non-canonical.
- **BDDs** achieve canonicity with size often polynomial in practice.

2.3 Shannon Expansion

The key to BDDs is the **Shannon expansion**, which decomposes a function by "case splitting" on a variable.

2.3.1 Cofactors

📘 **Definition (Cofactor)**

The **cofactor** of f with respect to variable x_i set to value $b \in \{0, 1\}$ is the function:

$$f|_{x_i=b} = f(x_1, ..., x_{i-1}, b, x_{i+1}, ..., x_n) \tag{21}$$

We write $f|_{x_i=0}$ as the **negative cofactor** (or **low cofactor**) and $f|_{x_i=1}$ as the **positive cofactor** (or **high cofactor**).

The cofactor $f|_{x_i=b}$ is the function f restricted to the case where x_i has value b. It is a function of $n - 1$ variables (since x_i is fixed).

✏️ **Example — Computing Cofactors**

Let $f = (x \land y) \lor z$. Then:

- $f|_{x=0} = (0 \land y) \lor z = z$
- $f|_{x=1} = (1 \land y) \lor z = y \lor z$
- $f|_{y=0} = (x \land 0) \lor z = z$
- $f|_{y=1} = (x \land 1) \lor z = x \lor z$

2.3.2 The Shannon Expansion Theorem

📐 **Theorem (Shannon Expansion)**

Every Boolean function $f(x_1, ..., x_n)$ can be decomposed with respect to any variable x_i:

$$f = \left(\neg x_i \land f|_{x_i=0}\right) \lor \left(x_i \land f|_{x_i=1}\right) \tag{22}$$

Equivalently, using the if-then-else notation:

$$f = \text{ite}\left(x_i, f|_{x_i=1}, f|_{x_i=0}\right) \tag{23}$$

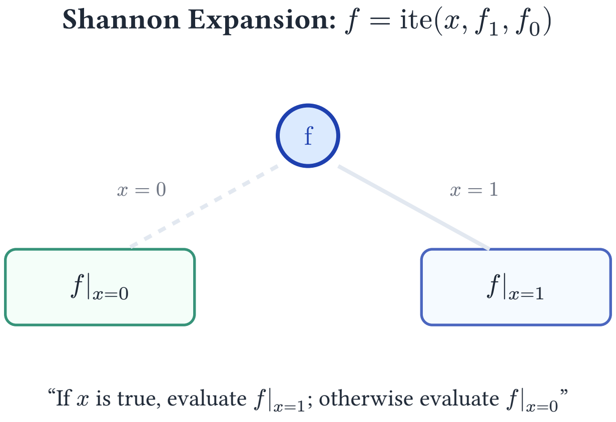

Shannon Expansion: $f = \text{ite}(x, f_1, f_0)$
f
$x = 0$
$x = 1$
$f|_{x=0}$
$f|_{x=1}$
"If x is true, evaluate $f|_{x=1}$; otherwise evaluate $f|_{x=0}$"

Figure 3: Shannon expansion decomposes f into two subfunctions based on a variable's value.

> *Proof.* Consider any assignment to the variables. If $x_i = 0$, then $\neg x_i = 1$ and the formula evaluates to $f|_{x_i=0}$, which equals f when $x_i = 0$. If $x_i = 1$, then $x_i = 1$ and the formula evaluates to $f|_{x_i=1}$, which equals f when $x_i = 1$. In both cases, the formula equals f. □

The Shannon expansion has a natural interpretation: "if x_i is true, then $f|_{x_i=1}$, else $f|_{x_i=0}$."

2.3.3 Recursive Structure

Applying Shannon expansion recursively yields a **decision tree**:

1. Start with f.
2. Pick a variable x_1 and decompose: two subfunctions $f|_{x_1=0}$ and $f|_{x_1=1}$.
3. For each subfunction, pick the next variable x_2 and decompose again.
4. Continue until reaching constant functions (0 or 1).

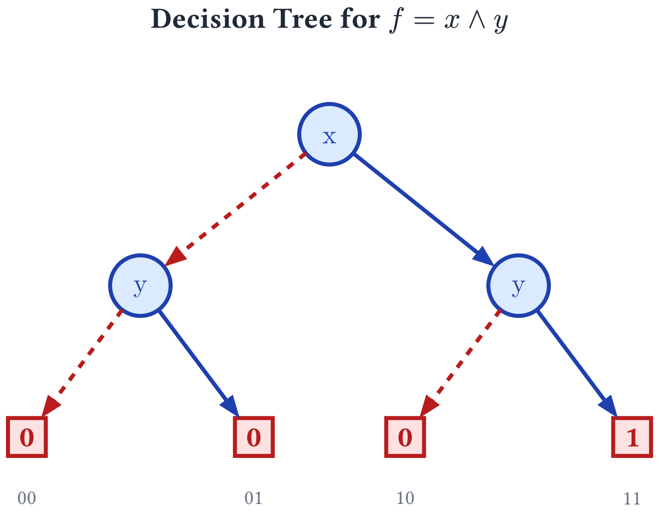


Figure 4: A decision tree for $x \wedge y$ has $2^n = 4$ leaves — one for each input combination.

This produces a binary tree with 2^n leaves, one for each input combination.

The key insight leading to BDDs: **many subtrees are identical**. If $f|_{x_1=0,x_2=1}$ equals $f|_{x_1=1,x_2=0}$, we can share a single subtree for both. This sharing is what transforms exponential decision trees into potentially compact BDDs.

2.4 Function Equivalence

Two Boolean functions f and g are **equivalent** (written $f \equiv g$) if they produce the same output on every input:

$$f \equiv g \text{ iff } \forall x \in \mathbb{B}^n : f(x) = g(x) \tag{24}$$

2.4.1 The Equivalence Checking Problem

Given two representations of Boolean functions, determine if they represent the same function.

With truth tables: Compare entry by entry. Complexity: $O(2^n)$ time and space.

With formulas: In general, this is co-NP-complete. Even syntactically different formulas can represent the same function.

With BDDs (canonical): Compare pointers. If two BDDs are constructed in the same manager with the same variable ordering, they represent the same function if and only if they are the same node. Complexity: $O(1)$.

💡 **Key Insight**

Canonicity transforms equivalence checking from a hard problem (co-NP-complete) into a trivial one ($O(1)$). This is the fundamental reason BDDs are powerful for verification.

2.4.2 Why Equivalence Matters

Equivalence checking appears throughout computer science:

- **Circuit verification**: Does an optimized circuit compute the same function as the specification?
- **Compiler optimization**: Is the optimized code equivalent to the original?
- **Theorem proving**: Are two logical formulas equivalent?
- **Test generation**: Does the implementation match the specification?

Any technique that makes equivalence checking efficient has broad applicability.

2.5 The Representation Problem

We have seen three representations:

Representation	Canonical?	Space	Equivalence
Truth table	Yes	$O(2^n)$	$O(2^n)$
Boolean formula	No	Variable	co-NP-complete
BDD (ROBDD)	Yes	Variable†	$O(1)$

† BDD size ranges from constant (for simple functions) to exponential (for multiplication), but is often polynomial for structured functions.

No representation is universally best. Truth tables guarantee polynomial-time operations but have exponential space. Formulas can be compact but have hard equivalence checking. BDDs occupy a middle ground: canonical representation with size that depends on the function's structure.

The BDD "gamble" is that many practical functions have compact BDDs. Decades of experience in verification, synthesis, and optimization have shown this gamble often pays off — but not always.

2.5.1 Preview: The BDD Solution

In the next chapter, we define BDDs formally. The key ideas are:

1. Apply Shannon expansion with a **fixed variable ordering**.
2. **Share** identical subfunctions (hash consing).
3. **Eliminate** redundant tests (where both branches lead to the same place).

These constraints yield the **Reduced Ordered Binary Decision Diagram** (ROBDD), which is canonical for any fixed ordering. The challenge then becomes managing the variable ordering to keep BDDs small — a topic we address in Section 12.

Chapter 3

BDD Definition and Structure

In the previous chapter, we saw that Boolean functions can be represented in many ways. Shannon expansion gives us a recursive decomposition, but naively applying it yields exponential-size decision trees.

The breakthrough insight of BDDs is simple yet profound: **share identical substructures**. By merging duplicate subtrees, we transform an exponential tree into a compact directed acyclic graph (DAG).

3.1 From Decision Trees to Decision Diagrams

Let's trace the evolution from trees to diagrams. This progression reveals why BDDs have the properties they do.

3.1.1 Decision Trees

A **decision tree** for a Boolean function $f(x_1, ..., x_n)$ is a rooted binary tree where:

- Each **internal node** is labeled with a variable x_i
- Each internal node has two children: **low** (for $x_i = 0$) and **high** (for $x_i = 1$)
- Each **leaf** is labeled with a Boolean constant (0 or 1)

To evaluate the function on an assignment, we start at the root and follow edges based on variable values until reaching a leaf. The leaf's label gives the function value.

✏️ **Example — Decision Tree for Majority**

Consider the majority function $\mathrm{Maj}(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (x \wedge z)$, which outputs 1 when at least two inputs are 1.

A decision tree testing variables in order x, y, z has:

- Root tests x
- Second level tests y (two nodes)
- Third level tests z (four nodes)
- Eight leaves with values based on majority

Even this simple function requires $2^3 - 1 = 7$ internal nodes plus 8 leaves.

The problem is clear: a decision tree for n variables has up to 2^n leaves. This exponential growth makes decision trees impractical for functions with many variables.

3.1.2 The Key Insight: Structure Sharing

The breakthrough comes from observing that decision trees contain redundancy. Many subtrees compute the **same** subfunction and could be merged.

Consider evaluating $\mathrm{Maj}(x, y, z)$ when we have already fixed $x = 0$ and $y = 1$. The remaining function is just z. Now consider fixing $x = 1$ and $y = 0$ — again, the remaining function is z. These two subtrees are **isomorphic**; they compute the same thing.

> 💡 **Key Insight**
>
> By merging isomorphic subtrees, we transform a tree into a DAG. The more structure a function has, the more sharing is possible.

3.1.3 From Tree to DAG

The transformation from decision tree to decision diagram proceeds bottom-up:

1. Start with a decision tree
2. Identify leaves with the same value and merge them (yielding just two terminal nodes: 0 and 1)
3. Identify internal nodes with the same variable and same children — merge them
4. Repeat until no more merging is possible

This process is called **reduction**, and the resulting structure is a Binary Decision Diagram.

3.2 Binary Decision Diagrams

We now give the formal definition of BDDs and establish notation used throughout this guide.

> 📘 **Definition (Binary Decision Diagram)**
>
> A **Binary Decision Diagram (BDD)** is a rooted directed acyclic graph $G = (V, E)$ where:
> - $V = V_T \cup V_D$ partitions into **terminal** and **decision** nodes
> - **Terminal nodes** $V_T = \{0, 1\}$ have no outgoing edges
> - Each **decision node** $v \in V_D$ is labeled with variable $\mathrm{var}(v) \in \{x_1, ..., x_n\}$ and has exactly two outgoing edges:
>   - $\mathrm{low}(v) \in V$: the **low child** (taken when $\mathrm{var}(v) = 0$)
>   - $\mathrm{high}(v) \in V$: the **high child** (taken when $\mathrm{var}(v) = 1$)
> - There is a distinguished **root node** $r \in V$

> 📘 **Definition (Semantics of BDDs)**
>
> A BDD with root r represents a Boolean function $f_r : \mathbb{B}^n \to \mathbb{B}$ defined recursively:

$$f_v(\boldsymbol{x}) = \begin{cases} 0 & \text{if } v = 0 \\ 1 & \text{if } v = 1 \\ \left(\overline{x_i} \wedge f_{\text{low}(v)}(\boldsymbol{x})\right) \vee \left(x_i \wedge f_{\text{high}(v)}(\boldsymbol{x})\right) & \text{if } \text{var}(v) = x_i \end{cases} \tag{25}$$

This is precisely Shannon expansion: $f = \overline{x_i} \cdot f|_{x_i=0} + x_i \cdot f|_{x_i=1}$.

3.2.1 Graphical Conventions

Throughout this guide, we draw BDDs using standard conventions:

- **Decision nodes**: Circles labeled with variable names
- **Terminal nodes**: Squares labeled 0 or 1 (sometimes drawn as $\perp$ and $\top$)
- **High edges**: Solid lines (taken when variable is 1)
- **Low edges**: Dashed lines (taken when variable is 0)
- Variables at the same **level** are drawn at the same vertical position

3.3 Ordered BDDs (OBDDs)

BDDs become much more useful when we impose an ordering constraint on variables.

📘 **Definition (Variable Ordering)**

A **variable ordering** is a total order π on the variables $\{x_1, \ldots, x_n\}$. We write $x_i <_\pi x_j$ to mean x_i comes before x_j in the ordering.

📘 **Definition (Ordered BDD)**

A BDD is **ordered** (OBDD) with respect to variable ordering π if on every path from the root to a terminal, variables are encountered in increasing order according to π.

Formally: for every decision node v with decision child u (either low or high), if both are decision nodes, then $\text{var}(v) <_\pi \text{var}(u)$.

The ordering constraint has profound implications:

- Variables can be **skipped** on a path (if the function does not depend on them in that branch)
- Variables can **never repeat** on a path
- The **same** variable appears at the **same level** throughout the BDD

i Why Ordering Matters

Without ordering, two BDDs for the same function could have completely different structures, making comparison difficult. Ordering is the first step toward canonicity.

3.3.1 The Impact of Variable Ordering

Different orderings can yield dramatically different BDD sizes for the same function.

✏️ **Example — Ordering Impact**

Consider $f = (x_1 \land y_1) \lor (x_2 \land y_2)$, which is true when at least one (x_i, y_i) pair is both true.

With ordering $x_1 < y_1 < x_2 < y_2$ (interleaved), the BDD has **linear** size in the number of pairs.

With ordering $x_1 < x_2 < y_1 < y_2$ (grouped), the BDD can have **exponential** size.

The difference arises because interleaved ordering allows decisions about each pair to be made together, while grouped ordering requires remembering partial information across many levels.

This sensitivity to ordering is both a strength and a weakness of BDDs:
- **Strength**: A good ordering can yield very compact representations
- **Weakness**: Finding optimal orderings is itself NP-hard

We discuss variable ordering heuristics in detail in Section 12.

3.4 Reduced BDDs (ROBDDs)

The final ingredient for canonicity is **reduction** — eliminating all redundancy.

📘 **Definition (Reduced Ordered BDD)**

An OBDD is **reduced** (ROBDD) if it satisfies two properties:
1. **No redundant tests**: For every decision node v, we have $\text{low}(v) \neq \text{high}(v)$
2. **No duplicate nodes**: No two distinct nodes have the same variable, low child, and high child

Equivalently: the BDD is maximally shared and contains no unnecessary nodes.

3.4.1 Reduction Rules

The two reduction properties correspond to two reduction rules:

Rule 1: Eliminate Redundant Tests

If a node v has $\text{low}(v) = \text{high}(v) = u$, then v is redundant. The function computed by v is $\overline{x} \cdot f_u + x \cdot f_u = f_u$, independent of x. We can redirect all edges pointing to v to point to u instead, then remove v.

Rule 2: Merge Isomorphic Subgraphs

If two distinct nodes v and w have the same variable and children: $\text{var}(v) = \text{var}(w)$, $\text{low}(v) = \text{low}(w)$, $\text{high}(v) = \text{high}(w)$

Then they compute the same function and can be merged. We keep one and redirect all edges to the other.

⚙️ **Algorithm: Reduction Procedure**

```
To reduce an OBDD:
```

```
1. Process nodes bottom-up (from terminals toward root)
2. For each node $v$:
      • If $\text{low}(v) = \text{high}(v)$, replace $v$ with its child (Rule 1)
      • Otherwise, check if an equivalent node already exists; if so,
        merge (Rule 2)
3. The result is a reduced OBDD
```

3.4.2 The Unique Table

In practice, reduction is achieved by maintaining a **unique table** — a hash table that maps (var, low, high) triples to nodes. When creating a node, we first check if it already exists. This ensures duplicate nodes are never created in the first place.

The unique table is fundamental to BDD implementations and is covered in detail in Section 8.

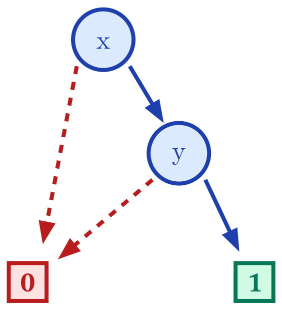
3.5 Visual Examples

Let us see these concepts in action with concrete examples.

3.5.1 Example: Conjunction ($x \wedge y$)

The function $f(x, y) = x \wedge y$ with ordering $x < y$:

- If $x = 0$: output is 0 regardless of y
- If $x = 1$: output is y



Evaluation paths:
$x = 0$: $\rightarrow 0$ (short-circuit)
$x = 1, y = 0$: $\rightarrow 0$
$x = 1, y = 1$: $\rightarrow 1$ ✓

Figure 5: ROBDD for $x \wedge y$. The y-node is only reached when $x = 1$.

Notice that the y-node is only reached when $x = 1$, reflecting short-circuit evaluation. The BDD has just 2 decision nodes — much smaller than the $2^2 = 4$ leaves of a decision tree.

3.5.2 Example: Exclusive Or ($x \oplus y$)

The function $f(x, y) = x \oplus y$ with ordering $x < y$:

- If $x = 0$: output is y
- If $x = 1$: output is $\neg y$

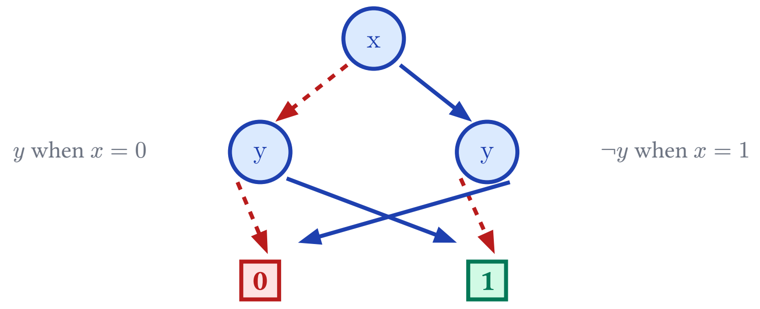


Figure 6: ROBDD for $x \oplus y$. Two y-nodes are needed because the subfunctions differ.

The ROBDD requires two y-nodes because y and $\neg y$ are different functions. No reduction is possible here — this is the minimal representation.

i XOR and Complement Edges

With **complement edges** (covered in Section 11), XOR can share structure with equivalence. The left y-node becomes the right one with a complement marker, halving the size.

3.5.3 Example: Majority Function

The majority function $\mathrm{Maj}(x, y, z)$ outputs 1 when at least two inputs are 1:

$$\mathrm{Maj}(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (x \wedge z) \tag{26}$$

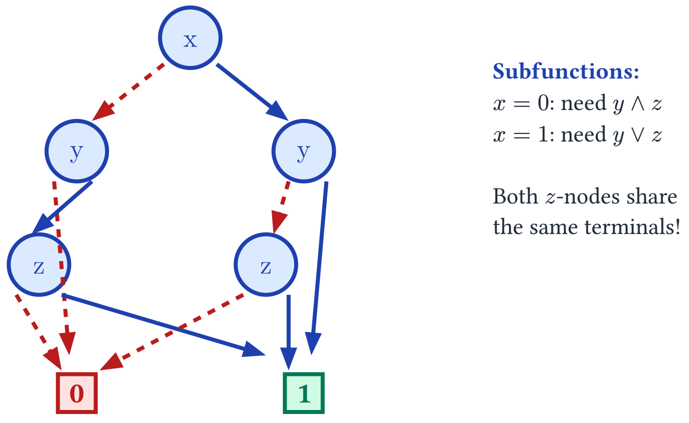


Figure 7: ROBDD for majority function $\mathrm{Maj}(x, y, z)$ with ordering $x < y < z$.

Interestingly, both z-nodes output the same value: z itself. The only difference is which paths reach them. With complement edges, further sharing would be possible.

3.6 Graph Properties and Metrics

Several metrics characterize BDD complexity:

📘 **Definition (BDD Size)**

The **size** of a BDD is the number of nodes, typically counting decision nodes only (excluding terminals). We denote the size of BDD representing function f as $|f|$ or $\text{size}(f)$.

📘 **Definition (BDD Width)**

The **width** at level i is the number of nodes with variable x_i. The **maximum width** is the maximum over all levels.

📘 **Definition (BDD Depth)**

The **depth** (or height) is the length of the longest path from root to a terminal. For ROBDDs with n variables, depth is at most n.

3.6.1 Size Bounds

The size of an ROBDD is bounded:

- **Lower bound**: 1 (for constant functions 0 or 1)
- **Upper bound**: Depends on the function and ordering

For **any** function and **any** ordering, the ROBDD has at most 2^n nodes (one per possible subfunction). However, many practical functions have polynomial-size BDDs with good orderings.

📐 **Theorem (Size Hierarchy)**

There exist functions with the following BDD sizes (for optimal orderings):

- $O(1)$: Constant functions, single variables
- $O(n)$: AND, OR, linear threshold functions, symmetric functions
- $O(n^2)$: Addition, comparison
- $O(2^n)$: Multiplication (output bits), hidden weighted bit

The size can vary exponentially between different orderings for the same function.

3.7 Summary

We have now established the formal foundation:

i Key Definitions

- **BDD**: A DAG with decision and terminal nodes representing a Boolean function via Shannon expansion
- **OBDD**: A BDD where variables appear in consistent order on all paths
- **ROBDD**: An OBDD with no redundant tests and no duplicate nodes

ROBDDs are **canonical**: two ROBDDs with the same ordering represent the same function if and only if they are identical.

The canonicity theorem (proved in Section 4) makes ROBDDs uniquely powerful. In the next chapter, we prove this theorem and explore its consequences for equivalence checking and other operations.

Chapter 4

Canonical Form and Uniqueness

Canonicity is the property that makes BDDs genuinely useful, not just compact. A representation is **canonical** if each function has exactly one representation — no exceptions, no ambiguity.

For reduced ordered BDDs with a fixed variable ordering, this property holds. The implications are profound: checking if two functions are equivalent reduces to checking if two pointers are equal.

This chapter proves the canonicity theorem and explores its far-reaching consequences.

4.1 The Canonicity Theorem

We state the central theorem precisely before proving it.

📐 **Theorem (Canonicity of ROBDDs)**

Let π be a fixed variable ordering on $\{x_1, \dots, x_n\}$. Then:

1. Every Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ has a unique ROBDD with respect to π
2. Two ROBDDs (with the same ordering) are structurally identical if and only if they represent the same function

Equivalently: there is a bijection between Boolean functions on n variables and ROBDDs with ordering π.

This theorem has two remarkable consequences:

- **Equivalence checking is trivial**: $f \equiv g$ if and only if their ROBDDs are identical
- **Satisfiability is trivial**: f is satisfiable if and only if its ROBDD is not the 0-terminal

Let us build toward the proof.

4.2 Proof of Canonicity

The proof proceeds by structural induction on the number of variables the function depends on. We show that the reduction rules uniquely determine the ROBDD structure.

4.2.1 Preliminaries

📘 **Definition (Essential Variables)**

A variable x_i is **essential** to function f if $f|_{x_i=0} \neq f|_{x_i=1}$, i.e., the function value changes when x_i changes (for some assignment to other variables).

A function depends only on its essential variables. Non-essential variables can be ignored in the ROBDD (this is what the "no redundant tests" rule achieves).

Lemma (Shannon Expansion Uniqueness)

Every Boolean function f can be uniquely decomposed as:

$$f = \overline{x_i} \cdot f|_{x_i=0} + x_i \cdot f|_{x_i=1} \tag{27}$$

where $f|_{x_i=0}$ and $f|_{x_i=1}$ are unique subfunctions not depending on x_i.

Proof. The cofactors $f|_{x_i=0}$ and $f|_{x_i=1}$ are defined pointwise and thus unique. The expansion follows from the definition of Boolean functions. □

4.2.2 Base Case: Constant Functions

For constant functions $f \equiv 0$ and $f \equiv 1$:

- They have no essential variables
- Their ROBDD is the respective terminal node (0 or 1)
- These are trivially unique

4.2.3 Inductive Step

Assume the theorem holds for all functions with fewer than k essential variables. Consider a function f with k essential variables.

Let x_i be the smallest essential variable according to ordering π. By Shannon expansion:

$$f = \overline{x_i} \cdot f|_{x_i=0} + x_i \cdot f|_{x_i=1} \tag{28}$$

Since x_i is essential, $f|_{x_i=0} \neq f|_{x_i=1}$. Both cofactors have at most $k - 1$ essential variables (they don't depend on x_i).

By the induction hypothesis, $f|_{x_i=0}$ and $f|_{x_i=1}$ have unique ROBDDs, call them B_0 and B_1.

The ROBDD for f must:

- Have root variable x_i (the smallest essential variable)
- Have low child B_0 (unique by induction)
- Have high child B_1 (unique by induction)
- Have $B_0 \neq B_1$ (since x_i is essential)

The "no duplicate nodes" rule ensures this node is unique. Therefore, the ROBDD for f is unique. □

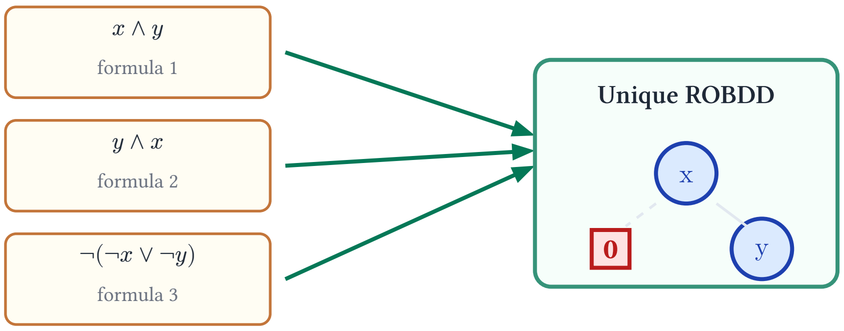
i Proof Strategy

The proof shows that:

1. The root must test the first essential variable (by ordering + no redundant tests)
2. The children are uniquely determined by the cofactors (by induction)
3. The node itself is unique (by no duplicate nodes)

Each reduction rule eliminates exactly one degree of freedom, leaving a unique representation.

Why BDDs Are Canonical



Same function $\rightarrow$ **Same BDD** (with fixed ordering)

Equivalence check: just compare pointers!

Figure 8: Different formulas for the same function all produce the identical BDD structure.

4.3 Consequences of Canonicity

The canonicity theorem enables several operations to be performed in constant time.

4.3.1 Equivalence Checking

📐 **Theorem (O(1) Equivalence)**

Given two ROBDDs B_f and B_g for functions f and g (with the same ordering), we can check whether $f \equiv g$ in $O(1)$ time.

Proof. By canonicity, $f \equiv g$ if and only if B_f and B_g are structurally identical. With hash consing (discussed below), structurally identical means pointer-equal. Comparing two pointers takes constant time. □

This is extraordinary. In contrast:

- CNF equivalence is coNP-complete
- Truth table comparison takes $O(2^n)$ time
- General circuit equivalence is coNP-complete

4.3.2 Satisfiability and Tautology

📐 Theorem (O(1) SAT and Tautology)

Given an ROBDD B_f:

- f is satisfiable if and only if $B_f \neq 0$
- f is a tautology if and only if $B_f = 1$

Both checks take $O(1)$ time.

Proof. By canonicity, the only ROBDD representing the constant-false function is the 0 terminal. Similarly, the only ROBDD for constant-true is the 1 terminal. Checking if a BDD is a terminal is a constant-time operation. □

⚠️ Complexity Caveat

These operations are $O(1)$ **given** the BDD. Building the BDD may take exponential time and space. The complexity is shifted from query time to construction time.

4.3.3 Solution Counting

📐 Theorem (Linear-Time Counting)

Given an ROBDD B_f with $|B_f|$ nodes, the number of satisfying assignments to f can be computed in $O(|B_f|)$ time.

The algorithm traverses the BDD bottom-up, computing at each node the number of paths to the 1 terminal, weighted by the number of variable assignments each path represents.

4.3.4 Model Enumeration

We can enumerate all satisfying assignments by traversing all paths from root to the 1 terminal. Each path corresponds to a (partial) assignment; variables not on the path can take any value.

4.4 Hash Consing: Implementing Canonicity

The theoretical canonicity theorem becomes practical through **hash consing**, a technique that maintains structural sharing.

📘 Definition (Hash Consing)

Hash consing is a technique where:

1. Every unique structure is stored exactly once
2. Creating a structure returns a reference to the existing copy if one exists
3. Structural equality reduces to pointer (reference) equality

For BDDs, this means maintaining a **unique table** — a hash table mapping $(\mathrm{var}, \mathrm{low}, \mathrm{high})$ triples to node references.

4.4.1 The Unique Table

📘 **Definition (Unique Table)**

The **unique table** is a hash map:

$$U : (\mathrm{Var} \times \mathrm{Node} \times \mathrm{Node}) \to \mathrm{Node} \qquad (29)$$

For any triple (x, l, h), either:

- $U(x, l, h)$ is undefined (no such node exists), or
- $U(x, l, h) = v$ where v is the unique node with $\mathrm{var}(v) = x$, $\mathrm{low}(v) = l$, $\mathrm{high}(v) = h$

⚙️ **Algorithm: mk — Create or Find Node**

```
function mk(var, low, high):
    // Rule 1: No redundant tests
    if low = high:
        return low

    // Rule 2: No duplicate nodes
    if (var, low, high) in UniqueTable:
        return UniqueTable[(var, low, high)]

    // Create new node
    node = new Node(var, low, high)
    UniqueTable[(var, low, high)] = node
    return node
```

The `mk` function enforces both reduction rules:

- If $\mathrm{low} = \mathrm{high}$, no node is created (Rule 1)
- If an equivalent node exists, it is returned (Rule 2)

💡 **Key Insight**

After every operation, the BDD manager maintains the invariant that structurally equal subgraphs are pointer-equal. This invariant is what makes $O(1)$ equivalence checking possible.

4.4.2 Implications for Operations

With hash consing:

- Creating a node is $O(1)$ amortized (hash table lookup/insert)
- Equivalence checking is $O(1)$ (pointer comparison)
- All operations that produce BDDs automatically produce reduced, canonical results

4.4.3 Per-Level vs Global Unique Tables

There are two common implementations:

Global Unique Table: One hash table for all nodes.
- Pro: Simple implementation
- Con: May have poor cache behavior

Per-Level Unique Tables: One hash table per variable level.
- Pro: Better cache locality during BDD operations
- Pro: Enables efficient variable reordering
- Con: Slightly more complex

Most modern implementations, including `bdd-rs`, use per-level unique tables.

4.5 The Cost of Canonicity

Canonicity is not free. The unique table and reduction rules impose constraints.

4.5.1 Memory Management

Since nodes are shared, we cannot simply delete a node when one reference disappears. BDD packages must use:
- **Reference counting**: Track how many references point to each node
- **Garbage collection**: Periodically reclaim unreachable nodes
- **Mark-and-sweep**: Identify live nodes from roots, reclaim the rest

4.5.2 Global State

The unique table is inherently global. All BDDs in a manager share the same table, which means:
- Thread safety requires synchronization
- All operations must go through the manager
- Mixing BDDs from different managers is invalid

4.5.3 Single Ordering

All BDDs in a manager share the same variable ordering. This is necessary for canonicity but means:
- You cannot have two BDDs with different orderings
- Changing the ordering requires rebuilding all BDDs
- Operations between BDDs require compatible orderings

4.5.4 The Trade-off

i Canonicity Trade-off

Canonicity trades **construction-time complexity** for **query-time efficiency**.

- Without canonicity: Construction might be faster, but every equivalence check requires full comparison
- With canonicity: Construction maintains invariants, but equivalence is free

For applications that perform many queries (verification, model checking), this trade-off is favorable.

4.6 Summary

The canonicity theorem is the foundation of BDD utility:

i Key Results

- **Theorem**: Every function has exactly one ROBDD (for fixed ordering)
- **Equivalence**: $f \equiv g$ iff their BDDs are pointer-equal — $O(1)$
- **SAT**: f is satisfiable iff BDD is not 0 — $O(1)$
- **Tautology**: f is valid iff BDD is 1 — $O(1)$
- **Counting**: Number of solutions in $O(|\text{BDD}|)$

Hash consing makes these theoretical results practical.

In the next chapter, we see how to build BDDs through Boolean operations, maintaining canonicity at every step.

Chapter 5

BDD Operations

The power of BDDs lies not just in compact representation, but in efficient manipulation. This chapter covers the core algorithms: combining BDDs (`Apply`), testing conditions (restriction), and abstracting variables (quantification).

The recurring theme: BDD operations are polynomial in the **BDD size**, not the exponential **function size**. When the BDD is compact, everything is fast.

5.1 Overview of Operations

BDD operations fall into several complexity classes:

Operation	Complexity	Example
Equivalence	$O(1)$	$f \equiv g?$
SAT check	$O(1)$	$f \equiv 0?$
Negation	$O(1)^*$	$\neg f$
Size query	$O(1)$	$\lvert f \rvert?$
Counting	$O(\lvert f \rvert)$	$\lvert\{\boldsymbol{x} : f(\boldsymbol{x}) = 1\}\rvert$
Cofactor	$O(\lvert f \rvert)$	$f\vert_{x=1}$
Apply (AND, OR, …)	$O(\lvert f \rvert \cdot \lvert g \rvert)$	$f \wedge g$
Quantification	$O(\lvert f \rvert^2)$	$\exists x . f$
Composition	$O(\lvert f \rvert^2 \cdot \lvert g \rvert^2)$	$f[x := g]$

$*$ With complement edges; $O(|f|)$ without.

The key insight: operations are polynomial in BDD size, which is often much smaller than 2^n.

5.2 The Apply Algorithm

The `Apply` algorithm is the workhorse of BDD manipulation. It combines two BDDs using any binary Boolean operation (AND, OR, XOR, etc.).

5.2.1 Intuition

To compute f `op` g where `op` is a binary operation:

1. Apply Shannon expansion to both functions on the same variable
2. Recursively compute the operation on cofactors
3. Combine results using `mk`

The trick is **memoization**: remember results to avoid redundant computation.

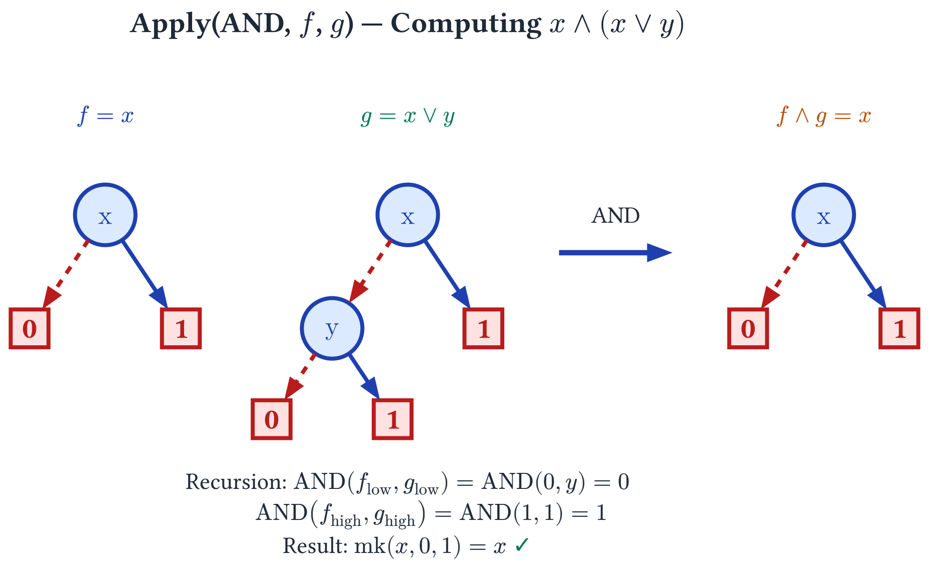

Apply(AND, f, g) — Computing $x \wedge (x \vee y)$
$f = x$
$g = x \vee y$
$f \wedge g = x$
x
y
0
1
AND
Recursion: $\text{AND}(f_{\text{low}}, g_{\text{low}}) = \text{AND}(0, y) = 0$
$\text{AND}(f_{\text{high}}, g_{\text{high}}) = \text{AND}(1, 1) = 1$
Result: $\text{mk}(x, 0, 1) = x$ ✓

Figure 9: Apply combines two BDDs by recursing on cofactors and rebuilding with `mk`*.*

5.2.2 The Algorithm

⚙️ **Algorithm: Apply**

```
function Apply(op, f, g):
    // Terminal cases
    if is_terminal(f) and is_terminal(g):
        return terminal(op(value(f), value(g)))

    // Short-circuit optimizations (for AND)
    if op = AND:
        if f = 0 or g = 0: return 0
        if f = 1: return g
        if g = 1: return f
        if f = g: return f

    // Cache lookup
    if (op, f, g) in ApplyCache:
        return ApplyCache[(op, f, g)]

    // Determine top variable (smallest in ordering)
    v = topvar(f, g)

    // Get cofactors (Shannon expansion)
    f_low  = cofactor(f, v, 0)
    f_high = cofactor(f, v, 1)
    g_low  = cofactor(g, v, 0)
    g_high = cofactor(g, v, 1)
    // Recursive calls
    low  = Apply(op, f_low, g_low)
    high = Apply(op, f_high, g_high)

    // Build result (mk handles reduction)
    result = mk(v, low, high)

    // Cache result
    ApplyCache[(op, f, g)] = result
    return result
```

The `topvar` function returns the smallest variable (according to ordering) that appears in either f or g. If a BDD does not depend on that variable, its cofactors are both equal to itself.

i Cofactor Computation

For a node v with variable x:

- If we're computing the cofactor for x: return low or high child
- If we're computing the cofactor for a variable $y < x$ (above in ordering): the function doesn't depend on y, return the node itself

5.2.3 Terminal Rules

Different operations have different terminal rules:

Operation	Terminal Rule	Short-circuits
AND ($\wedge$)	$0 \wedge x = 0, 1 \wedge x = x$	Either arg is 0
OR ($\vee$)	$1 \vee x = 1, 0 \vee x = x$	Either arg is 1
XOR ($\oplus$)	$0 \oplus x = x, 1 \oplus x = \neg x$	Both terminals
IMPLIES ($\rightarrow$)	$0 \rightarrow x = 1, 1 \rightarrow x = x$	$f = 0$
IFF ($\leftrightarrow$)	$x \leftrightarrow x = 1, 0 \leftrightarrow x = \neg x$	$f = g$

5.2.4 Example: Computing $f \wedge g$

Let $f = x_1$ and $g = x_1 \vee x_2$ with ordering $x_1 < x_2$.

1. `Apply(AND, f, g)`:
    - Neither is terminal, so expand on x_1
    - $f_{\text{low}} = 0$, $f_{\text{high}} = 1$
    - $g_{\text{low}} = x_2$, $g_{\text{high}} = 1$
2. Recursive calls:
    - `Apply(AND, 0, x_2)` $= 0$ (short-circuit)
    - `Apply(AND, 1, 1)` $= 1$ (terminals)
3. Build result:
    - `mk(x_1, 0, 1)` $= x_1$

Result: $x_1 \wedge (x_1 \vee x_2) = x_1$ ✓

5.3 Complexity Analysis

📐 **Theorem (Apply Complexity)**

For BDDs f and g, `Apply(op, f, g)` runs in $O(|f| \times |g|)$ time.

Proof. The cache ensures each pair of nodes (n_f, n_g) is processed at most once. There are at most $|f| \times |g|$ such pairs. Each non-cached call does $O(1)$ work (excluding recursive calls). Therefore, total time is $O(|f| \times |g|)$. □

📐 **Theorem (Result Size)**

The result of `Apply(op, f, g)` has at most $O(|f| \times |g|)$ nodes.

This bound is tight in the worst case but rarely achieved in practice. Many operations produce results much smaller than the theoretical maximum.

💡 **Key Insight**

The key insight: memoization transforms exponential recursion into polynomial time. Without the cache, we'd explore up to 2^n paths. With it, we explore at most $|f| \times |g|$ unique subproblems.

5.3.1 Cache Management

The **apply cache** (or **computed table**) is crucial for performance. Without caching, Apply would have exponential complexity.

Implementation considerations:

- **Cache key**: (`op`, f, g) triple, often hashed
- **Cache size**: Bounded; old entries may be evicted
- **Symmetry**: For commutative operations, normalize (f, g) to avoid duplicate entries
- **Clearing**: Cache must be invalidated when garbage collecting nodes

5.4 If-Then-Else (ITE)

The If-Then-Else operation is a ternary operation that subsumes all binary operations.

📘 **Definition (ITE Operation)**

$\text{ite}(f, g, h) = (f \land g) \lor (\neg f \land h)$

Semantically: "if f then g else h"

5.4.1 ITE is Universal

Every binary Boolean operation can be expressed as ITE:

Operation	ITE Expression
$f \wedge g$	$\text{ite}(f, g, 0)$
$f \vee g$	$\text{ite}(f, 1, g)$
$\neg f$	$\text{ite}(f, 0, 1)$
$f \oplus g$	$\text{ite}(f, \neg g, g)$
$f \to g$	$\text{ite}(f, g, 1)$
$f \leftrightarrow g$	$\text{ite}(f, g, \neg g)$

Some BDD libraries implement only ITE and derive other operations from it. Others implement Apply directly for common operations (more efficient).

5.4.2 ITE Algorithm

⚙️ Algorithm: ITE

```
function ITE(f, g, h):
    // Terminal cases
    if f = 1: return g
    if f = 0: return h
    if g = 1 and h = 0: return f
    if g = 0 and h = 1: return NOT(f)
    if g = h: return g

    // Cache lookup
    if (f, g, h) in ITECache:
        return ITECache[(f, g, h)]

    // Determine top variable
    v = topvar(f, g, h)

    // Recursive calls
    low  = ITE(cofactor(f,v,0), cofactor(g,v,0), cofactor(h,v,0))
    high = ITE(cofactor(f,v,1), cofactor(g,v,1), cofactor(h,v,1))

    // Build result
    result = mk(v, low, high)
    ITECache[(f, g, h)] = result
    return result
```

ITE complexity is $O(|f| \times |g| \times |h|)$ in the worst case.

5.5 Negation

Negation (complement) is conceptually simple: flip every 0 to 1 and vice versa.

5.5.1 Without Complement Edges

⚙️ Algorithm: Negation (Recursive)

```
function NOT(f):
    if f = 0: return 1
    if f = 1: return 0
    if f in NotCache: return NotCache[f]

    result = mk(var(f), NOT(low(f)), NOT(high(f)))
    NotCache[f] = result
    return result
```

This takes $O(|f|)$ time — we visit each node once and create a mirror node.

5.5.2 With Complement Edges

With complement edges (Section 11), negation is $O(1)$: just flip a bit in the reference.

💡 **Key Insight**

Complement edges are an optimization where the "negated" bit is stored in the pointer, not the node. This makes negation free but complicates other operations slightly. Most modern BDD packages use complement edges.

5.6 Restriction (Cofactor)

Restriction substitutes a constant for a variable: $f|_{x=b}$.

⚙️ **Algorithm: Restrict**

```
function Restrict(f, x, b):
    if is_terminal(f): return f
    if var(f) > x: return f          // x not in f's cone
    if var(f) = x:
        return (b = 0) ? low(f) : high(f)

    // var(f) < x: recurse
    if f in RestrictCache: return RestrictCache[f]

    result = mk(var(f),
                Restrict(low(f), x, b),
                Restrict(high(f), x, b))
    RestrictCache[f] = result
    return result
```

Restriction runs in $O(|f|)$ time.

5.6.1 Cube Restriction

Often we want to restrict multiple variables at once. A **cube** is a conjunction of literals: $x_1 \land \neg x_3 \land x_5$.

Restricting by a cube means setting all those variables to their specified values. This can be done in a single pass through the BDD.

5.6.2 Applications

- **Evaluation**: $f(1,0,1) = f|_{x_1=1,x_2=0,x_3=1}$
- **Simplification**: Given constraint c, simplify f under c
- **Composition building block**: Used in existential quantification

5.7 Composition (Substitution)

Composition replaces a variable with a function: $f[x := g]$.

This is more expensive than restriction because g can be an arbitrary BDD, not just a constant.

⚙️ Algorithm: Compose

```
function Compose(f, x, g):
    if is_terminal(f): return f
    if var(f) > x: return f          // x not in f's cone

    if (f, x, g) in ComposeCache:
        return ComposeCache[(f, x, g)]

    if var(f) = x:
        // f = x ? high(f) : low(f)
        // f[x := g] = g ? high(f)[x := g] : low(f)[x := g]
        result = ITE(g,
                     Compose(high(f), x, g),
                     Compose(low(f), x, g))
    else:
        // var(f) < x
        result = mk(var(f),
                    Compose(low(f), x, g),
                    Compose(high(f), x, g))

    ComposeCache[(f, x, g)] = result
    return result
```

Composition complexity is $O(|f|^2 \times |g|^2)$ in the worst case, but often much better in practice.

⚠️ Composition Pitfall

Repeated composition can be expensive. If you need $f[x_1 := g_1, x_2 := g_2, ..., x_k := g_k]$, the order matters and naive sequential composition can blow up. Consider vector composition or careful ordering.

5.8 Existential and Universal Quantification

Quantification abstracts away a variable.

📘 Definition (Existential Quantification)

$\exists x.f = f|_{x=0} \vee f|_{x=1}$

The function is true if f is true for **some** value of x.

📘 **Definition (Universal Quantification)**

$\forall x.f = f|_{x=0} \wedge f|_{x=1}$

The function is true if f is true for **all** values of x.

5.8.1 Algorithm

⚙️ **Algorithm: Existential Quantification**

```
function Exists(f, x):
    f0 = Restrict(f, x, 0)
    f1 = Restrict(f, x, 1)
    return Apply(OR, f0, f1)
```

This takes two restriction passes plus one Apply, so complexity is $O(|f|^2)$.

5.8.2 Multiple Variables

Quantifying multiple variables: $\exists x_1, x_2, x_3.f$

Naive approach: quantify one at a time. This can be inefficient because intermediate results may explode.

Better approaches:
- **Order by proximity**: Quantify variables that are close in the ordering together
- **Early quantification**: Interleave quantification with conjunction
- **Conjunctive decomposition**: Split f and quantify pieces

i Quantification in Model Checking

In symbolic model checking, image computation involves: $\text{Image}(S) = \exists \boldsymbol{x}.T(\boldsymbol{x}, \boldsymbol{x}') \wedge S(\boldsymbol{x})$

Efficient quantification is critical for scalability. Techniques like “partition TR” and “early quantification” address this.

5.9 Optimization Techniques

Several techniques improve operation performance:

5.9.1 Cache Sharing

Different operations can share cache entries:
- $f \wedge g$ and $g \wedge f$ are the same (commutativity)
- Some ITE patterns reduce to Apply calls

5.9.2 Terminal Case Optimization

Recognizing terminal cases early avoids recursion:
- $f \wedge 0 = 0$, $f \vee 1 = 1$

- $f \wedge f = f$, $f \vee f = f$
- $f \oplus f = 0$, $f \to f = 1$

5.9.3 Cofactor Computation

For efficiency, cofactor computation should be $O(1)$:

- If the node's variable matches: return child
- Otherwise: return the node itself (function doesn't depend on that variable)

5.10 Summary

i Core Operations

- **Apply**: Combine BDDs with any binary operation — $O(|f| \times |g|)$
- **ITE**: Universal ternary operation — $O(|f| \times |g| \times |h|)$
- **Negation**: $O(1)$ with complement edges, $O(|f|)$ without
- **Restrict**: Substitute constant for variable — $O(|f|)$
- **Compose**: Substitute function for variable — $O(|f|^2 \times |g|^2)$
- **Quantify**: Abstract away variables — $O(|f|^2)$

All operations preserve canonicity: results are automatically reduced.

The Apply algorithm is the foundation. Understanding it deeply is essential for effective BDD use. In Part II, we dive into implementation details that make these operations fast in practice.

PART II

Implementation

Chapter 6

Manager-Centric Architecture

Picture a library where every book can reference any page in any other book, but the references only work because all books live on the same shelf. Move a book to a different shelf, and the references shatter into meaningless numbers.

This is the essence of BDD architecture. Every BDD node can reference any other node, and these references **must** remain consistent across the entire system. This chapter explains why BDD libraries universally adopt a **manager-centric architecture** and how `bdd-rs` implements it.

6.1 The Central Challenge: Sharing

What makes BDDs powerful is also what makes them tricky to implement: **maximal sharing**.

Consider building the BDD for $f = (a \land b) \lor c$. Now build $g = (a \land b) \lor d$. Both formulas contain the subexpression $(a \land b)$. In a well-designed BDD library, this subexpression is represented **exactly once** — both f and g point to the same physical nodes in memory.

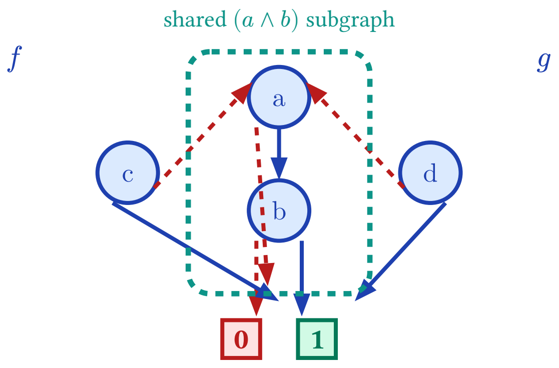


Figure 10: Two BDDs sharing a common subgraph. Both f and g reference the same nodes for $(a \land b)$.

This sharing requires centralized control. If f and g lived in separate data structures, they couldn't share nodes. Every BDD library solves this the same way: a **manager** that owns all nodes.

6.2 Why a Manager?

The manager pattern is not a stylistic choice — it's a necessity. BDDs need four guarantees that only centralized control can provide:

Requirement	Why It Needs a Manager
Shared storage	Nodes must live in one pool so references work across all BDDs
Hash consing	Before creating a node, check if it exists — needs global view
Consistent ordering	All BDDs must agree on variable order
Shared caches	Operation results must be reusable everywhere

Without a manager, two BDDs built independently would represent the same function with different structures. The fundamental $O(1)$ equivalence check — "are these two pointers equal?" — would break.

💡 **Key Insight**

Every major BDD library uses a manager:
- **CUDD**: `DdManager` (the gold standard)
- **BuDDy**: Global manager via `bdd_init()`
- **Sylvan**: Lock-free parallel manager
- **bdd-rs**: `Bdd` struct with interior mutability

6.3 The Bdd Manager in bdd-rs

Here is the actual `Bdd` struct from the source code:

```rust
pub struct Bdd {
    /// Node storage: index 0 is the terminal node
    nodes: RefCell<Vec<Node>>,
    /// Free node indices available for reuse
    free_set: RefCell<HashSet<NodeId>>,

    /// Operation cache (memoizes ITE results)
    cache: RefCell<Cache<OpKey, Ref>>,
    /// Size computation cache
    size_cache: RefCell<Cache<Ref, u64>>,

    /// Variable ordering: level → variable
    var_order: RefCell<Vec<Var>>,
    /// Reverse mapping: variable → level
    level_map: RefCell<HashMap<Var, Level>>,

    /// Per-level hash tables for uniqueness
    subtables: RefCell<Vec<Subtable>>,

    /// Next variable ID to allocate
    next_var_id: Cell<u32>,
    /// Configuration settings
    config: BddConfig,
}
```

Let us visualize how these components fit together:

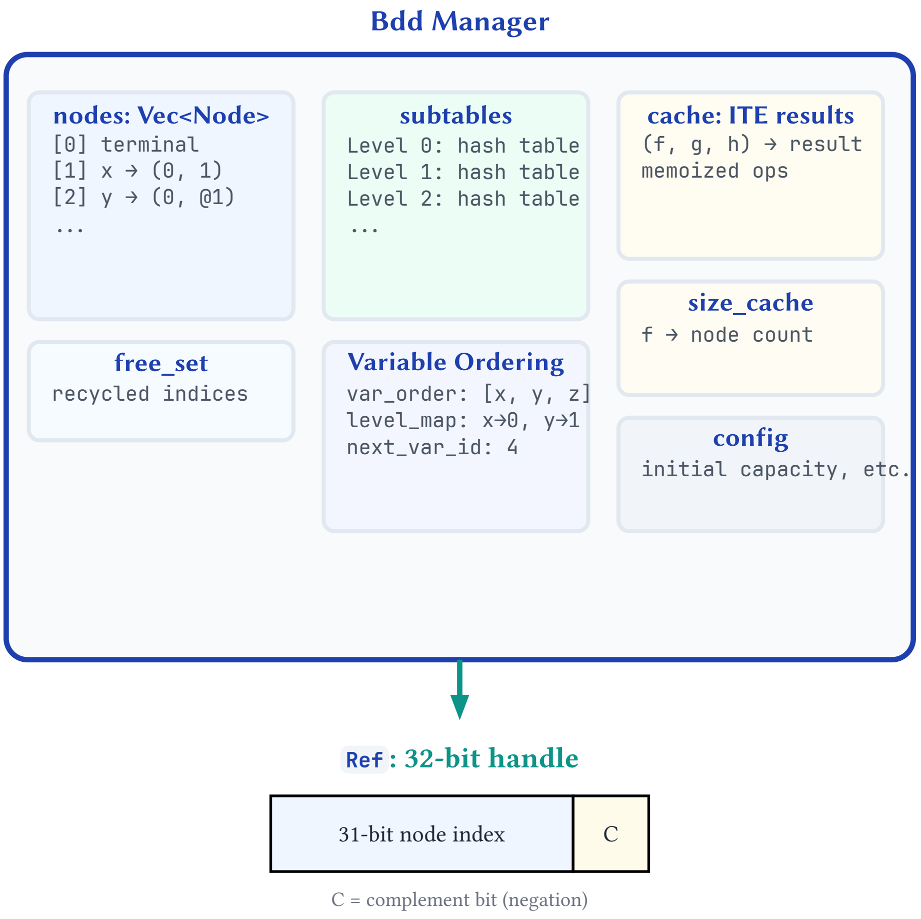


Figure 11: Architecture of the `Bdd` manager showing all major components.

6.3.1 Interior Mutability

Notice all the `RefCell` wrappers? They enable **interior mutability** — the ability to modify data through a shared reference (`&self`).

This is a deliberate ergonomic choice. Without it, every BDD operation would require `&mut self`, meaning you could not hold multiple `Ref` values while building a formula:

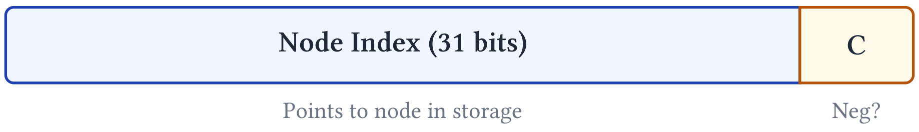
```
// With interior mutability (what we have):
let x = bdd.mk_var(1);  // &self
let y = bdd.mk_var(2);  // &self
let f = bdd.apply_and(x, y);  // &self - x and y still valid!

// Without (hypothetical):
let x = bdd.mk_var(1);  // &mut self
// x is now invalid because we'd need &mut self again!
```

The trade-off: runtime borrow checking instead of compile-time. In practice, BDD operations don't nest in ways that cause panics.

6.4 References: The User-Facing Handle

Users never touch `Node` structs directly. Instead, they work with `Ref` — a compact 32-bit handle:



Examples:

- `Ref::ONE` = `0b...0` (terminal, positive)
- `Ref::ZERO` = `0b...1` (terminal, negated)
- Node 5, positive = `0b...1010`
- Node 5, negated = `0b...1011`

Figure 12: Bit layout of `Ref`: 31 bits for node index, 1 bit for complement flag.

The complement bit is the key to $O(1)$ negation:

```rust
impl Neg for Ref {
    fn neg(self) → Self {
        Self(self.0 ^ 1)  // Just flip the lowest bit!
    }
}
```

Instead of traversing a BDD and creating new nodes for $\neg f$, we simply flip one bit. This single optimization cascades through the entire library, making XOR, equivalence, and implication faster.

Aspect	Ref	Node
Size	4 bytes	24 bytes
Copy	Trivial	Trivial but larger
Negation	$O(1)$ bit flip	Would need new node
Comparison	Single integer compare	Field-by-field

6.5 The API Surface

The manager exposes operations through method calls. Here's a quick tour:

6.5.1 Construction

```rust
// Create variables (1-indexed by convention)
let x = bdd.mk_var(1);
let y = bdd.mk_var(2);

// Build cubes and clauses efficiently
let cube = bdd.mk_cube([1, -2, 3]);    // x₁ ∧ ¬x₂ ∧ x₃
let clause = bdd.mk_clause([1, -2, 3]); // x₁ ∨ ¬x₂ ∨ x₃
```

6.5.2 Boolean Operations

```rust
// Binary operations
let f_and_g = bdd.apply_and(f, g);
let f_or_g = bdd.apply_or(f, g);
let f_xor_g = bdd.apply_xor(f, g);
```

```
// The universal if-then-else
let result = bdd.apply_ite(f, g, h);  // f ? g : h

// Negation is just syntax
let not_f = -f;
```

6.5.3 Queries

```
// O(1) checks (after construction)
bdd.is_zero(f)    // Unsatisfiable?
bdd.is_one(f)     // Tautology?
f == g            // Equivalent? (pointer compare!)

// O(|f|) operations
bdd.size(f)               // Node count
bdd.sat_count(f, n_vars) // Solution count
```

6.5.4 Quantification

```
// Existential: ∃x. f  (is there some x making f true?)
let ex = bdd.exists(f, [Var::new(1)]);

// Universal: ∀x. f  (is f true for all x?)
let fa = bdd.forall(f, [Var::new(1)]);

// Relational product: ∃vars. (f ∧ g)
let rp = bdd.rel_product(f, g, &quant_vars);
```

6.6 Creating and Configuring

For most uses, the default configuration works well:

```
let bdd = Bdd::default();
```

For large problems, customize the initial capacities:

```
use bdd_rs::bdd::{Bdd, BddConfig};

let config = BddConfig::default()
    .with_initial_nodes(1 << 20)  // ~1M nodes
    .with_cache_bits(18);         // 256K cache entries

let bdd = Bdd::with_config(config);
```

⚠️ Cross-Manager Pitfall

A `Ref` is only valid for its originating manager. Mixing them causes silent corruption:

```
let bdd1 = Bdd::default();
let bdd2 = Bdd::default();
let x = bdd1.mk_var(1);
```

```
// WRONG: x is from bdd1!
// bdd2.apply_and(x, bdd2.mk_var(2));  // Undefined behavior
```

Rust's type system cannot catch this — the indices look the same. Be careful when working with multiple managers.

6.7 What's Next

The following chapters dive deep into each component:

- **Section 7**: How nodes store variable, children, and hash data
- **Section 8**: Per-level subtables for $O(1)$ node lookup
- **Section 9**: The ITE algorithm that powers all operations
- **Section 10**: How memoization prevents exponential blowup
- **Section 11**: The elegant trick behind $O(1)$ negation

Chapter 7

Node Representation

At the lowest level, a BDD is just memory: bytes laid out in a particular way. How those bytes are organized directly impacts every operation's performance — from cache locality during traversals to memory consumption with millions of nodes.

This chapter explores `bdd-rs` 's node structure, the type-safe wrappers that prevent common bugs, and the memory layout decisions that keep things fast.

7.1 The Node Structure

Each BDD node represents a Shannon decomposition:

$$f = (\neg v \land f_{\text{low}}) \lor (v \land f_{\text{high}}) \tag{30}$$

The `Node` struct captures this with five fields:

```
#[derive(Debug, Copy, Clone)]
pub struct Node {
    pub variable: Var,    // Decision variable at this node
    pub low: Ref,         // Child when variable = 0
    pub high: Ref,        // Child when variable = 1
    pub next: NodeId,     // Hash collision chain pointer
    hash: u64,            // Precomputed hash for fast lookup
}
```

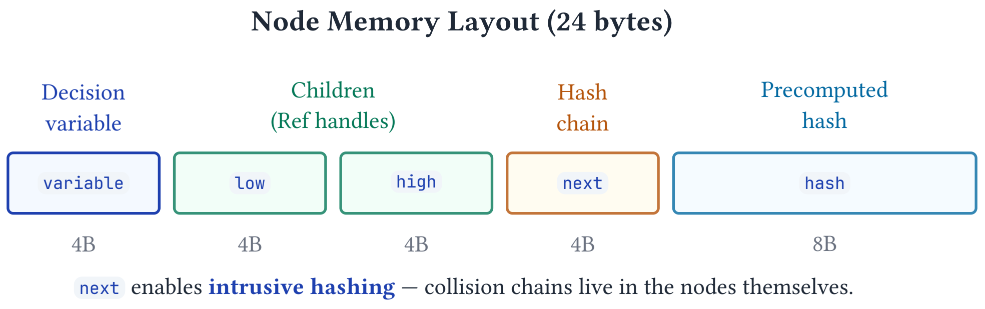

Node Memory Layout (24 bytes)
Decision variable
Children (Ref handles)
Hash chain
Precomputed hash
variable
low
high
next
hash
4B
4B
4B
4B
8B
next enables intrusive hashing — collision chains live in the nodes themselves.

Figure 13: Node memory layout: 24 bytes with precomputed hash and intrusive collision chain.

i Why Store the Hash?

Computing `hash(variable, low, high)` requires three multiplications and XORs. By precomputing and storing it in the node, we avoid recalculating during hash table operations. This trades 8 bytes of memory per node for faster lookups.

The `next` field implements **intrusive hashing** — nodes themselves form the collision chains for the hash table, rather than using a separate wrapper struct. This CUDD-inspired design eliminates allocation overhead and improves cache locality.

7.2 Terminal Nodes

The terminal node (representing both $\top$ and $\bot$) receives special handling:

```
// During manager construction:
let mut nodes = Vec::with_capacity(capacity);
nodes.push(Node::new(Var::ZERO, Ref::INVALID, Ref::INVALID));

// Terminal references:
let one = Ref::positive(0);   // @0 (non-negated terminal)
let zero = -one;               // ~@0 (negated terminal)
```

Key properties of the terminal:

1. **Index 0**: The terminal always lives at index 0 in the node array
2. **Variable = 0**: `Var::ZERO` marks this as a terminal, not a decision node
3. **Invalid children**: `Ref::INVALID` signals that children don't exist
4. **Both constants share it**: `one = @0` and `zero = ~@0` (complement)

This design means checking for terminals is a simple index comparison:

```
impl Bdd {
    pub fn is_terminal(&self, r: Ref) -> bool {
        r.id().raw() == 0
    }

    pub fn is_one(&self, r: Ref) -> bool {
        r.raw() == self.one.raw()
    }

    pub fn is_zero(&self, r: Ref) -> bool {
        r.raw() == self.zero.raw()
    }
}
```

7.3 Type-Safe Wrappers

`bdd-rs` uses the **newtype pattern** extensively to prevent mixing up different indices:

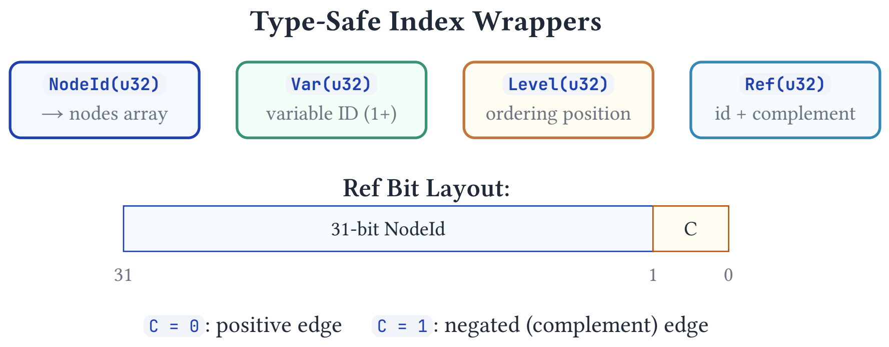


Figure 14: Type-safe wrappers prevent accidentally mixing indices. The `Ref` type packs node ID and complement flag into 32 bits.

```rust
pub struct NodeId(u32);   // Index into nodes array
pub struct Var(u32);      // Variable identifier (1-indexed)
pub struct Level(u32);    // Position in variable ordering
pub struct Ref(u32);      // Handle: (id << 1) | negated
```

These types are all `u32` internally, but the type system prevents accidental confusion:

```rust
fn process(v: Var, l: Level) { ... }

let var = Var::new(1);
let level = Level::new(0);

process(var, level);     // ✓ Compiles
// process(level, var);  // × Type error!
```

📘 Definition (NodeId)

An index into the `nodes: Vec<Node>` array. Valid range is $[0, \text{nodes.len})$. Index 0 is the terminal node.

📘 Definition (Var)

A semantic variable identifier. Variables are **1-indexed** by convention — `Var(0)` is reserved for terminals. This aligns with DIMACS format used in SAT solvers.

📘 Definition (Level)

Position in the current variable ordering. Level 0 is the root (top) of the BDD. Lower levels are closer to terminals.

📘 Definition (Ref)

A user-facing handle combining `NodeId` and a complement bit. The 31 high bits store the node index; the low bit stores negation.

🔧 **Implementation Note**

In `bdd-rs`, variables are 1-indexed by convention. Variable 0 is reserved for internal use (marking terminals). This aligns with DIMACS format used in SAT solvers.

7.4 Variables vs. Levels

A common source of confusion is the distinction between **variables** and **levels**:

Concept	Meaning	Example
Variable	Semantic identity $(x_1, x_2, \ldots)$	`Var(1)` = “input A”
Level	Position in current ordering	`Level(0)` = root position

Why does this distinction matter? Because **variable reordering** can change which variable is at which level.

✏️ **Example — Variable vs. Level**

With ordering $x_2 < x_1 < x_3$:

- x_2 is at level 0
- x_1 is at level 1
- x_3 is at level 2

The variable x_1 always represents the same Boolean input, but its position in the BDD changes with the ordering.

The manager maintains bidirectional mappings:

```rust
// In Bdd struct:
var_order: RefCell<Vec<Var>>,           // level → variable
level_map: RefCell<HashMap<Var, Level>>, // variable → level

// Usage:
fn var_at_level(&self, level: Level) -> Var {
    self.var_order()[level.index()]
}

fn level_of_var(&self, var: Var) -> Level {
    self.level_map()[&var]
}
```

This separation is crucial for dynamic reordering — we can swap positions without changing variable semantics.

7.5 The Ref Type in Detail

`Ref` is the most important type for users. It's a 32-bit value that encodes both **which node** and **whether it's negated**:

```rust
#[repr(transparent)]
pub struct Ref(u32);

impl Ref {
    // Bit layout: [31-bit node index][1-bit complement]

    pub fn new(id: NodeId, negated: bool) → Self {
        Self((id.raw() << 1) | (negated as u32))
    }

    pub fn id(self) → NodeId {
        NodeId::from_raw(self.0 >> 1)
    }

    pub fn is_negated(self) → bool {
        (self.0 & 1) ≠ 0
    }
}
```

💡 **Key Insight**

The complement bit enables $O(1)$ negation. Instead of building a new BDD for $\neg f$, we just flip the low bit of the `Ref`. This is why BDD libraries with complement edges outperform those without for negation-heavy operations.

7.5.1 Ref Operations

```rust
// Negation: flip the complement bit
impl Neg for Ref {
    fn neg(self) → Self {
        Self(self.0 ^ 1)
    }
}

// Comparison: direct integer compare
impl PartialEq for Ref {
    fn eq(&self, other: &Self) → bool {
        self.0 == other.0
    }
}

// Hashing: use raw value
impl Hash for Ref {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.0.hash(state);
    }
}
```

The raw `u32` comparison for equality is **exactly** why BDD equivalence is $O(1)$. Two `Ref` values are equal if and only if they represent the same Boolean function.

7.6 Memory and Cache Considerations

7.6.1 Contiguous Storage

Nodes are stored in a simple `Vec<Node>`:

```
nodes: RefCell<Vec<Node>>
```

This provides:

- **O(1) access**: Index directly into array
- **Cache locality**: Sequential nodes are adjacent in memory
- **Simple growth**: `Vec` handles reallocation automatically

7.6.2 Cache-Friendly Traversal

BDD algorithms typically traverse nodes in a pattern that respects variable ordering:

1. Start at root (highest level)
2. Recursively process children (lower levels)
3. Combine results bottom-up

Because nodes at the same level are often created together, they tend to be adjacent in the array. This improves CPU cache hit rates during traversal.

7.6.3 Size Analysis

With the current 24-byte node structure:

- 1M nodes $\approx$ 24 MB
- 16M nodes $\approx$ 384 MB
- 256M nodes $\approx$ 6 GB

The cache size (for memoization) is typically smaller — `bdd-rs` defaults to 2^{16} cache entries regardless of node capacity.

> ⚡ **Performance**
>
> For large problems, the bottleneck is usually cache misses during graph traversal, not raw memory bandwidth. Keep your working set small by avoiding unnecessary intermediate results.

7.7 Putting It Together

Here's how the types interact during a simple operation:

```rust
let bdd = Bdd::default();
let x = bdd.mk_var(1);  // Returns Ref

// mk_var internally:
// 1. Creates Var(1)
// 2. Registers var at next available Level
// 3. Creates Node { variable: Var(1), low: zero, high: one, ... }
// 4. Stores node at some NodeId
// 5. Returns Ref::positive(node_id)
```

```
let not_x = -x;  // Just flips the complement bit

// is_one check:
// 1. Compare x.raw() with one.raw()
// 2. Direct u32 comparison --- O(1)
```

The layered type system — `Node`, `NodeId`, `Var`, `Level`, `Ref` — may seem complex, but each type serves a specific purpose and prevents a class of bugs.

Chapter 8

The Unique Table

What stops a BDD from creating the same node twice? The answer is the **unique table** — a hash-based lookup structure ensuring that every $(v, \text{low}, \text{high})$ triple maps to exactly one node.

This is the mechanism behind canonicity. Without it, two separately constructed BDDs might represent the same function but have different structures. The $O(1)$ equivalence check — the killer feature of BDDs — would break entirely.

This chapter explains how the unique table works and why `bdd-rs` uses per-level subtables for efficiency.

8.1 The Problem: Duplicate Nodes

Imagine building the BDD for $f = (x \land y) \lor (x \land z)$. The variable x appears twice in the formula. Without care, we might create two separate x-nodes:

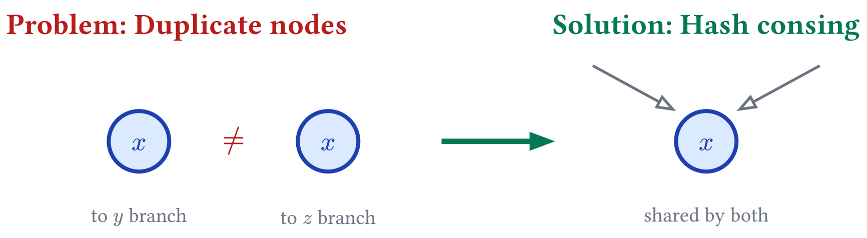


Figure 15: Without hash consing (left), duplicate nodes destroy canonicity. With it (right), identical structures are shared.

Hash consing prevents this. Before creating any node, we check: "Does this node already exist?" If yes, return the existing one. If no, create and register it.

8.2 The Unique Table

The unique table maintains the invariant:

$$\forall (\text{var}, \text{low}, \text{high}) : \text{at most one node exists} \tag{31}$$

It's a hash map from triples to node IDs:

$$U : (\text{Var} \times \text{Ref} \times \text{Ref}) \to \text{NodeId} \tag{32}$$

📘 Definition (Hash Consing)

Hash consing is a technique where:

1. Before creating a structure, check if an identical one exists
2. If it exists, return a reference to the existing one
3. If not, create it and add it to the table

This ensures structural sharing: identical structures are represented exactly once.

8.3 The `mk` Operation

The `mk` (make) function is the gatekeeper. Every node creation goes through it, and it enforces all BDD invariants:

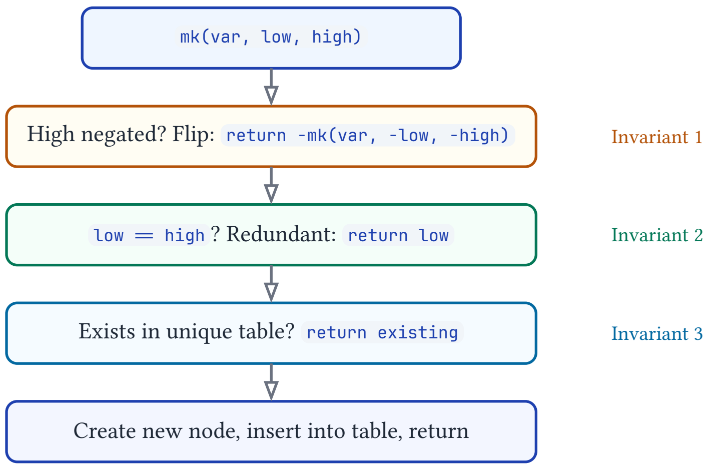

mk(var, low, high)
High negated? Flip: return -mk(var, -low, -high)
Invariant 1
low == high ? Redundant: return low
Invariant 2
Exists in unique table? return existing
Invariant 3
Create new node, insert into table, return

Figure 16: The `mk` function enforces three invariants in sequence.

⚙️ Algorithm: mk (Make Node)

```
mk(var, low, high):
  // Invariant 1: Canonicity (high edge positive)
  if high.is_negated():
    return -mk(var, -low, -high)

  // Invariant 2: Reduction (no redundant tests)
  if low == high:
    return low

  // Invariant 3: Uniqueness (hash consing)
  level = get_level(var)
  if (low, high) in subtables[level]:
    return subtables[level].find(low, high)

  // Create and register new node
  node = Node::new(var, low, high)
  id = allocate_node(node)
  subtables[level].insert(low, high, id)
  return Ref::positive(id)
```

8.4 Per-Level Subtables

`bdd-rs` doesn't use a single global hash table. Instead, it uses **per-level subtables** — one hash table per variable:

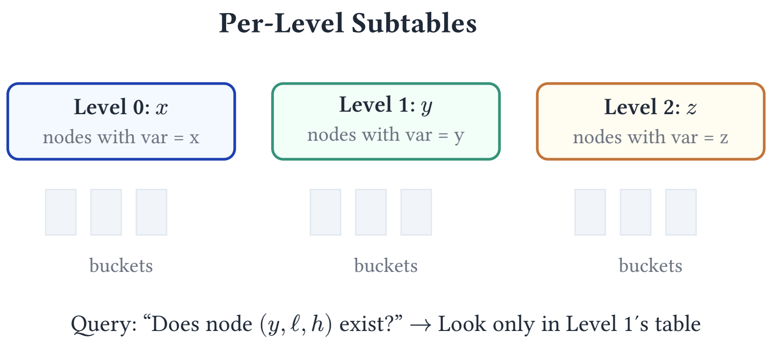


Figure 17: Each level maintains its own hash table. Queries only search the relevant level.

```
subtables: RefCell<Vec<Subtable>>  // level → subtable

pub struct Subtable {
    pub variable: Var,
    buckets: Vec<NodeId>,  // Hash bucket heads
    bitmask: u64,          // For fast modulo
    count: usize,          // Node count at this level
}
```

8.4.1 Why Per-Level?

i Advantages of Per-Level Subtables

1. **Smaller tables**: Each level has fewer nodes than the total, reducing collision rates
2. **Better locality**: Operations often work within one or two levels
3. **Simpler reordering**: Swapping variable positions means swapping subtables
4. **Parallelism-friendly**: Different levels can be processed independently

The key insight: when creating a node for variable v, we **know** which subtable to search. We don't need to include v in the hash — it's implicit from which table we're querying.

```
fn bucket_index(&self, low: Ref, high: Ref) → usize {
    let hash = hash_children(low, high);
    (hash & self.bitmask) as usize
}
```

8.5 The Unique Table vs. Computed Table

Two hash-based structures exist in a BDD library. Don't confuse them:

Aspect	Unique Table	Computed Table (Cache)
Purpose	Node deduplication	Operation memoization

Key	$(\text{low}, \text{high})$ at level	(op, f, g, h)
Value	NodeId	Ref (result)
On collision	Chain (must be complete)	Evict (optimization only)
Correctness	**Required**	Performance only

The unique table must be **complete** — every node must be findable. The computed table can **evict** entries on collision; it only affects speed, not correctness.

8.5.1 Intrusive Hashing

Following CUDD's design, `bdd-rs` uses **intrusive hashing** — collision chains are stored in the nodes themselves via the `next` field:

```
pub struct Node {
    pub variable: Var,
    pub low: Ref,
    pub high: Ref,
    pub next: NodeId,  // Next in collision chain
    hash: u64,
}
```

The `buckets` array stores head pointers. Following `next` fields traverses the collision chain:

```
Subtable for level k:
┌──────────────────────────────────────────────────┐
│ buckets: [NodeId; 2^bits]                        │
│   [0] ───→ Node@5 ──→ Node@12 ──→ ∅              │
│   [1] ───→ ∅                                     │
│   [2] ───→ Node@3 ──→ ∅                          │
│   ...                                            │
└──────────────────────────────────────────────────┘
```

Approach	**Memory**	**Locality**
Complexity	Intrusive (CUDD-style)	No extra allocation
Better — nodes are data	Harder to implement	External chaining
Entry wrapper per node	Worse — indirection	Easier
Open addressing	No chains	Excellent
Resizing tricky		

8.5.2 Lookup Operation

Finding a node with given children:

```
impl Subtable {
    pub fn find(&self, low: Ref, high: Ref, nodes: &[Node]) → Option<NodeId> {
        let idx = self.bucket_index(low, high);
        let mut current = self.buckets[idx];

        // Walk collision chain
        while current != Node::NO_NEXT {
            let node = &nodes[current.index()];
            if node.low == low && node.high == high {
                return Some(current);
            }
            current = node.next;
        }
        None
    }
}
```

Average case is $O(1)$ assuming a good hash function and reasonable load factor. Worst case is $O(n)$ if all nodes hash to the same bucket.

8.5.3 Insert Operation

Adding a new node to the subtable:

```
impl Subtable {
    pub fn insert(&mut self, low: Ref, high: Ref, id: NodeId, nodes: &mut [Node]) {
        let idx = self.bucket_index(low, high);

        // Prepend to collision chain
        nodes[id.index()].next = self.buckets[idx];
        self.buckets[idx] = id;

        self.count += 1;
    }
}
```

Insertion is $O(1)$ — we prepend to the chain head. No need to check for duplicates; we assume `find` was called first.

8.6 Hash Function Design

A good hash function for $(\text{low}, \text{high})$ pairs should:

1. **Distribute evenly**: Minimize collisions
2. **Be fast**: Called frequently during BDD construction
3. **Handle similar inputs**: Nearby Ref values shouldn't cluster

```
fn hash_children(low: Ref, high: Ref) -> u64 {
    let x = low.raw() as u64;
    let y = high.raw() as u64;
    // Mix using multiplication and XOR
    x.wrapping_mul(PRIME1) ^ y.wrapping_mul(PRIME2)
}
```

The node's hash is precomputed during creation:

```
impl Node {
    pub fn new(variable: Var, low: Ref, high: Ref) -> Self {
        let hash = {
            let x = variable.id() as u64;
            let y = hash(&low);
            let z = hash(&high);
            hash(&(y, z, x))
        };
        Self { variable, low, high, next: Self::NO_NEXT, hash }
    }
}
```

8.7 Load Factor and Resizing

The **load factor** is the ratio of nodes to buckets:

$$\text{load factor} = \frac{\text{count}}{\text{num\_buckets}} \tag{33}$$

High load factors mean longer collision chains and slower lookups. `bdd-rs` initializes subtables with 2^{16} buckets by default:

```
const DEFAULT_BUCKET_BITS: usize = 16;  // 65536 buckets

impl Subtable {
    pub fn new(variable: Var) → Self {
        Self::with_bucket_bits(variable, DEFAULT_BUCKET_BITS)
    }
}
```

For most applications, this is sufficient. Dynamic resizing (rehashing) adds complexity and is not currently implemented in `bdd-rs`.

⚠️ Load Factor Implications

If a single level grows to millions of nodes with only 65536 buckets, the average chain length becomes ≈ 15 nodes. For pathological cases, consider initializing with larger subtables.

8.8 Maintaining Invariants

The unique table and node storage must stay synchronized:

📐 Theorem (Unique Table Invariants)

At all times:

1. Every node in storage (except free slots) is in exactly one subtable
2. Every entry in a subtable points to a valid node with matching children
3. No two nodes have the same $(\text{var}, \text{low}, \text{high})$ triple

Violations indicate bugs in the implementation. Debugging techniques include:

```
// Verify all nodes are reachable from subtables
fn validate_unique_table(&self) {
    let mut seen = HashSet::new();
    for subtable in self.subtables().iter() {
        for id in subtable.all_nodes(&self.nodes()) {
            assert!(!seen.contains(&id), "Duplicate entry");
            seen.insert(id);
        }
    }
    // Check free_set nodes are not in subtables
    for &id in self.free_set().iter() {
        assert!(!seen.contains(&id), "Free node in table");
    }
}
```

8.9 Alternative Designs

Other BDD libraries make different choices:

Library	Table Structure	Hash Strategy
CUDD	Per-level subtables	Intrusive chaining
BuDDy	Global table	External chaining
Sylvan	Lock-free global table	Open addressing
bdd-rs	Per-level subtables	Intrusive chaining

The per-level approach scales better for dynamic reordering, while global tables are simpler to implement. Lock-free designs like Sylvan's enable parallelism but add significant complexity.

💡 **Key Insight**

The unique table is where BDD libraries spend most of their implementation effort. A well-tuned hash table directly impacts the speed of every BDD operation.

Chapter 9

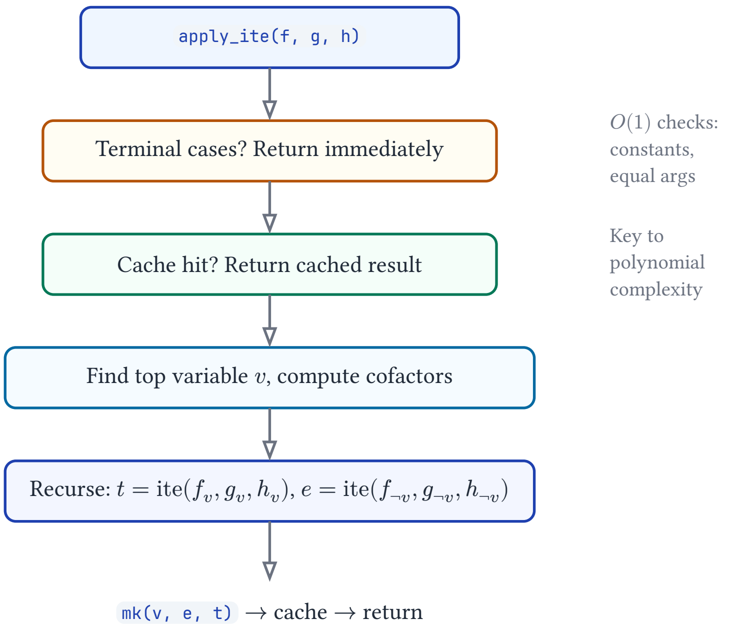
The Apply Algorithm in Detail

Apply is the workhorse of BDD manipulation. It takes two (or three) BDDs and a Boolean operation, and produces a new BDD representing the combined function. Nearly **every** BDD operation — from simple AND/OR to complex quantification — flows through Apply.

Understanding Apply deeply means understanding BDD performance. This chapter dissects the algorithm: terminal cases, normalization tricks, and the crucial role of caching that makes everything polynomial.

9.1 The Big Picture

Apply computes f `op` g for any binary Boolean operation `op` (AND, OR, XOR, etc.). The structure follows the recursive decomposition of Boolean functions:



apply_ite(f, g, h)
Terminal cases? Return immediately
$O(1)$ checks: constants, equal args
Cache hit? Return cached result
Key to polynomial complexity
Find top variable v, compute cofactors
Recurse: $t = \text{ite}(f_v, g_v, h_v)$, $e = \text{ite}(f_{\neg v}, g_{\neg v}, h_{\neg v})$
mk(v, e, t) $\to$ cache $\to$ return

Figure 18: High-level flow of the Apply (ITE) algorithm.

The algorithm has polynomial complexity because of **memoization**: each unique triple (f, g, h) is computed at most once.

9.2 Everything is ITE

In `bdd-rs`, all operations go through the **ITE** (if-then-else) primitive:

$$\text{ite}(f, g, h) = (f \land g) \lor (\neg f \land h) \quad (34)$$

Every binary operation can be expressed as ITE:

Operation	Formula	ITE Encoding
$f \land g$	$f \cdot g$	$\text{ite}(f, g, 0)$
$f \lor g$	$f + g$	$\text{ite}(f, 1, g)$
$f \oplus g$	$f \oplus g$	$\text{ite}(f, \neg g, g)$
$f \to g$	$\overline{f} + g$	$\text{ite}(f, g, 1)$
$f \equiv g$	$\overline{f \oplus g}$	$\text{ite}(f, g, \neg g)$

This unification simplifies implementation — one algorithm handles everything.

9.3 Terminal Cases

Terminal cases are where recursion stops. More terminal cases mean faster computation.

9.3.1 Constant Arguments

```
// ite(1, g, h) = g
if self.is_one(f) { return g; }

// ite(0, g, h) = h
if self.is_zero(f) { return h; }
```

9.3.2 Equal and Complementary Arguments

```
// ite(f, g, g) = g  (doesn't matter what f is)
if g == h { return g; }

// ite(f, 1, 0) = f
if self.is_one(g) && self.is_zero(h) { return f; }

// ite(f, 0, 1) = ¬f
if self.is_zero(g) && self.is_one(h) { return -f; }
```

9.3.3 Advanced Terminal Cases

These catch more patterns:

```
// ite(f, 1, ¬f) = 1
if self.is_one(g) && h == -f { return self.one(); }

// ite(f, f, 0) = f  (f AND f = f)
if g == f && self.is_zero(h) { return f; }

// ite(f, ¬f, 0) = 0  (f AND ¬f = 0)
if g == -f && self.is_zero(h) { return self.zero(); }
```

✏️ **Example — Terminal Cases for AND**

Since $f \wedge g = \text{ite}(f, g, 0)$, terminal cases include:

- `ite(0, g, 0) = 0` (absorbing element)
- `ite(1, g, 0) = g` (identity)
- `ite(f, f, 0) = f` (idempotent)
- `ite(f, ¬f, 0) = 0` (complement)

9.4 Standard Triples: Cache Optimization

Standard triples normalize equivalent ITE calls to improve cache hit rates.

The insight: `ite(f, 1, g)` and `ite(g, 1, f)` compute the same function ($f \vee g$). If we always put the "smaller" BDD first, they'll hit the same cache entry.

```
// ite(f, f, h) → ite(f, 1, h)  (f in "then" position is redundant)
if g == f {
    return self.apply_ite(f, self.one, h);
}
// ite(f, g, f) → ite(f, g, 0)  (f in "else" position is redundant)
if h == f {
    return self.apply_ite(f, g, self.zero);
}
// ite(f, ¬f, h) → ite(f, 0, h)
if g == -f {
    return self.apply_ite(f, self.zero, h);
}
// ite(f, g, ¬f) → ite(f, g, 1)
if h == -f {
    return self.apply_ite(f, g, self.one);
}
```

9.4.1 Argument Ordering

When possible, reorder arguments so the smallest-variable BDD comes first:

```
let i = self.variable(f.id());
let j = self.variable(g.id());
let k = self.variable(h.id());

// ite(f, 1, h) == ite(h, 1, f) == f ∨ h
// Choose the one with smaller top variable
if self.is_one(g) && self.var_precedes(k, i) {
    return self.apply_ite(h, self.one, f);
}
// ite(f, g, 0) == ite(g, f, 0) == f ∧ g
if self.is_zero(h) && self.var_precedes(j, i) {
    return self.apply_ite(g, f, self.zero);
}
```

This normalization ensures that `ite(f, 1, g)` and `ite(g, 1, f)` hit the same cache entry.

9.5 Complement Edge Handling

The canonical form requires that the "then" branch (g) is never negated:

```
let (mut f, mut g, mut h) = (f, g, h);

// ite(¬f, g, h) → ite(f, h, g)
if f.is_negated() {
    f = -f;
    std::mem::swap(&mut g, &mut h);
}
assert!(!f.is_negated());

// ite(f, ¬g, h) → ¬ite(f, g, ¬h)
let mut n = false;
if g.is_negated() {
    n = true;
    g = -g;
    h = -h;
}
assert!(!g.is_negated());
```

The `n` flag tracks whether we need to negate the final result. This normalization is crucial for cache efficiency — without it, `ite(f, g, h)` and `ite(f, -g, -h)` would be cached separately.

💡 **Key Insight**

Complement edge normalization can **double** cache hit rates. The small overhead of checking and swapping is far outweighed by reduced redundant computation.

9.6 Cofactor Computation

Given the top variable v, we need cofactors of all three arguments:

```
// Determine top variable (smallest in ordering)
let mut m = i;  // f's variable
if !j.is_terminal() {
    m = self.top_variable(m, j);
}
if !k.is_terminal() {
    m = self.top_variable(m, k);
}

// Get cofactors
let (ft, fe) = self.top_cofactors(f, m);  // f|_{m=1}, f|_{m=0}
let (gt, ge) = self.top_cofactors(g, m);
let (ht, he) = self.top_cofactors(h, m);
```

The `top_cofactors` function handles three cases:

```
pub fn top_cofactors(&self, node_ref: Ref, v: Var) -> (Ref, Ref) {
    // Terminal: cofactors are the terminal itself
    if self.is_terminal(node_ref) {
        return (node_ref, node_ref);
    }

    let node = self.node(node_ref.id());

    // Variable not at this node: function doesn't depend on v
    if self.var_precedes(v, node.variable) {
        return (node_ref, node_ref);
    }

    // Variable matches: return children (respecting complement)
    assert_eq!(v, node.variable);
    if node_ref.is_negated() {
        (-node.low, -node.high)
    } else {
        (node.low, node.high)
    }
}
```

i Cofactor Computation

For a node with variable x:

- If we're computing the cofactor for x: return low or high child
- If we're computing the cofactor for a variable $y < x$ (above in ordering): the function doesn't depend on y, return the node itself

9.7 The Recursive Step

With cofactors computed, we recurse:

```
// Recursive calls on cofactors
let t = self.apply_ite(ft, gt, ht);  // "then" branch
let e = self.apply_ite(fe, ge, he);  // "else" branch

// Build result
let result = self.mk_node(m, e, t);  // Note: low = else, high = then

// Cache and return
self.cache_mut().insert(key, result);
if n { -result } else { result }
```

Note the argument order to `mk_node`: low (else) comes before high (then). This matches the Shannon decomposition $f = (\neg v \land f_{\text{low}}) \lor (v \land f_{\text{high}})$.

9.8 The Complete ITE Implementation

Here's the full implementation structure:

⚙️ Algorithm: apply_ite (ITE Operation)

```
apply_ite(f, g, h):
  // Terminal cases (constants)
  if f == 1: return g
  if f == 0: return h
  if g == h: return g
  if g == 1 and h == 0: return f
  if g == 0 and h == 1: return -f
  // ... more terminal cases ...

  // Standard triple normalizations
```

```
if g == f: return apply_ite(f, 1, h)
if h == f: return apply_ite(f, g, 0)
// ... more normalizations ...

// Argument ordering (smallest variable first)
// ... reordering logic ...

// Complement edge normalization
if f.is_negated():
  f = -f; swap(g, h)
n = false
if g.is_negated():
  n = true; g = -g; h = -h

// Cache lookup
key = (f, g, h)
if key in cache: return cache[key] (negated if n)

// Determine top variable
m = top_var(f, g, h)

// Get cofactors
(ft, fe) = top_cofactors(f, m)
(gt, ge) = top_cofactors(g, m)
(ht, he) = top_cofactors(h, m)

// Recurse
t = apply_ite(ft, gt, ht)
e = apply_ite(fe, ge, he)

// Build and cache result
result = mk(m, e, t)
cache[key] = result
return result (negated if n)
```

9.9 Complexity Analysis

📐 **Theorem (ITE Complexity)**

For BDDs f, g, and h, `apply_ite(f, g, h)` runs in $O(|f| \times |g| \times |h|)$ time.

Proof. The cache key is the triple (f, g, h) after normalization. There are at most $|f| \times |g| \times |h|$ distinct triples. Each non-cached call does $O(1)$ work (excluding recursive calls). Therefore, total time is bounded by $O(|f| \times |g| \times |h|)$. □

For binary operations where one argument is constant, this simplifies:

- $f \land g = \mathrm{ite}(f, g, 0)$: $O(|f| \times |g|)$ since $|0| = 1$
- $f \lor g = \mathrm{ite}(f, 1, g)$: $O(|f| \times |g|)$

9.10 Iterative vs. Recursive Implementation

The recursive implementation is clean but has a limitation: deep BDDs can overflow the call stack.

```
// Recursive (current bdd-rs approach)
fn apply_ite(&self, f: Ref, g: Ref, h: Ref) -> Ref {
    // ... base cases ...
    let t = self.apply_ite(ft, gt, ht);  // Stack frame
    let e = self.apply_ite(fe, ge, he);  // Another stack frame
    self.mk_node(m, e, t)
}
```

For very large BDDs (millions of nodes), an **iterative** implementation with an explicit stack avoids overflow:

```
// Iterative alternative
fn apply_ite_iterative(&self, f: Ref, g: Ref, h: Ref) → Ref {
    let mut stack = Vec::new();
    stack.push(Task::Compute(f, g, h));

    while let Some(task) = stack.pop() {
        match task {
            Task::Compute(f, g, h) ⇒ {
                // Check cache, terminal cases...
                // Push continuation and recursive tasks
                stack.push(Task::Combine(m, key));
                stack.push(Task::Compute(fe, ge, he));
                stack.push(Task::Compute(ft, gt, ht));
            }
            Task::Combine(m, key) ⇒ {
                // Pop results, build node, cache
            }
        }
    }
    // Return final result
}
```

The trade-off:
- **Recursive**: Cleaner code, limited by stack size
- **Iterative**: More complex, handles arbitrarily deep BDDs

⚡ **Performance**

BDD operations are exponentially faster in release mode. Debug builds have significant overhead from bounds checking and unoptimized recursion. Always benchmark with `--release`.

9.11 Operation-Specific Optimizations

While ITE is universal, specialized implementations can be faster:

9.11.1 AND Optimization

```
pub fn apply_and(&self, f: Ref, g: Ref) → Ref {
    // Special terminal rules for AND
    if self.is_zero(f) || self.is_zero(g) {
        return self.zero;  // Short-circuit
    }
    if self.is_one(f) {
        return g;
    }
    if self.is_one(g) {
        return f;
    }
    if f == g {
        return f;  // Idempotent
    }
    if f == -g {
        return self.zero;  // Contradiction
    }
    // Fall back to ITE
    self.apply_ite(f, g, self.zero)
}
```

9.11.2 XOR and Complement Edges

XOR has a special relationship with complement edges:

$$f \oplus g = \neg(f \oplus \neg g) \tag{35}$$

This means XOR can often be computed by just flipping a complement bit:

```
pub fn apply_xor(&self, f: Ref, g: Ref) -> Ref {
    // f ⊕ 0 = f
    if self.is_zero(g) {
        return f;
    }
    // f ⊕ 1 = ¬f
    if self.is_one(g) {
        return -f;
    }
    // f ⊕ f = 0
    if f == g {
        return self.zero;
    }
    // f ⊕ ¬f = 1
    if f == -g {
        return self.one;
    }
    // General case
    self.apply_ite(f, -g, g)
}
```

9.12 Summary

The Apply/ITE algorithm is the workhorse of BDD manipulation. Its efficiency comes from:

1. **Aggressive terminal case checking**: Stop recursion as early as possible
2. **Standard triple normalization**: Maximize cache reuse
3. **Complement edge handling**: Unify equivalent computations
4. **Memoization**: Never recompute the same subproblem

The implementation in `bdd-rs` follows the classic CUDD approach, optimized for single-threaded use with interior mutability.

Chapter 10

Caching and Computed Tables

The Apply algorithm's polynomial complexity hinges on **memoization** — remembering results to avoid redundant computation. Without caching, BDD operations degrade to exponential time, no better than brute-force enumeration. This chapter explores the cache (also called the **computed table**) that transforms BDDs from a theoretical curiosity into a practical powerhouse.

10.1 Why Caching Matters

Consider computing $f \wedge g$ where both BDDs have n nodes. The recursive structure of Apply spawns a call tree that branches at every non-terminal node:

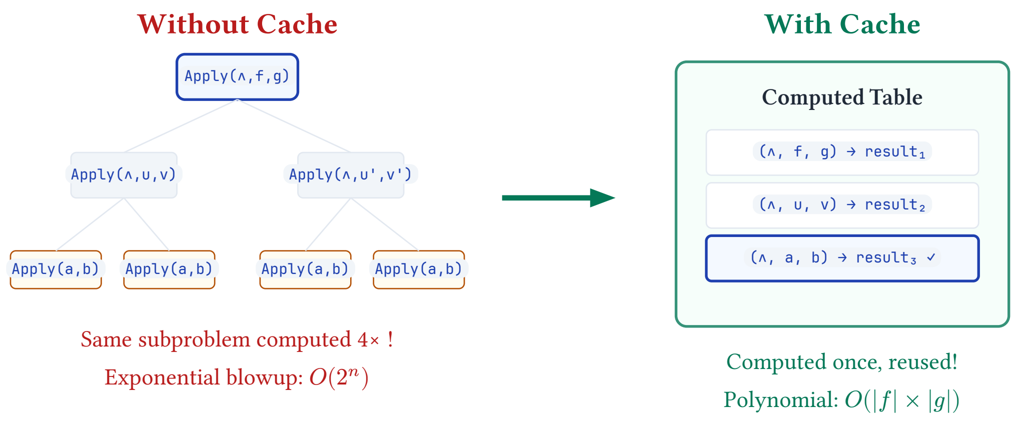


Figure 19: Without caching (left), identical subproblems are recomputed exponentially. With caching (right), each unique subproblem is solved once.

Without memoization, this tree can have exponentially many leaves. The same subproblem `Apply(AND, u, v)` appears repeatedly from different branches — and each time, we would naively recompute it from scratch.

With caching, each unique (op, u, v) triple is computed **exactly once** and stored. Since there are at most $O(|f| \times |g|)$ such triples, the algorithm achieves polynomial time.

📐 **Theorem (Caching Complexity)**

> Without memoization: $O(2^n)$ worst-case (exponential). With memoization: $O(|f| \times |g|)$ (polynomial).

10.2 Cache Structure

The cache in `bdd-rs` is a fixed-size hash table mapping operation keys to results:

```rust
pub struct Cache<K, V> {
    data: Vec<Option<Entry<K, V>>>,
    bitmask: u64,        // For fast index computation
    hits: Cell<usize>,   // Successful lookups
    faults: Cell<usize>, // Collisions (wrong key at index)
    misses: Cell<usize>, // Total unsuccessful lookups
}

struct Entry<K, V> {
    key: K,
    value: V,
}
```

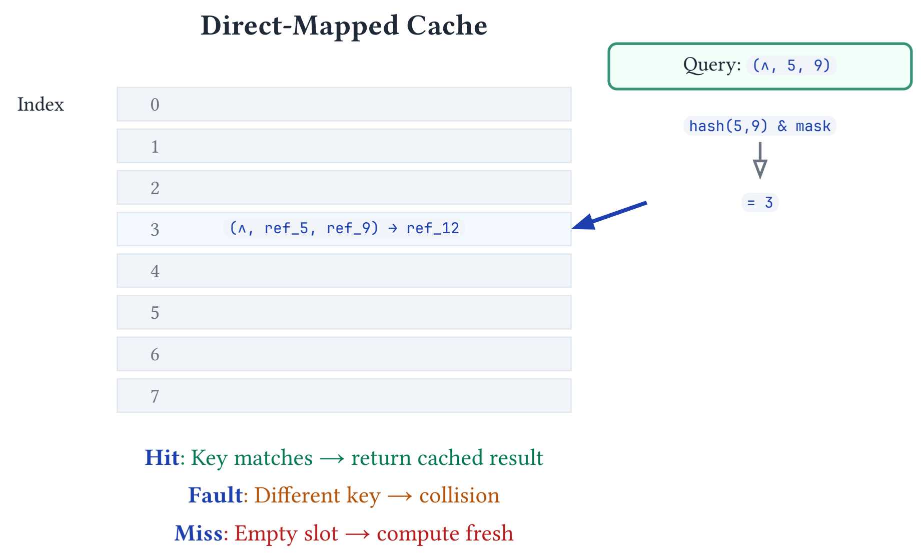

Direct-Mapped Cache
Index
0
1
2
3
(∧, ref_5, ref_9) → ref_12
4
5
6
7
Query: (∧, 5, 9)
hash(5,9) & mask
= 3
Hit: Key matches ⟶ return cached result
Fault: Different key ⟶ collision
Miss: Empty slot ⟶ compute fresh

Figure 20: The cache uses direct-mapped hashing. A query hashes to one slot; if the key matches, we have a hit.

10.2.1 Key Structure

For ITE operations, the key is a triple of references:

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub enum OpKey {
    Ite(Ref, Ref, Ref),  // ite(f, g, h)
}
```

The key is hashed to find an index:

```rust
fn index(&self, key: &K) → usize {
    (key.hash() & self.bitmask) as usize
}
```

The `bitmask` is `size - 1` where size is a power of two, making the modulo operation a simple bitwise AND.

10.2.2 Cache vs. Unique Table

It's important to distinguish these two hash-based structures:

Property	Unique Table	Computed Table (Cache)
Purpose	Hash consing (node dedup)	Memoization (result reuse)
Key	(var, low, high)	(op, f, g, h)
Value	NodeId	Ref (result)
Lifetime	Permanent	May evict on collision
Correctness	Required for canonicity	Only affects performance

The unique table is **essential** — without it, BDDs lose canonicity. The cache is an **optimization** — without it, BDDs are correct but slow.

10.3 Cache Operations

10.3.1 Lookup

```rust
pub fn get(&self, key: &K) → Option<&V>
where
    K: Eq,
{
    let index = self.index(key);
    match &self.data[index] {
        Some(entry) ⇒ {
            if &entry.key == key {
                // Cache hit: exact key match
                self.hits.set(self.hits.get() + 1);
                Some(&entry.value)
            } else {
                // Cache fault: collision (different key)
                self.faults.set(self.faults.get() + 1);
                self.misses.set(self.misses.get() + 1);
                None
            }
        }
        None ⇒ {
            // Cache miss: empty slot
            self.misses.set(self.misses.get() + 1);
            None
        }
    }
}
```

10.3.2 Insert

```rust
pub fn insert(&mut self, key: K, value: V) {
    let index = self.index(&key);
    self.data[index] = Some(Entry { key, value });
}
```

Insert unconditionally overwrites the slot. If another entry was there, it's lost (but this only affects performance, not correctness).

10.4 Cache Sizing

The cache size is specified in bits:

```rust
impl<K, V> Cache<K, V> {
    pub fn new(bits: usize) -> Self {
        assert!(bits <= 31);
        let size = 1 << bits;  // 2^bits entries
        let bitmask = (size - 1) as u64;
        // ...
    }
}
```

In `bdd-rs`, the default is 16 bits = 65,536 entries:

```rust
impl Bdd {
    pub fn new(storage_bits: usize) -> Self {
        let cache_bits = 16;
        // ...
    }
}
```

10.4.1 Sizing Trade-offs

Cache Size	Memory	Hit Rate
Use Case	2^{14} (16K)	0.5 MB
Lower	Small problems	2^{16} (64K)
2 MB	Good	Default
2^{18} (256K)	8 MB	Better
Large problems	2^{20} (1M)	32 MB
Excellent	Very large problems	

Memory estimates assume 32-byte entries (key + value + padding).

i When to Increase Cache Size

If you observe:
- Hit rate below 70%

- Large BDDs (millions of nodes)
- Many repeated operations

Consider increasing cache bits. Double the bits roughly quadruples memory but can significantly improve hit rates.

10.5 Collision Handling

`bdd-rs` uses **direct-mapped** caching: each key maps to exactly one slot. If two keys hash to the same index, the newer one overwrites the older.

10.5.1 Why Direct-Mapped?

Strategy	Complexity	Hit Rate
Memory	Direct-mapped	Simple — $O(1)$
Lower	Minimal	Set-associative
Medium — $O(k)$ for k-way	Better	Slight overhead
Fully associative	Complex — $O(n)$ or LRU	Best
Significant overhead		

Direct-mapped is the simplest and fastest, at the cost of more collisions. For BDD operations, this trade-off usually favors simplicity:

- Operations are fast, so cache overhead matters
- Collisions lose performance but not correctness
- Memory efficiency allows larger caches

10.5.2 Collision Statistics

```rust
impl Cache<K, V> {
    pub fn hits(&self) → usize;    // Successful lookups
    pub fn faults(&self) → usize; // Key mismatch (collision)
    pub fn misses(&self) → usize; // Total failures (empty + fault)
}
```

The **fault rate** indicates collision frequency:

$$\text{fault rate} = \frac{\text{faults}}{\text{hits} + \text{misses}} \tag{36}$$

High fault rates suggest:

- Cache is too small
- Hash function has poor distribution
- Working set exceeds cache capacity

🔧 **Implementation Note**

> `bdd-rs` provides cache statistics via `cache.hits()`, `cache.misses()`, and `cache.faults()`. A hit rate below 80% may indicate the cache is too small.

10.6 Hash Function Design

The cache's effectiveness depends on a good hash function. `bdd-rs` uses a combination of pairing functions and bit mixing:

```
// Szudzik pairing function
pub const fn pairing_szudzik(a: u64, b: u64) -> u64 {
    if a < b {
        b.wrapping_mul(b).wrapping_add(a)
    } else {
        a.wrapping_mul(a).wrapping_add(a).wrapping_add(b)
    }
}

// MurmurHash3 finalizer for bit mixing
pub const fn mix64(mut x: u64) -> u64 {
    x = x.wrapping_mul(0xff51afd7ed558ccd);
    x ^= x >> 33;
    x = x.wrapping_mul(0xc4ceb9fe1a85ec53);
    x ^= x >> 33;
    x
}
```

For ITE keys, the hash combines three `Ref` values:

```
impl MyHash for OpKey {
    fn hash(&self) -> u64 {
        match self {
            OpKey::Ite(f, g, h) => {
                combine3(f.raw() as u64, g.raw() as u64, h.raw() as u64)
            }
        }
    }
}
```

10.7 Multiple Caches

`bdd-rs` maintains separate caches for different purposes:

10.7.1 Operation Cache

```
cache: RefCell<Cache<OpKey, Ref>>
```

This is the main cache for ITE results. All binary operations (AND, OR, XOR, etc.) go through ITE, so this single cache covers them all.

10.7.2 Size Cache

```
size_cache: RefCell<Cache<Ref, u64>>
```

The `size` function (counting nodes in a BDD) is cached separately:

```
pub fn size(&self, node_ref: Ref) -> u64 {
    if let Some(&size) = self.size_cache().get(&node_ref) {
        return size;
    }
    // ... compute size ...
    self.size_cache_mut().insert(node_ref, size);
    size
}
```

Why separate caches?

- **Different key types**: OpKey vs. Ref
- **Different access patterns**: Size is queried less frequently
- **Cache pollution**: Mixing would reduce hit rates

10.7.3 When to Clear Caches

Caches should be cleared when:

- After variable reordering (node indices change)
- When memory pressure is high
- Starting a new, unrelated computation phase

```
// In bdd-rs reordering code:
self.cache_mut().clear();
self.size_cache_mut().clear();
```

10.8 Cache in the Apply Flow

Here’s how caching fits into the overall Apply/ITE algorithm:

⚙️ **Algorithm: Apply with Caching**

```
apply_ite(f, g, h):
  // 1. Terminal cases (no cache needed)
  if f == 1: return g
  if f == 0: return h
  // ... more terminals ...

  // 2. Normalize for cache efficiency
  if f.is_negated(): f = -f; swap(g, h)
  if g.is_negated(): negate_result = true; g = -g; h = -h

  // 3. Cache lookup
  key = (f, g, h)
  if key in cache:
    return cache[key]  // HIT: avoid recursion

  // 4. Recursive computation (cache MISS)
  m = top_variable(f, g, h)
  (f0, f1) = cofactors(f, m)
  (g0, g1) = cofactors(g, m)
  (h0, h1) = cofactors(h, m)
  e = apply_ite(f0, g0, h0)
  t = apply_ite(f1, g1, h1)
  result = mk(m, e, t)
```

```
// 5. Cache insert
cache[key] = result
return result
```

The normalization step (2) is crucial for cache efficiency. Without it, `ite(f, g, h)` and `ite(-f, h, g)` would be cached separately, wasting space and missing reuse opportunities.

10.9 Performance Analysis

10.9.1 Cache Hit Rate Impact

Consider a computation making N recursive calls:

- With 0% hit rate: All N calls execute fully
- With 50% hit rate: $N/2$ calls execute fully
- With 90% hit rate: $N/10$ calls execute fully

Since each call involves node construction, cache lookup, and potentially allocation, the savings compound.

10.9.2 Measuring Cache Effectiveness

```rust
let bdd = Bdd::default();
// ... perform operations ...

let cache = bdd.cache();
let hits = cache.hits();
let misses = cache.misses();
let faults = cache.faults();

let total = hits + misses;
let hit_rate = hits as f64 / total as f64;
let fault_rate = faults as f64 / total as f64;

println!("Hit rate: {:.1}%", hit_rate * 100.0);
println!("Fault rate: {:.1}%", fault_rate * 100.0);
```

⚠️ **Cache Pitfall**

A high hit rate doesn't always mean good performance. If the working set is small, even a tiny cache has high hit rate. Compare **absolute** hit counts and execution time, not just percentages.

10.10 Summary

The computed table is simple in concept but critical in practice:

1. **Structure**: Fixed-size hash table with direct mapping
2. **Key**: Normalized operation triple (f, g, h)
3. **Collision handling**: Overwrite (lossy but fast)
4. **Sizing**: Power of two, typically 2^{16} to 2^{20}
5. **Statistics**: Track hits, misses, faults for diagnostics

The cache transforms BDD operations from exponential to polynomial complexity, making the entire data structure practical for real-world use.

Chapter 11

Complement Edges

What if negating a BDD took zero time? Not “fast” — literally **zero**. Just flip one bit, and f becomes $\neg f$.

This is what **complement edges** achieve. They are one of the most elegant optimizations in BDD technology: negation becomes $O(1)$, memory usage drops (since f and $\neg f$ share all structure), and the `Apply` algorithm gets powerful new terminal cases.

The tradeoff? Careful bookkeeping to maintain canonicity. This chapter explains the concept, the implementation, and the subtle invariants.

11.1 The Problem: Redundant Negations

In standard BDDs, f and $\neg f$ are completely separate structures.

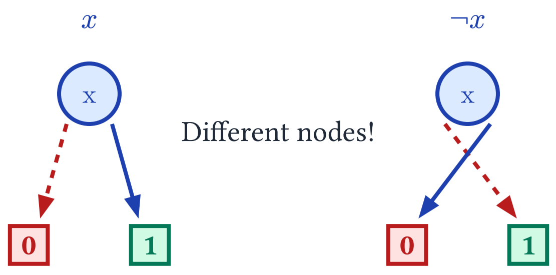
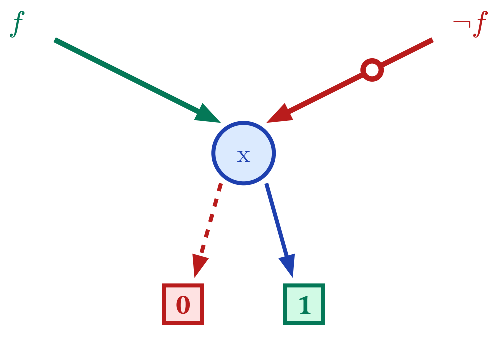


Figure 21: Without complement edges: x and $\neg x$ require separate nodes with swapped children.

This is wasteful. If your formula uses both g and $\neg g$ for some complex subformula g, you store the entire g structure twice. Negation takes $O(|g|)$ time to rebuild everything.

11.2 The Solution: Annotated Edges

The insight: instead of negating the **nodes**, negate the **edge**.



Same node, different edge annotations.
The circle marks a **complement edge**.

Figure 22: With complement edges: f and $\neg f$ share the same node. The small circle indicates negation.

📘 **Definition (Complement Edge)**

A **complement edge** is an edge annotated with a negation flag. Following a complemented edge **inverts** the semantics of the subgraph. This allows f and $\neg f$ to share the same underlying structure.

11.3 The Benefits

11.3.1 Space Reduction

Every function f in the BDD implicitly provides $\neg f$ for free. In practice, this can reduce node count by 30–50%.

Consider a formula like $(a \wedge b) \oplus (c \wedge d)$. XOR involves negation: $p \oplus q = (p \wedge \neg q) \vee (\neg p \wedge q)$. Without complement edges, we'd duplicate the $(a \wedge b)$ and $(c \wedge d)$ subgraphs. With them, we just mark edges as complemented.

11.3.2 $O(1)$ Negation

This is the killer feature. Negating a BDD becomes a single bit operation:

```rust
impl Neg for Ref {
    fn neg(self) -> Self {
        Self(self.0 ^ 1)  // XOR flips the lowest bit
    }
}
```

In Rust, you just write `-f` and get the negated BDD instantly. No traversal. No new nodes. No allocation.

💡 **Key Insight**

The $O(1)$ negation ripples through the entire library. XOR, equivalence, and implication all involve negation internally. They all become faster because negation is free.

11.4 Implementation in bdd-rs

The complement bit is packed into the `Ref` type itself:

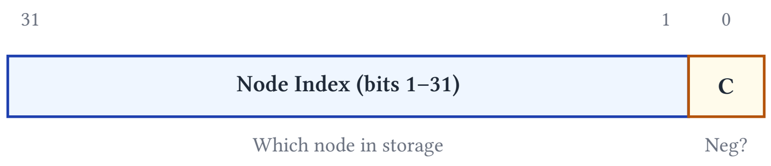


Figure 23: Bit layout of `Ref`: the complement flag occupies the lowest bit.

This encoding means:

- `Ref(0)` = node 0, positive = **TRUE** (terminal)
- `Ref(1)` = node 0, negated = **FALSE**
- `Ref(4)` = node 2, positive
- `Ref(5)` = node 2, negated

The API is straightforward:

```rust
impl Ref {
    pub fn id(self) → NodeId {
        NodeId::from_raw(self.0 >> 1)  // Shift right to get index
    }

    pub fn is_negated(self) → bool {
        (self.0 & 1) ≠ 0  // Check lowest bit
    }
}
```

11.5 The Canonicity Challenge

Complement edges create an ambiguity. Consider a node for variable x:

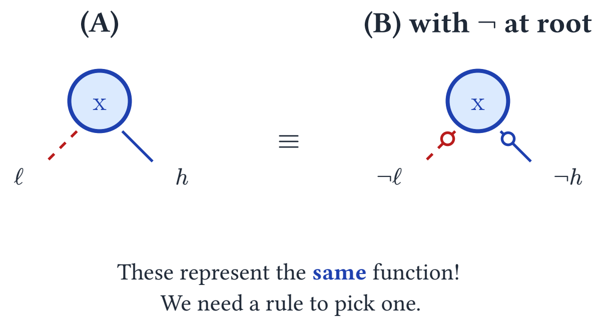

(A)
(B) with ¬ at root
x
ℓ
h
≡
x
¬ℓ
¬h
These represent the same function!
We need a rule to pick one.

Figure 24: Ambiguity: `mk(x, l, h)` and `¬mk(x, ¬l, ¬h)` represent the same function.

Without a normalization rule, canonicity breaks — two different structures for the same function.

11.5.1 The Solution: High Edge Convention

`bdd-rs` enforces: **high edges are never complemented**.

When `mk_node` would create a node with a complemented high edge, it flips everything:

```
pub fn mk_node(&self, v: Var, low: Ref, high: Ref) → Ref {
    // Canonicity rule: high edge must be positive
    if high.is_negated() {
        // Flip: ¬mk(v, ¬low, ¬high) = mk(v, low, high)
        return -self.mk_node(v, -low, -high);
    }
    // ... proceed with normal node creation
}
```

⚠️ **The Rule Must Be Consistent**

The choice of "high never complemented" vs "low never complemented" is arbitrary. What matters is **consistency**. CUDD uses low-never-complemented. `bdd-rs` uses high-never-complemented. Pick one and stick to it.

11.6 Traversal with Complements

When traversing a BDD, complement flags must propagate correctly:

```
pub fn low_node(&self, node_ref: Ref) → Ref {
    let low = self.low(node_ref.id());
    if node_ref.is_negated() {
        -low  // Propagate the complement
    } else {
        low
    }
}

pub fn high_node(&self, node_ref: Ref) → Ref {
    let high = self.high(node_ref.id());
    if node_ref.is_negated() {
        -high  // Propagate the complement
    } else {
        high
    }
}
```

If you're at a complemented reference, both children become complemented. This propagation continues to the terminals, flipping the final result.

i **Evaluation Rule**

A path evaluates to TRUE if it reaches the TRUE terminal through an **even** number of complement edges. An odd number of complements flips the result to FALSE.

11.7 Terminal Handling

With complements, there's one physical terminal but two logical constants:

Complement edges introduce an ambiguity: f and $\neg f$ have the same underlying graph. To maintain canonicity, we need a **normalization rule**.

11.7.1 The Problem

Consider a node with children `low` and `high`:

- `mk(x, low, high)` represents $(\neg x \land \text{low}) \lor (x \land \text{high})$
- `mk(x, -low, -high)` represents $(\neg x \land \neg\, \text{low}) \lor (x \land \neg\, \text{high})$

These are negations of each other! Without normalization, we'd have two representations for the same function.

11.7.2 The Solution: High Edge Convention

`bdd-rs` uses the convention: **high edges are never complemented**.

If we try to create a node where `high.is_negated()`, we flip everything:

```rust
pub fn mk_node(&self, v: Var, low: Ref, high: Ref) -> Ref {
    // Canonicity: high edge must be positive
    if high.is_negated() {
        return -self.mk_node(v, -low, -high);
    }
    // ... rest of mk_node
}
```

⚠️ **Normalization Required**

With complement edges, we must enforce a normalization rule to maintain canonicity. In `bdd-rs`, the convention is: **high edges are never complemented**. If a high edge would be complemented, we complement the entire node instead.

11.7.3 Why High Edge?

The choice of "high edge never complemented" vs "low edge never complemented" is arbitrary but must be consistent. The high-edge convention is common in the literature and has some advantages:

- Terminal 1 is the "natural" positive terminal
- Following the high edge often represents the "true" case

11.7.4 Alternative: Low Edge Convention

Some libraries use the opposite: **low edges are never complemented**. CUDD, for example, uses this convention. The important thing is consistency, not the specific choice.

11.8 Impact on Traversal

When traversing a BDD with complement edges, we must propagate the complement flag:

```rust
pub fn low_node(&self, node_ref: Ref) -> Ref {
    let low = self.low(node_ref.id());
    if node_ref.is_negated() {
        -low  // Propagate complement
    } else {
        low
    }
}
```

```
pub fn high_node(&self, node_ref: Ref) -> Ref {
    let high = self.high(node_ref.id());
    if node_ref.is_negated() {
        -high  // Propagate complement
    } else {
        high
    }
}
```

If we're at a negated reference, both children become negated. This maintains the semantic invariant: following a complemented edge negates everything below.

i Propagation Rule

When traversing through a complemented reference:

- The children are also complemented
- This propagates down to the terminals

A path to terminal 1 through an odd number of complemented edges evaluates to 0.

11.9 Impact on Terminal Detection

With complement edges, we have one physical terminal node but two logical constants:

```
// Physical: one terminal node at index 0
// Logical: one = @0, zero = ~@0

impl Bdd {
    pub fn is_one(&self, r: Ref) -> bool {
        r.raw() == self.one.raw()  // Non-negated terminal
    }

    pub fn is_zero(&self, r: Ref) -> bool {
        r.raw() == self.zero.raw()  // Negated terminal
    }

    pub fn is_terminal(&self, r: Ref) -> bool {
        r.id().raw() == 0  // Either constant
    }
}
```

The distinction:

- `is_terminal(r)`: Is this a constant (either 0 or 1)?
- `is_one(r)` / `is_zero(r)`: Which specific constant?

11.10 Impact on Operations

11.10.1 ITE Normalization

The Apply/ITE algorithm must normalize its arguments for cache efficiency:

```
pub fn apply_ite(&self, f: Ref, g: Ref, h: Ref) -> Ref {
    // ...
    // Normalize: f should not be negated
    // ite(¬f, g, h) = ite(f, h, g)
    let (mut f, mut g, mut h) = (f, g, h);
    if f.is_negated() {
        f = -f;
        std::mem::swap(&mut g, &mut h);
    }

    // Normalize: g should not be negated
    // ite(f, ¬g, h) = ¬ite(f, g, ¬h)
    let mut negate_result = false;
    if g.is_negated() {
        negate_result = true;
        g = -g;
        h = -h;
    }

    // ... compute result ...

    if negate_result { -result } else { result }
}
```

This normalization ensures that equivalent computations hit the same cache entry:

- `ite(f, g, h)` and `ite(-f, h, g)` ⟶ same cache key
- `ite(f, g, h)` and `ite(f, -g, -h)` ⟶ same cache key (result negated)

11.10.2 XOR Simplification

XOR has a special relationship with complement edges:

$$f \oplus g = \neg(f \oplus \neg g) \tag{37}$$

This means many XOR computations reduce to existing results with a complement:

```
pub fn apply_xor(&self, f: Ref, g: Ref) → Ref {
    // f ⊕ 1 = ¬f (just flip the bit!)
    if self.is_one(g) {
        return -f;
    }
    // f ⊕ ¬f = 1
    if f == -g {
        return self.one;
    }
    // General case: ite(f, ¬g, g)
    self.apply_ite(f, -g, g)
}
```

11.10.3 Equivalence Testing

Equivalence becomes truly $O(1)$:

```
// f ≡ g iff they're the same Ref (including complement bit)
f == g  // Direct comparison

// f ≡ ¬g iff f == -g
f == -g
```

No traversal needed — just compare the 32-bit values.

11.11 Trade-offs

11.11.1 Advantages

1. **Space**: Up to 50% fewer nodes
2. **Negation**: $O(1)$ instead of $O(|f|)$
3. **XOR/Equivalence**: Significant speedups
4. **Equality**: $O(1)$ for both $f = g$ and $f = \neg g$

11.11.2 Disadvantages

1. **Complexity**: Every algorithm must handle complements correctly
2. **Bugs**: Easy to forget complement propagation
3. **Cache keys**: Must normalize to avoid redundant entries
4. **Debugging**: Output is harder to interpret

Aspect	Without Complement Edges	With Complement Edges
Negation	$O(\|f\|)$	$O(1)$
Node count	Higher	Up to 50% less
Equality check	$O(1)$	$O(1)$
Algorithm complexity	Simpler	More complex
Debugging	Easier	Harder

11.11.3 When Complement Edges Hurt

In rare cases, complement edges can slightly hurt cache performance. The normalization in ITE can cause cache thrashing if the working set has many complementary pairs. However, this is unusual — the space and negation benefits almost always dominate.

11.12 Visualization Considerations

When outputting BDDs (e.g., to DOT format), complement edges require special handling:

```
// In DOT output:
if edge.is_negated() {
    // Show as dashed line or with "~" label
    println!("  {} -> {} [style=dashed];", from, to);
} else {
    println!("  {} -> {};", from, to);
}
```

Without this, the output would be confusing — two identical-looking graphs could represent different functions.

11.13 Summary

Complement edges are a powerful optimization:

1. **Representation**: Encode negation in the low bit of `Ref`
2. **Canonicity**: Enforce "high edge never complemented" rule

3. **Traversal**: Propagate complement flag through children
4. **Operations**: Normalize arguments for cache efficiency
5. **Benefits**: $O(1)$ negation, space reduction, faster XOR

The added complexity is well worth the performance gains. Nearly all production BDD libraries use complement edges.

PART III

Advanced Topics

Chapter 12

Variable Ordering

Variable ordering is the Achilles' heel of BDDs. The same Boolean function can have a linear-sized BDD under one ordering and an exponential-sized BDD under another. Understanding and controlling variable ordering is essential for practical BDD use — it often makes the difference between a solution in milliseconds and a computation that never terminates.

12.1 Why Ordering Matters

The size of a BDD depends **critically** on the variable ordering. This is not merely a constant factor — it is the difference between tractable and intractable, between success and failure.

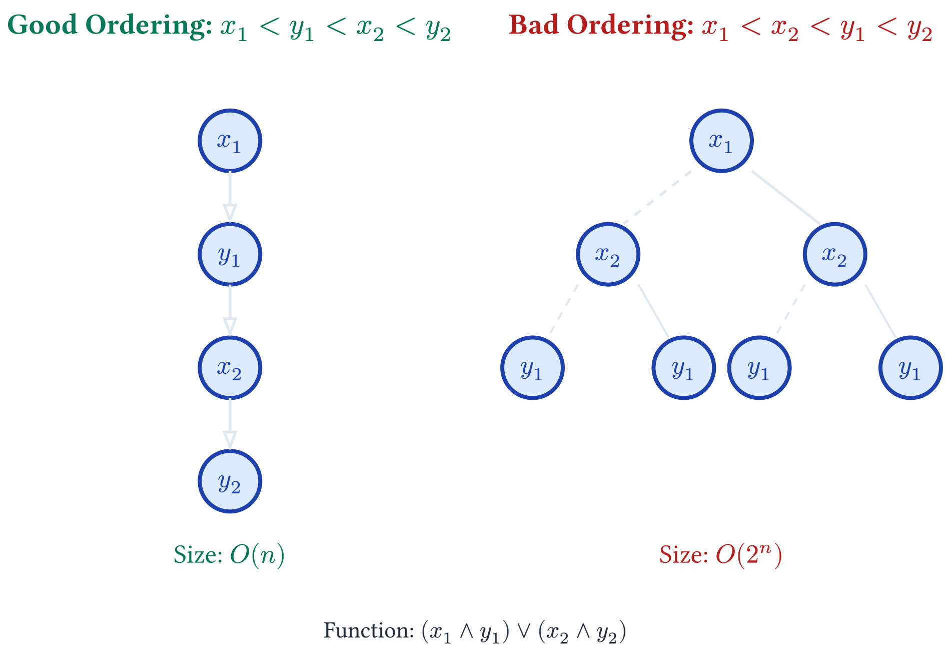

Good Ordering: $x_1 < y_1 < x_2 < y_2$
Bad Ordering: $x_1 < x_2 < y_1 < y_2$
x_1
y_1
x_2
y_2
x_1
x_2
x_2
y_1
y_1
y_1
y_1
Size: $O(n)$
Size: $O(2^n)$
Function: $(x_1 \wedge y_1) \vee (x_2 \wedge y_2)$

Figure 25: The same function with different orderings: linear vs. exponential size.

⚠️ **Ordering Can Make or Break Performance**

For the function $f = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \dots \vee (x_n \wedge y_n)$:
- **Interleaved ordering** $x_1 < y_1 < x_2 < y_2 < \dots$: $O(n)$ nodes

- **Separated ordering** $x_1 < x_2 < \ldots < y_1 < y_2 < \ldots$: $O(2^n)$ nodes

The difference grows with n — for $n = 20$, this is 40 nodes vs. over a million!

12.1.1 The Intuition

Why does interleaved ordering work better? Consider what happens when we decide $x_1 = 0$ in each ordering:

- **Interleaved**: The term $x_1 \land y_1$ becomes 0. Variable y_1 is tested next, but it doesn't matter — we move directly to x_2. The subproblem simplifies.
- **Separated**: After deciding all x_i, we still need to track **which** x_i were 1 to know which y_i matter. This "remembering" causes exponential blowup.

💡 **Key Insight**

Good orderings keep **related variables close together**. When x and y appear in a term $x \land y$, testing them consecutively allows early simplification.

12.2 Static Ordering Heuristics

Before building a BDD, we can choose an initial ordering based on the structure of the input.

12.2.1 DFS Ordering

For circuits, a depth-first traversal from outputs to inputs often produces good orderings:

```
fn dfs_ordering(circuit: &Circuit) → Vec<Var> {
    let mut order = Vec::new();
    let mut visited = HashSet::new();

    for output in circuit.outputs() {
        dfs_visit(output, &mut order, &mut visited);
    }
    order
}
```

Variables encountered earlier in DFS tend to be "closer" to outputs and get lower indices.

12.2.2 FORCE Algorithm

The FORCE algorithm iteratively improves ordering by minimizing a "span" metric — how far apart related variables are placed:

⚙️ **Algorithm: FORCE Heuristic**

```
FORCE(clauses, iterations):
  ordering = initial_random_ordering()

  for i in 1..iterations:
    for each variable v:
      // Compute center of gravity
      cog = average position of variables
            that share a clause with v

      // Move v toward its COG
      target[v] = cog

    // Sort variables by target position
    ordering = sort_by(target)

  return ordering
```

12.3 Dynamic Variable Reordering

Even with good heuristics, the initial ordering may become suboptimal as computation proceeds. **Dynamic reordering** adjusts the ordering during BDD operations.

12.3.1 The Sifting Algorithm

Sifting (Rudell, 1993) is the most widely used reordering algorithm:

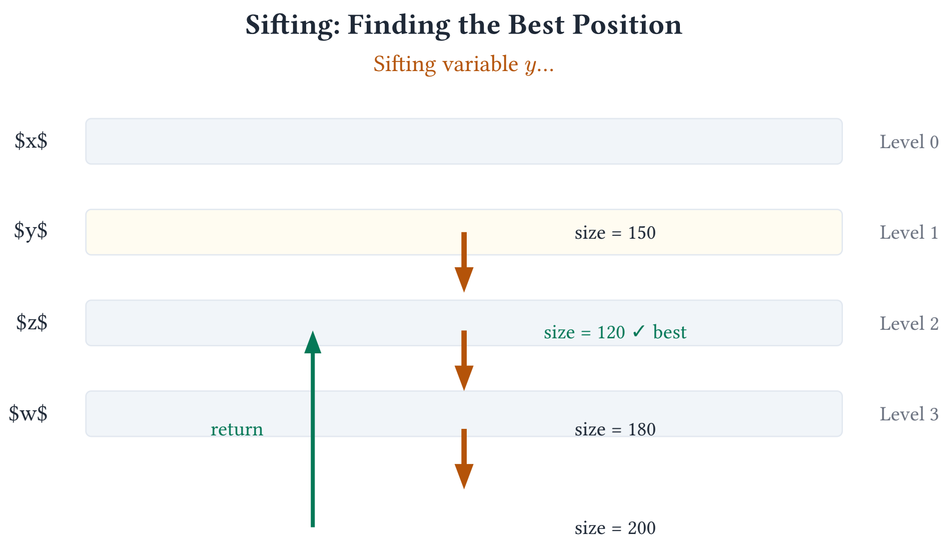


Figure 26: Sifting moves a variable through all positions, tracking where total BDD size is minimized.

Algorithm: Sifting (Rudell 1993)

```
Sifting():
  for each variable v (in decreasing size order):
    best_pos = current_level(v)
    best_size = total_nodes()

    // Move v down through all levels
    while not at_bottom(v):
      swap_adjacent(v, below(v))
      if total_nodes() < best_size:
        best_pos = current_level(v)
        best_size = total_nodes()

    // Move v back up to best position
    while current_level(v) > best_pos:
      swap_adjacent(above(v), v)
```

12.3.2 Level Swapping

The core operation in sifting is **swapping adjacent levels**. This is a local operation that only affects nodes at those two levels:

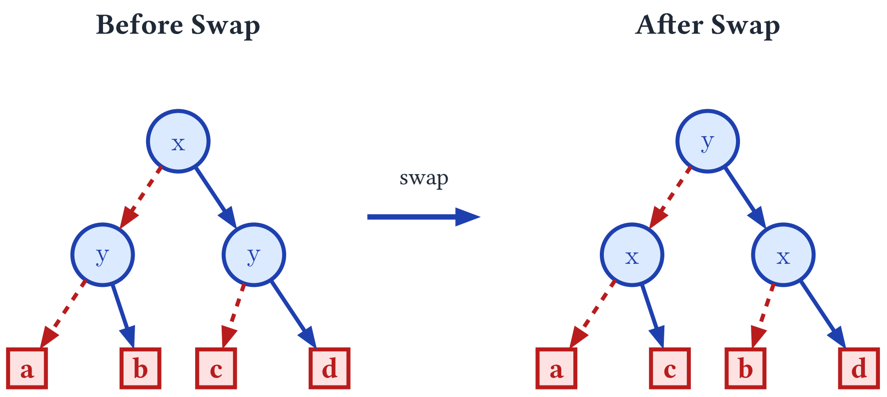


Figure 27: Swapping levels x and y restructures the BDD while preserving the function.

The swap operation:

1. Takes each node at the upper level
2. Computes new children based on the grandchildren
3. Creates new nodes at both levels as needed
4. Updates the unique table

12.4 Reordering in bdd-rs

`bdd-rs` provides explicit variable ordering control through the `var_order` and `level_map` data structures:

```rust
// Get the level of a variable
let level: Level = bdd.get_level(var)?;

// Get the variable at a level
let var: Var = bdd.get_var_at_level(level);

// Iterate variables in order
for level in 0..bdd.num_levels() {
    let var = bdd.get_var_at_level(Level(level));
    // Process variables top-to-bottom
}
```

i Current Capabilities

`bdd-rs` currently supports:
- Explicit variable ordering during construction
- Level/variable mapping queries
- Manual ordering specification

Dynamic reordering (sifting) is planned for future releases.

12.5 Ordering for Specific Domains

Different problem domains have different optimal ordering strategies.

12.5.1 Circuits

For circuits, follow signal flow — inputs near the top, internal signals in the middle, outputs near the bottom. Variables that share gates should be adjacent.

12.5.2 Transition Relations

For symbolic model checking with transition relations on state variables $(s_1, ..., s_n)$ and next-state variables $(s'_1, ..., s'_n)$:

- **Interleaved**: $s_1 < s'_1 < s_2 < s'_2 < ...$ (usually best)
- **Separated**: $s_1 < s_2 < ... < s'_1 < s'_2 < ...$ (often exponential)

12.5.3 Arithmetic

For arithmetic functions like addition or multiplication:
- **MSB-first**: Often better for multiplication
- **LSB-first**: Often better for addition
- Experiment with both!

Domain	Strategy	Rationale
Circuits	Follow signal flow	Related signals stay close
FSMs	Interleave state/next	Transition locality
Arithmetic	Try both MSB/LSB	Problem-dependent
Feature models	Respect hierarchy	Parent before children

Chapter 13

Garbage Collection

BDD operations create nodes. Lots of nodes. An operation like `f AND g` may produce thousands of intermediate nodes, only to discard most of them when the final result emerges. Without cleanup, memory consumption grows without bound.

This chapter covers garbage collection — the art of reclaiming dead nodes while preserving live ones.

13.1 The Memory Problem

Every `mk` call potentially allocates a new node. Every Apply operation recursively calls `mk` many times. Consider this innocent-looking code:

```
let a = bdd.and(x, y);
let b = bdd.and(a, z);
let c = bdd.or(b, w);    // We only care about c
```

What happens to the intermediate nodes created for `a` and `b`? If `c` shares structure with them, they're still needed. If not, they're **garbage** — consuming memory but serving no purpose.

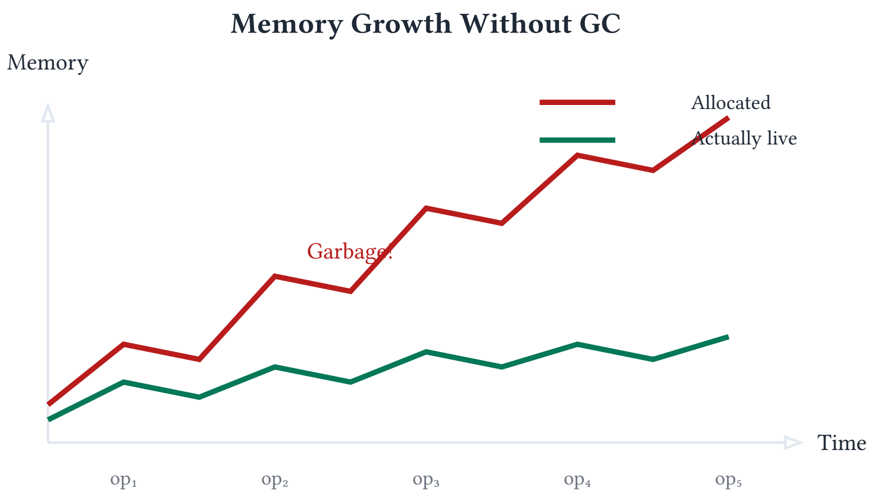


Figure 28: Without garbage collection, allocated memory grows far beyond what's actually needed.

⚠️ **The Danger of Unbounded Growth**

A verification run might perform millions of BDD operations. Without GC, you'll run out of memory long before finding the bug you're looking for.

13.2 What Counts as Garbage?

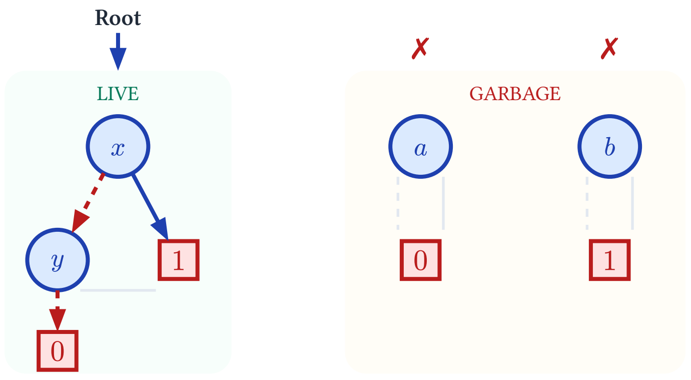
📘 **Definition (Reachability)**

A node is **live** (reachable) if:

1. It's a **root** — a BDD the user explicitly keeps, or
2. It's reachable from a root via low/high edges

Everything else is **garbage**.

Live vs. Garbage Nodes



No root points to these nodes ⇒ they can be reclaimed

Figure 29: Live nodes are reachable from a root; garbage nodes have no path from any root.

💡 **Key Insight**

Shared structure complicates things. A node might be reachable from **multiple** roots. We can only reclaim it when **all** roots that could reach it are gone.

13.3 Mark-and-Sweep Collection

The classic garbage collection algorithm has two phases:

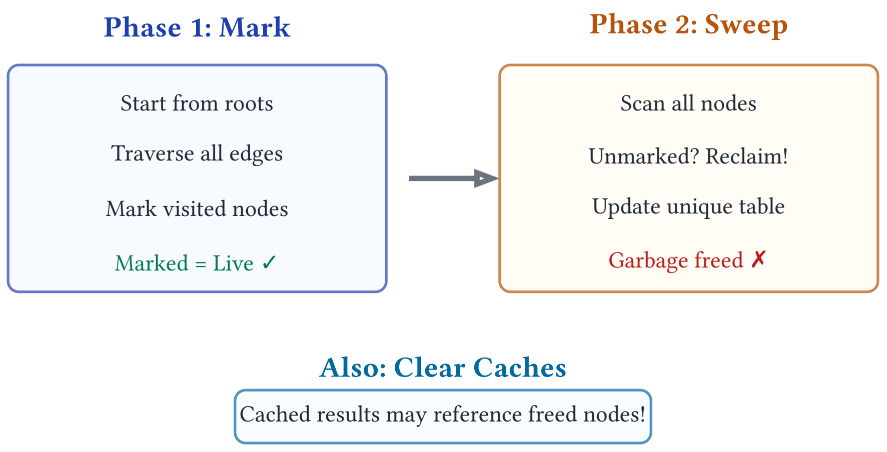


Figure 30: Mark-and-sweep GC: mark what's live, sweep away the rest.

⚙️ Algorithm: Mark-and-Sweep GC

```
collect_garbage(roots):
  // ═══ MARK PHASE ═══
  marked = empty_bitset(num_nodes)
  for root in roots:
    mark_recursive(root, marked)

  // ═══ SWEEP PHASE ═══
  for node_id in 1..num_nodes:  // Skip terminal at 0
    if not marked[node_id]:
      // Remove from unique table
      level = get_level(node_id)
      subtables[level].remove(node_id)
      // Add to free list for reuse
      free_set.insert(node_id)

  // ═══ CACHE INVALIDATION ═══
  operation_cache.clear()
  size_cache.clear()

mark_recursive(ref, marked):
  if ref.is_terminal() or marked[ref.id()]:
    return
  marked[ref.id()] = true
  node = nodes[ref.id()]
  mark_recursive(node.low, marked)
  mark_recursive(node.high, marked)
```

13.3.1 Why Clear Caches?

The operation cache stores entries like `(AND, f, g) → h`. If node `h` gets garbage-collected, this cache entry becomes **dangling** — it points to freed memory.

⚠️ Dangling Cache Entries

If we don't clear caches after GC:

1. A future operation looks up `(AND, f, g)`
2. Cache returns stale reference `h`

3. The slot for `h` might now hold a completely different node!
4. Result: silent corruption, wrong answers

Always invalidate caches after garbage collection.

13.4 Reference Counting Alternative

Instead of periodic mark-and-sweep, we could track references to each node:

Aspect	Mark-and-Sweep	Reference Counting
When reclaimed	During GC pauses	Immediately when count hits 0
Overhead	None between GCs	Every reference update
Pause times	Can be long	None (incremental)
Cycles	Handles fine	Problematic (but DAGs don't cycle)
Implementation	Simpler	Pervasive ref management

i Why bdd-rs Uses Mark-and-Sweep

BDDs are DAGs (no cycles), so reference counting **would** work. However:
- Reference count updates add overhead to **every** operation
- BDD operations are cache-bound; extra memory traffic hurts
- Batch GC integrates well with explicit root management

Most BDD libraries, including CUDD, use mark-and-sweep.

13.5 GC in bdd-rs

`bdd-rs` uses **explicit** garbage collection — you decide when to collect and what to keep:

```
// Build some BDDs
let formula1 = bdd.and(x, y);
let formula2 = bdd.or(y, z);
let combined = bdd.and(formula1, formula2);

// We only need 'combined' going forward
bdd.collect_garbage(&[combined]);

// Now formula1 and formula2 may have been freed
// (unless 'combined' shares structure with them)
```

The `free_set` tracks which node slots are available for reuse:

```
pub struct Bdd {
    nodes: RefCell<Vec<Node>>,
    free_set: RefCell<HashSet<NodeId>>,  // Available slots
    // ...
}
```

```
fn allocate_node(&self, node: Node) → NodeId {
    let mut free_set = self.free_set.borrow_mut();
    if let Some(&id) = free_set.iter().next() {
        // Reuse freed slot
        free_set.remove(&id);
        self.nodes.borrow_mut()[id.raw()] = node;
        id
    } else {
        // Allocate new slot
        let id = NodeId::new(self.nodes.borrow().len() as u32);
        self.nodes.borrow_mut().push(node);
        id
    }
}
```

13.6 When to Collect

💡 **Key Insight**

GC is expensive — it touches **all** live nodes and clears **all** caches. Time it wisely.

Good times to collect:

1. **After major phases**: Finished building a transition relation? Collect before model checking.
2. **When memory is tight**: Monitor allocation and trigger GC at thresholds.
3. **Before long-running operations**: A clean heap means better cache behavior.

Bad times to collect:

- **Inside tight loops**: The overhead dominates.
- **Mid-computation**: You might need those “intermediate” results!
- **Without knowing your roots**: You’ll lose data.

```
// Good: collect between phases
let transition_rel = build_transition_relation(&bdd);
bdd.collect_garbage(&[transition_rel]);

let reachable = compute_reachability(&bdd, transition_rel);
bdd.collect_garbage(&[reachable]);

// Bad: collect inside a loop
for i in 0..1000 {
    let step = bdd.and(current, constraint);
    bdd.collect_garbage(&[step]);  // DON'T DO THIS!
    current = step;
}
```

⚡ **Performance**

As a rule of thumb: if your BDD manager has grown to 2× the live node count, it’s time to collect. Many libraries trigger automatic GC at such thresholds.

Chapter 14

Quantification and Abstraction

Quantification eliminates variables from Boolean functions. It answers questions like "does **some** assignment to x make f true?" (existential) or "is f true for **every** value of x?" (universal).

These operations are the bridge between BDDs and symbolic reasoning. Model checking, constraint solving, and image computation all rely heavily on quantification to project away variables and reduce problem dimensionality.

14.1 Existential Quantification

📘 **Definition (Existential Quantification)**

The existential quantification of f with respect to variable x is:

$$\exists x.f = f|_{x=0} \lor f|_{x=1} \tag{38}$$

Intuitively, $\exists x.f$ is true if f can be made true by **some** choice of x.

The result is a function that no longer depends on x. We've "projected away" that variable.

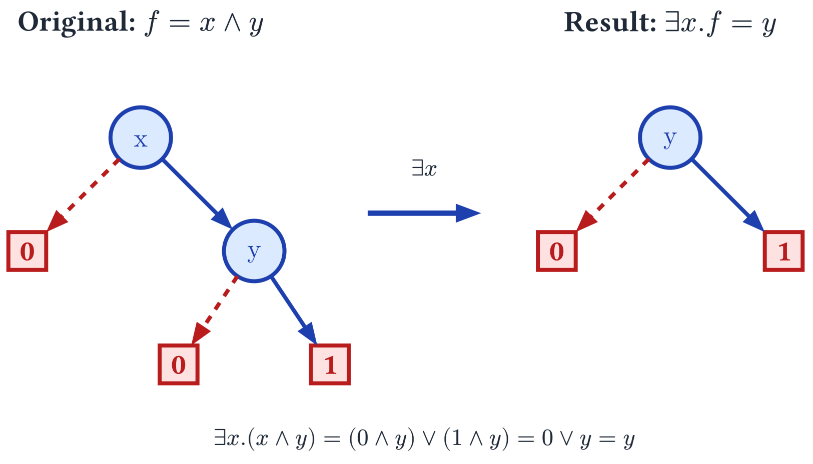

Original: $f = x \land y$
Result: $\exists x.f = y$
x
y
0
0
1
$\exists x$
y
0
1
$\exists x.(x \land y) = (0 \land y) \lor (1 \land y) = 0 \lor y = y$

Figure 31: Existential quantification eliminates x by OR-ing the two cofactors.

14.1.1 Computing Existential Quantification

```
pub fn exists(&self, f: Ref, x: Var) → Ref {
    let low = self.cofactor(f, x, false);   // f|_{x=0}
    let high = self.cofactor(f, x, true);   // f|_{x=1}
    self.or(low, high)                       // low ∨ high
}
```

14.2 Universal Quantification

Definition (Universal Quantification)

The universal quantification of f with respect to variable x is:

$$\forall x.f = f|_{x=0} \land f|_{x=1} \quad (39)$$

$\forall x.f$ is true only if f is true for **all** choices of x.

Universal quantification is the dual of existential quantification. It's more restrictive — requiring f to hold regardless of x.

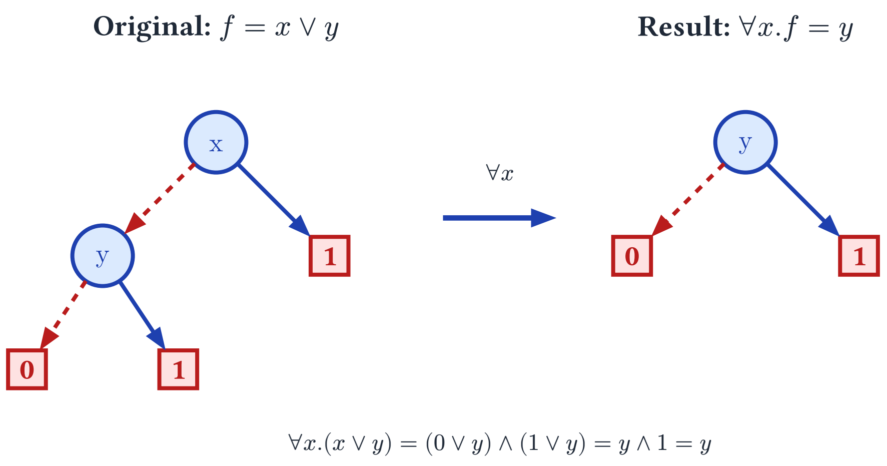

Original: $f = x \lor y$
Result: $\forall x.f = y$
x
y
0
1
1
$\forall x$
y
0
1
$\forall x.(x \lor y) = (0 \lor y) \land (1 \lor y) = y \land 1 = y$

Figure 32: Universal quantification eliminates x by AND-ing the two cofactors.

14.2.1 Computing Universal Quantification

```
pub fn forall(&self, f: Ref, x: Var) → Ref {
    let low = self.cofactor(f, x, false);   // f|_{x=0}
    let high = self.cofactor(f, x, true);   // f|_{x=1}
    self.and(low, high)                      // low ∧ high
}
```

14.3 Multiple Variable Quantification

Often we need to quantify over multiple variables at once. The order of quantification matters for performance (though not correctness).

💡 **Key Insight**

Rule of thumb: Quantify variables from the bottom of the BDD upward. This keeps intermediate results smaller because bottom variables affect fewer nodes.

14.3.1 Cube-Based Quantification

Variables to quantify are often represented as a **cube** — a conjunction of variables:

```
// Quantify over {x, y, z} represented as cube = x ∧ y ∧ z
pub fn exists_cube(&self, f: Ref, cube: Ref) → Ref {
    if self.is_one(cube) {
        return f;  // No more variables to quantify
    }
    let var = self.top_var(cube);
    let remaining = self.high(cube);  // Rest of cube
    let quantified = self.exists(f, var);
    self.exists_cube(quantified, remaining)
}
```

14.4 Relational Product

The **relational product** (also called **and-exists**) combines conjunction with existential quantification:

$$\text{RelProd}(f, g, X) = \exists X.(f \wedge g) \tag{40}$$

This operation is critical for symbolic model checking, where it computes reachable states.

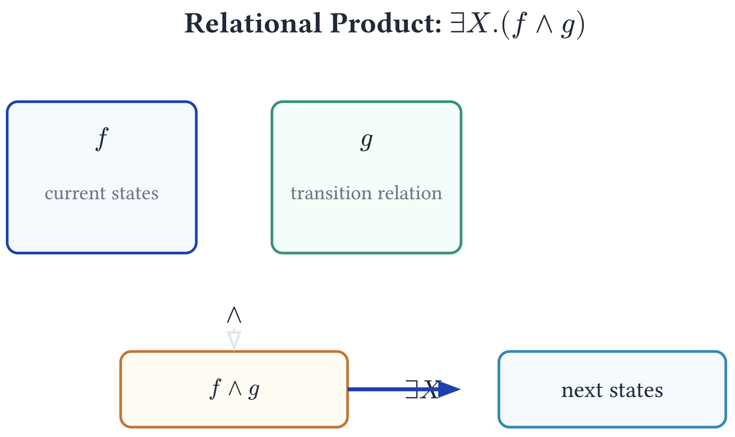


Figure 33: Relational product computes the image (next states) in one combined operation.

14.4.1 Why Combine Operations?

Computing $\exists X.(f \wedge g)$ as two separate steps can create huge intermediate results. The combined algorithm quantifies variables **during** the conjunction, keeping BDDs smaller.

⚙️ **Algorithm: Relational Product (And-Exists)**

```
RelProduct(f, g, cube):
  if f == 0 or g == 0:
    return 0
  if is_one(cube):
    return And(f, g)
  if f == 1 and g == 1:
    return 1

  v = top_var_of(f, g, cube)

  if v in cube:
    // Quantify this variable
    f_low, f_high = cofactors(f, v)
    g_low, g_high = cofactors(g, v)
    cube' = remove(v, cube)

    r_low  = RelProduct(f_low, g_low, cube')
    r_high = RelProduct(f_high, g_high, cube')
    return Or(r_low, r_high)
  else:
    // Regular conjunction recursion
    // ... standard Apply-style recursion
```

14.5 Complexity Considerations

Quantification can cause significant BDD growth.

⚠️ Quantification Blowup

While cofactors can only shrink, their OR (or AND) can create BDDs much larger than the original. A single existential quantification can square the BDD size in the worst case.

14.5.1 Early Quantification

When computing $\exists x.(f_1 \wedge f_2 \wedge \ldots \wedge f_n)$:

- **Naive**: Compute all conjunctions, then quantify
- **Better**: Quantify x as soon as it appears in only one remaining term

💡 Key Insight

Early quantification keeps intermediate BDDs smaller by eliminating variables as soon as they're no longer needed. This is a key optimization in symbolic model checking.

14.6 Applications

Quantification is used throughout symbolic reasoning:

Application	Operation	Purpose
Reachability	$\exists X.(R \wedge T)$	Compute next states
Verification	$\forall X.(P \to Q)$	Check implication

Projection	$\exists Y.f$	Hide internal variables
Consensus	$\forall x.f$	Variables that don't matter

Chapter 15

BDD Variants

Standard ROBDDs are just one point in a rich design space. By tweaking the reduction rules, terminal values, or structural constraints, we get different data structures — each optimized for different problem domains.

This chapter surveys the most important variants: ZDDs for sparse combinatorics, ADDs for numeric computations, and others that have found their niche.

15.1 Beyond Binary Decisions

ROBDDs represent functions $f : \mathbb{B}^n \to \mathbb{B}$ — Boolean inputs, Boolean output. But many real problems involve:

- **Sparse sets**: Most elements absent (SAT solving, graph algorithms)
- **Numeric values**: Probabilities, costs, timing constraints
- **Multi-valued logic**: More than two truth values

Different BDD variants address each of these needs.

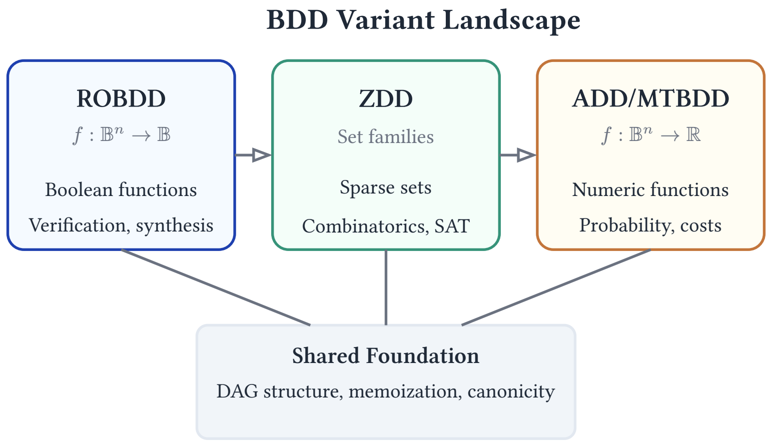


Figure 34: BDD variants share core ideas but differ in reduction rules and terminal values.

15.2 Zero-Suppressed BDDs (ZDDs)

ZDDs, introduced by Minato in 1993, are optimized for **sparse set families** — collections where most elements are absent from most sets.

📘 **Definition (Zero-Suppressed Decision Diagram)**

A **ZDD** uses a different reduction rule than BDDs:

- **BDD rule**: Delete node if low = high
- **ZDD rule**: Delete node if high = 0 (redirect to low)

This makes ZDDs compact when sets are sparse.

The key insight: in a family of sparse sets, most elements are **absent** from most sets. The ZDD rule lets us skip mentioning absent elements entirely.

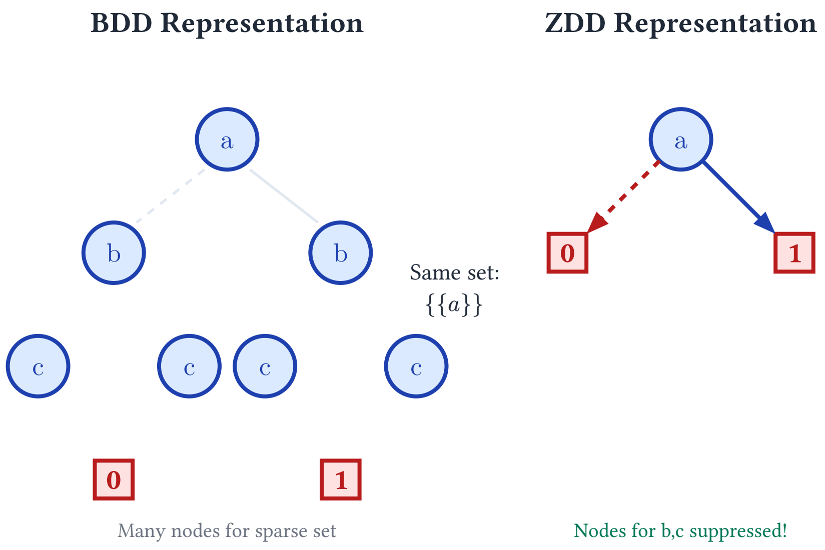


Figure 35: For the set family $\{\{a\}\}$ over universe $\{a, b, c\}$: BDD needs nodes for all variables; ZDD only needs one.

15.2.1 When to Use ZDDs

Use Case	BDD	ZDD
Dense Boolean functions	Excellent	Poor
Sparse set families	Poor	Excellent
Combinatorial enumeration	Moderate	Excellent
Graph problems (cliques, paths)	Moderate	Excellent
Circuit verification	Excellent	Moderate

💡 **Key Insight**

ZDDs shine in problems where you're enumerating **sets of things** — solutions to combinatorial problems, satisfying assignments, graph substructures. Knuth dedicated an entire section of TAOCP Volume 4B to ZDDs.

15.3 Algebraic Decision Diagrams (ADDs)

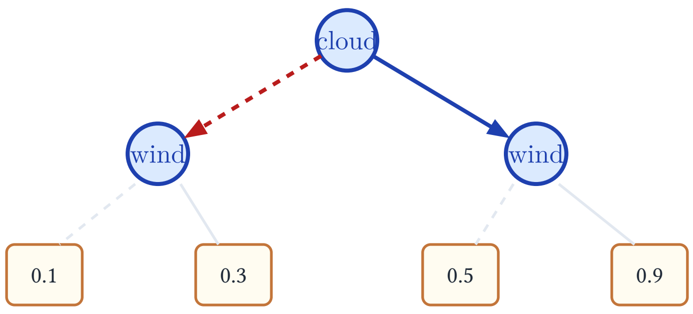
ADDs (also called MTBDDs — Multi-Terminal BDDs) generalize BDDs to functions with arbitrary terminal values.

📘 **Definition (Algebraic Decision Diagram)**

An **ADD** represents functions $f : \mathbb{B}^n \to D$ where D is any set. Common choices:

- $D = \mathbb{R}$ (real numbers) for probabilities, costs
- $D = \mathbb{Z}$ (integers) for counts
- $D = \{0, 1, ..., k\}$ for multi-valued logic

ADD Example: Probability Function



$P(\text{rain} \mid \text{cloud}, \text{wind})$: probability depends on weather conditions

Figure 36: An ADD representing probability of rain given cloud cover and wind conditions.

15.3.1 Applications of ADDs

- **Probabilistic model checking**: Represent transition probabilities
- **Cost functions**: Assign costs to state combinations
- **Reward structures**: Accumulate rewards over paths
- **Matrix operations**: Sparse matrix-vector multiplication

```
// Conceptual ADD operations
fn add_value(&self, f: AddRef, g: AddRef) → AddRef {
    // Terminal case: add numeric values
    // Recursive case: like BDD Apply
}

fn max_value(&self, f: AddRef, g: AddRef) → AddRef {
    // Take maximum at terminals
}
```

15.4 Edge-Valued BDDs (EVBDDs)

EVBDDs put numeric values on **edges** rather than at terminals. This can be more compact for certain functions.

📘 **Definition (Edge-Valued BDD)**

An **EVBDD** has:

- A single terminal node (value 0)
- Numeric weights on each edge
- Function value = sum (or product) of edge weights on path

This is particularly compact for linear arithmetic functions.

EVBDD for $f = 2x + 3y$

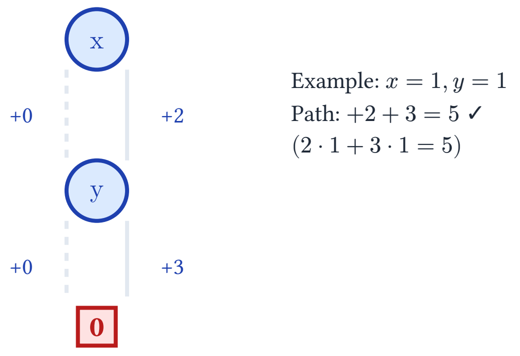


Figure 37: EVBDD encodes $2x + 3y$ with edge weights; only one terminal needed.

15.5 Free BDDs (FBDDs)

Free BDDs relax the variable ordering constraint:

📘 **Definition (Free BDD)**

In an **FBDD** (Free BDD), each **path** can use a different variable ordering. The only constraint: each variable appears **at most once** per path.

FBDDs are also called **read-once branching programs**.

Property	ROBDD	FBDD
Ordering	Global (same for all paths)	Per-path (can vary)
Canonicity	Yes (given ordering)	No
Compactness	Depends on ordering	Always at least as good
Operations	Efficient (polynomial)	Can be expensive
Equivalence check	$O(1)$	coNP-complete

⚠️ **The Price of Freedom**

FBDDs can be exponentially smaller than any ROBDD, but:

- They're **not canonical** — equivalence checking is hard
- Operations like AND can be expensive
- Most BDD libraries don't support them

FBDDs are mainly of theoretical interest and for specific one-shot computations.

15.6 Choosing the Right Variant

💡 **Key Insight**

The best variant depends on your problem:

- **Verification/synthesis**: ROBDD (canonical, efficient operations)
- **Combinatorial enumeration**: ZDD (sparse sets)
- **Probabilistic reasoning**: ADD/MTBDD (numeric terminals)
- **Arithmetic functions**: EVBDD (edge values)

`bdd-rs` focuses on ROBDDs with complement edges — the sweet spot for most verification tasks.

PART IV

Applications

Chapter 16

Model Checking with BDDs

In 1987, Ken McMillan demonstrated that BDDs could verify systems with 10^{20} states — far beyond what explicit enumeration could ever hope to reach. This breakthrough, called **symbolic model checking**, revolutionized hardware verification and remains one of BDDs' most celebrated applications.

This chapter shows how BDDs transform the impossible into routine.

16.1 The State Explosion Problem

Consider a simple system: 100 flip-flops. How many possible states? $2^{100} \approx 10^{30}$. At a billion states per second, explicit enumeration would take 10^{13} **years** — longer than the age of the universe.

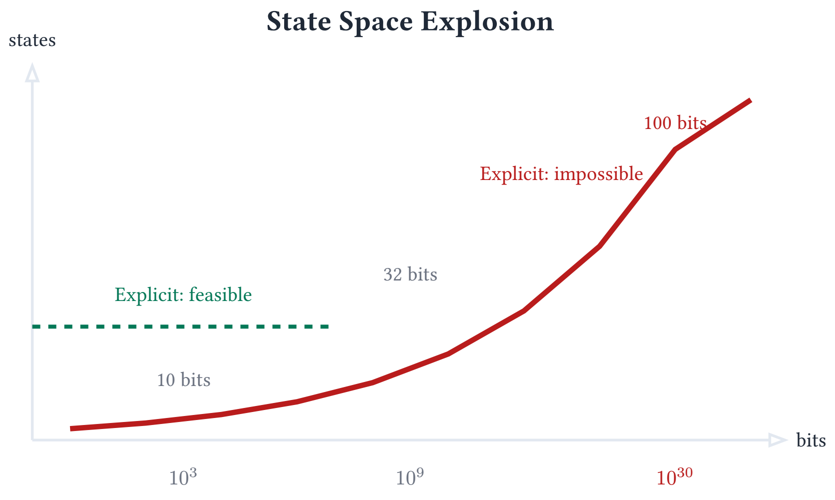


Figure 38: State spaces grow exponentially with the number of state bits.

Yet real hardware has thousands of flip-flops. The key insight: **we don't need to enumerate states individually**.

💡 **Key Insight**

BDDs represent **sets of states** as Boolean functions. A BDD with 1000 nodes can represent 10^{20} states. Operations on BDDs manipulate entire sets at once.

16.2 Encoding States and Transitions

The key insight of symbolic model checking is representing state **sets** as Boolean functions rather than enumerating individual states.

A finite-state system consists of:

- **State variables** $\boldsymbol{x} = (x_1, \ldots, x_n)$: Boolean variables encoding the current state
- **Next-state variables** $\boldsymbol{x}' = (x'_1, \ldots, x'_n)$: Primed copies for the next state
- **Transition relation** $T(\boldsymbol{x}, \boldsymbol{x}')$: A Boolean function capturing **all** legal transitions
- **Initial states** $I(\boldsymbol{x})$: Boolean function true exactly on starting states
- **Property** $P(\boldsymbol{x})$: Boolean function characterizing “good” or “bad” states

✏️ **Example — Two-Bit Counter**

A counter with state bits x_0, x_1 counts $0 \to 1 \to 2 \to 3 \to 0 \to \ldots$

Transition relation:

$$T(x_0, x_1, x'_0, x'_1) = (x'_0 \leftrightarrow \neg x_0) \wedge (x'_1 \leftrightarrow (x_1 \oplus x_0)) \tag{41}$$

Initial state (start at 0):

$$I(x_0, x_1) = \neg x_0 \wedge \neg x_1 \tag{42}$$

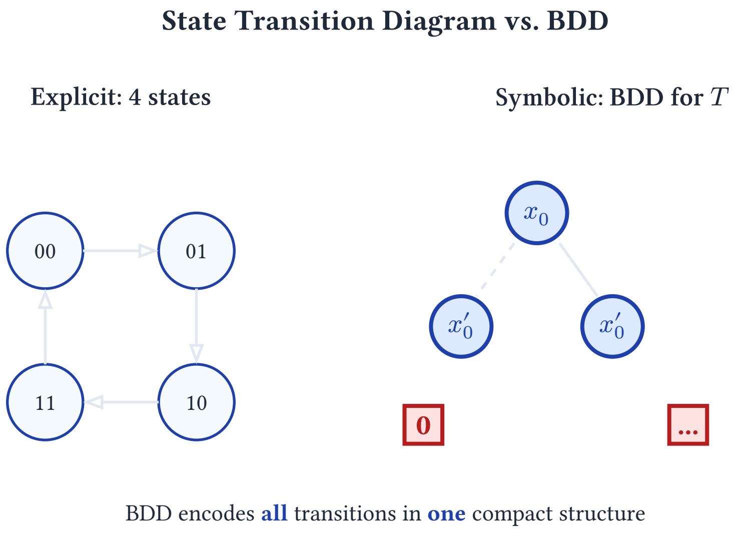

State Transition Diagram vs. BDD
Explicit: 4 states
Symbolic: BDD for T
00
01
10
11
x_0
x'_0
x'_0
0
...
BDD encodes **all** transitions in **one** compact structure

Figure 39: Explicit representation lists states; symbolic representation encodes the transition function.

16.3 Reachability Analysis

The fundamental question: **can the system ever reach a bad state?**

If we can compute the set of all reachable states, verification becomes trivial: check whether any bad state is reachable. The challenge is computing this set without enumerating states individually.

16.3.1 Forward Reachability

Starting from initial states, repeatedly compute successors until reaching a **fixpoint** — a point where no new states appear:

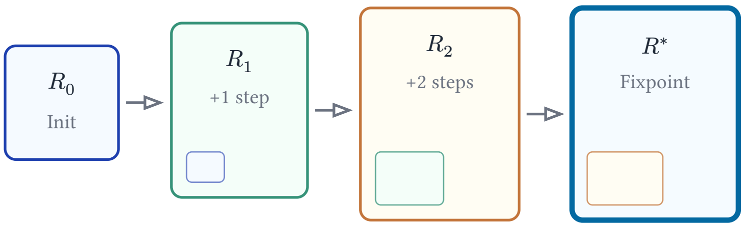
⚙️ Algorithm: Symbolic Reachability (Forward)

```
Reachable(Init, Trans):
  Reached = Init
  repeat:
    Reached_old = Reached
    Reached = Reached U Image(Reached, Trans)
  until Reached == Reached_old
  return Reached
```

The **Image** operation computes successor states:

$$\text{Image}(S,T) = \exists \boldsymbol{x}.T(\boldsymbol{x},\boldsymbol{x}') \wedge S(\boldsymbol{x}) \quad (43)$$

Symbolic Reachability



Each R_i is a **single BDD** representing 10^{20} states

Figure 40: Reachability iterates until the set of reached states stabilizes.

16.3.2 Why This Works

The magic: each iteration manipulates **BDDs**, not individual states. Consider the scale difference:

- S might represent 10^{15} states — more than atoms in a human body
- The BDD for S might have only 5000 nodes
- Image computation operates on these 5000 nodes, taking milliseconds
- Result: a BDD for the next frontier, perhaps 6000 nodes representing 10^{16} states

💡 Key Insight

BDD size depends on the **structure** of the function, not the number of satisfying assignments. Regular, structured systems often have compact BDD representations even with huge state spaces.

16.4 CTL Model Checking

CTL (Computation Tree Logic) expresses temporal properties:

- EF P: "Eventually P might hold" (exists a path where P eventually true)

- AG P: “Always globally P” (on all paths, P always true)
- EG P: “Exists a path where P is always true”
- AF P: “On all paths, P eventually holds”

📘 Definition (CTL Semantics via Fixpoints)

CTL operators can be computed as fixpoints:

- $\text{EF } P = \mu Z.(P \lor \text{EX } Z)$ — least fixpoint
- $\text{EG } P = \nu Z.(P \land \text{EX } Z)$ — greatest fixpoint
- $\text{AF } P = \mu Z.(P \lor \text{AX } Z)$
- $\text{AG } P = \nu Z.(P \land \text{AX } Z)$

Where $\text{EX } S = \text{PreImage}(S)$ and $\text{AX } S = \neg \text{ EX}(\neg S)$.

⚙️ Algorithm: Computing EF (Backward Reachability)

```
EF(P, Trans):
  Z = False        // Start with empty set
  repeat:
    Z_old = Z
    Z = P v PreImage(Z, Trans)
  until Z == Z_old
  return Z         // States that can reach P
```

16.5 Image and PreImage Computation

These are the workhorses of symbolic model checking:

Operation	Computes	Formula
Image	Successors of S	$\exists \boldsymbol{x}.T(\boldsymbol{x}, \boldsymbol{x}') \land S(\boldsymbol{x})$
PreImage	Predecessors of S	$\exists \boldsymbol{x}'.T(\boldsymbol{x}, \boldsymbol{x}') \land S(\boldsymbol{x}')$

Implementation uses **relational product** (And-Exists) for efficiency:

```
fn image(&self, states: Ref, trans: Ref) → Ref {
    // Conjoin states with transition relation
    let conjoined = self.and(states, trans);
    // Quantify out current-state variables
    let next = self.exists_cube(conjoined, current_vars);
    // Rename x' back to x
    self.rename(next, next_to_current)
}
```

⚠️ The Image Bottleneck

Image computation is often the bottleneck in model checking. Optimizations include:

- **Partitioned transition relations**: Split T into smaller pieces
- **Early quantification**: Quantify variables as soon as possible
- **Transition clustering**: Group related transitions

16.6 Practical Example: Mutual Exclusion

Consider verifying Peterson's mutual exclusion algorithm with two processes:

```
// State variables per process: {idle, trying, critical}
// Need at least 2 bits per process = 4 bits total

let trans = build_peterson_transition(&bdd);
let init = build_initial_state(&bdd);

// Bad states: both processes in critical section
let bad = bdd.and(p1_critical, p2_critical);

// Check: can we reach bad states?
let reachable = symbolic_reachability(&bdd, init, trans);
let bad_reachable = bdd.and(reachable, bad);

if bdd.is_zero(bad_reachable) {
    println!("Verified: mutual exclusion holds!");
} else {
    println!("Bug found! Extracting counterexample...");
    let cex = extract_counterexample(&bdd, init, trans, bad);
}
```

16.7 Limitations and Modern Approaches

BDD-based model checking, despite its power, has fundamental limits:

- **BDD blowup**: Some functions (like multiplication) have exponential BDDs regardless of variable ordering
- **Memory pressure**: Intermediate BDDs during image computation can be 10-100× larger than the final result
- **Ordering sensitivity**: The "right" ordering can mean the difference between seconds and days

Modern alternatives have emerged:

- **SAT-based BMC**: Bounded model checking unrolls transitions k times and asks "is a bad state reachable in k steps?"
- **IC3/PDR**: Property-directed reachability builds proofs incrementally, often without BDDs entirely
- **Hybrid approaches**: Use BDDs for control logic, SAT for data paths

i When to Use BDDs

BDDs excel when:

- The system is highly regular (hardware, protocols)
- You need to **count** or **enumerate** states
- The property involves complex temporal patterns

- You'll verify many properties on the same system

Consider SAT/SMT when:

- The instance is too big for BDDs
- You only need to find **one** bug (not prove absence)
- The system has irregular, data-dependent structure

Chapter 17

Combinatorial Problems

BDDs aren't just for verification — they're a remarkably powerful tool for combinatorics. Where SAT solvers find **one** satisfying assignment, BDDs can count **all** of them. Where constraint solvers enumerate solutions one by one, BDDs represent the entire solution space as a single, compact data structure.

This chapter explores encoding constraints, counting solutions, and solving classic combinatorial puzzles.

17.1 Constraint Encoding

The pattern is elegant in its simplicity:

1. Encode each constraint as a Boolean function (BDD)
2. Conjoin all constraints: $f = c_1 \wedge c_2 \wedge \ldots \wedge c_k$
3. The resulting BDD represents **every** feasible solution

Constraint Conjunction

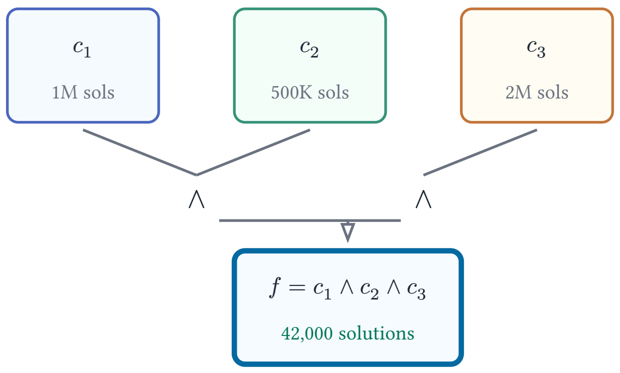


Figure 41: Conjoining constraint BDDs yields a BDD representing exactly the feasible solutions.

💡 **Key Insight**

Once you have the constraint BDD:

- **SAT check**: Is the BDD non-zero? ($O(1)$)

- **Count solutions**: Traverse the BDD ($O(|f|)$)
- **Enumerate solutions**: Walk all paths to 1
- **Random sample**: Weighted path selection

17.2 The N-Queens Problem

Place n queens on an $n \times n$ chessboard so no two attack each other. This classic problem illustrates BDD-based constraint solving beautifully.

17.2.1 Encoding

For each cell (i, j), variable $x_{i,j}$ indicates whether a queen is present.

✏️ **Example — N-Queens Constraints**

For an $n \times n$ board:

1. **At least one queen per row**: $\bigvee_{j=1}^{n} x_{i,j}$ for each row i
2. **At most one queen per row**: For each row i, pairs (j, k) with $j < k$:

$$\neg(x_{i,j} \wedge x_{i,k}) \tag{44}$$

3. **Column constraints**: Similar to rows
4. **Diagonal constraints**: For each diagonal, at most one queen

4-Queens: One Solution

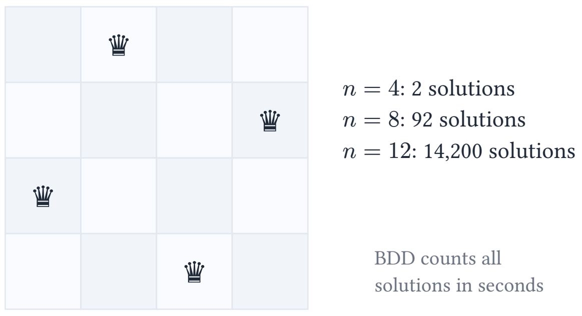

$n = 4$: 2 solutions
$n = 8$: 92 solutions
$n = 12$: 14,200 solutions
BDD counts all solutions in seconds

Figure 42: One of the two solutions to the 4-Queens problem.

17.2.2 Implementation

```rust
fn n_queens(bdd: &Bdd, n: usize) → Ref {
    let mut vars = vec![vec![Ref::default(); n]; n];

    // Create variables for each cell
    for i in 0..n {
        for j in 0..n {
            vars[i][j] = bdd.variable(/* (i, j) index */);
        }
    }
    let mut constraint = bdd.one();

    // Row constraints
    for i in 0..n {
        // At least one queen in row i
        let at_least_one = vars[i].iter().fold(bdd.zero(), |acc, &v| bdd.or(acc, v));
        constraint = bdd.and(constraint, at_least_one);

        // At most one queen in row i
        for j in 0..n {
            for k in (j + 1)..n {
                let not_both = bdd.nand(vars[i][j], vars[i][k]);
                constraint = bdd.and(constraint, not_both);
            }
        }
    }

    // Similar for columns and diagonals...
    constraint
}

// Count solutions
let queens_bdd = n_queens(&bdd, 8);
let count = bdd.sat_count(queens_bdd, 64);  // 64 variables for 8x8
println!("8-Queens has {} solutions", count);  // 92
```

17.3 Graph Coloring

Given a graph $G = (V, E)$, can we color vertices with k colors so adjacent vertices differ?

17.3.1 Encoding

For each vertex v and color c, variable $x_{v,c}$ indicates "vertex v has color c".

✏️ **Example — Graph Coloring Constraints**

For k-coloring:

1. **Each vertex has at least one color**: $\bigvee_{c=1}^{k} x_{v,c}$
2. **Each vertex has at most one color**: $\neg\left(x_{v,c_1} \wedge x_{v,c_2}\right)$ for $c_1 \neq c_2$
3. **Adjacent vertices differ**: For each edge $(u, v) \in E$ and color c:

$$\neg\left(x_{u,c} \wedge x_{v,c}\right) \tag{45}$$

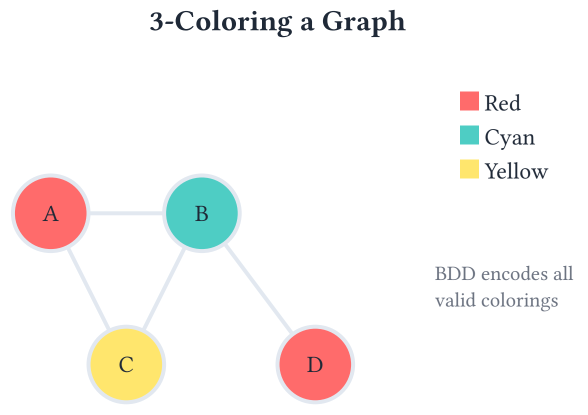


Figure 43: A valid 3-coloring. The BDD encodes all possible valid colorings simultaneously.

17.4 Satisfiability and #SAT

BDDs provide **O(1)** SAT checking and **O(|f|)** solution counting:

Query	BDD Complexity	Notes
SAT (is f satisfiable?)	$O(1)$	Just check if BDD ≠ 0
#SAT (count solutions)	$O(\|f\|)$	Dynamic programming on BDD
All-SAT (enumerate all)	$O(\text{output})$	Traverse paths to 1
Random solution	$O(n)$	Weighted random walk

17.4.1 Solution Counting Algorithm

⚙️ Algorithm: Solution Counting

```
SatCount(f, n):   // n = number of variables
  if f == 0: return 0
  if f == 1: return 2^n  // All assignments satisfy

  // Memoized recursion
  if f in CountCache: return CountCache[f]

  v = var(f)
  // Account for "skipped" variables above v
  skip_factor = 2^(v - expected_var)

  low_count = SatCount(low(f), n - v - 1)
  high_count = SatCount(high(f), n - v - 1)

  result = skip_factor * (low_count + high_count)
  CountCache[f] = result
  return result
```

⚠️ Skipped Variables

If the BDD jumps from variable x_1 to x_5, variables x_2, x_3, x_4 are “don’t cares”. Each skipped variable doubles the solution count (either value works).

17.5 Combinatorial Enumeration

Beyond counting, BDDs support efficient enumeration:

```rust
// Enumerate all solutions
fn all_solutions(bdd: &Bdd, f: Ref) → Vec<Vec<bool>> {
    let mut solutions = Vec::new();
    let mut path = Vec::new();

    fn traverse(bdd: &Bdd, node: Ref, path: &mut Vec<bool>, solutions: &mut Vec<Vec<bool>>) {
        if bdd.is_zero(node) { return; }
        if bdd.is_one(node) {
            solutions.push(path.clone());
            return;
        }

        // Try low branch (variable = false)
        path.push(false);
        traverse(bdd, bdd.low(node), path, solutions);
        path.pop();

        // Try high branch (variable = true)
        path.push(true);
        traverse(bdd, bdd.high(node), path, solutions);
        path.pop();
    }

    traverse(bdd, f, &mut path, &mut solutions);
    solutions
}
```

💡 **Key Insight**

Enumeration is efficient when you need **all** solutions. For huge solution spaces, use **random sampling**: at each node, randomly choose low/high weighted by solution counts in each subtree.

17.6 Comparison with SAT Solvers

Aspect	BDDs	SAT Solvers
Finding one solution	Build BDD, then $O(n)$	Often faster directly
Counting solutions	Excellent: $O(\|f\|)$	Hard: specialized #SAT
Enumerating all	Excellent: traverse BDD	Requires blocking clauses
Memory	Can explode	Usually modest
Incremental solving	Add constraints = AND	Native support
Certificates	BDD is certificate	Proof traces

i Choosing the Right Tool

Use BDDs when:

- You need to count or enumerate solutions
- The same constraint structure appears repeatedly
- The problem has exploitable regularity

Use SAT solvers when:

- You only need one solution (or unsatisfiability)
- The instance is very large
- The structure is irregular or data-dependent

Chapter 18

Symbolic Execution and Program Analysis

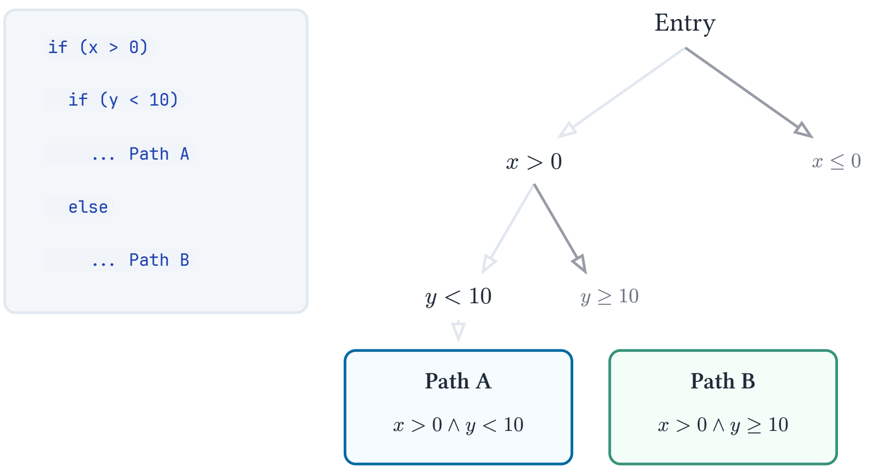
Symbolic execution flips the script on program testing: instead of feeding concrete inputs and watching what happens, we feed **symbols** and track **all** possible behaviors at once. The key challenge? A program with just 32 `if` statements has 2^{32} paths.

BDDs cut through this explosion. By representing path conditions — the Boolean constraints determining which paths are feasible — as BDDs, we can reason about exponentially many paths without enumerating them.

18.1 Path Conditions as BDDs

In symbolic execution, each program path has a **path condition**: a Boolean formula over input symbols that is satisfied exactly when execution takes that path.

Path Conditions in a Program



```
if (x > 0)
  if (y < 10)
    ... Path A
  else
    ... Path B
```

Figure 44: Each path through the program has a corresponding path condition.

💡 **Key Insight**

BDDs for path conditions enable:

- **Feasibility checking**: Is a path reachable? (SAT check)
- **Path merging**: Combine paths with same effect (OR)
- **Condition refinement**: Add constraints along execution (AND)
- **Complement paths**: What inputs avoid this path? (NOT)

18.2 Control Flow Encoding

We encode program structure as Boolean constraints:

✏️ **Example — Boolean Program Encoding**

For each program point p_i and branch condition c_j:

- Variable at_i: "execution is at point i"
- Transition: $\mathrm{at}_{i+1} \leftrightarrow \mathrm{at}_i \wedge c_j$ (if branch taken)

The BDD f satisfying $\mathrm{at}_{\mathrm{error}}$ represents all inputs reaching an error state.

Control Flow as BDD Constraints

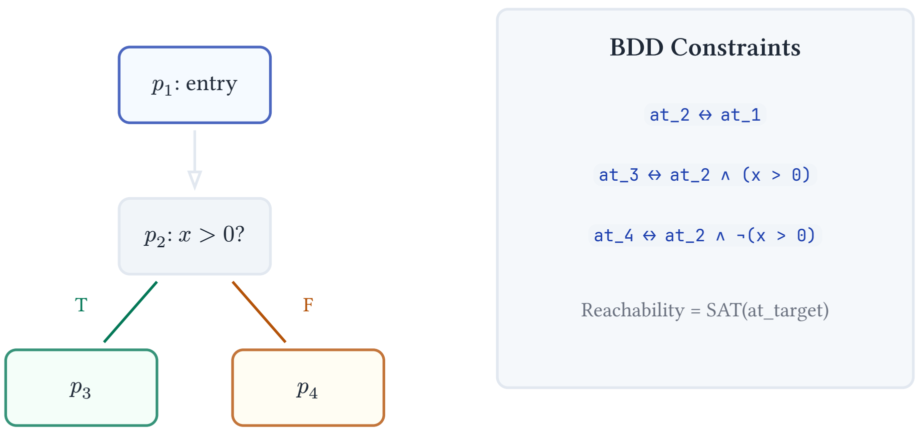


Figure 45: Control flow graph encoded as BDD constraints.

18.3 Path Merging

A key advantage of BDD-based analysis: **merge** paths that reach the same point.

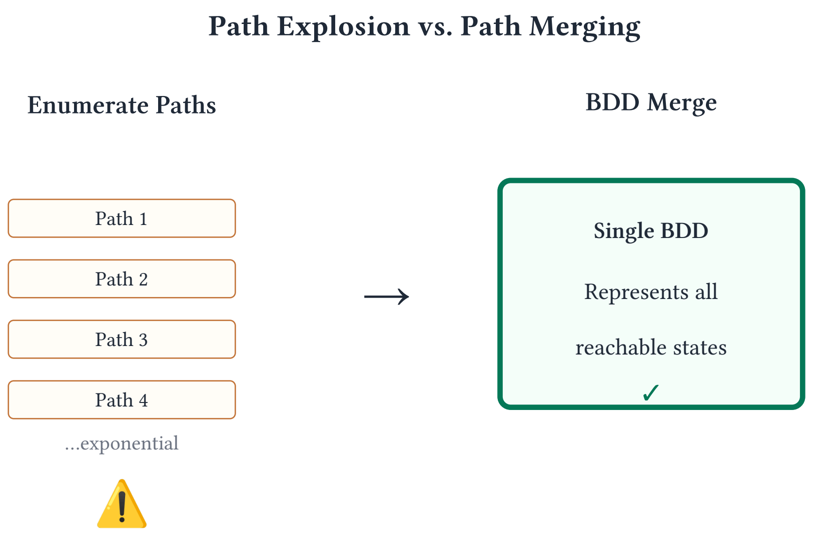


Figure 46: BDDs merge exponentially many paths into a single symbolic representation.

⚠️ When Merging Loses Information

Path merging is **sound** but may lose **precision**. If you need to distinguish paths (e.g., for debugging), track path conditions separately. The trade-off: precision vs. scalability.

18.4 Abstract Interpretation with BDDs

BDDs serve as an **abstract domain** for path-sensitive analysis:

📘 Definition (BDD Abstract Domain)

A program state is abstracted as:

- **Path condition** (BDD): Which inputs reach this point
- **Value constraints** (another domain): What values variables hold

Abstract operations:

- **Branch**: Conjoin branch condition to path BDD
- **Join**: Disjoin (OR) path BDDs at merge points
- **Widen**: For loops, extrapolate to fixed point

Analysis Type	Path Sensitivity	BDD Role
Flow-insensitive	None	Not needed
Flow-sensitive	Statement order	Track reaching definitions
Path-sensitive	Branch conditions	BDDs track full path conditions
Context-sensitive	Call sites	BDDs encode calling context

18.5 Test Case Generation

BDDs enable systematic test generation:

```
// Generate test inputs from path condition BDD
fn generate_tests(bdd: &Bdd, path_condition: Ref, input_vars: &[Var]) → Vec<TestInput> {
    let mut tests = Vec::new();

    // Each satisfying assignment is a test case
    for assignment in bdd.all_sat(path_condition) {
        let test = TestInput {
            values: input_vars.iter()
                .map(|&v| assignment[v])
                .collect(),
        };
        tests.push(test);
    }

    tests
}

// Incremental: find input for uncovered path
fn cover_new_path(bdd: &Bdd, uncovered: Ref, covered: Ref) → Option<TestInput> {
    // Find path not yet covered
    let new_path = bdd.and(uncovered, bdd.not(covered));
    if bdd.is_zero(new_path) {
        return None;  // All paths covered!
    }

    // Get one satisfying assignment
    bdd.any_sat(new_path).map(|a| a.into())
}
```

💡 **Key Insight**

For **path coverage**, generate one test per BDD path. For **branch coverage**, generate tests that flip each branch. BDDs make this systematic: enumerate paths, sample representative inputs.

18.6 Concolic Testing

Concolic (concrete + symbolic) execution combines:

- **Concrete execution**: Run on actual inputs
- **Symbolic tracking**: Build path condition BDD

⚙️ **Algorithm: Concolic Testing Loop**

```
ConcolisTest(program):
  worklist = {random_input}
  covered = ∅ (empty BDD)

  while worklist not empty:
    input = worklist.pop()
    (path_cond, _) = execute_symbolically(program, input)

    // Mark this path as covered
    covered = covered OR path_cond

    // Generate inputs for alternative branches
    for branch in path_cond.branches():
      // Negate this branch, keep prefix
      alt_cond = prefix(branch) AND NOT(branch)
      if alt_cond SAT:
        new_input = solve(alt_cond)
        worklist.add(new_input)
```

Concolic Testing: Negate and Explore

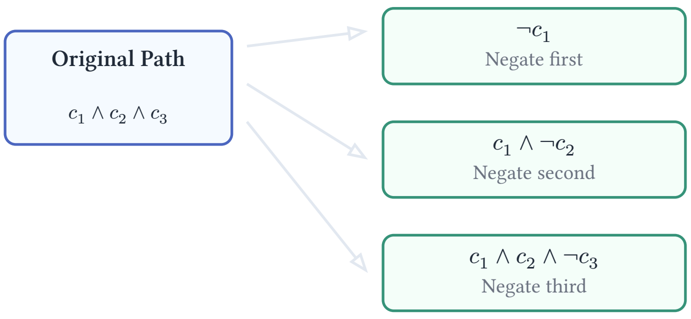


Figure 47: Concolic testing systematically explores alternative paths by negating branch conditions.

18.7 Security Analysis

BDDs excel at security-relevant program analysis:

✏️ **Example — Taint Analysis with BDDs**

Track information flow from untrusted sources:

- Variable tainted_x: "variable x holds untrusted data"
- Propagation: $\text{tainted}_y \leftarrow \text{tainted}_y \vee \text{tainted}_x$ (on `y = f(x)`)
- Violation: $\text{tainted}_z \wedge \text{sink}_z$ (tainted data reaches sensitive sink)

The BDD for the violation condition represents all inputs causing a security bug.

i Applications in Security

- **SQL injection**: Taint user input, check query strings
- **Buffer overflow**: Track array bounds symbolically
- **Information leak**: Path condition reveals secret bits?
- **Access control**: Encode policy, verify enforcement

18.8 Practical Considerations

Challenge	**Mitigation**	**BDD blowup on complex conditions**
Abstract predicates, simplify constraints	Floating-point arithmetic	Interval abstraction, not BDDs
Pointer aliasing	Points-to analysis pre-pass	Loops
Widening, bounded unrolling	Scalability	Modular analysis, summaries

BDD-based symbolic execution shines when:

- Path conditions are Boolean or small integer comparisons
- You need **complete** coverage analysis (all paths)
- The same analysis runs on multiple inputs/programs

Chapter 19

Configuration and Feature Models

Modern software rarely ships as a single product. A car configurator might offer 50 optional features, from heated seats to autonomous parking. With 2^{50} combinations — more than a quadrillion — how do you ensure every valid configuration actually builds, boots, and behaves correctly?

BDDs are the hidden engine behind **software product line analysis**, compactly representing astronomical configuration spaces and enabling instant queries about variability.

19.1 Software Product Lines

A **software product line** (SPL) is a family of related products sharing a common architecture but varying in **features**.

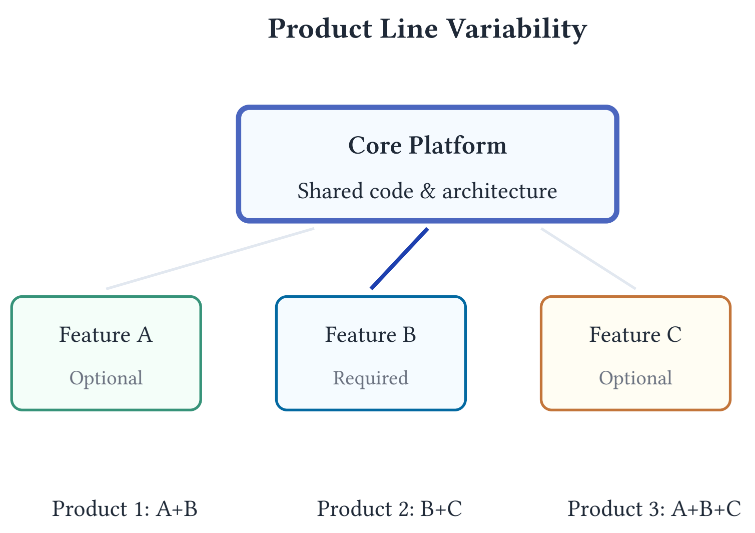


Figure 48: A product line derives multiple products from shared assets plus optional features.

💡 **Key Insight**

With n optional features, there are potentially 2^n configurations. Not all are valid — dependencies and conflicts constrain the space. BDDs compactly represent **exactly** the valid configurations.

19.2 Feature Models

A **feature model** captures variability as a tree with constraints:

✏️ **Example — Feature Model Elements**

1. **Mandatory**: Child must be selected if parent is $(A \to B)$
2. **Optional**: Child may be selected if parent is $(A \to (B \lor \neg B))$
3. **Alternative (XOR)**: Exactly one child $(A \to (B \oplus C))$
4. **Or-group**: At least one child $(A \to (B \lor C))$
5. **Cross-tree constraints**: $B \to C$ (selecting B requires C)

Feature Model: Mobile Phone

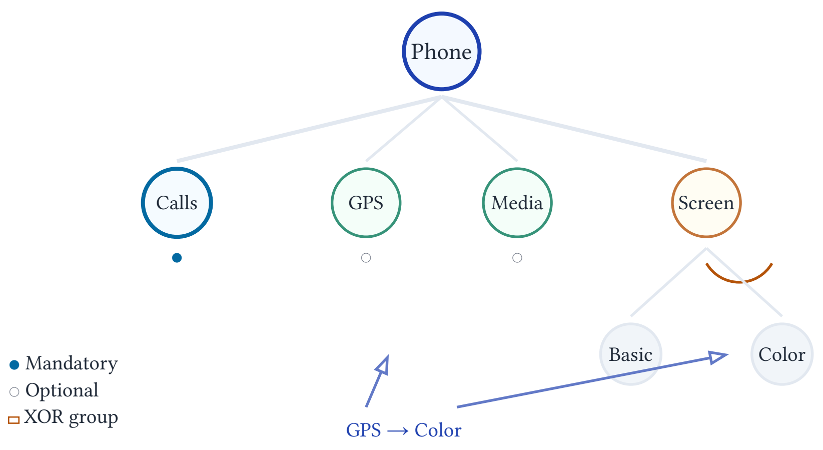


Figure 49: Feature model for a mobile phone with mandatory, optional, and alternative features.

19.2.1 BDD Encoding

```
fn encode_feature_model(bdd: &Bdd) → Ref {
    // Feature variables
    let phone = bdd.variable(1);
    let calls = bdd.variable(2);
    let gps = bdd.variable(3);
    let media = bdd.variable(4);
    let screen = bdd.variable(5);
    let basic = bdd.variable(6);
    let color = bdd.variable(7);

    // Mandatory: Phone → Calls
    let c1 = bdd.implies(phone, calls);

    // Mandatory: Phone → Screen
    let c2 = bdd.implies(phone, screen);

    // XOR: Screen → (Basic ⊕ Color)
    let xor_screen = bdd.xor(basic, color);
    let c3 = bdd.implies(screen, xor_screen);

    // Cross-tree: GPS → Color
    let c4 = bdd.implies(gps, color);

    // Root must be selected
    let c5 = phone;

    // Conjoin all constraints
    bdd.and_many(&[c1, c2, c3, c4, c5])
}
```

19.3 Configuration Analysis Queries

BDDs answer configuration questions efficiently:

Query	BDD Operation	Complexity
Is config valid?	Evaluate BDD path	$O(n)$
Any valid config?	BDD ≠ 0?	$O(1)$
Count valid configs	SatCount	$O(\|f\|)$
List all configs	AllSat enumeration	$O(\text{output})$
Feature always selected?	$f \land \neg x = 0$?	$O(\|f\|)$
Features compatible?	$f \land x \land y \neq 0$?	$O(\|f\|)$

✏️ **Example — Common Analysis Queries**

```
// Is this configuration valid?
let config = phone & calls & gps & color & !basic;
let valid = !bdd.is_zero(bdd.and(feature_model, config));

// How many valid configurations?
let count = bdd.sat_count(feature_model, num_features);

// Is GPS always selected in valid configs?
let without_gps = bdd.and(feature_model, bdd.not(gps));
let gps_mandatory = bdd.is_zero(without_gps);

// Are GPS and Media compatible?
let both = bdd.and(feature_model, bdd.and(gps, media));
let compatible = !bdd.is_zero(both);
```

19.4 Interactive Configuration

When users configure products interactively, BDDs enable smart assistance:

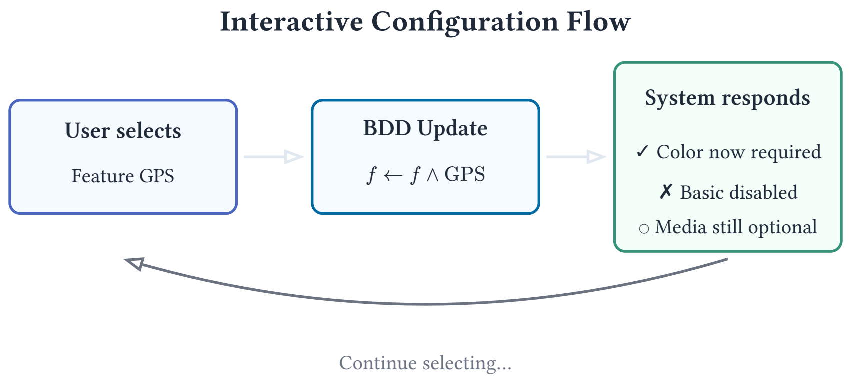


Figure 50: Interactive configuration with BDD-backed constraint propagation.

⚙️ Algorithm: Propagate Selection

```
PropagateSelection(bdd, current_config, selected_feature):
  // Add selection to configuration
  new_config = current_config AND selected_feature

  // Check validity
  if new_config == 0:
    return Error("Selection violates constraints")

  // Find implied selections (dead features = features that must be false)
  for each unselected feature f:
    if (new_config AND f) == 0:
      f.status = DISABLED  // Can't be selected
    else if (new_config AND NOT f) == 0:
      f.status = REQUIRED  // Must be selected
    else:
      f.status = OPTIONAL  // User can choose

  return new_config
```

💡 Key Insight

For large feature models (thousands of features), caching partial BDDs and using incremental operations keeps interactive response times under 100ms.

19.5 Optimization over Configurations

Often we want not just **any** valid configuration, but the **best** one.

✏️ Example — Configuration Optimization

Each feature has attributes:

- **Cost**: $\mathrm{cost}(\mathrm{GPS}) = 50$, $\mathrm{cost}(\mathrm{Color}) = 30$
- **Performance**: $\mathrm{perf}(\mathrm{Color}) = 0.8$

Goals:

- Minimize cost while meeting requirements
- Find Pareto-optimal configurations (cost vs. features)

Configuration Search Space

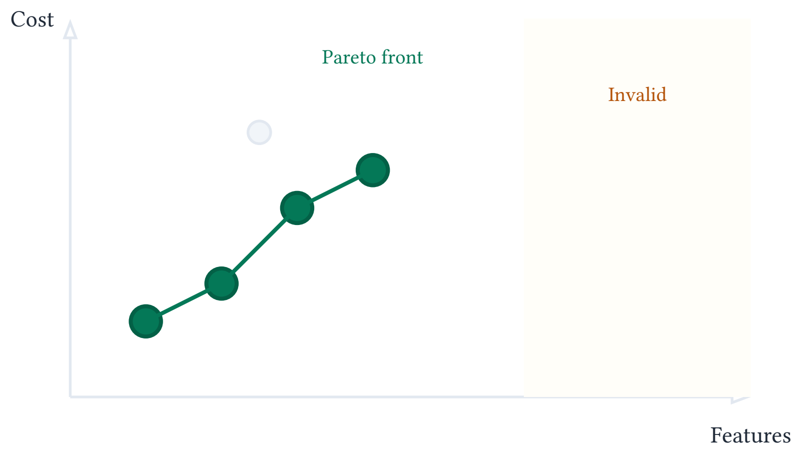


Figure 51: Pareto-optimal configurations balance cost against features.

19.6 Industrial Applications

19.6.1 Linux Kernel Configuration

The Linux kernel has over 15,000 configuration options with complex dependencies.

i Linux Kernel Kconfig

- **Options**: 15,000+ Boolean and choice features
- **Constraints**: Thousands of dependencies and conflicts
- **BDD use**: Configuration tools like `make menuconfig` use constraint solvers
- **Challenge**: Full BDD can be huge; partitioned/approximate methods used

19.6.2 Automotive Configuration

Modern cars have thousands of electronic control units (ECUs) with variant configurations:

Domain	Features	BDD Benefit
Automotive ECUs	Engine, safety, comfort options	Validate combinations pre-production
Cloud services	VM sizes, regions, features	Pricing, compatibility checks
Product configurators	E-commerce customization	Real-time validity feedback
Build systems	Compiler flags, dependencies	Detect incompatible flag combinations

⚠️ **Scalability Limits**

Very large feature models (10,000+ features) can cause BDD blowup. Industrial tools use:

- **Partitioning**: Split model into independent sub-models
- **Approximation**: Over-approximate the valid space
- **Hybrid**: BDD + SAT solver combination

19.7 The Power of Feature Model BDDs

Once you have a BDD for a feature model:

💡 **Key Insight**

- **Validation**: Instantly check any configuration
- **Counting**: Know exactly how many products are possible
- **Sampling**: Generate random valid configurations for testing
- **Analysis**: Find dead features, redundant constraints
- **Optimization**: Search only valid configurations

The BDD is a **complete, compact encoding** of your product space.

PART V

Ecosystem and Comparison

Chapter 20

Library Comparison

The BDD library ecosystem spans three decades of research and engineering. From CUDD's 1996 release — still the gold standard — to modern parallel implementations, each library embodies different design philosophies.

Understanding this landscape helps you choose the right tool for your problem and appreciate where `bdd-rs` fits in the lineage.

20.1 Landscape of BDD Libraries

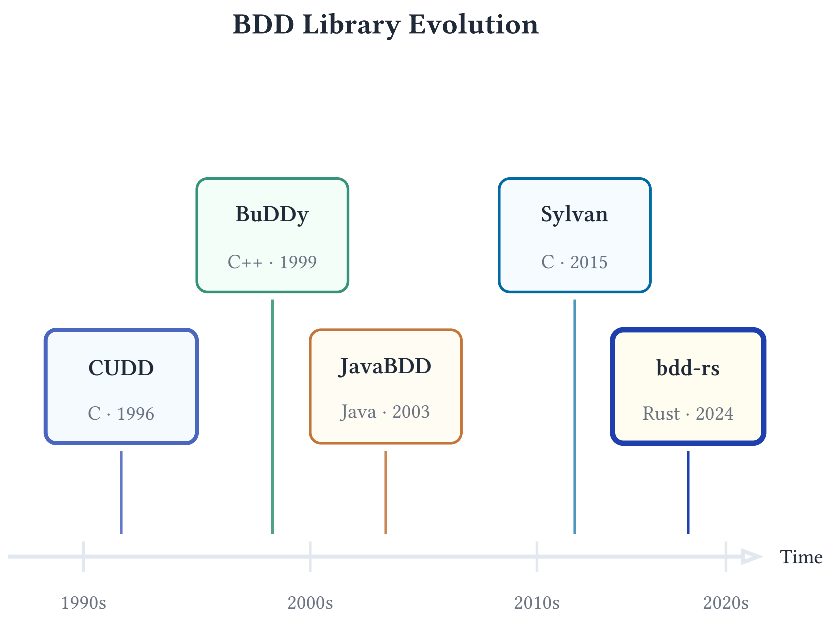


Figure 52: Major BDD libraries across three decades of development.

20.2 CUDD (Colorado University Decision Diagram)

CUDD is the **reference implementation** — the library against which all others are measured. Developed at the University of Colorado, Boulder, it has powered countless research projects and industrial tools.

i CUDD at a Glance

- **Language**: C (with C++ wrapper)
- **Features**: BDDs, ZDDs, ADDs (Algebraic Decision Diagrams)
- **Reordering**: Sifting, window permutation, simulated annealing
- **Memory**: Reference counting with periodic garbage collection
- **Status**: Mature, stable, widely used

20.2.1 Strengths

1. **Comprehensive**: Supports BDDs, ZDDs, and ADDs in one library
2. **Battle-tested**: Decades of use in research and industry
3. **Excellent reordering**: Dynamic variable reordering is well-tuned
4. **Rich API**: Every conceivable operation is available

20.2.2 Limitations

1. **C-style API**: Manual memory management, no type safety
2. **Single-threaded**: No built-in parallelism
3. **Complex setup**: Configuration can be tricky
4. **Documentation**: Comprehensive but dense

```
// CUDD usage example
DdManager *manager = Cudd_Init(0, 0, CUDD_UNIQUE_SLOTS, CUDD_CACHE_SLOTS, 0);
DdNode *x = Cudd_bddIthVar(manager, 0);
DdNode *y = Cudd_bddIthVar(manager, 1);
DdNode *f = Cudd_bddAnd(manager, x, y);
Cudd_Ref(f);  // Must manually reference!
```

20.3 BuDDy

BuDDy is the workhorse of many academic tools — simpler than CUDD but still powerful.

✏️ **Example — BuDDy Philosophy**

BuDDy prioritizes **simplicity** and **ease of use**. The API is cleaner than CUDD's, with better C++ integration. It's the library of choice when you want something that "just works."

```
// BuDDy usage example
bdd_init(1000000, 100000);  // nodes, cache
bdd_setvarnum(10);

bdd x = bdd_ithvar(0);
bdd y = bdd_ithvar(1);
bdd f = x & y;  // Operator overloading!
// Automatic reference counting
```

20.4 Sylvan

Sylvan brings **parallelism** to BDD operations, exploiting multi-core processors.

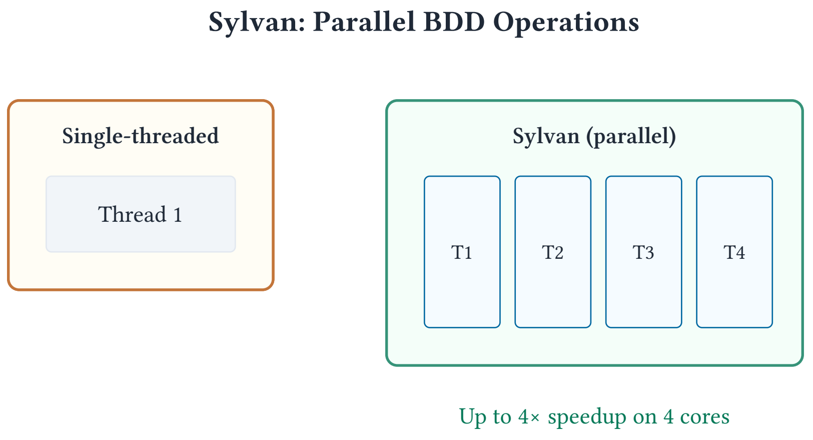


Figure 53: Sylvan distributes BDD operations across multiple threads.

💡 **Key Insight**

Sylvan uses **work-stealing** parallelism: threads that finish early steal work from busy threads. The unique table and operation cache use lock-free data structures for thread safety.

⚠️ **When Parallelism Helps**

Parallel BDD operations help most when:

- BDDs are large (millions of nodes)
- Operations are compute-bound (complex Apply)
- Multiple cores are available

For small BDDs, single-threaded libraries may actually be faster due to lower overhead.

20.5 Comprehensive Comparison

Feature	CUDD	BuDDy	Sylvan	bdd-rs
Language	C	C++	C	Rust
Reordering	✓	✓	✓	◐
Complement Edges	✓	✓	✓	✓
Parallel	✗	✗	✓	✗
ZDD Support	✓	✗	✓	✗
ADD Support	✓	✗	✗	✗

Memory Safety	✗	✗	✗	✓
Reference Counting	Manual	Auto	Lock-free	Mark-sweep

20.6 bdd-rs: Rust-Native Design

`bdd-rs` takes a different approach: **memory safety first**, enabled by Rust's ownership model.

✏️ **Example — bdd-rs Philosophy**

- **Safe by default**: No undefined behavior, no memory leaks
- **Ergonomic API**: Rust idioms, not C idioms
- **Modern design**: Complement edges, level-based ordering
- **Transparent**: Simple implementation you can understand

```
// bdd-rs usage
let bdd = Bdd::new();
let x = bdd.variable(1);
let y = bdd.variable(2);
let f = bdd.and(x, y);  // No manual reference counting!
// Drop automatically cleans up
```

20.6.1 What bdd-rs Does Well

1. **Memory safety**: Rust's type system prevents common bugs
2. **Clean API**: `Ref` handles are lightweight and copyable
3. **Explicit management**: Control when garbage collection happens
4. **Readable source**: Learn BDD internals by reading the code

20.6.2 Current Limitations

1. **No dynamic reordering** (yet): Manual ordering only
2. **Single-threaded**: No parallelism
3. **Fewer features**: No ZDDs, no ADDs
4. **Younger project**: Less battle-tested than CUDD

20.7 Choosing a Library

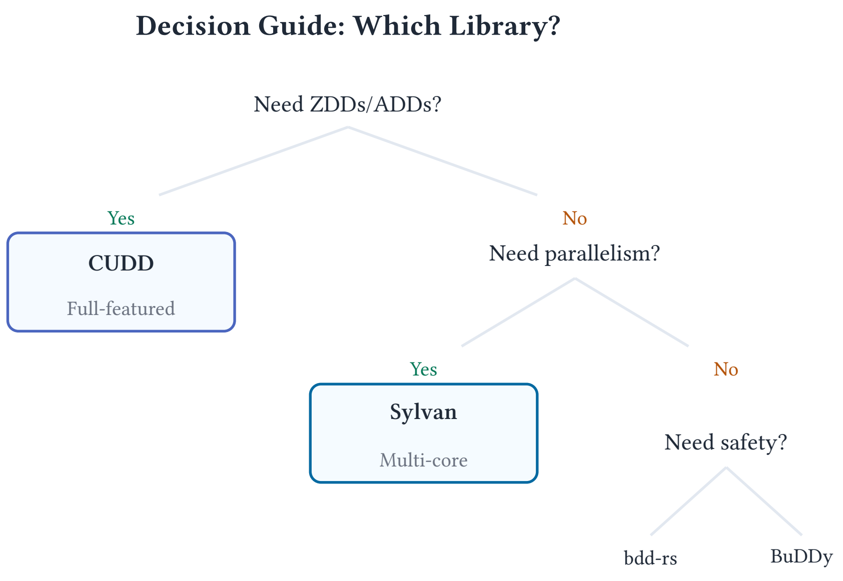


Figure 54: Choosing a BDD library based on your requirements.

i Recommendation Summary

- **Research prototyping**: `bdd-rs` (safe, readable) or BuDDy (simple)
- **Production systems**: CUDD (proven) or Sylvan (if parallel)
- **Learning BDDs**: `bdd-rs` (clean implementation to study)
- **Java projects**: JavaBDD (JNI wrapper around CUDD)

Chapter 21

Design Trade-offs

Building a BDD library is an exercise in trade-offs. Every decision — how to store nodes, when to garbage collect, whether to cache — trades off between competing concerns: speed vs. memory, simplicity vs. features, safety vs. flexibility.

This chapter dissects the key engineering choices, explaining why reasonable libraries make radically different decisions.

21.1 Memory vs. Time

The fundamental trade-off: **cache more** to avoid recomputation, or **compute more** to save memory.

Memory-Time Trade-off

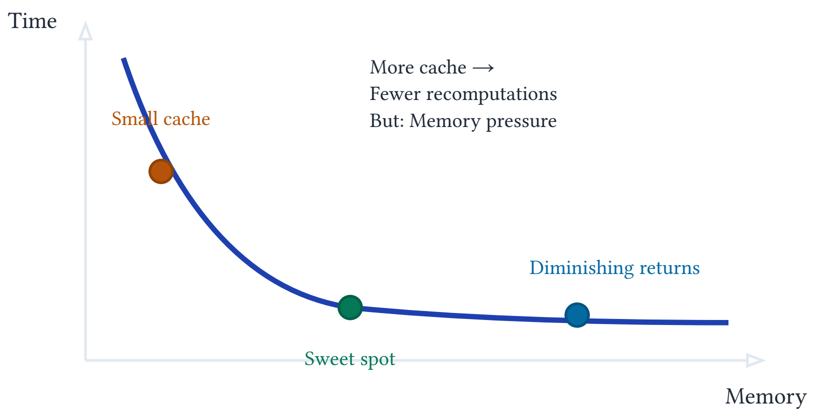


Figure 55: Increasing cache size improves performance up to a point, then provides diminishing returns.

💡 **Key Insight**

The “sweet spot” depends on:
- **Working set**: How many operations are repeated?
- **Available memory**: How much can you afford?
- **Access patterns**: Temporal locality matters

Most libraries default to generous caching — memory is cheaper than time.

21.2 Node Representation Choices

How do you store BDD nodes in memory? This low-level decision affects every operation.

Approach	Pros	Cons
Pointer-based	Direct access, familiar	64-bit overhead, GC complexity
Index-based	Compact (32-bit)	Extra indirection
Array-of-structs	Cache-friendly traversal	Fragmentation on deletion
Struct-of-arrays	SIMD potential	Scattered access patterns

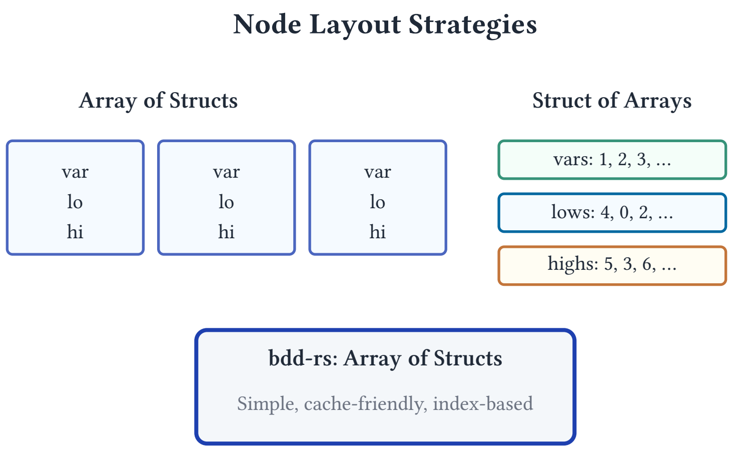


Figure 56: Different memory layouts trade locality for flexibility.

21.3 Complement Edge Trade-offs

Complement edges provide significant benefits but add complexity:

Aspect	Without Complements	With Complements
Node count	Baseline	Up to 50% fewer
Negation	$O(n)$ copy	$O(1)$ flip bit
Canonicity	Simple	Normalization rules required
Algorithm complexity	Straightforward	Extra edge-case handling
Memory per node	Baseline	Same or less total

✏️ **Example — The Complement Edge Payoff**

For symmetric functions like XOR, complement edges can reduce node count dramatically. The XOR of 10 variables:

- Without complements: 2000 nodes
- With complements: 10 nodes

The normalization complexity is worth it for the space savings.

21.4 Unique Table Design

The unique table is the heart of any BDD library — every node lookup goes through it.

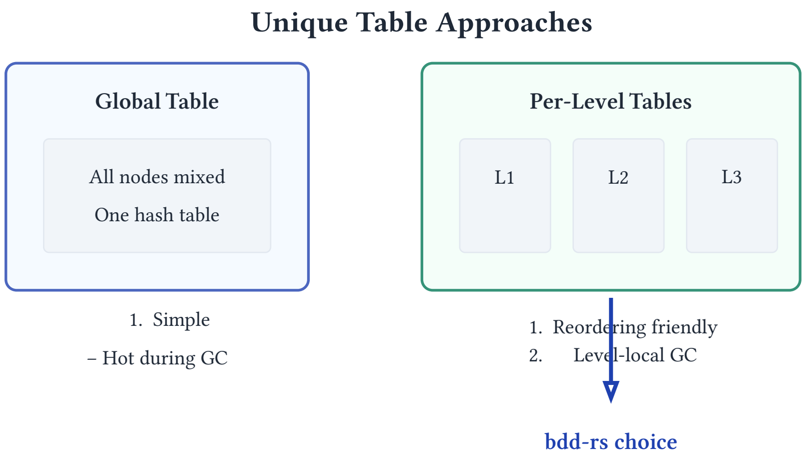


Figure 57: Per-level tables enable efficient variable reordering.

💡 **Key Insight**

Per-level (subtable) design enables:

- **Local operations**: Only touch affected levels during reordering
- **Incremental GC**: Collect one level at a time
- **Better locality**: Nodes at same level accessed together

21.5 Cache Strategies

Operation caches trade memory for avoiding recomputation:

⚙️ **Algorithm: Cache Design Decisions**

```
Key decisions:
1. Single cache vs. per-operation caches
   - Single: simpler, possible conflicts
   - Multiple: more memory, fewer conflicts

2. Cache associativity
   - Direct-mapped: simple, high conflict rate
   - Set-associative: balance
   - Fully-associative: complex, lowest conflicts

3. Eviction policy
   - Random: simple, works well in practice
   - LRU: optimal but expensive
   - FIFO: simple, reasonable
```

```
4. Persistence across GC
   - Clear cache: safe but lose warmth
   - Update cache: complex, preserves work
```

21.6 Garbage Collection Approaches

BDD libraries must reclaim memory from dead nodes:

Approach	Pros	Cons
Manual (user calls)	Predictable, explicit control	Burden on user
Reference counting	Immediate reclamation	Cycle issues, overhead per op
Mark-and-sweep	No per-op overhead	Pause time, needs root tracking
Incremental/concurrent	Low latency	Complex implementation

GC Strategy Comparison

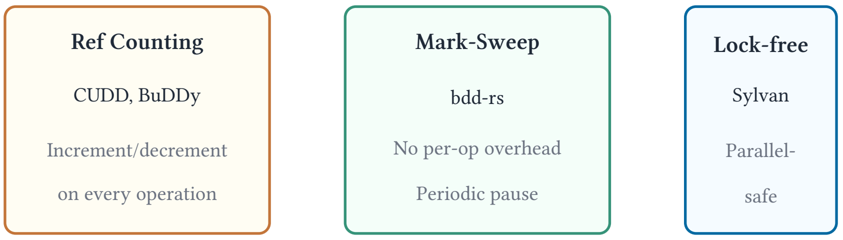


Figure 58: Different GC strategies suit different use cases.

21.7 API Design Philosophy

The public API shapes how users interact with your library:

Philosophy	Example	Trade-off
Maximize safety	Rust's `Ref` type	May limit advanced use
Maximize flexibility	CUDD's raw pointers	User can shoot themselves
Hide internals	Opaque handles	Limits optimization opportunities
Expose internals	Public node structure	Hard to change later

✏️ **Example — bdd-rs API Philosophy**

```
// Safe: Ref is Copy, no manual memory management
let f = bdd.and(x, y);  // Returns Ref, not raw pointer
```

```
// Explicit: GC is manual, user controls timing
bdd.gc();  // User decides when

// Transparent: Can inspect internals if needed
let node = bdd.get_node(f);  // Access node data
```

The goal: **safe by default**, **powerful when needed**.

21.8 The Unifying Theme

Every design decision involves trade-offs. The “right” choice depends on your priorities:

i Design Priority Matrix

- **Performance-critical applications**: Optimize for speed, accept complexity
- **Research prototypes**: Optimize for simplicity, accept slower speed
- **Production systems**: Optimize for reliability, accept some inefficiency
- **Learning/teaching**: Optimize for clarity, accept naive implementations

`bdd-rs` prioritizes **safety** and **clarity**, making it ideal for learning and correct-by-construction implementations.

Chapter 22

Future Directions

BDDs have been around for four decades, yet the field is far from stagnant. Researchers continue to push boundaries: scaling to larger problems, exploiting modern hardware, and finding unexpected applications in machine learning and quantum computing.

This chapter surveys the frontier — where BDD research is headed and what new capabilities may emerge.

22.1 Parallelism and Distribution

Modern CPUs have dozens of cores; servers have hundreds. Can BDDs exploit this parallelism, or are they inherently sequential?

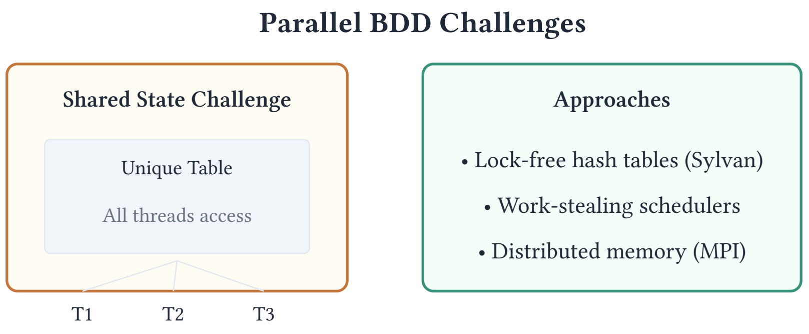


Figure 59: Parallel BDD operations must coordinate access to shared data structures.

i Parallelism Research Frontiers

- **GPU acceleration**: BDD traversal is irregular, challenging for GPUs
- **FPGA implementation**: Custom hardware for Apply operations
- **Distributed BDDs**: Partitioning across cluster nodes
- **Speculative execution**: Compute both branches, discard unused

22.2 Integration with Machine Learning

BDDs meet neural networks in surprising ways:

✏️ **Example — BDDs for Explainable AI**

A neural network classifies images, but **why** did it decide "cat"?

Approach: Compile the (simplified) decision boundary into a BDD. The BDD reveals:
- Which input features matter
- What combinations trigger each output
- Minimal explanations for decisions

Neural Network → BDD Extraction

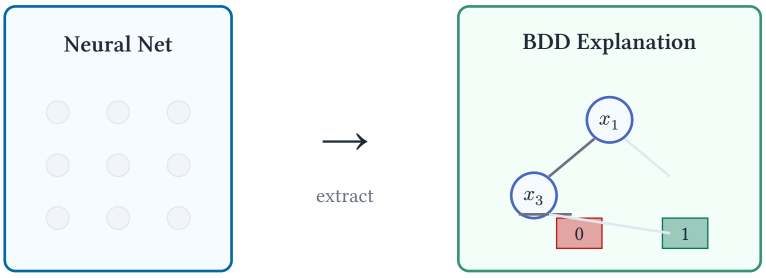


Figure 60: BDDs can explain neural network decisions by extracting symbolic decision rules.

22.3 Modern Hardware Considerations

Today's CPUs are complex — cache hierarchies, NUMA, branch prediction. BDD libraries must adapt:

Hardware Feature	BDD Impact	Optimization
L1/L2/L3 caches	Node access patterns	Cache-oblivious algorithms
NUMA (multi-socket)	Memory locality	Per-socket unique tables
Branch prediction	If-then-else traversal	Branchless operations
Prefetching	BDD traversal	Explicit prefetch hints
Persistent memory	Large BDDs	Memory-mapped structures

💡 **Key Insight**

A cache-aware BDD library can be 2-10× faster than a naive implementation on the same algorithm, purely from better memory access patterns.

22.4 BDDs in Quantum Computing

Quantum computing introduces new uses for decision diagrams:

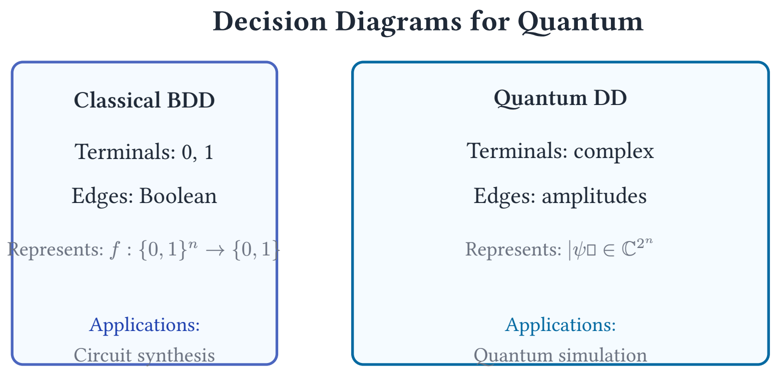


Figure 61: Quantum decision diagrams extend BDDs to represent quantum states.

✏️ Example — Quantum Circuit Synthesis

Given a quantum operation as a unitary matrix, find a circuit implementing it.

BDD-based approaches:
1. Encode the transformation symbolically
2. Search for gate sequences using BDD operations
3. Verify equivalence with quantum decision diagrams

This connects classical BDD techniques to quantum compilation.

22.5 Incremental and Online Algorithms

What if the BDD changes continuously?

⚙️ Algorithm: Incremental BDD Update

```
Traditional:
  constraints = [c1, c2, c3, ..., cn]
  bdd = AND(c1, c2, ..., cn)  // Build from scratch

Incremental:
  bdd = c1
  bdd = AND(bdd, c2)  // Add constraint
  bdd = AND(bdd, c3)
  ...
  // Later: remove c2
  // Challenge: Can we "subtract" c2 efficiently?
```

⚠️ The Subtraction Problem

BDD conjunction is irreversible — you can't efficiently "remove" a constraint. Workarounds:
- Keep constraint BDDs separate, reconstruct on removal
- Use "soft" constraints with indicator variables
- Maintain incremental history for rollback

22.6 Domain-Specific Extensions

BDDs have spawned many specialized variants:

Variant	Extension	Application
ZBDD	Zero-suppressed rules	Combinatorics, SAT
ADD	Arbitrary terminal values	Probabilistic inference
EVBDD	Edge-valued	Arithmetic, probability
MTBDD	Multi-terminal	Multi-valued logic
SDD	Sentential DD	Knowledge compilation

💡 **Key Insight**

The BDD concept generalizes to **Decision Diagrams** — any data structure that:

1. Uses a fixed variable ordering
2. Applies reduction rules for canonicity
3. Enables efficient operations via dynamic programming

22.7 The Enduring Role of BDDs

After 40 years, why do BDDs persist?

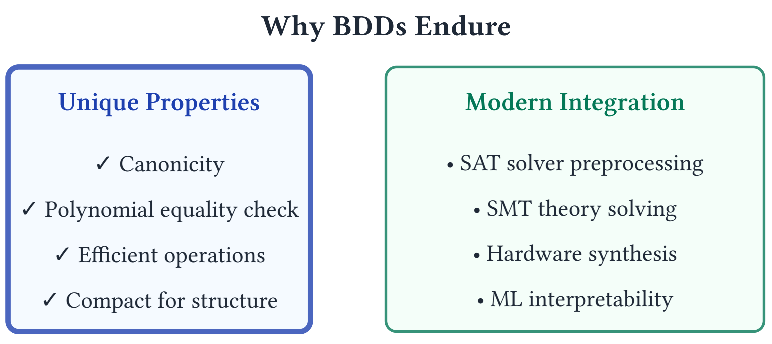


Figure 62: BDDs offer unique properties that complement modern techniques.

i The Future: Integration, Not Replacement

BDDs won't replace SAT solvers or neural networks. But they fill a unique niche:

- **SAT** finds one solution — **BDDs** represent all solutions
- **Neural nets** approximate — **BDDs** are exact
- **SMT** reasons about theories — **BDDs** provide Boolean backbone

The future is **hybrid** systems combining the strengths of each approach.

💡 **Key Insight**

For the practitioner: BDDs are a power tool. Like a good compiler or database, they solve a class of problems so well that reinventing them is rarely worthwhile.

For the researcher: BDDs are a living field. Parallelism, quantum, ML integration — there's still much to discover.

For the student: BDDs teach fundamentals. Canonical forms, dynamic programming, memory management — skills that transfer everywhere.

Chapter A

API Reference

Quick reference for the `bdd-rs` API. For complete documentation, see the rustdoc.

A.1 Manager Creation

```rust
// Create with default settings
let bdd = Bdd::default();

// Create with specific capacity (2^bits nodes)
let bdd = Bdd::new(20);  // Up to ~1M nodes
```

A.2 Variable Creation

```rust
// Create a variable (1-indexed)
let x = bdd.variable(1);
let y = bdd.variable(2);

// Variables are cached — calling again returns same Ref
assert_eq!(bdd.variable(1), x);
```

⚠️ **Variable Indexing**

Variables are **1-indexed**. Variable 0 is reserved for internal use. Use `Var(1)`, `Var(2)`, etc.

A.3 Boolean Operations

Operation	Method	Notes
NOT	`bdd.not(x)` or `-x`	$O(1)$ with complement edges
AND	`bdd.and(x, y)`	
OR	`bdd.or(x, y)`	
XOR	`bdd.xor(x, y)`	
NAND	`bdd.nand(x, y)`	

NOR	`bdd.nor(x, y)`	
Implies	`bdd.implies(x, y)`	$x \to y$
IFF	`bdd.equiv(x, y)`	$x \leftrightarrow y$
ITE	`bdd.ite(c, t, e)`	If c then t else e

```
// Examples
let f = bdd.and(x, y);           // x ∧ y
let g = bdd.or(x, bdd.not(y));   // x ∨ ¬y
let h = bdd.ite(c, f, g);        // if c then f else g
```

A.4 Queries

```
// Terminal checks
bdd.is_zero(f)     // Is f ≡ 0?
bdd.is_one(f)      // Is f ≡ 1?
bdd.is_const(f)    // Is f terminal?

// Satisfiability
bdd.is_sat(f)      // f ≢ 0
bdd.is_tautology(f) // f ≡ 1

// Size metrics
bdd.size(f)                    // Nodes in subgraph
bdd.sat_count(f, num_vars)     // Satisfying assignments
bdd.node_count()               // Total nodes in manager
```

A.5 Cofactors and Restriction

```
// Low/high cofactors
let f_low = bdd.low(f);   // f|_{top_var=0}
let f_high = bdd.high(f); // f|_{top_var=1}

// Restriction to specific variable
let f_x0 = bdd.restrict(f, Var(1), false);  // f|_{x₁=0}
let f_x1 = bdd.restrict(f, Var(1), true);   // f|_{x₁=1}
```

A.6 Quantification

```
// Existential: ∃x. f = f|_{x=0} ∨ f|_{x=1}
let exists_x = bdd.exists(f, Var(1));

// Universal: ∀x. f = f|_{x=0} ∧ f|_{x=1}
let forall_x = bdd.forall(f, Var(1));

// Over multiple variables
let cube = bdd.cube(&[Var(1), Var(2)]);
let exists_xy = bdd.exists_cube(f, cube);
```

A.7 Composition

```
// Substitute g for variable x in f
let result = bdd.compose(f, Var(1), g);  // f[x₁ := g]
```

A.8 Visualization

```
// Generate DOT format
let dot = bdd.to_dot(f);

// With custom options
let dot = bdd.to_dot_opts(f, DotOpts {
    show_complement: true,
    ..Default::default()
});

// Write to file, then render
std::fs::write("bdd.dot", dot)?;
// $ dot -Tpng bdd.dot -o bdd.png
```

A.9 Garbage Collection

```
// Mark roots and collect
bdd.gc(&[f, g, h]);

// Statistics
let stats = bdd.stats();
println!("Nodes: {}, Peak: {}", stats.nodes, stats.peak_nodes);
```

i GC Best Practices

- Call `gc()` periodically during long computations
- Always pass **all** BDDs you need to keep as roots
- Unreachable nodes are freed, invalid `Ref` s will panic

Chapter B

Complexity Analysis

Time and space complexity of BDD operations. These are worst-case bounds; actual performance is often much better due to sharing and caching.

B.1 Operation Complexity

Operation	Time	Space	Notes
Negation	$O(1)$	$O(1)$	Complement edge flip
AND, OR, XOR	$O(\|f\| \cdot \|g\|)$	$O(\|f\| \cdot \|g\|)$	Apply algorithm
ITE	$O(\|f\| \cdot \|g\| \cdot \|h\|)$	$O(\|f\| \cdot \|g\| \cdot \|h\|)$	Three operands
Restrict	$O(\|f\|)$	$O(\|f\|)$	Single variable
Compose	$O(\|f\|^2 \cdot \|g\|)$	$O(\|f\|^2 \cdot \|g\|)$	$f[x := g]$
$\exists x.f$	$O(\|f\|^2)$	$O(\|f\|^2)$	Single variable
$\forall x.f$	$O(\|f\|^2)$	$O(\|f\|^2)$	Same as $\exists$
Relational Product	$O(\|f\| \cdot \|g\| \cdot 2^k)$	$O(\|f\| \cdot \|g\| \cdot 2^k)$	k = vars quantified
Equivalence Check	$O(1)$	$O(1)$	Pointer comparison
SAT Check	$O(1)$	$O(1)$	Is BDD $\neq 0$?
SAT Count	$O(\|f\|)$	$O(\|f\|)$	Dynamic programming
SAT Witness	$O(n)$	$O(n)$	n = variables
Level Swap	$O(w^2)$	$O(1)$	w = level width

Table 1: Complexity of BDD operations. $|f|$ denotes the number of nodes in BDD f.

B.2 Function Size Bounds

Some functions have inherently small or large BDDs:

Function Class	BDD Size	Example
Constants	$O(1)$	0, 1
Single variable	$O(1)$	x_i
AND/OR of n vars	$O(n)$	$x_1 \wedge x_2 \wedge \ldots \wedge x_n$
XOR of n vars	$O(n)$	$x_1 \oplus x_2 \oplus \ldots \oplus x_n$
Symmetric	$O(n^2)$	Majority, threshold
Comparator	$O(n)$	$x < y$ (interleaved)
Addition	$O(n)$	$x + y$ (interleaved)
Multiplication	$\Omega(2^{n/3})$	$x \times y$
Hidden weighted bit	$\Omega(2^{n/2})$	Specific construction

Table 2: BDD sizes for various function classes (with optimal variable ordering).

⚠️ **Ordering Dependence**

These bounds assume **optimal** variable ordering. With a bad ordering, even simple functions like addition can become exponential. For example, $x < y$ with all x-bits before all y-bits is exponential.

B.3 Caching Effects

Without caching, Apply is exponential. With caching:

📘 **Definition (Memoization Guarantee)**

Each unique subproblem (f, g, op) is computed **at most once**. Total work is bounded by the number of distinct subproblems, which is $O(|f| \cdot |g|)$.

💡 **Key Insight**

The cache is what makes BDDs practical. A "small" cache (e.g., 10% of unique table size) is usually sufficient — most operations have high temporal locality.

B.4 Memory Analysis

Per-node memory in `bdd-rs`:

Component	Size
Variable index	16 bits
Low child (index + complement)	32 bits
High child (index only)	31 bits
Hash chain pointer	32 bits
Total per node	14 bytes (aligned to 16)

Table 3: Memory layout of a BDD node in `bdd-rs`.

For a BDD with N nodes:
- **Unique table**: $N \times 16$ bytes (nodes) $+ O(N)$ (hash buckets)
- **Operation cache**: Typically $0.1 N$ to $0.5 N$ entries $\times$ 16 bytes/entry
- **Total**: Roughly $20 N$ to $30 N$ bytes

Chapter C

Bibliography and Further Reading

Key references for deeper study, organized by topic.

C.1 Foundational Papers

i Essential Reading

If you read nothing else, read Bryant (1986). It's the paper that established BDDs as we know them today.

1. **R. E. Bryant** (1986). "Graph-Based Algorithms for Boolean Function Manipulation." *IEEE Transactions on Computers*, C-35(8):677–691.

   The seminal paper introducing ROBDDs with the canonicity theorem.
2. **C. Y. Lee** (1959). "Representation of Switching Circuits by Binary-Decision Programs." *Bell System Technical Journal*, 38:985–999.

   Early work on binary decision programs, predating modern BDDs.
3. **S. B. Akers** (1978). "Binary Decision Diagrams." *IEEE Transactions on Computers*, C-27(6):509–516.

   Formalization of binary decision diagrams.

C.2 Algorithms and Optimizations

1. **K. S. Brace, R. L. Rudell, R. E. Bryant** (1990). "Efficient Implementation of a BDD Package." *DAC '90*.

   Practical implementation techniques including complement edges and caching.
2. **R. Rudell** (1993). "Dynamic Variable Ordering for Ordered Binary Decision Diagrams." *ICCAD '93*.

   The sifting algorithm for dynamic reordering.
3. **S. Minato** (1993). "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems." *DAC '93*.

Introduction of ZDDs for sparse set families.

4. **R. I. Bahar et al.** (1993). “Algebraic Decision Diagrams and Their Applications.” *ICCAD ‘93*.

   ADDs for non-Boolean terminal values.

C.3 Model Checking

1. **J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang** (1992). “Symbolic Model Checking: 10^{20} States and Beyond.” *Information and Computation*, 98(2):142–170.

   Landmark paper on symbolic model checking with BDDs.
2. **K. L. McMillan** (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.

   Comprehensive book on symbolic verification.
3. **E. M. Clarke, O. Grumberg, D. Peled** (1999). *Model Checking*. MIT Press.

   Standard reference on model checking, including BDD methods.

C.4 Books and Surveys

1. **C. Meinel, T. Theobald** (1998). *Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications*. Springer.

   Thorough treatment of BDD theory and applications.
2. **D. E. Knuth** (2009). *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley.

   Extensive coverage of BDDs and ZDDs with careful analysis.
3. **H. R. Andersen** (1999). “An Introduction to Binary Decision Diagrams.” *Lecture Notes, IT University of Copenhagen*.

   Excellent pedagogical introduction, freely available online.

C.5 Software Libraries

Library	Language	URL
CUDD	C	`vlsi.colorado.edu/~fabio/CUDD`
BuDDy	C++	`github.com/jgcoded/BuDDy`
Sylvan	C	`github.com/trolando/sylvan`
JavaBDD	Java	`javabdd.sourceforge.net`
bdd-rs	Rust	(this library)

C.6 Online Resources

- **Bryant’s website**: Original papers and slides from the BDD inventor
- **CUDD documentation**: Detailed manual for the reference implementation
- **Knuth’s web pages**: Errata, additional examples, implementations