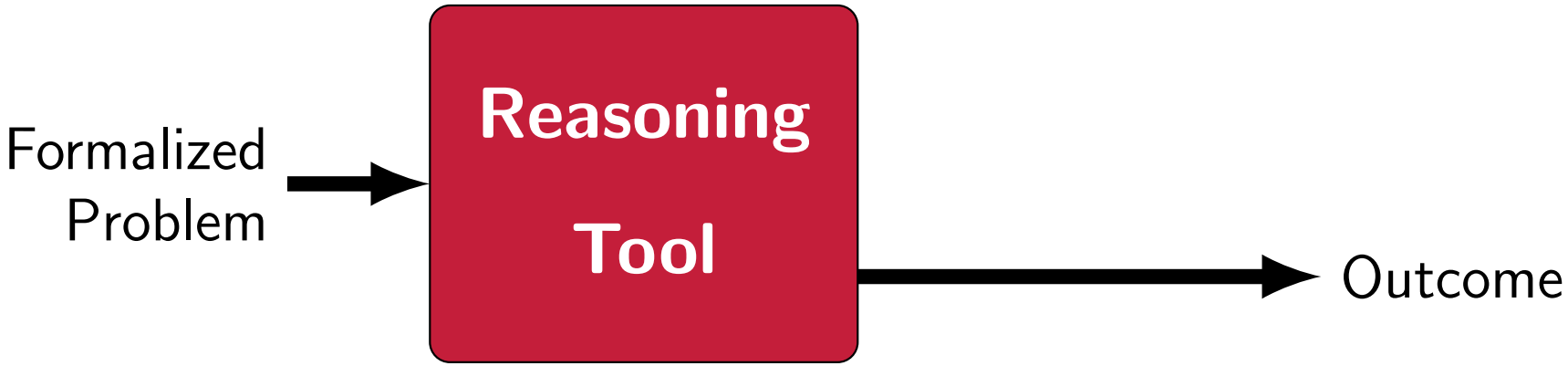
Binary Decision Diagrams Applied to Verifiable Automated Reasoning

Randal E. Bryant

Carnegie Mellon University

2022

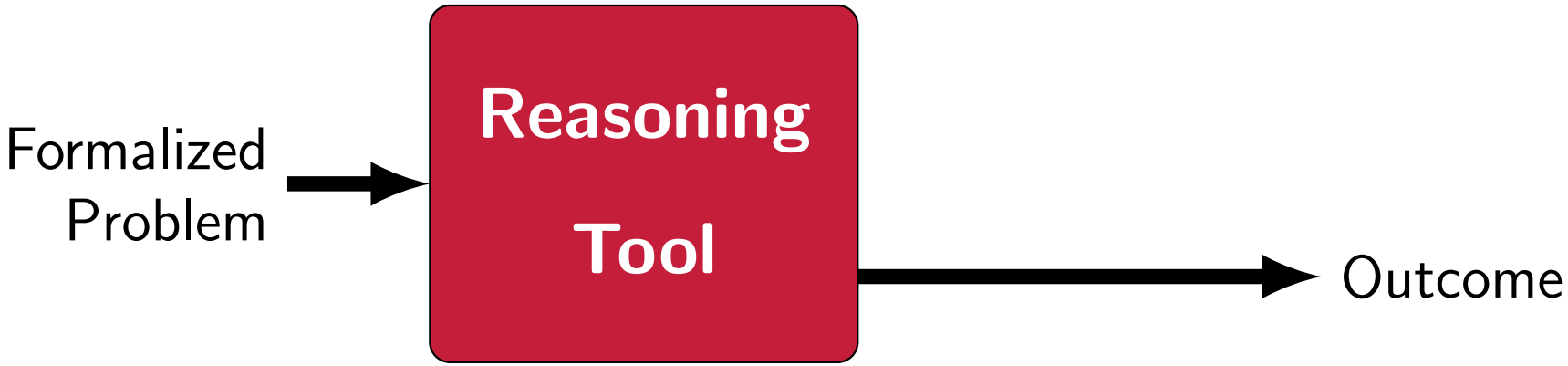
Automated Reasoning Programs



Example Applications

- Verifying hardware and software systems
- Analyzing security protocols
- Proving mathematical theorems
- Solving optimization problems

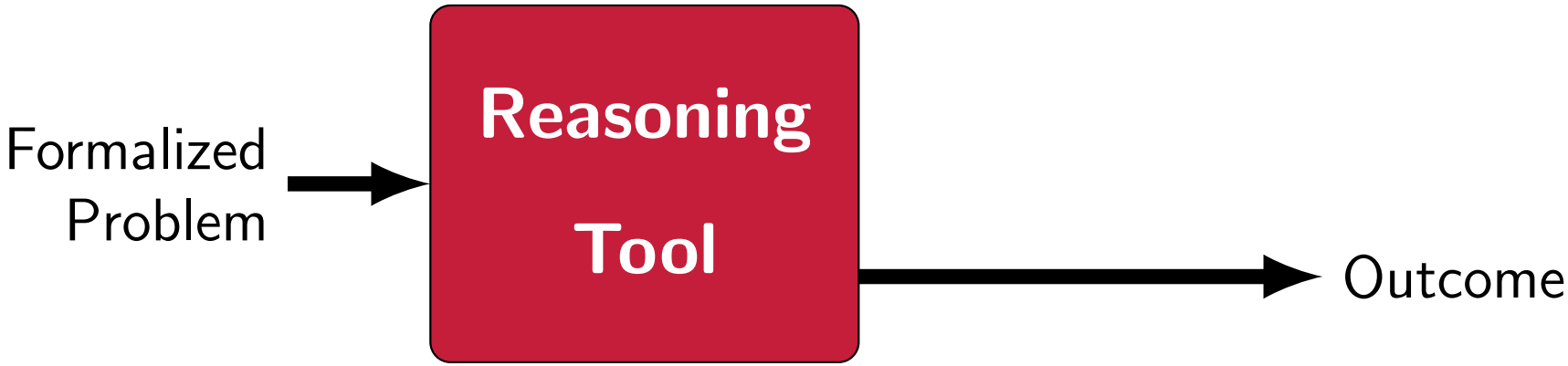
Automated Reasoning Programs



Can We Trust the Results?

- *No!*
- Complex software with many optimizations

Automated Reasoning Programs



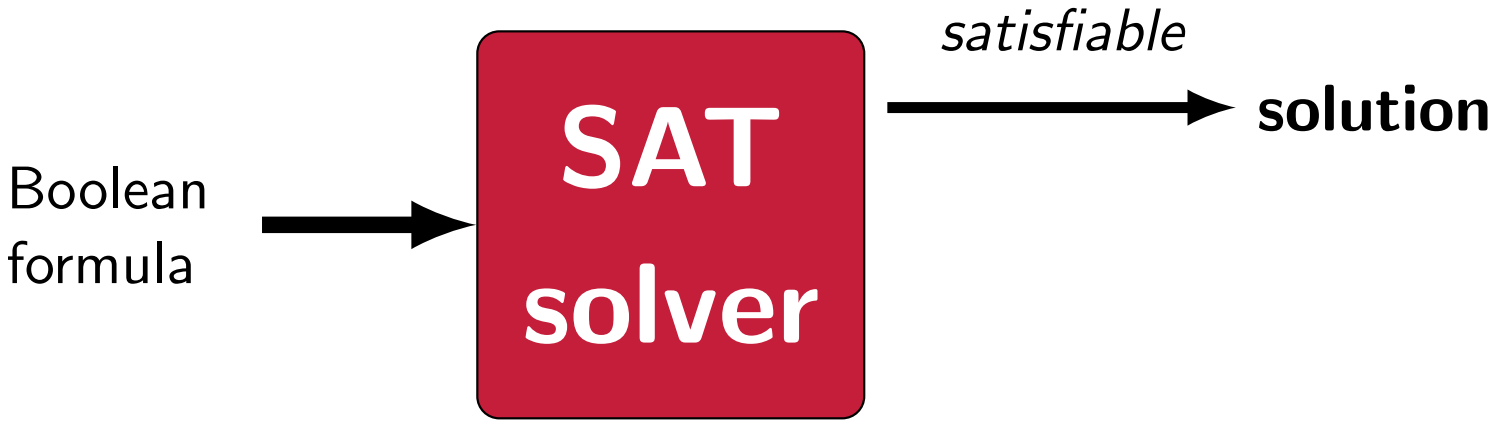
Can We Trust the Results?

- *No!*
- Complex software with many optimizations

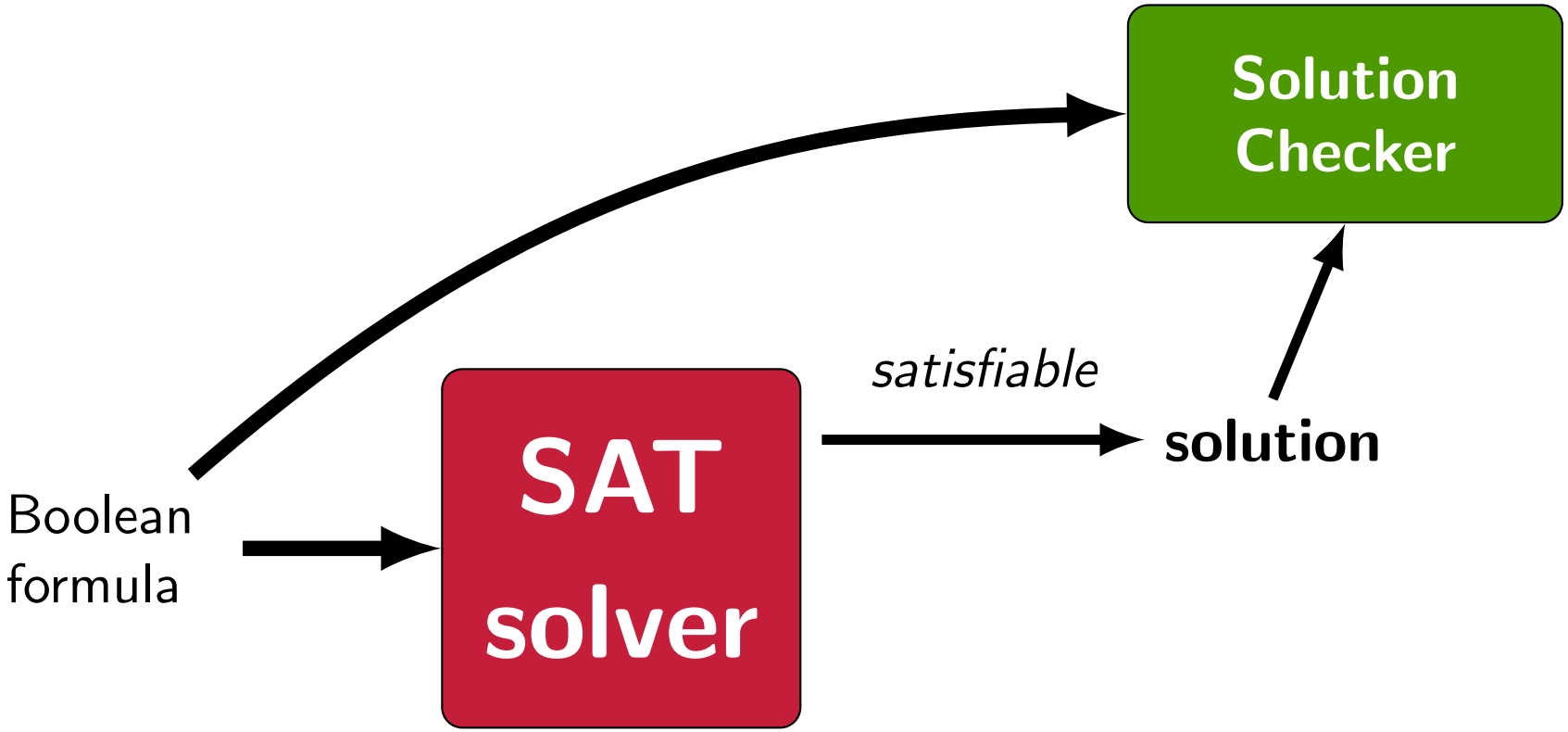
Is This a Problem?

- *Yes!*
- Automated reasoning is cornerstone of trusted system development

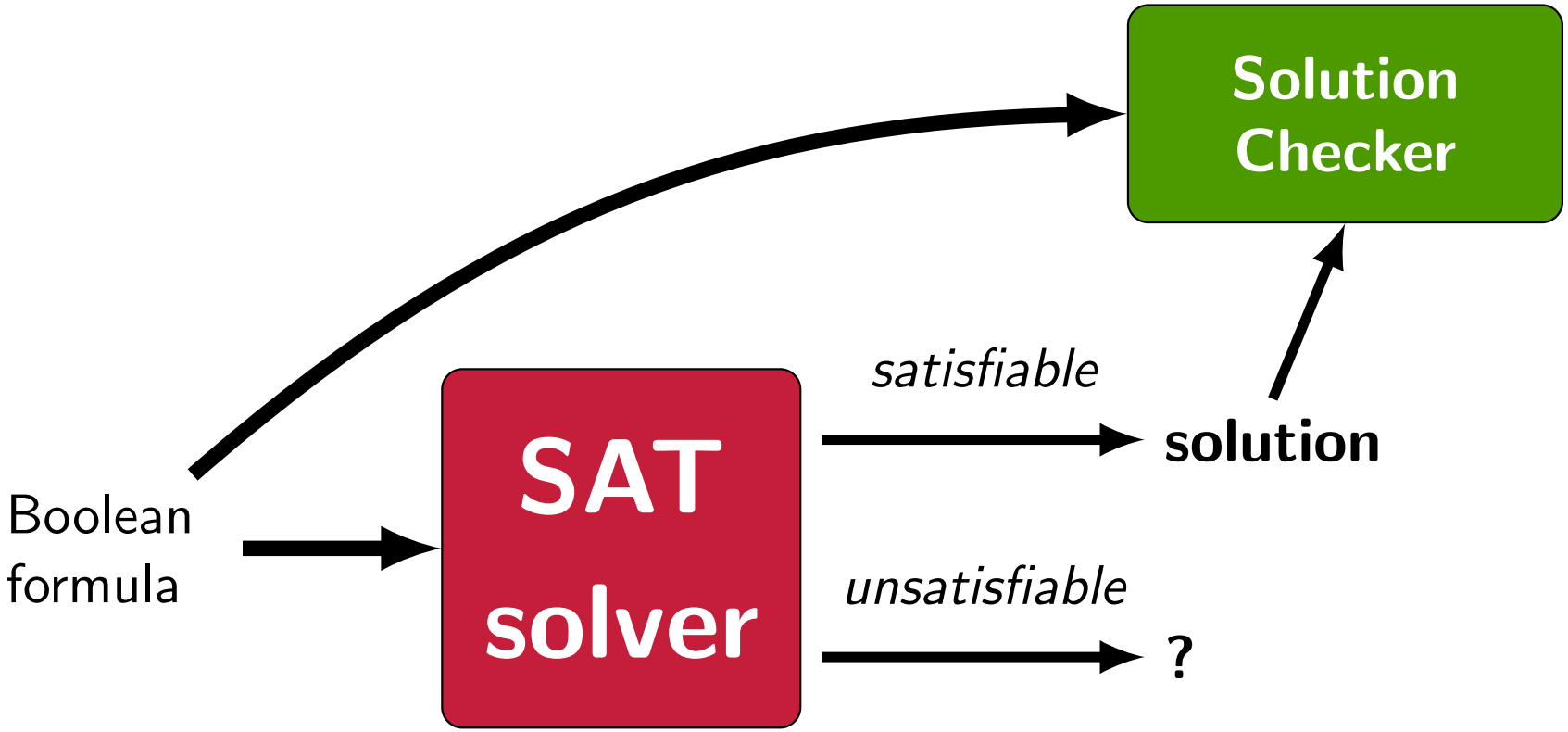
Boolean Satisfiability Solvers



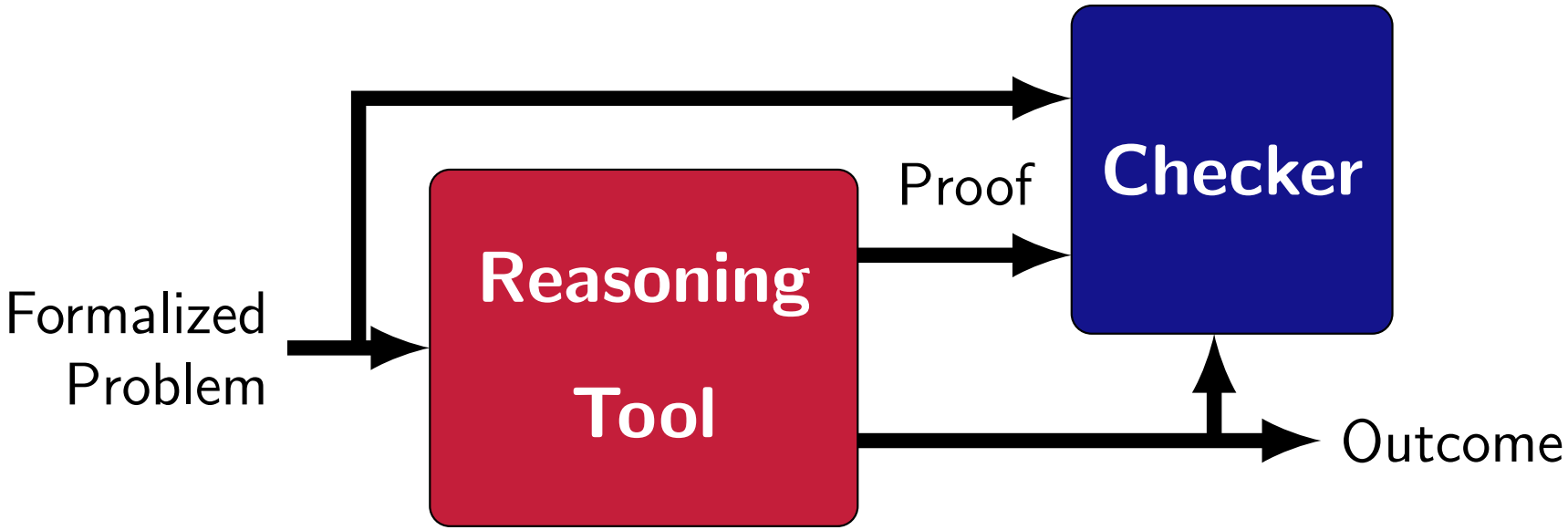
Boolean Satisfiability Solvers



Boolean Satisfiability Solvers



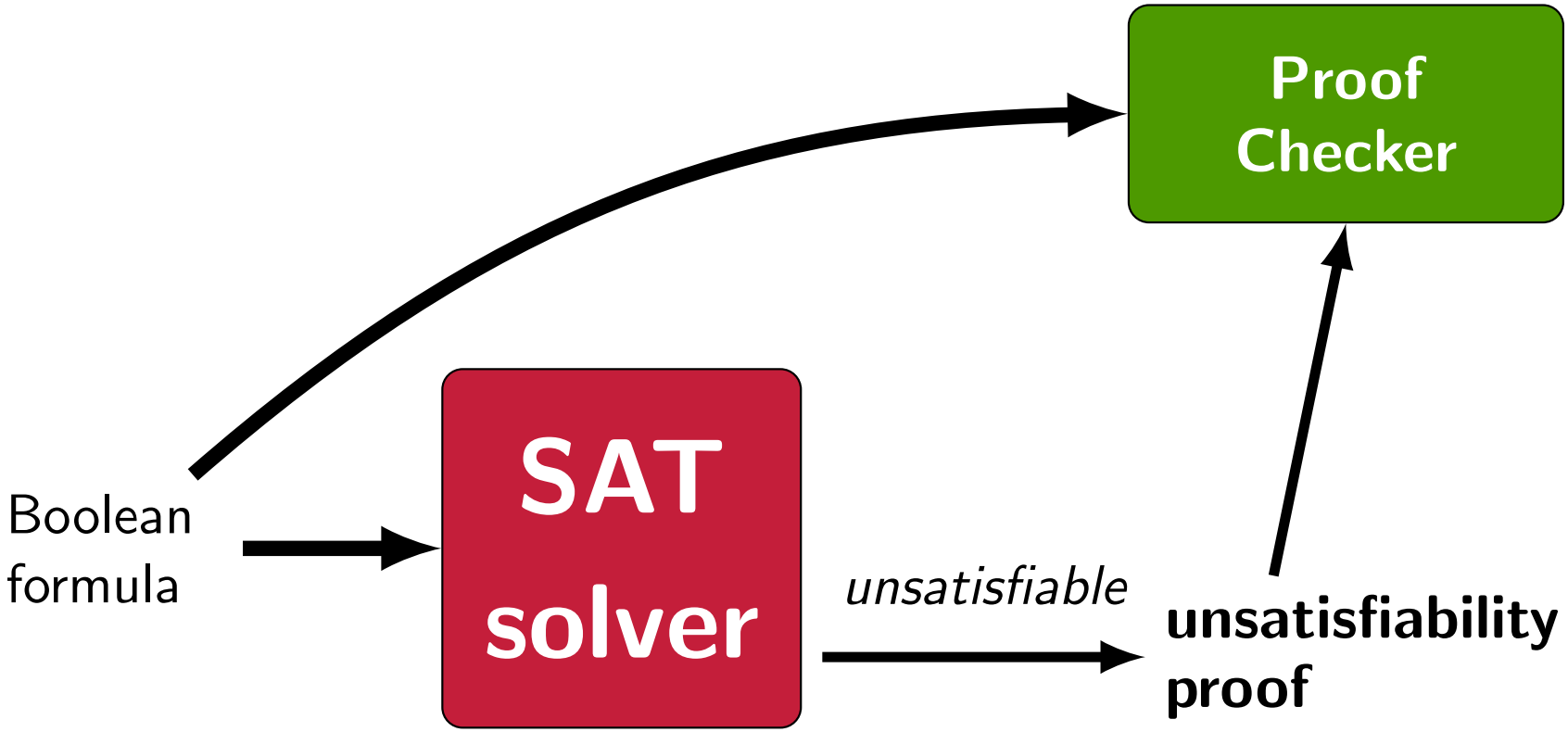
Proof-Generating Automated Reasoning Programs



Checkable Proofs

- Step-by-step proof in some logical framework
- Independently validated by proof checker
- Checker should operate in low-degree polynomial time
  - Relative to proof size
- Checker should be based on well-understood logical framework

Proof-Generating SAT Solvers



Impact

- Since 2016: Entrants to SAT competition must produce UNSAT proofs
- 2020: No entrants had errors
  - Even on new benchmarks

Motivating Example: Parity Benchmark

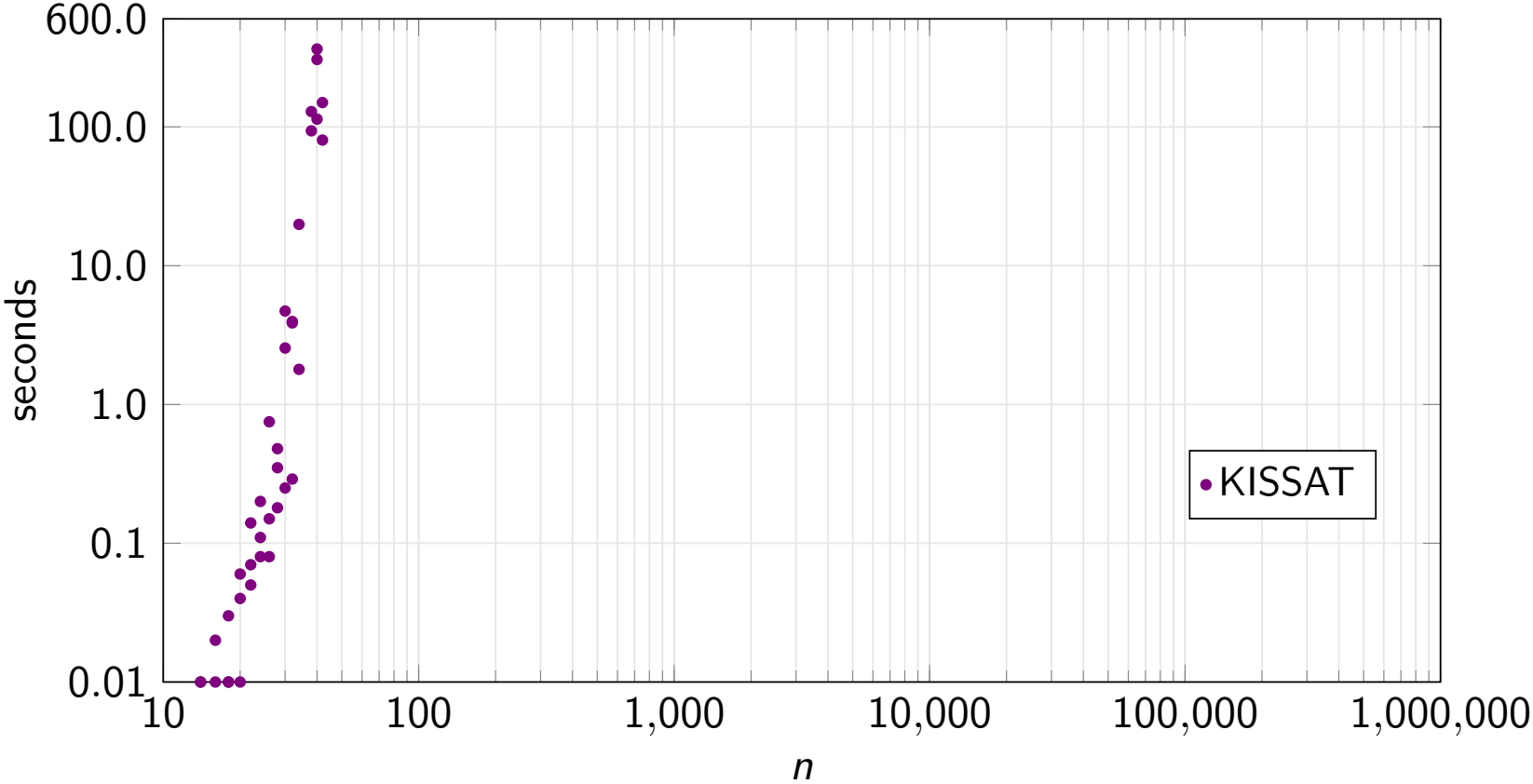
- Chew and Heule, SAT 2020

For n Boolean variables and random permutation π:

$$
\begin{array}{ccccccccccc}
x_1 & \oplus & x_2 & \oplus & \cdots & \oplus & x_n & = & 1 & \text{Odd parity} \\
x_{\pi(1)} & \oplus & x_{\pi(2)} & \oplus & \cdots & \oplus & x_{\pi(n)} & = & 0 & \text{Even parity}
\end{array}
$$

- Conjunction unsatisfiable

Parity Benchmark Runtime



- KISSAT: State-of-the-art CDCL solver
- 3 different seeds for each value of n
- Cannot get beyond $n = 42$ within 600 seconds

Reduced Ordered Binary Decision Diagrams (BDDs)

- Bryant, 1986
- Based on earlier work by Lee (1959) and Akers (1978)

Graph Representation of Boolean Functions

- Canonical Form
- Compact for many useful problems
- Simple algorithms to construct & manipulate

Used in SAT, QBF, Model Checking, ...

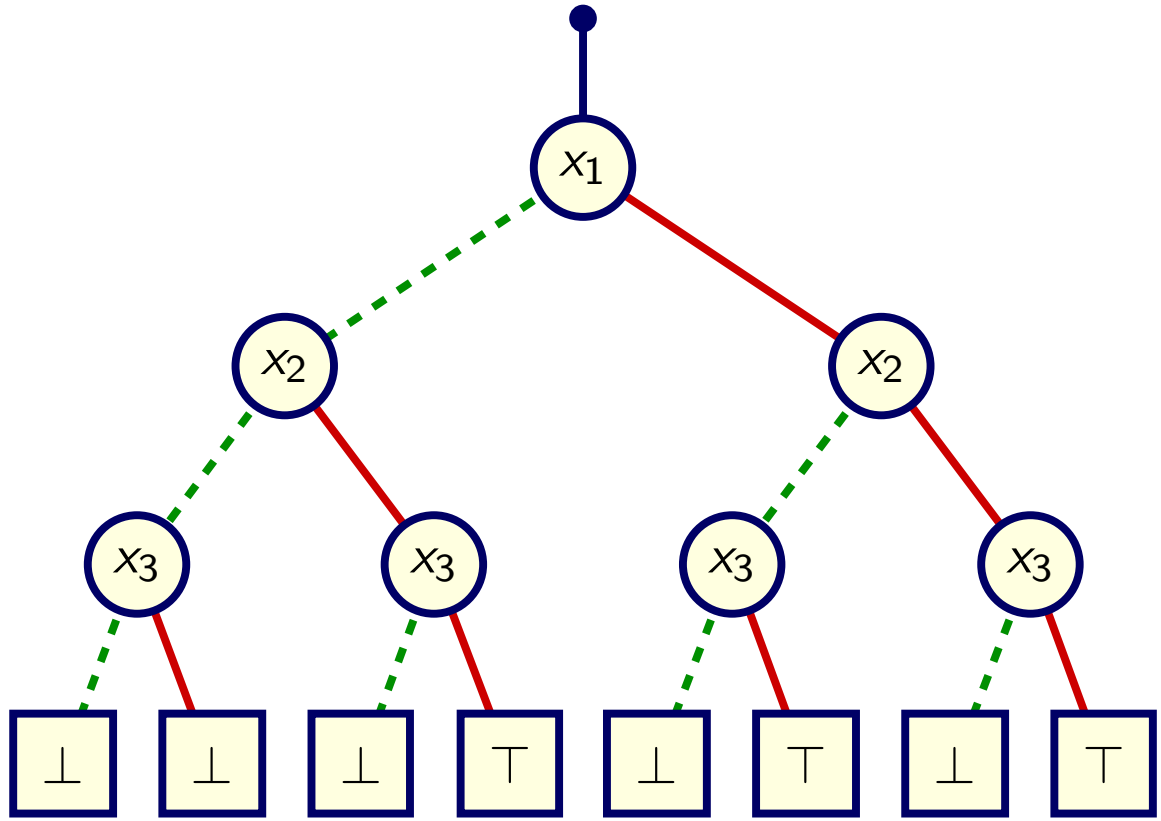
- Bottom-up approach
  - Construct canonical representation of problem
  - Generate solutions
- Compare to search-based methods
  - E.g., DPLL, CDCL
  - Top-down approaches
  - Keep branching on variables until find solution

Boolean Function Representations

Truth Table

x_1	x_2	x_3	f
0	0	0	$\bot$
0	0	1	$\bot$
0	1	0	$\bot$
0	1	1	$\top$
1	0	0	$\bot$
1	0	1	$\top$
1	1	0	$\bot$
1	1	1	$\top$

Decision Tree



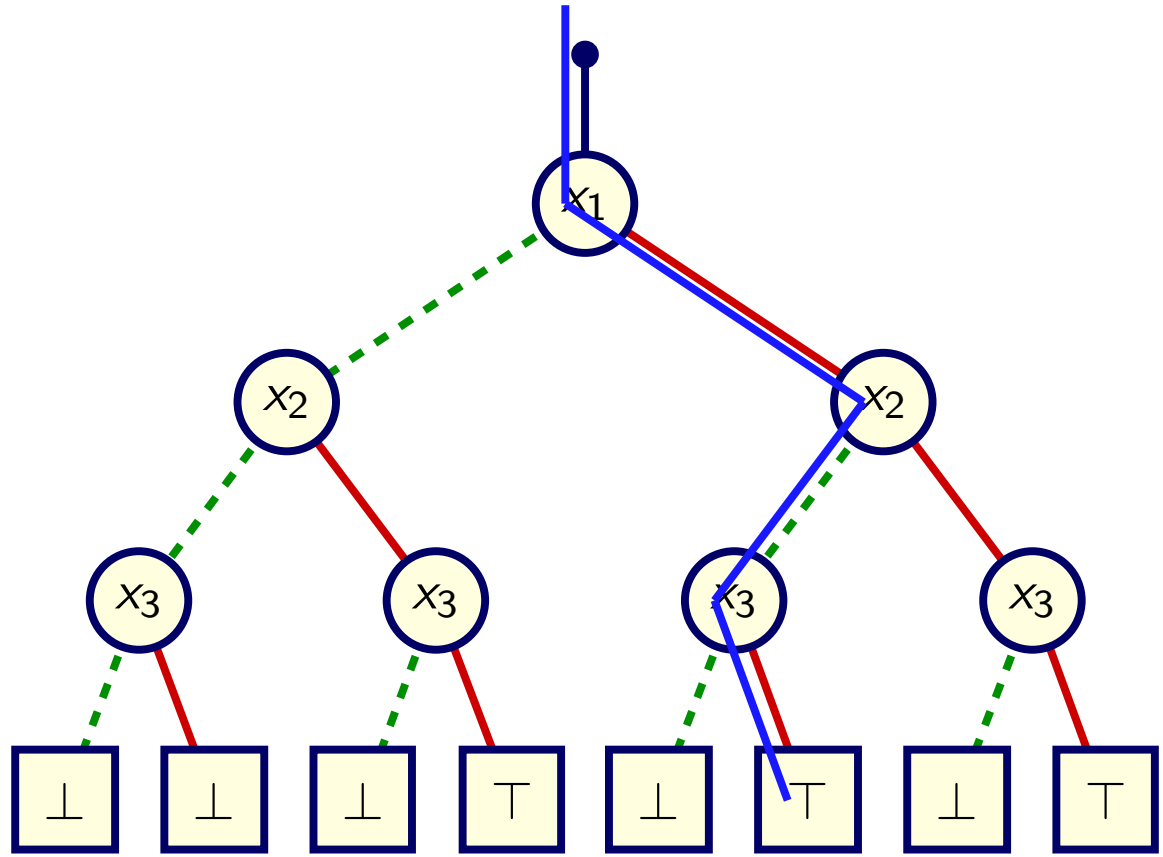
- Size = $O(2^n)$

Boolean Function Representations

Truth Table

x_1	x_2	x_3	f
0	0	0	$\bot$
0	0	1	$\bot$
0	1	0	$\bot$
0	1	1	$\top$
1	0	0	$\bot$
1	0	1	$\top$
1	1	0	$\bot$
1	1	1	$\top$

Decision Tree



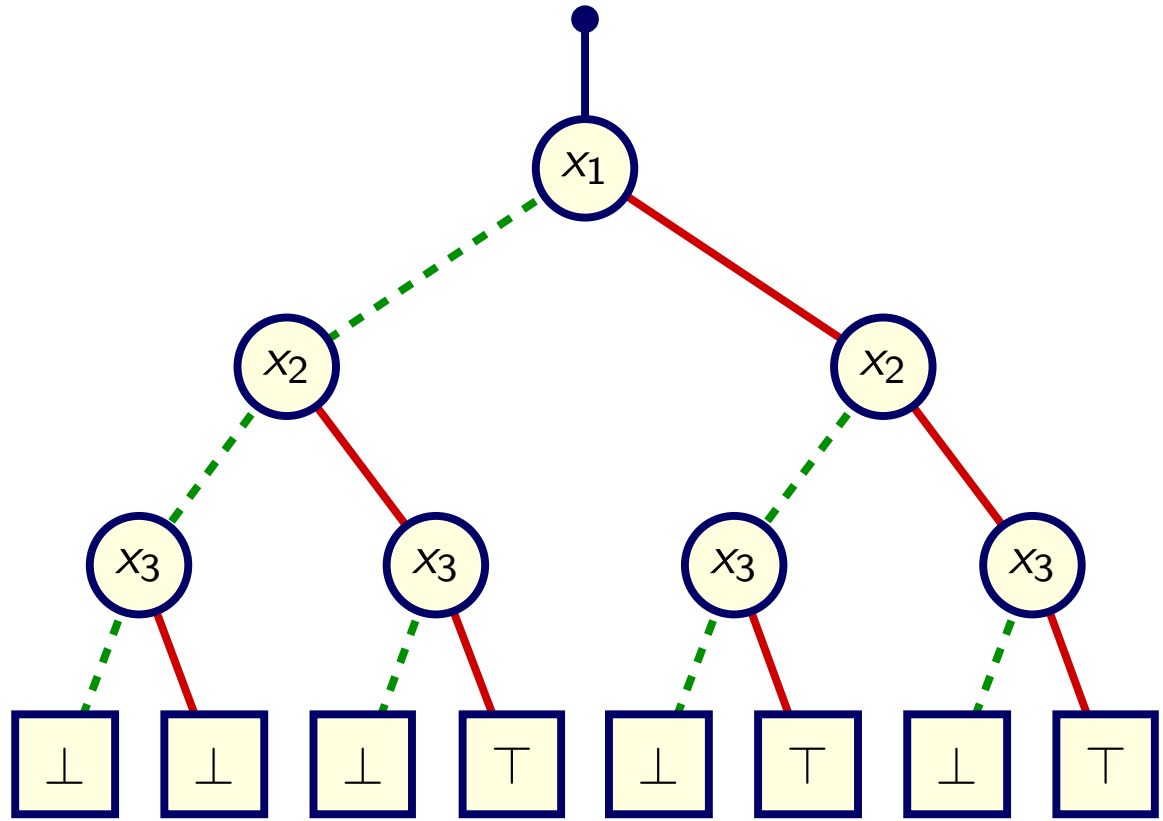
- Size = $O(2^n)$
- Assignment defines path from root to leaf

Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	$\bot$
0	0	1	$\bot$
0	1	0	$\bot$
0	1	1	$\top$
1	0	0	$\bot$
1	0	1	$\top$
1	1	0	$\bot$
1	1	1	$\top$

Graph Representation



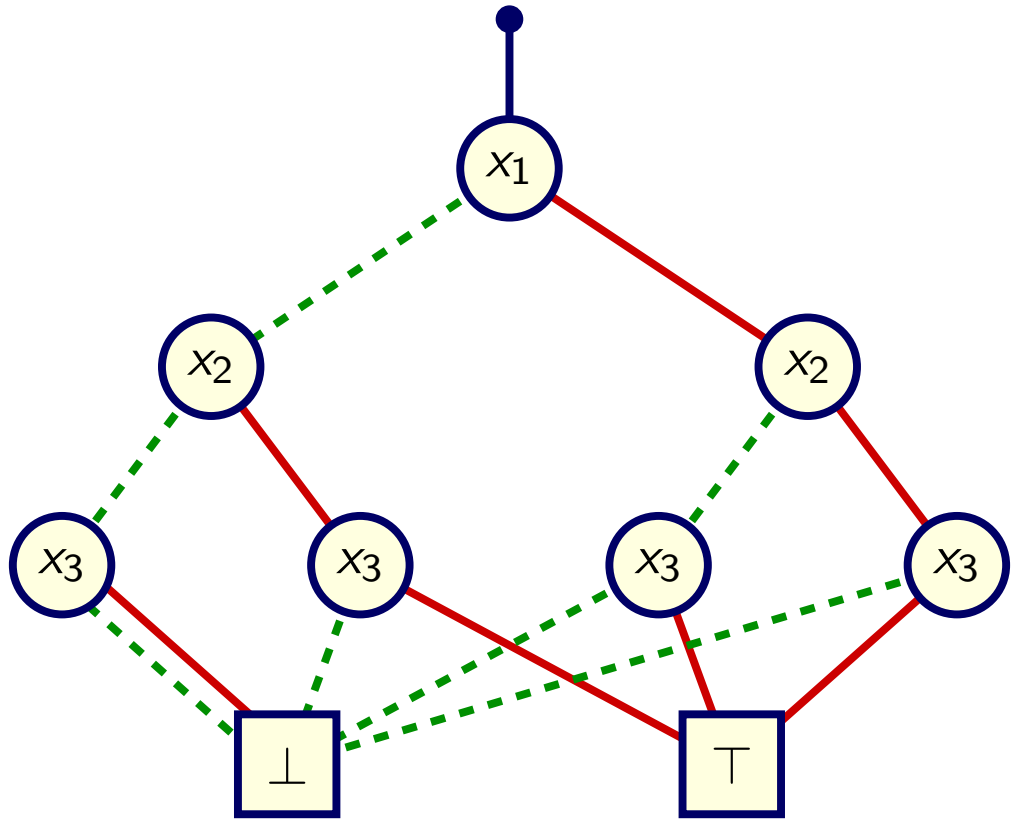
- Merge isomorphic nodes
- Eliminate redundant tests

Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	$\bot$
0	0	1	$\bot$
0	1	0	$\bot$
0	1	1	$\top$
1	0	0	$\bot$
1	0	1	$\top$
1	1	0	$\bot$
1	1	1	$\top$

Graph Representation



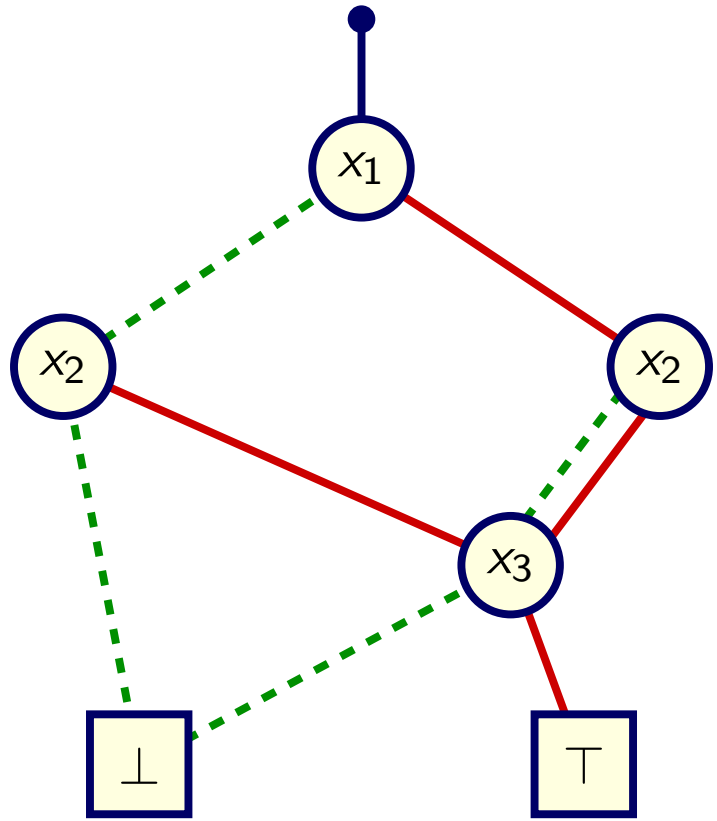
- Merge isomorphic nodes
- Eliminate redundant tests

Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	$\bot$
0	0	1	$\bot$
0	1	0	$\bot$
0	1	1	$\top$
1	0	0	$\bot$
1	0	1	$\top$
1	1	0	$\bot$
1	1	1	$\top$

Graph Representation



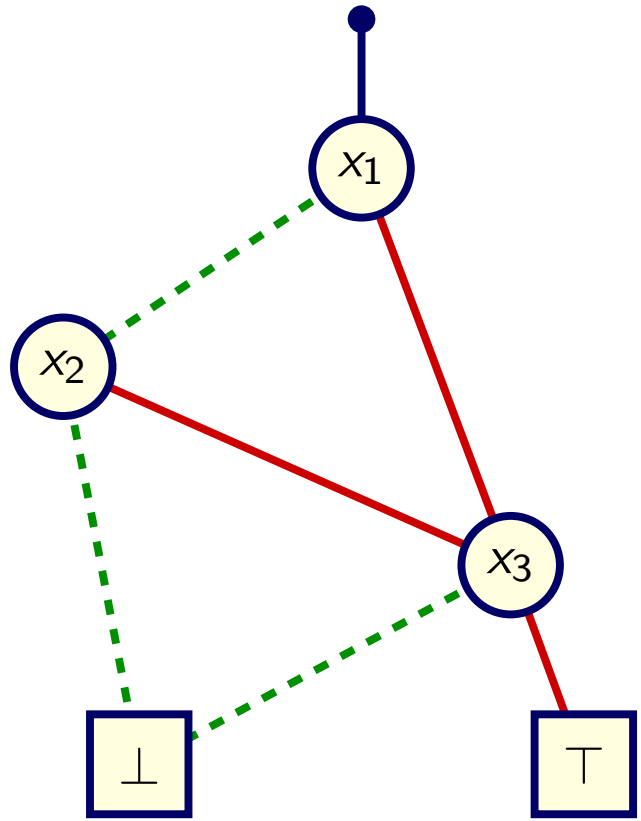
- Merge isomorphic nodes
- Eliminate redundant tests

Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	$\bot$
0	0	1	$\bot$
0	1	0	$\bot$
0	1	1	$\top$
1	0	0	$\bot$
1	0	1	$\top$
1	1	0	$\bot$
1	1	1	$\top$

Graph Representation



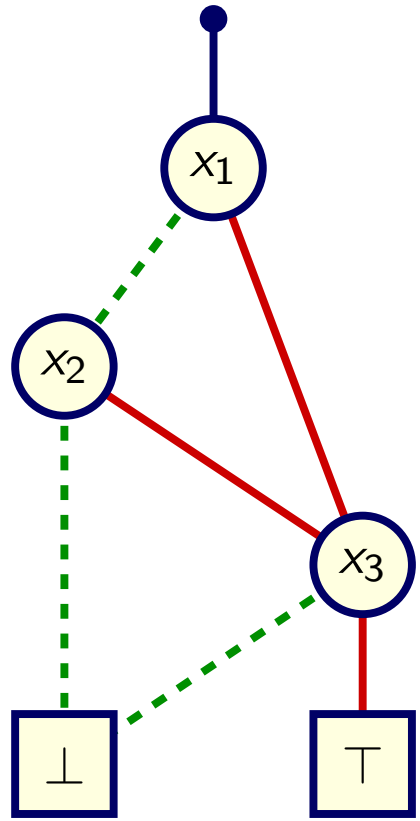
- Merge isomorphic nodes
- Eliminate redundant tests

Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	$\bot$
0	0	1	$\bot$
0	1	0	$\bot$
0	1	1	$\top$
1	0	0	$\bot$
1	0	1	$\top$
1	1	0	$\bot$
1	1	1	$\top$

Reduced Ordered Binary Decision Diagram

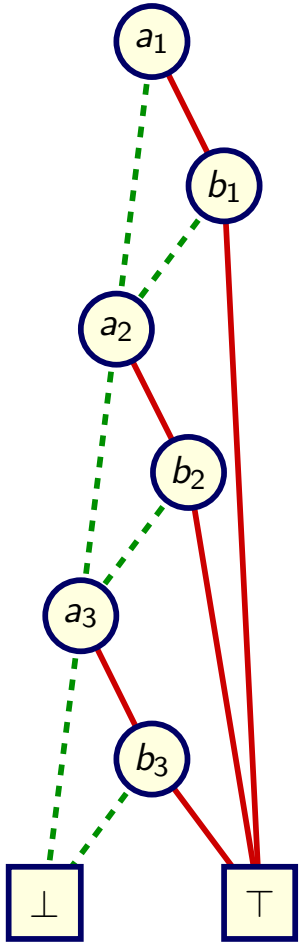


- Canonical representation of Boolean function
- No further simplifications possible

Effect of Variable Ordering

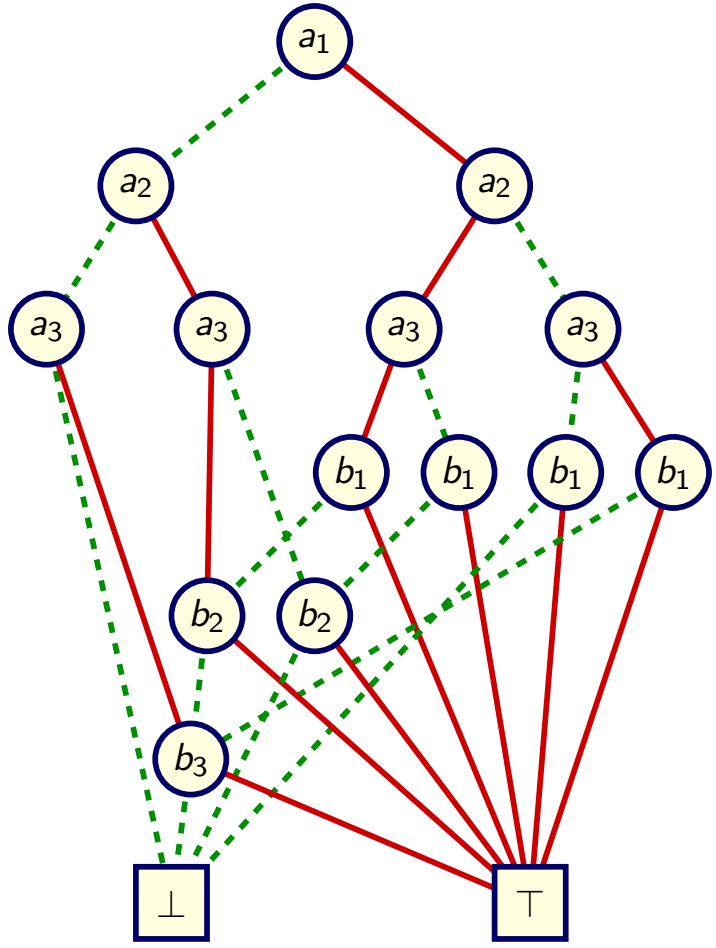
$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$$

Good Ordering



- Linear growth

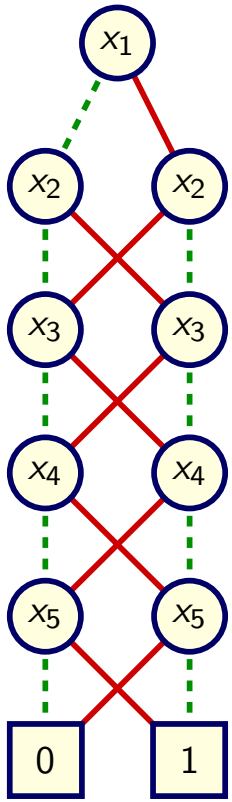
Bad Ordering



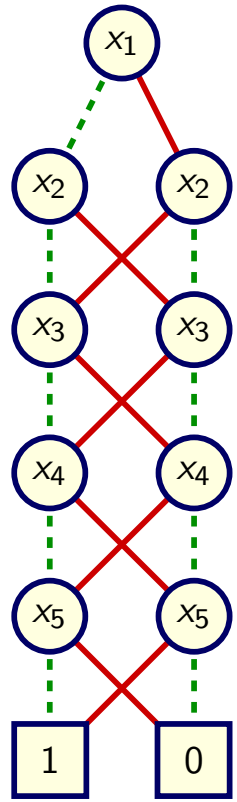
- Exponential growth

BDD Representation of Parity Constraints

Odd Parity



Even Parity



- Linear complexity
- Insensitive to variable order
- Potential major advantage over CDCL

Symbolic Manipulation with BDDs

Strategy

- Represent data as set of BDDs
  - All with same variable ordering
- Express method as sequence of symbolic operations
  - Generate new BDDs. Test properties of BDDs
- Implement each operation via BDD manipulation
  - Never enumerate individual cases
  - Efficient, as long as BDDs stay small

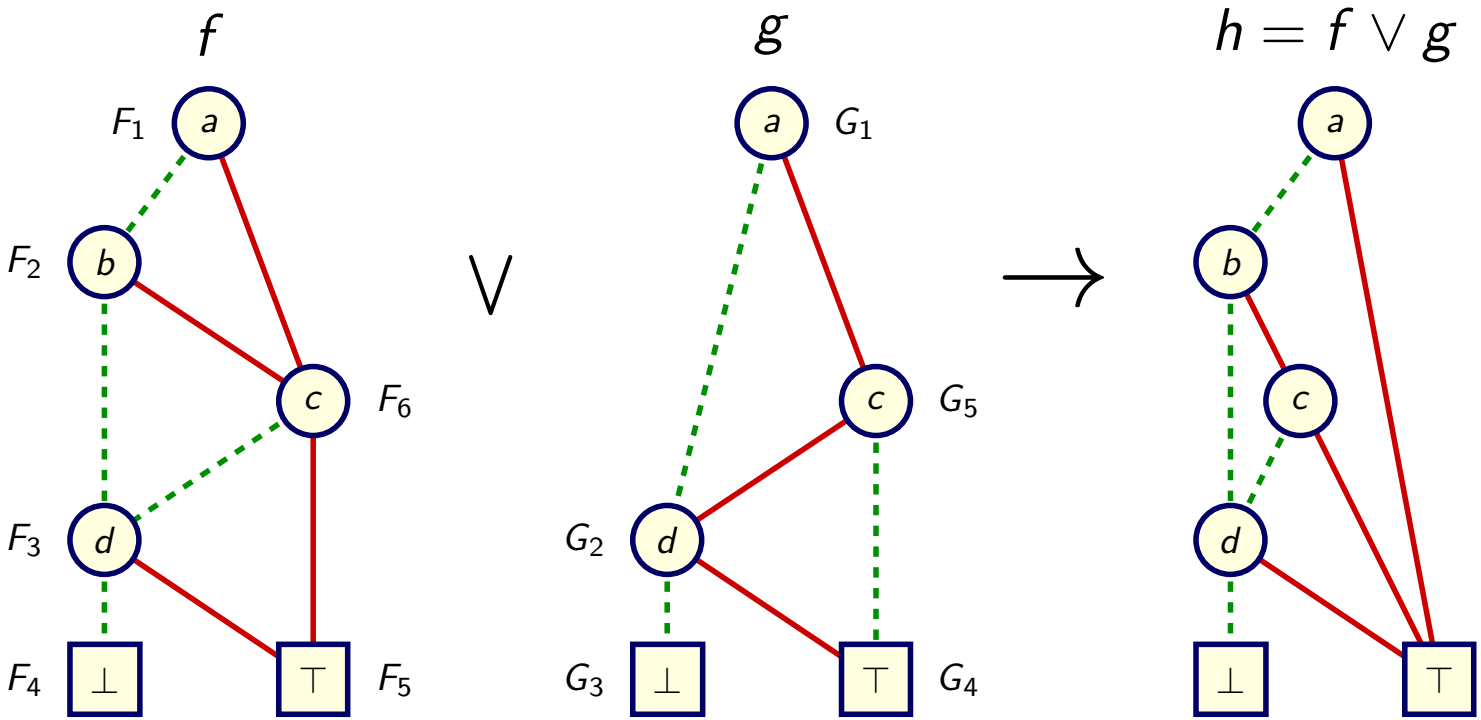
Key Algorithmic Properties

- Arguments at each step are BDDs with same variable ordering
- Result is BDD with same ordering
- Each step has polymomial complexity

Apply Algorithm

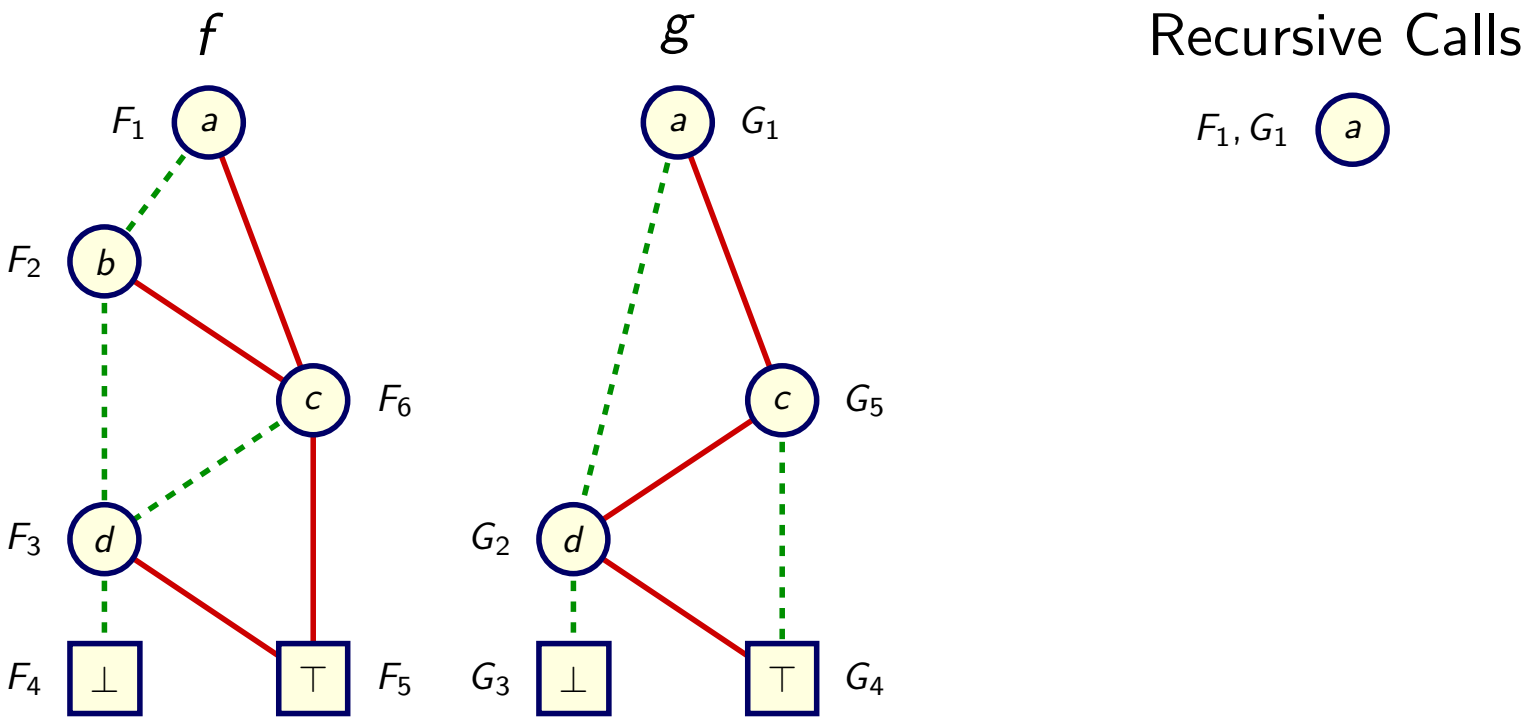
$h \leftarrow f \odot g$

- f, g, h functions represented as BDDs
- $\odot$ binary Boolean operator
  - E.g., $\wedge$, $\vee$, $\oplus$



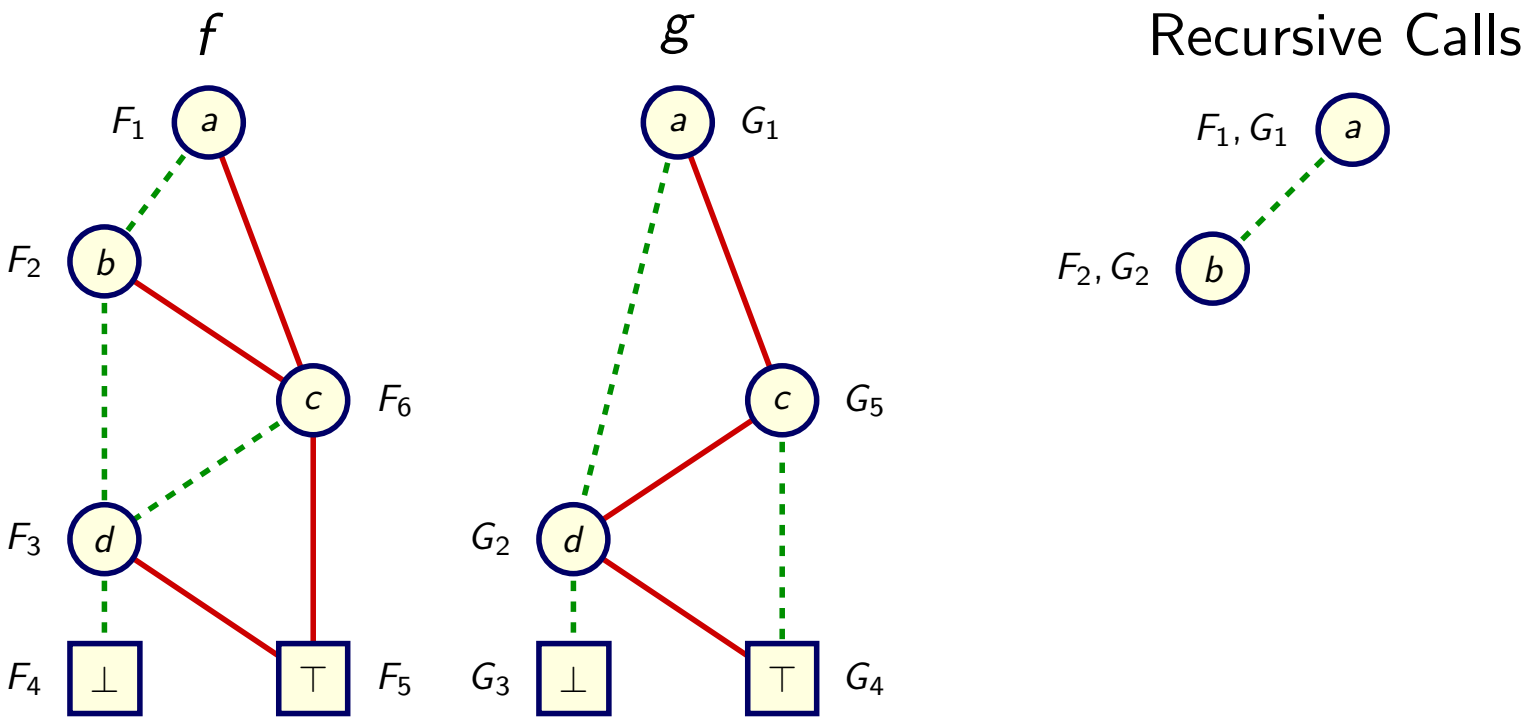
Apply Algorithm Recursion

- Recurse through argument graphs
- Stop when hit terminal case
- Save results in cache to reuse when hit same arguments



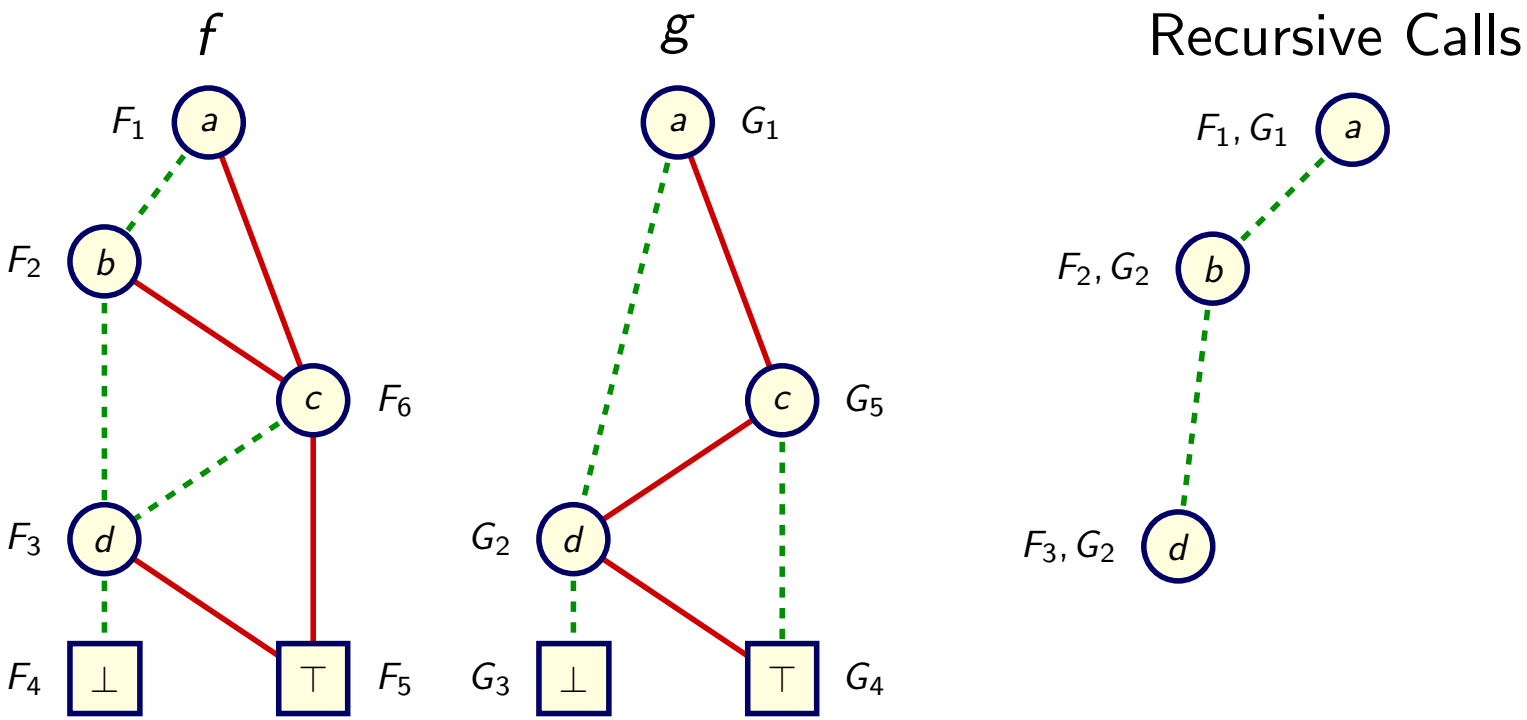
Apply Algorithm Recursion

- Recurse through argument graphs
- Stop when hit terminal case
- Save results in cache to reuse when hit same arguments



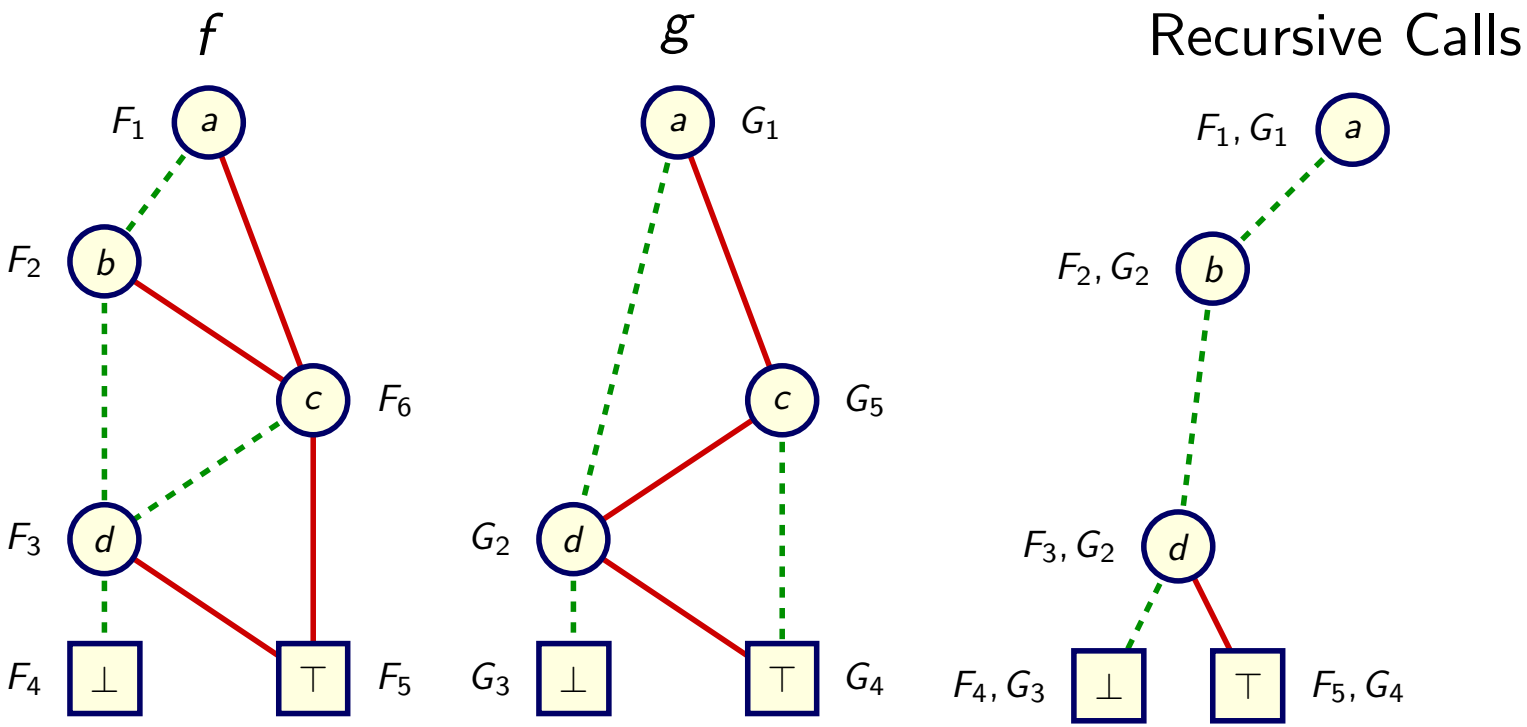
Apply Algorithm Recursion

- Recurse through argument graphs
- Stop when hit terminal case
- Save results in cache to reuse when hit same arguments



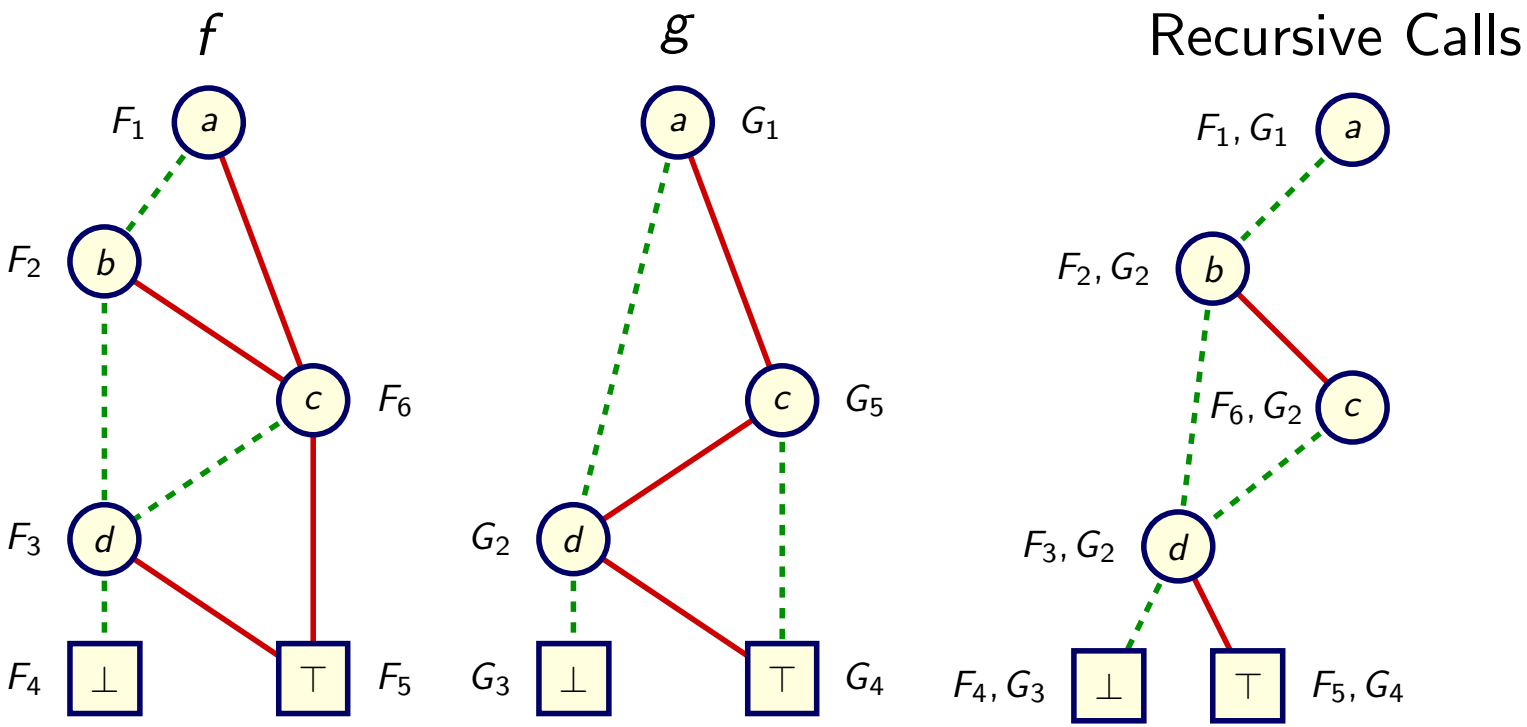
Apply Algorithm Recursion

- Recurse through argument graphs
- Stop when hit terminal case
- Save results in cache to reuse when hit same arguments



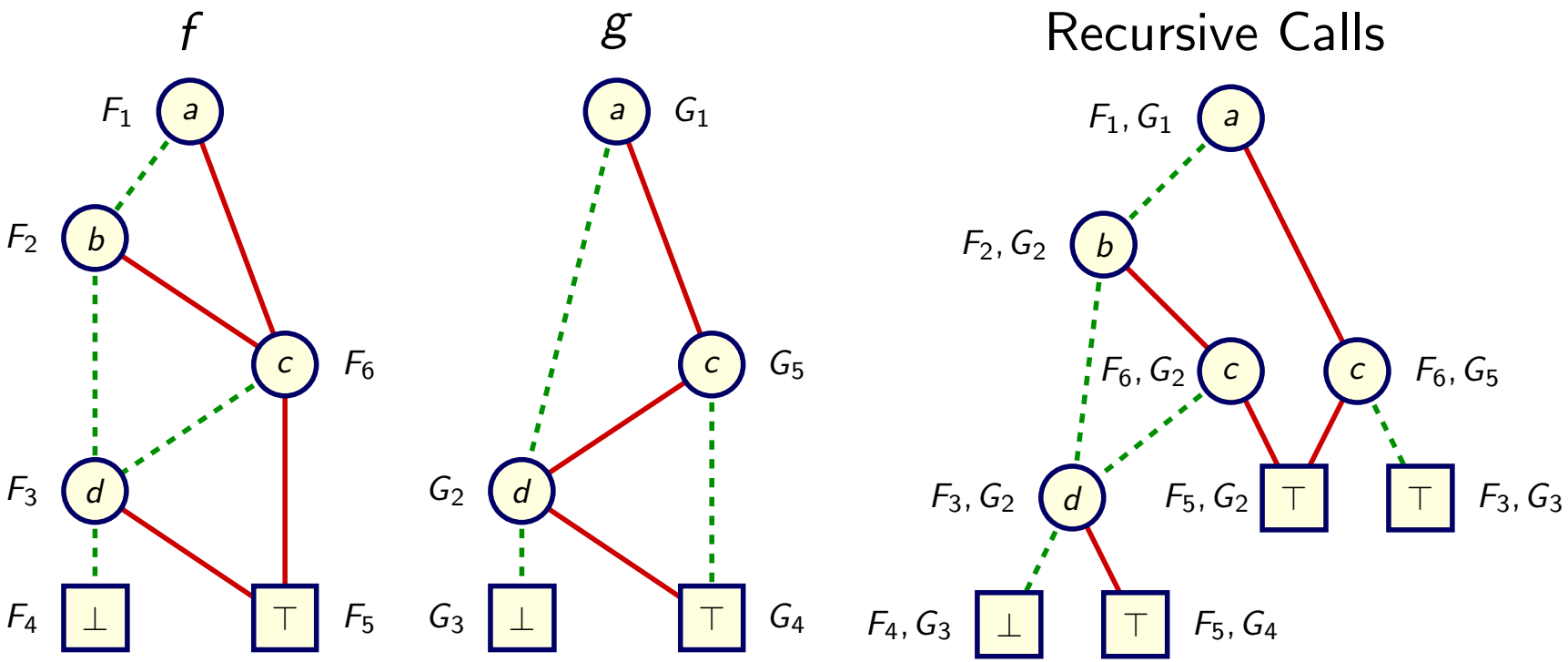
Apply Algorithm Recursion

- Recurse through argument graphs
- Stop when hit terminal case
- Save results in cache to reuse when hit same arguments



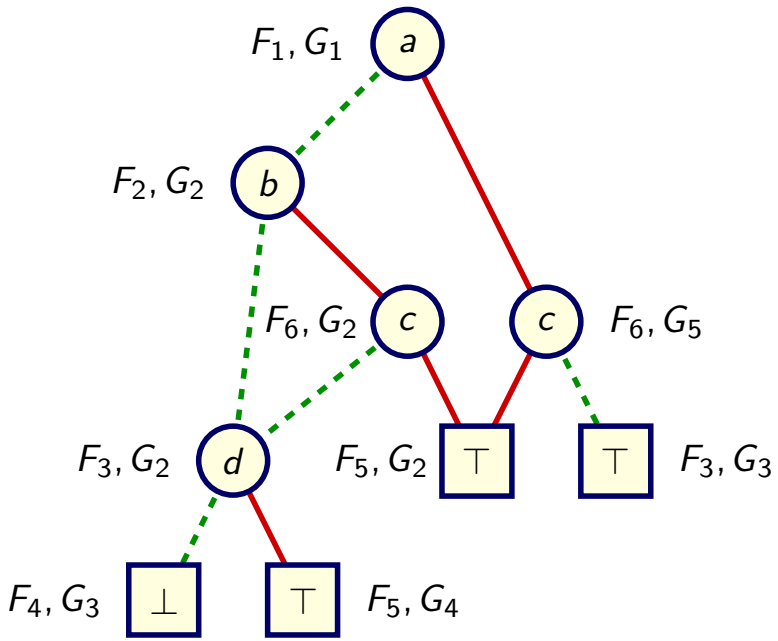
Apply Algorithm Recursion

- Recurse through argument graphs
- Stop when hit terminal case
- Save results in cache to reuse when hit same arguments

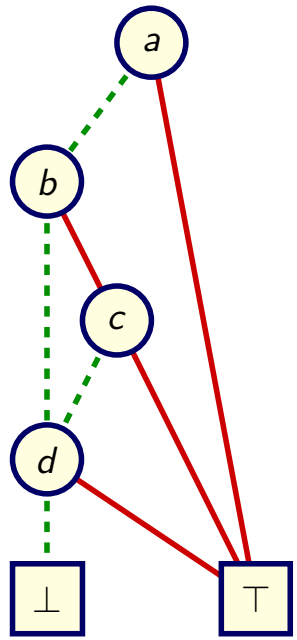


Apply Algorithm Result

Recursive Calls



Reduced Result



Clausal Proofs

Conjunctive Normal Form (CNF) Input Formula

$$C_1, C_2, \ldots, C_m$$

Unsatisfiability Proof

$$C_1, C_2, \ldots, C_m, C_{m+1}, \ldots, C_t$$

- For all $i > m$:

  If $C_1, \ldots, C_{i-1}$ has a satisfying assignment, then so does $C_1, \ldots, C_{i-1}, C_i$.

- $C_t = [\,]$
  - Empty clause unsatisfiable

Clausal Proof Frameworks

Resolution (Robinson, 1965)

- Proof rule guarantees *implication redundancy*:

$$\bigwedge_{1 \leq j < i} C_j \;\rightarrow\; C_i$$

Clausal Proof Frameworks

Resolution (Robinson, 1965)

- Proof rule guarantees *implication redundancy*:

$$\bigwedge_{1 \leq j < i} C_j \;\rightarrow\; C_i$$

Extended Resolution (Tseitin, 1967)

- Allow *extension variables*
  - Variable e shorthand for some formula F over input and previous extension variables
  - Add clauses encoding $e \leftrightarrow F$ to proof
- Can make proofs exponentially more compact

Clausal Proof Frameworks

Resolution (Robinson, 1965)

- Proof rule guarantees *implication redundancy*:

$$\bigwedge_{1 \leq j < i} C_j \;\rightarrow\; C_i$$

Extended Resolution (Tseitin, 1967)

- Allow *extension variables*
  - Variable e shorthand for some formula F over input and previous extension variables
  - Add clauses encoding $e \leftrightarrow F$ to proof
- Can make proofs exponentially more compact

Deletion Resolution Asymmetric Tautology (DRAT)

- Superset of extended resolution
- Variety of efficient checkers, including formally verified ones

Proof-Generating Solvers Based on BDDs

Implementations

- EBDDRES: Sinz, Biere, Jussila, 2006
- PGBDD: Bryant, Heule, 2021
- PGPBS: Bryant, Heule, 2022
  - Supports pseudo-Boolean reasoning

Extended-Resolution Proof Generation

- Introduce extension variable for each BDD node
- Generate proof steps based on recursive structure of BDD algorithms
- Proof is (very) detailed justification of each BDD operation

Proof Comparison

UNSAT Proof from CDCL Solver

- Clauses describe detected conflicts
- Keep narrowing search space until it becomes empty

UNSAT Proof from BDD-Based Solver

- Step by step justification for each node generated
- Sequence of BDD operations leading to leaf node L_0.

Both within DRAT Framework

- Proof system can accomodate variety of styles

TBUDDY Trusted BDD Package

Concept

- BDD package with built-in support for proof generation
- Generate clausal proof as BDD operations proceed

Applications

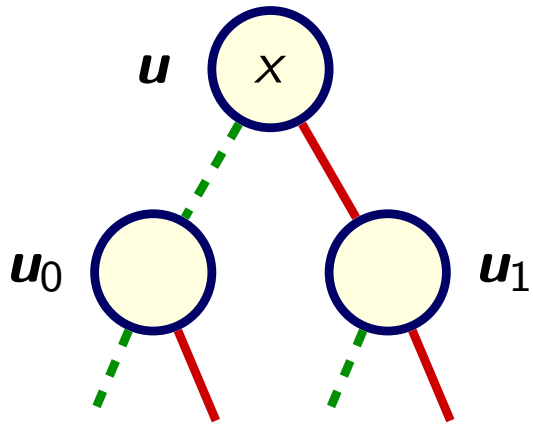
- Implement standalone solver TBSAT
- Incorporate into other solvers

Implementation

- Build on BUDDY BDD package
- Also support parity reasoning

Generating Extended Resolution Proofs

- Extension variable u for each node $\boldsymbol{u}$ in BDD

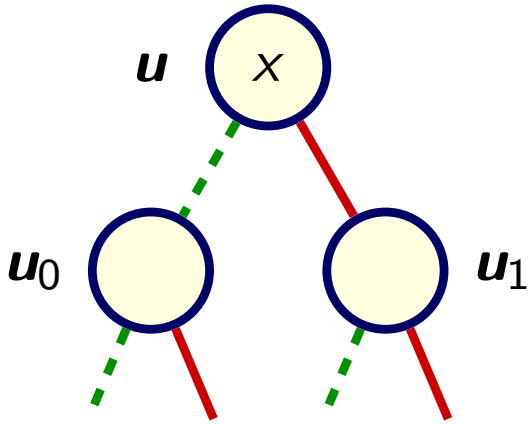


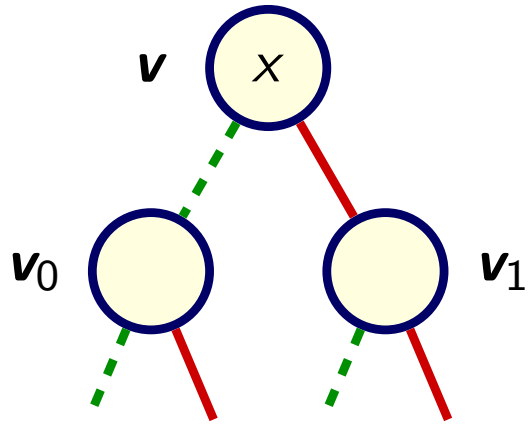
- Defining clauses encode constraint $u \leftrightarrow \mathit{ITE}(x, u_1, u_0)$

Clause name	Formula	Clausal form
HD($\boldsymbol{u}$)	$x \rightarrow (u \rightarrow u_1)$	$[\overline{x} \vee \overline{u} \vee u_1]$
LD($\boldsymbol{u}$)	$\overline{x} \rightarrow (u \rightarrow u_0)$	$[x \vee \overline{u} \vee u_0]$
HU($\boldsymbol{u}$)	$x \rightarrow (u_1 \rightarrow u)$	$[\overline{x} \vee \overline{u}_1 \vee u]$
LU($\boldsymbol{u}$)	$\overline{x} \rightarrow (u_0 \rightarrow u)$	$[x \vee \overline{u}_0 \vee u]$

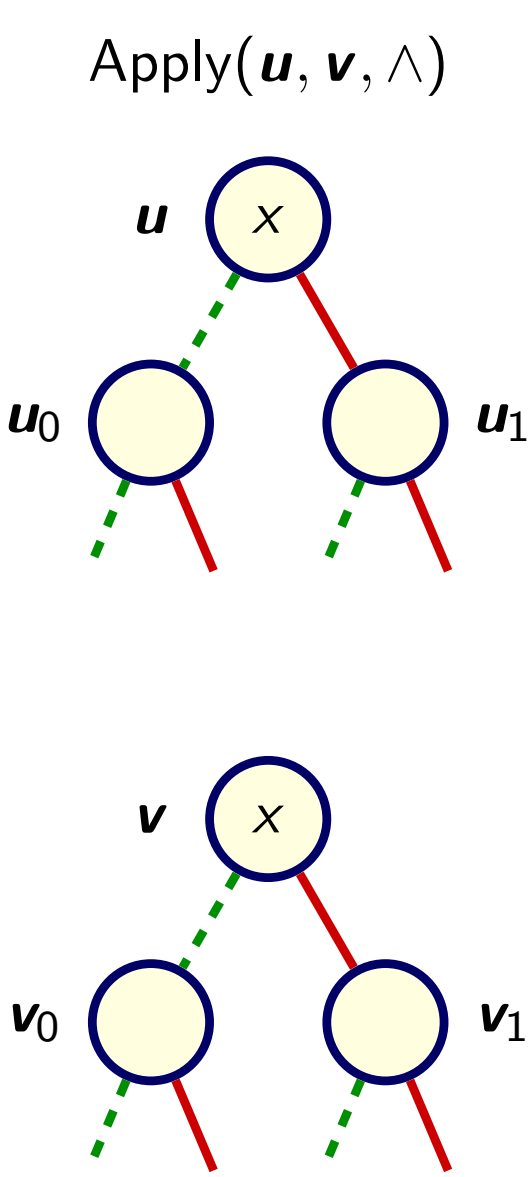
Apply Algorithm Recursion

$\text{Apply}(\boldsymbol{u}, \boldsymbol{v}, \wedge)$





Apply Algorithm Recursion

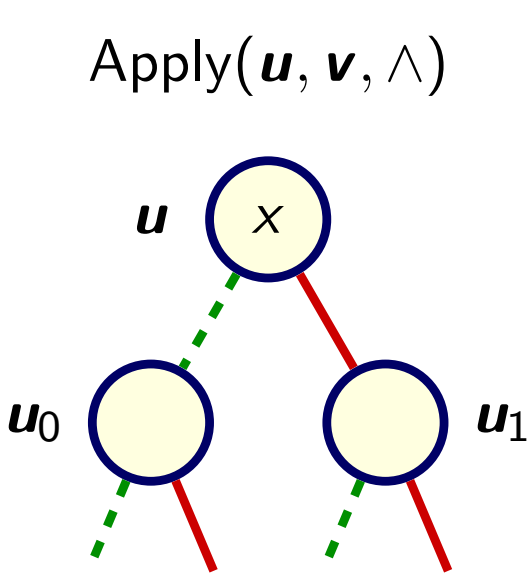


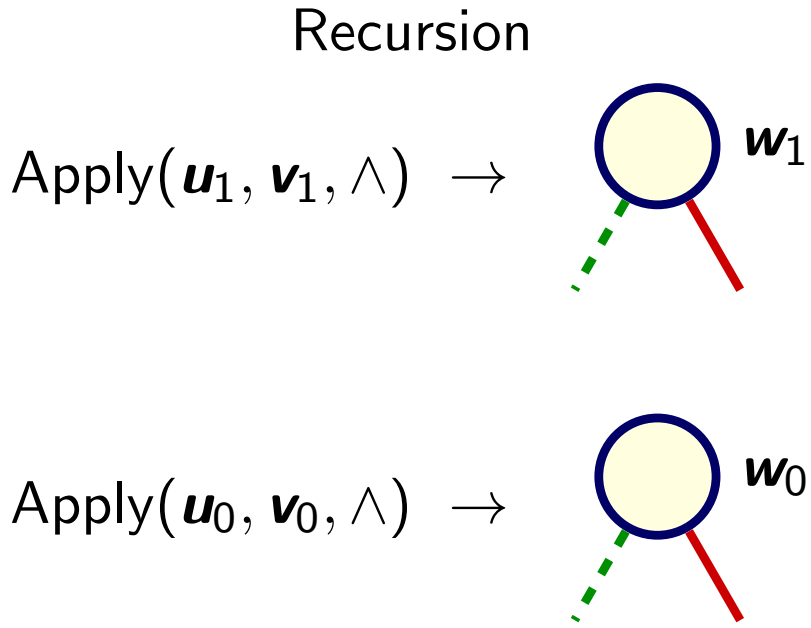
Recursion

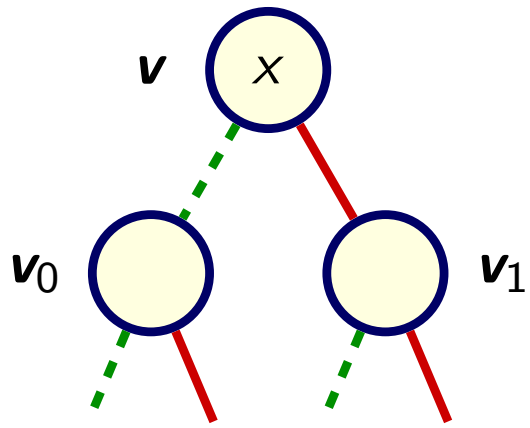
Apply(u_1, v_1, $\wedge$) $\rightarrow$ w_1

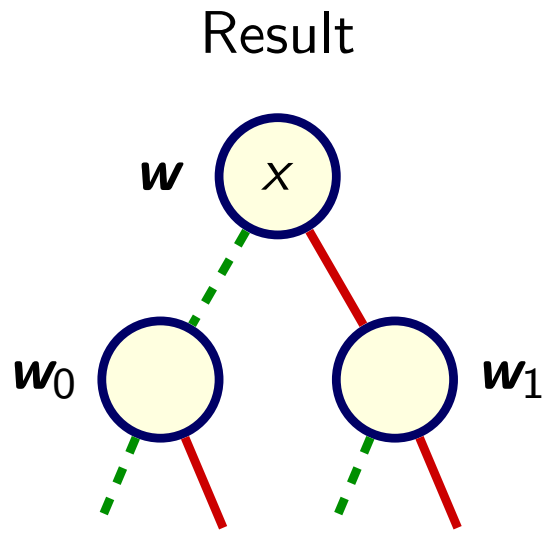
Apply(u_0, v_0, $\wedge$) $\rightarrow$ w_0

Apply Algorithm Recursion









Proof-Generating Apply Operation

Integrate Proof Generation into Apply Operation

- When Apply($\boldsymbol{u}, \boldsymbol{v}, \wedge$) returns $\boldsymbol{w}$, also generate proof $u \wedge v \rightarrow w$
- **Key Idea:** Proof based on the underlying logic of the Apply algorithm

Proof Structure

- Assume recursive calls generate proofs
  - $u_1 \wedge v_1 \rightarrow w_1$
  - $u_0 \wedge v_0 \rightarrow w_0$
- Combine with defining clauses for nodes $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$

Apply Proof Structure

Defining Clauses

Clause	Formula	Clause	Formula
HD(u)	$x \rightarrow (u \rightarrow u_1)$	LD(u)	$\overline{x} \rightarrow (u \rightarrow u_0)$
HD(v)	$x \rightarrow (v \rightarrow v_1)$	LD(v)	$\overline{x} \rightarrow (v \rightarrow v_0)$
HU(w)	$x \rightarrow (w_1 \rightarrow w)$	LU(w)	$\overline{x} \rightarrow (w_0 \rightarrow w)$

Resolution Steps

$$\dfrac{\dfrac{\dfrac{x \rightarrow (u \rightarrow u_1) \quad x \rightarrow (v \rightarrow v_1) \quad x \rightarrow (w_1 \rightarrow w) \quad u_1 \wedge v_1 \rightarrow w_1}{x \rightarrow (u \wedge v \rightarrow w)} \quad \dfrac{\overline{x} \rightarrow (u \rightarrow u_0) \quad \overline{x} \rightarrow (v \rightarrow v_0) \quad \overline{x} \rightarrow (w_0 \rightarrow w) \quad u_0 \wedge v_0 \rightarrow w_0}{\overline{x} \rightarrow (u \wedge v \rightarrow w)}}{u \wedge v \rightarrow w}$$

Quantification Operation

Operation EQuant($\boldsymbol{u}, x$)

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

- Abstract away details of satisfying solutions
- Not logically required for SAT solver
  - But, critical for obtaining good performance

Proof Generation

- Do not attempt to follow recursive structure of algorithm
- Instead, follow with separate implication proof generation
  - EQuant($\boldsymbol{u}, x$) $\rightarrow \boldsymbol{w}$
  - Generate proof $u \rightarrow w$
  - Algorithm similar to proof-generating Apply operation

Trusted BDDs (TBDDs)

Components of TBDD $\dot{u}$

- BDD with root node u.
- Associated extension variable u
- Proof step for unit clause $[u]$

Interpretation. For input formula ϕ:

- $\phi \models u$
- Any variable assignment that satisfies ϕ must yield 1 for BDD with root u

TBDD API

`tbdd tbdd_from_clause_id(int i);`

- Create TBDD representation $\dot{\boldsymbol{u}}_i$ of input clause C_i
  - Add proof step for $C_i \models u_i$

`tbdd tbdd_and(tbdd` $\dot{\boldsymbol{u}}$`, tbdd` $\dot{\boldsymbol{v}}$`);`

- Form conjunction $\dot{\boldsymbol{w}}$ of TBDDs $\dot{\boldsymbol{u}}$ and $\dot{\boldsymbol{v}}$.
  - Apply operation generates proof $u \wedge v \rightarrow w$
  - Resolution with unit clauses $[u]$ and $[v]$ yields unit clause $[w]$

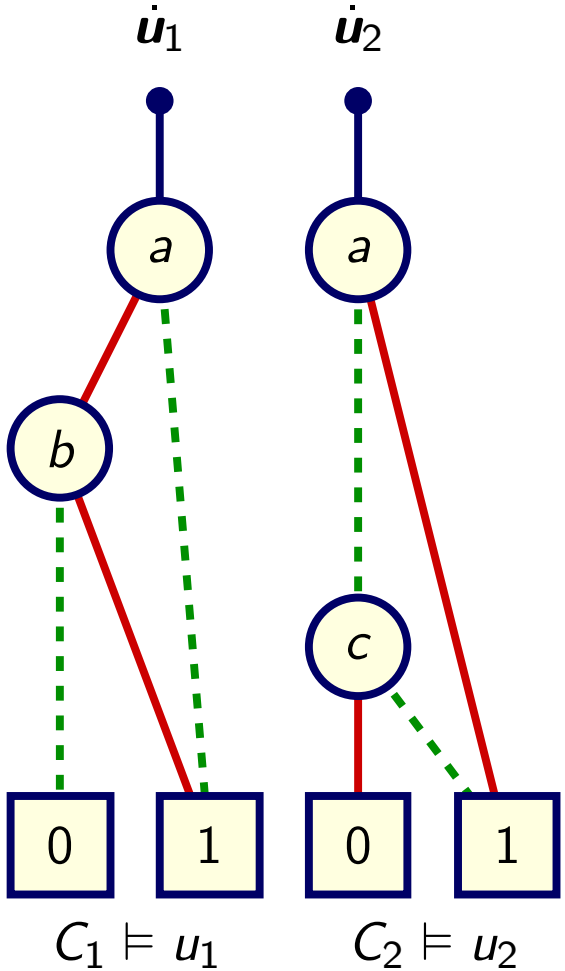
`tbdd tbdd_validate(bdd` $\boldsymbol{v}$`, tbdd` $\dot{\boldsymbol{u}}$`);`

- Upgrade BDD $\boldsymbol{v}$ to TBDD $\dot{\boldsymbol{v}}$
  - Apply operation generates proof $u \rightarrow v$
  - Resolution with unit clause $[u]$ yields unit clause $[v]$

TBDD Execution Example

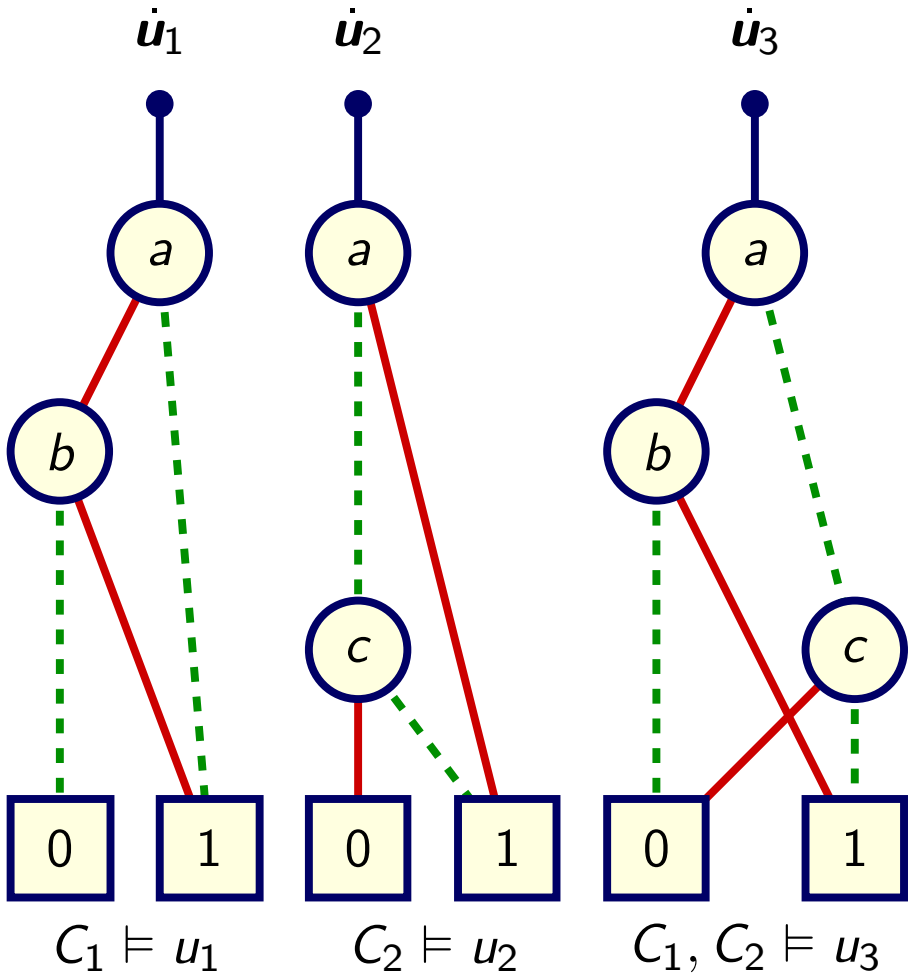
$\dot{u}_1 \longleftarrow$ `tbdd_from_clause`(C_1)

$\dot{u}_2 \longleftarrow$ `tbdd_from_clause`(C_2)



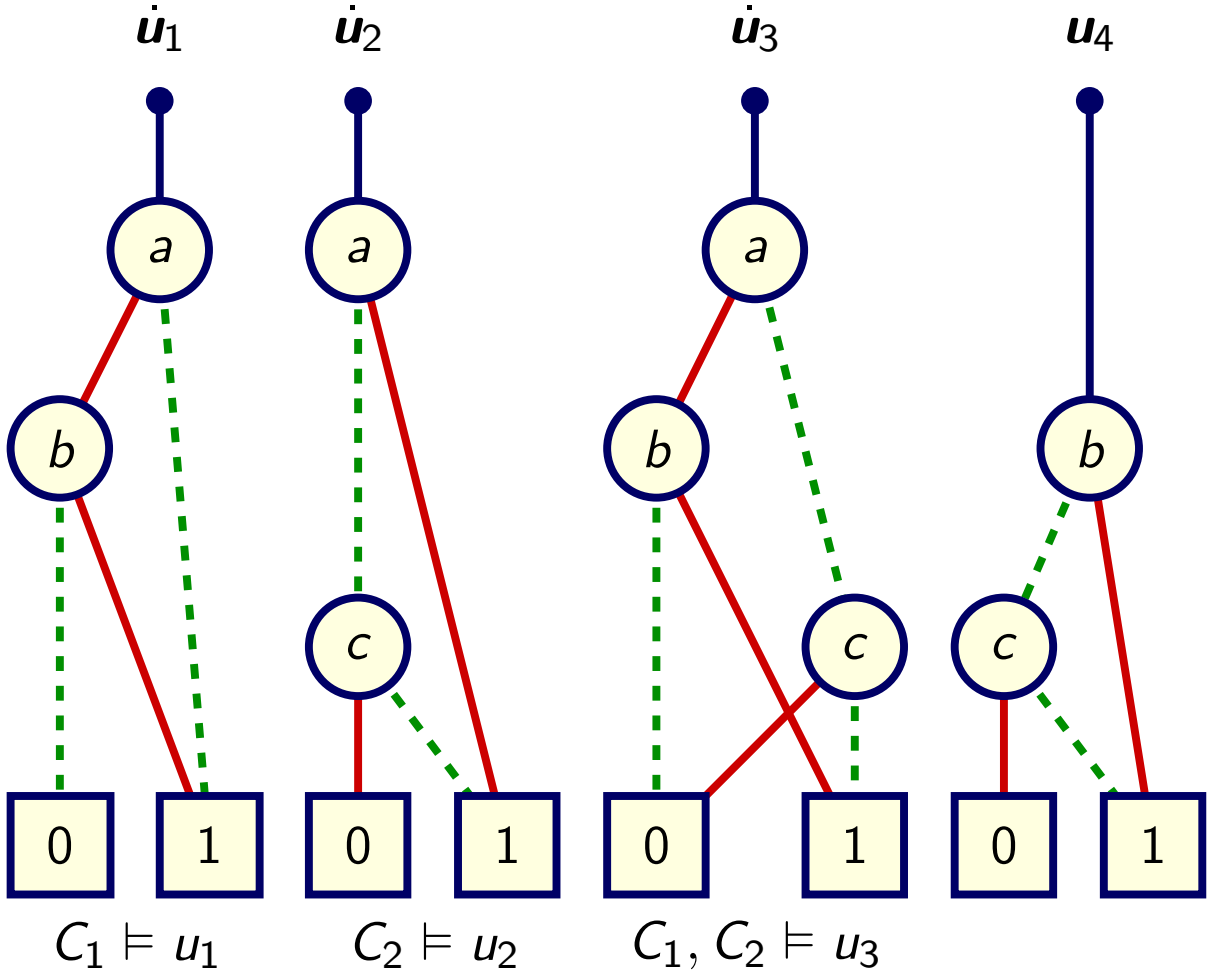
TBDD Execution Example

$\dot{u}_3 \longleftarrow$ tbdd_and($\dot{u}_1$, $\dot{u}_2$)



TBDD Execution Example

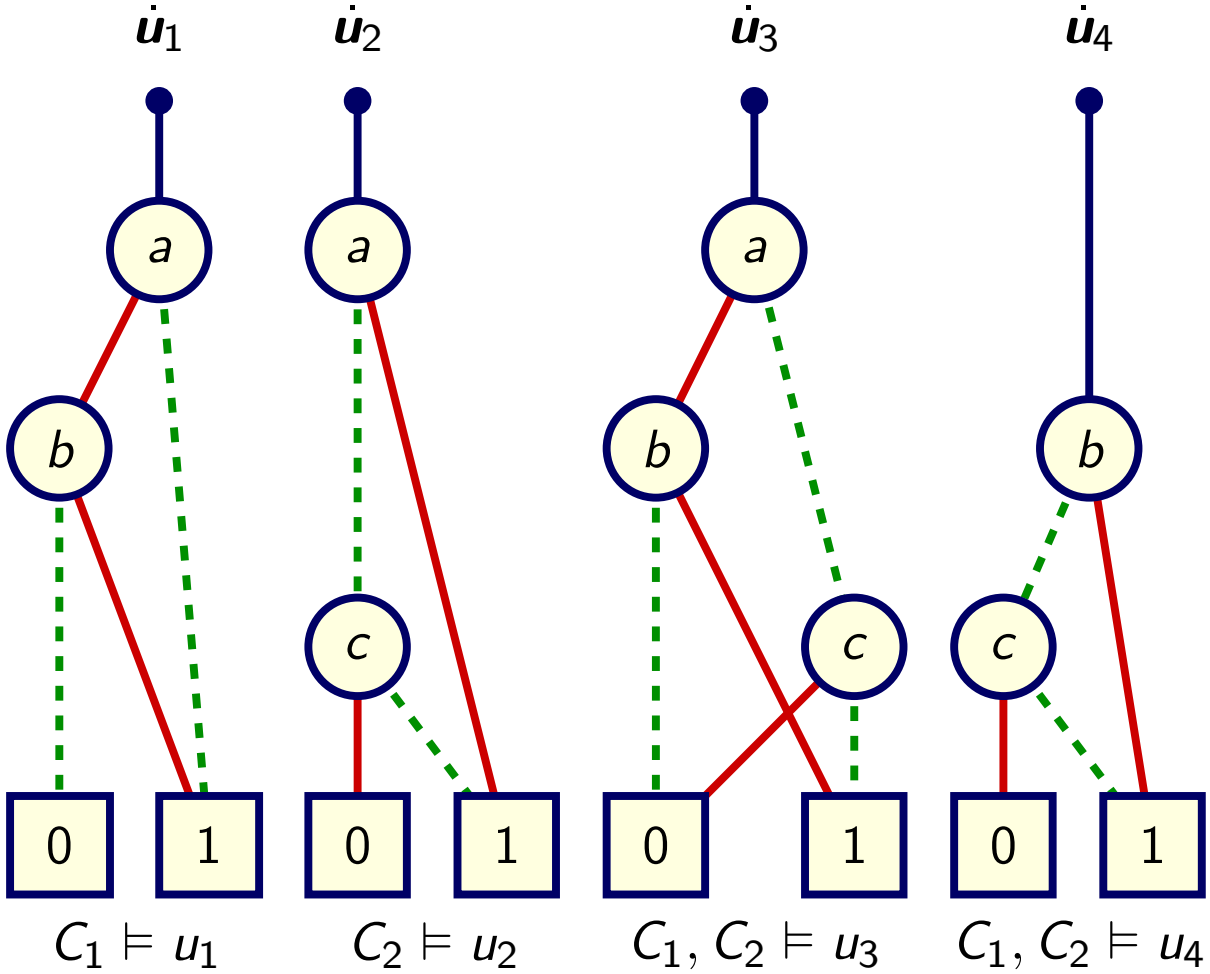
$u_4 \longleftarrow$ `bdd_exists`(u_3, a)



TBDD Execution Example

$\boldsymbol{u}_4 \longleftarrow$ `bdd_exists`$(\boldsymbol{u}_3, a)$

$\dot{\boldsymbol{u}}_4 \longleftarrow$ `tbdd_validate`$(\boldsymbol{u}_4, \dot{\boldsymbol{u}}_3)$



BuDDy BDD Package

BuDDy: Binary Decision Diagram package
Release 2.2

Jørn Lind-Nielsen
IT-University of Copenhagen (ITU)
e-mail: buddy@itu.dk
November 9, 2002

- ∼12K lines of code
- Clean, robust, and well documented
- Benchmark comparisons demonstrate good performance
- Node identified by 32-bit index into table
  - Rather than as 64-bit pointer

TBUDDY Data Structures

Node data

level, mark, rc
low
high
next
head
xvar
dclause

Cache entry

op
arg1
arg2
arg3
res
jclause

Key

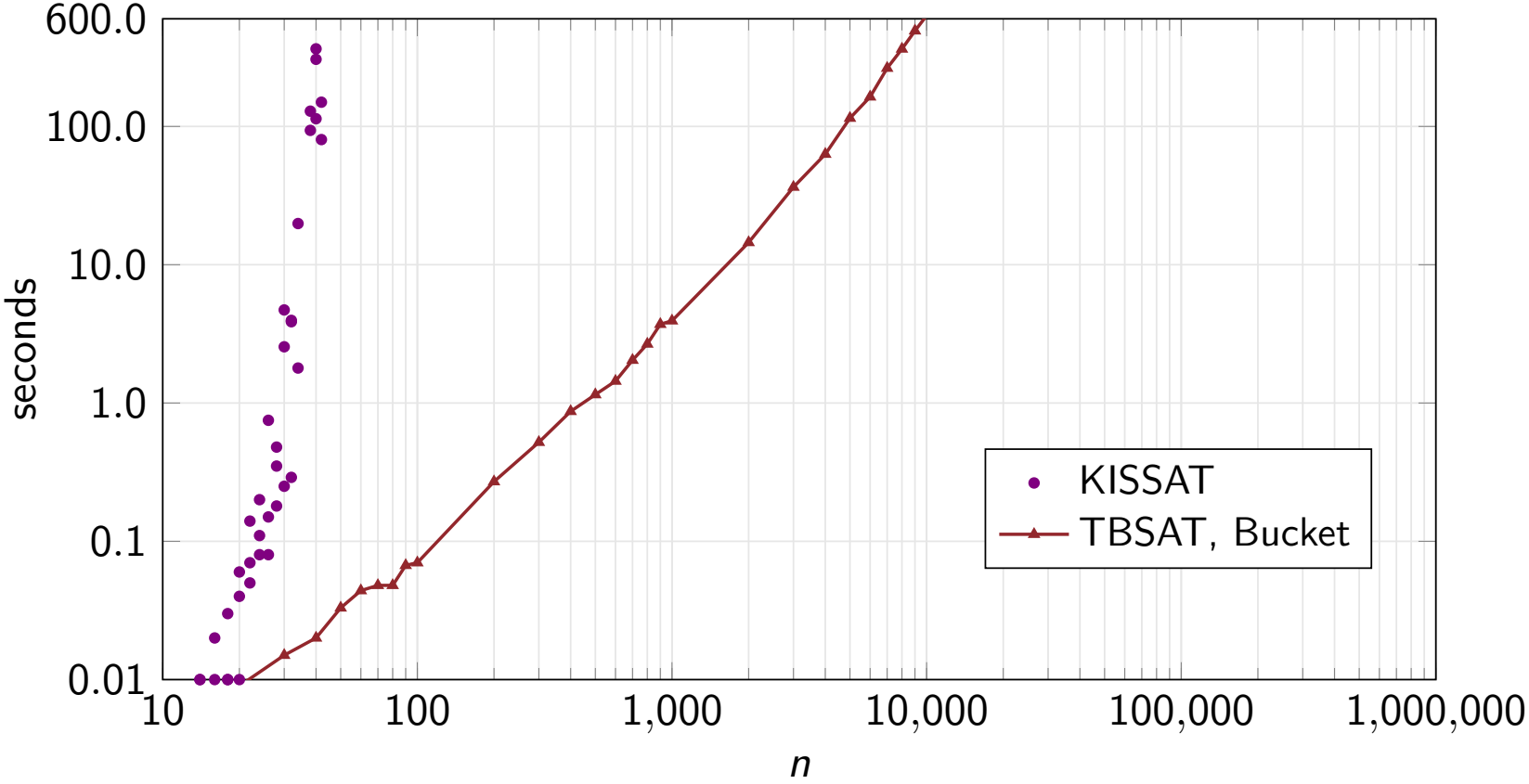
Node representation

Unique table

Proof information

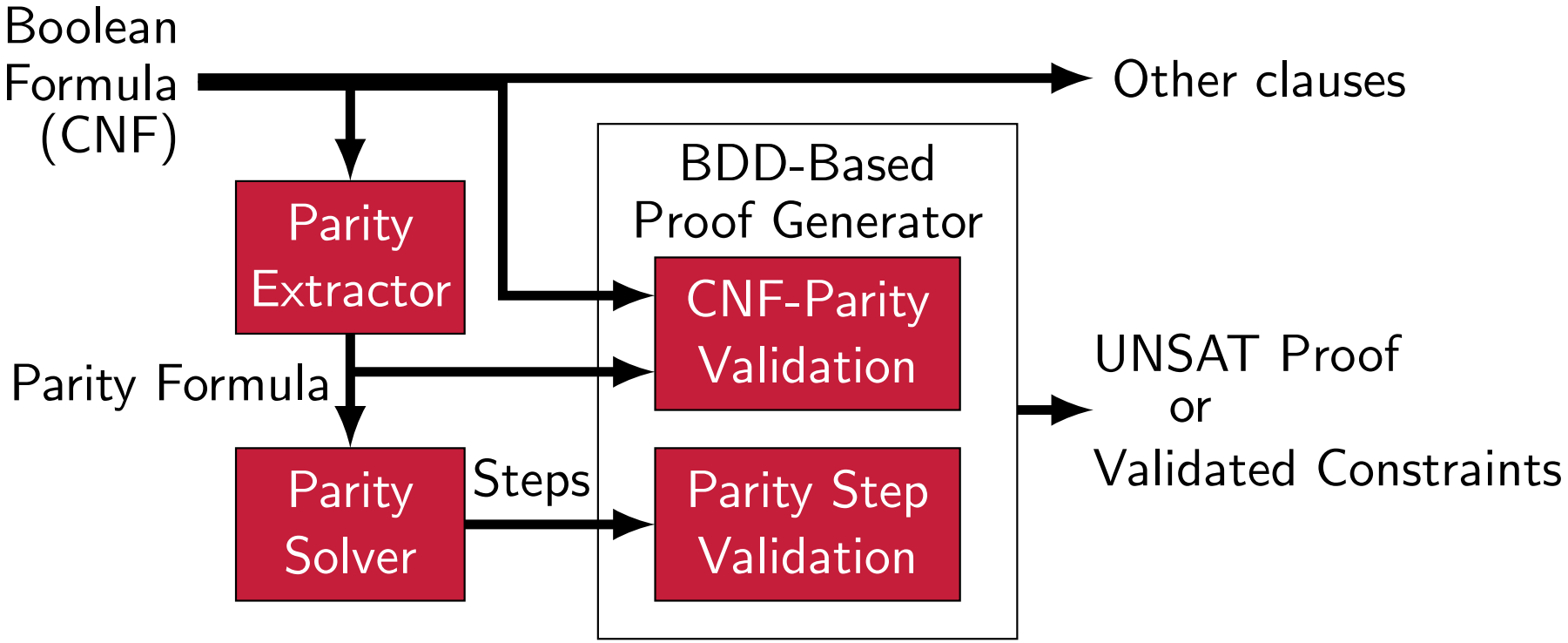
- Node entries: 20 bytes $\rightarrow$ 28 to store proof information
- Cache entry: Existing 24 bytes can also hold proof information
- Total memory overhead 1.35$\times$

Parity Benchmark Runtime



- Bucket elimination
  - Systematic way to perform conjunctions and quantifications
- Random variable ordering
- No guidance from user

Integrating Parity Reasoning



- Fully automated
- UNSAT if constraints infeasible
- Otherwise, supply validated constraints to BDD-based solver

Gaussian Elimination Over GF2

Parity Constraints $\mathcal{P} = P_1, P_2, \ldots, P_m$, each of form

$$a_1 \cdot x_1 \oplus a_2 \cdot x_2 \oplus \cdots \oplus a_n \cdot x_n \quad = \quad p$$

- Coefficients $a_i \in \{0, 1\}$
- Phase $p \in \{0, 1\}$

Elimination Steps

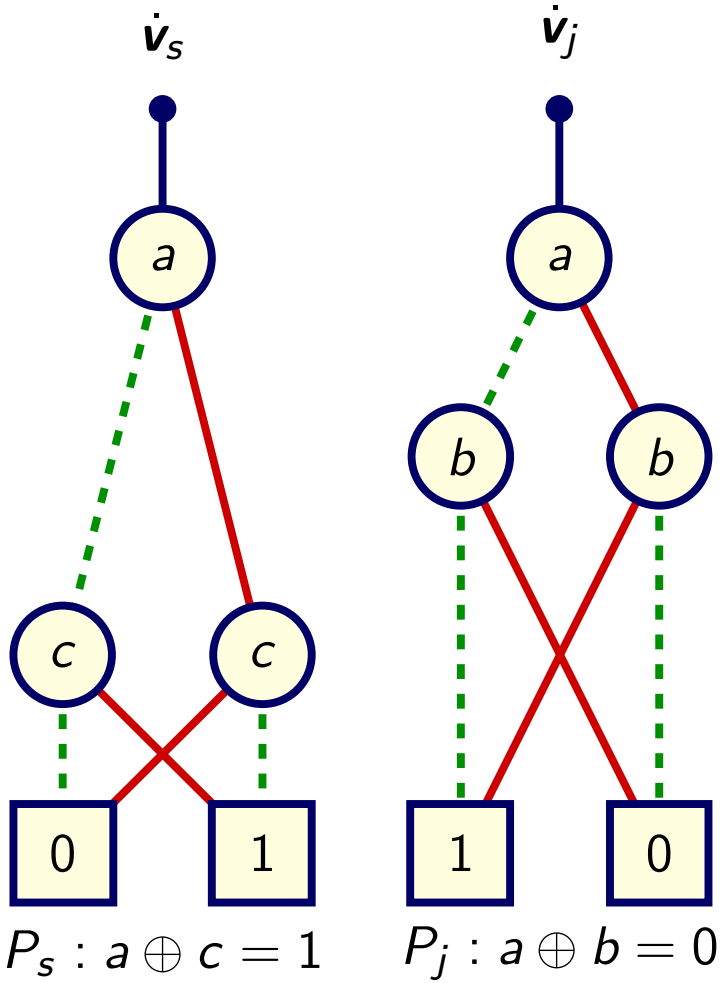
- For pivot P_s, replace each constraint P_j by $P'_j = P_j \oplus P_s$

	a_1	a_2	a_3	a_4	p
P_s	1	0	1	1	0
P_j	1	1	0	1	1
P'_j	0	1	1	0	1

- Stop with infeasible constraint $0 = 1$

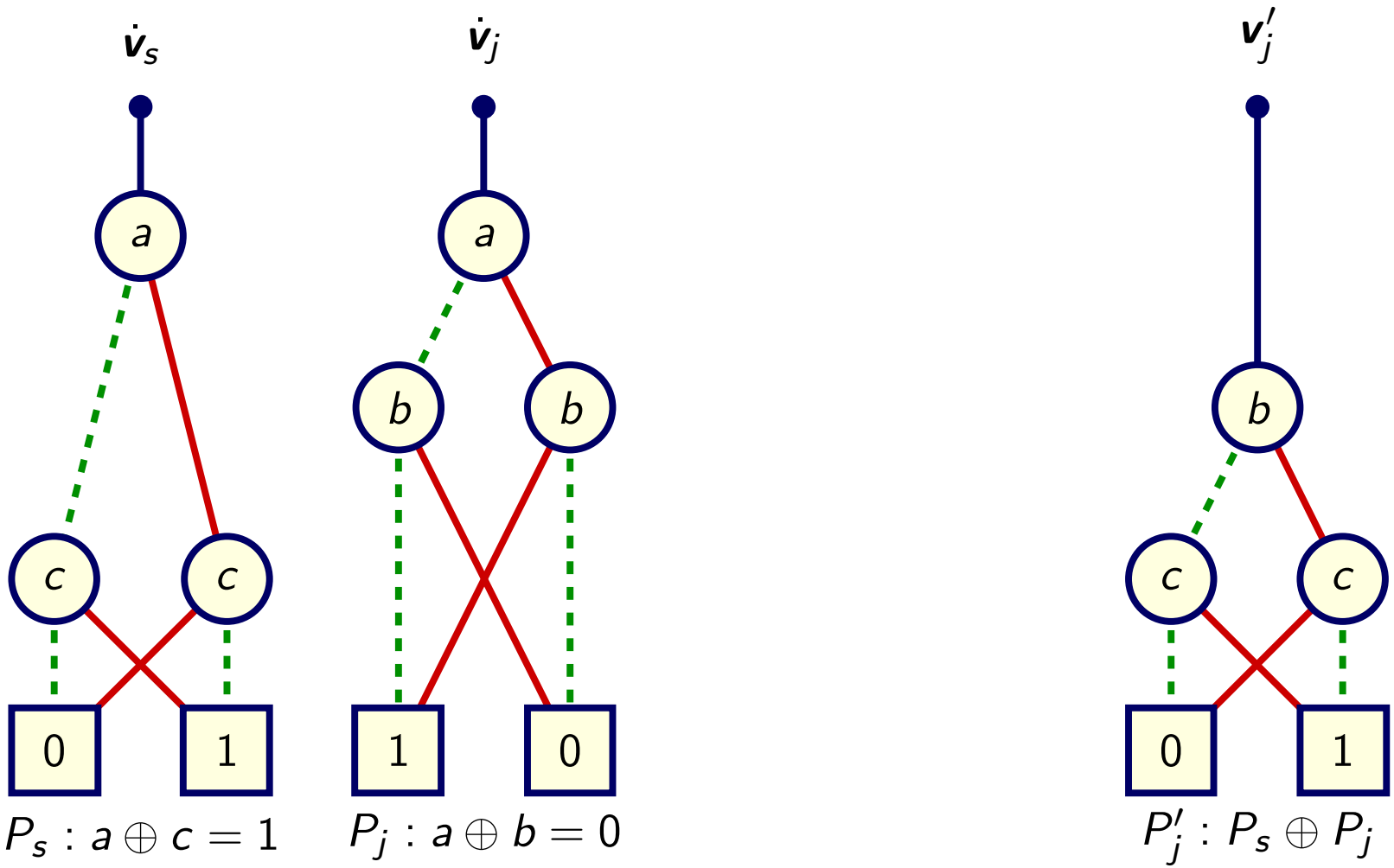
TBDD-Based Parity Reasoning Example

Goal: Compute $P_j' \longleftarrow P_s \oplus P_j$



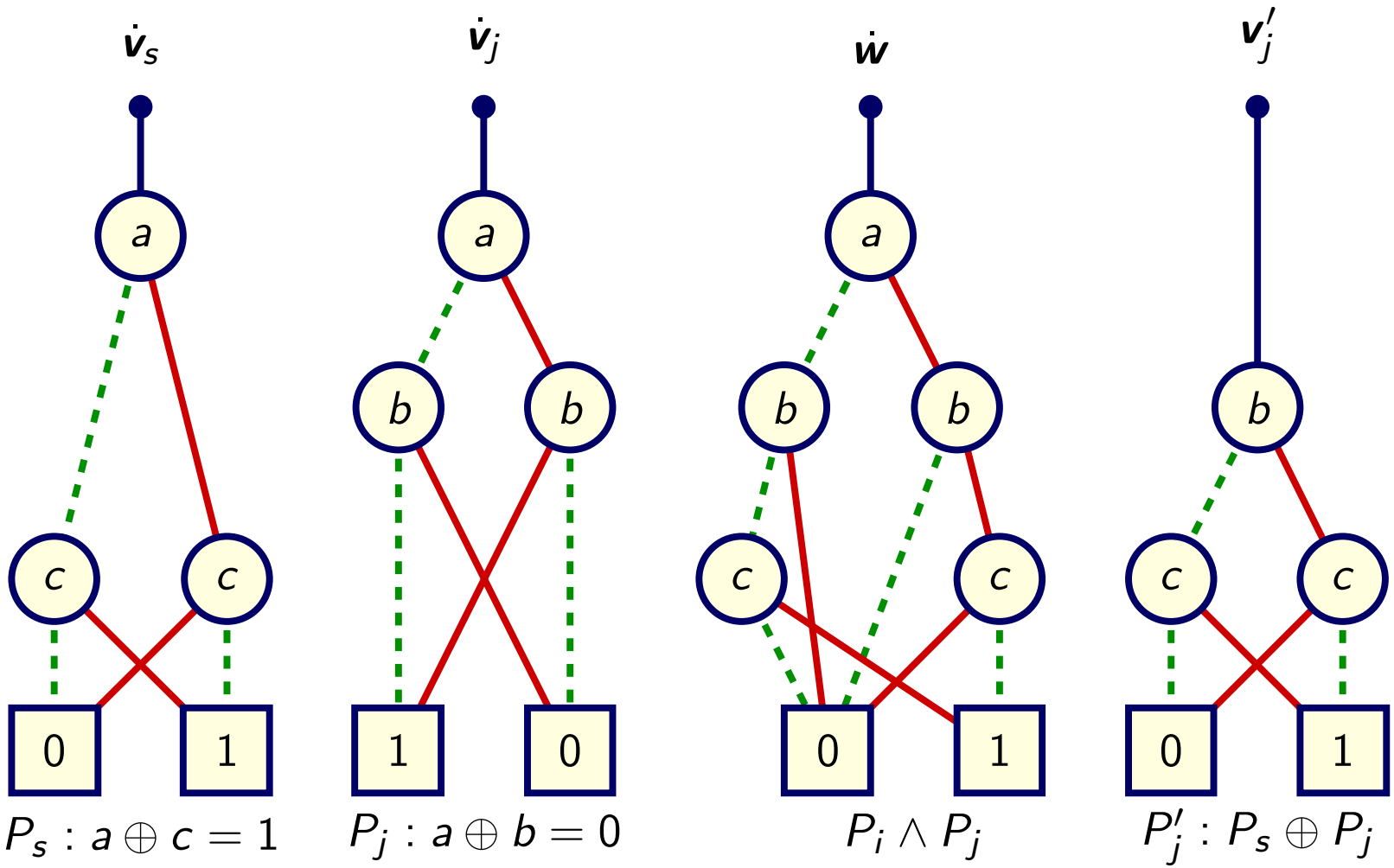
TBDD-Based Parity Reasoning Example

$\boldsymbol{v}_j' \longleftarrow \texttt{bdd\_xnor}(\boldsymbol{v}_s, \boldsymbol{v}_j)$



TBDD-Based Parity Reasoning Example

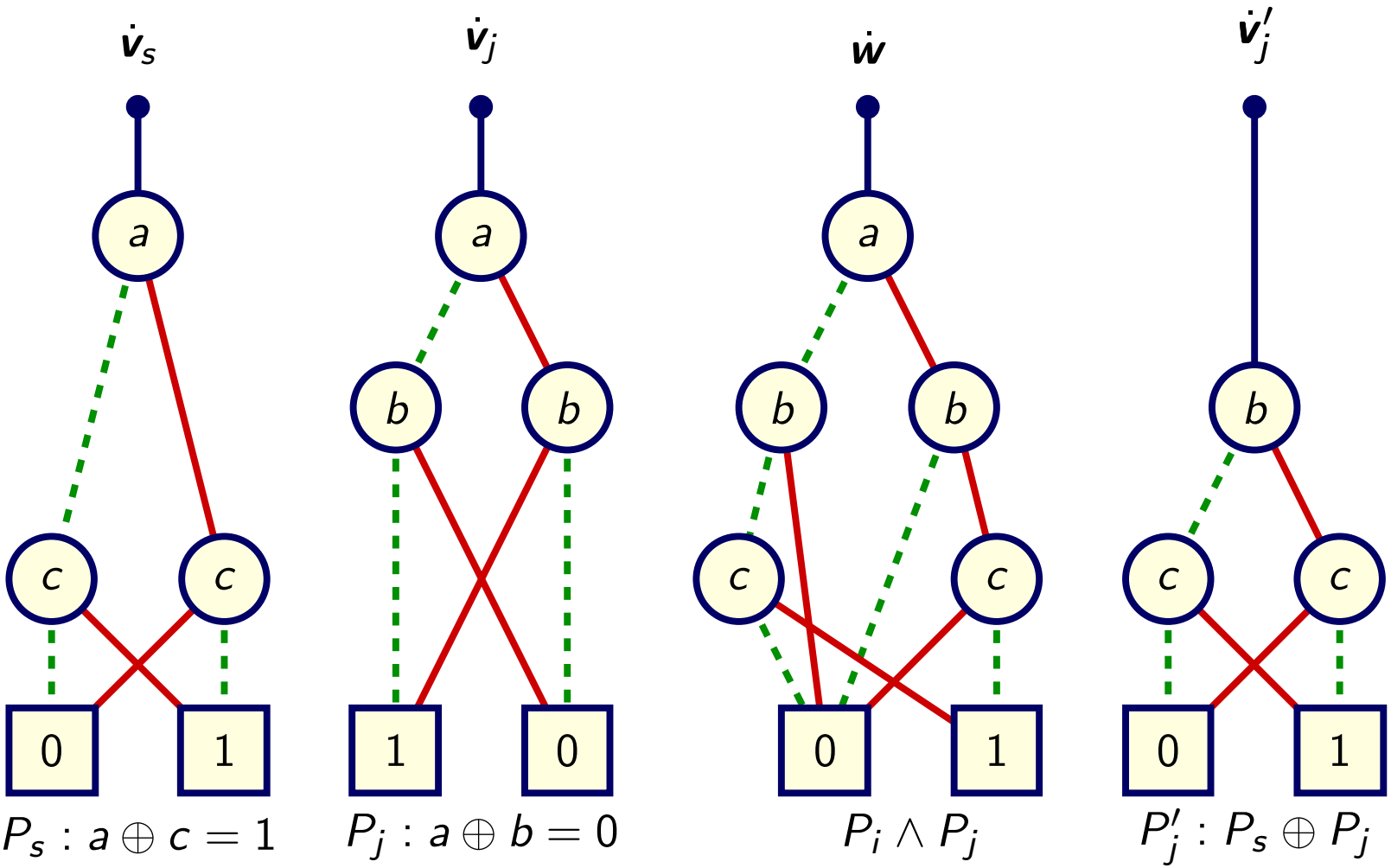
$\dot{w} \longleftarrow$ tbdd_and($\dot{v}_s$, $\dot{v}_j$)



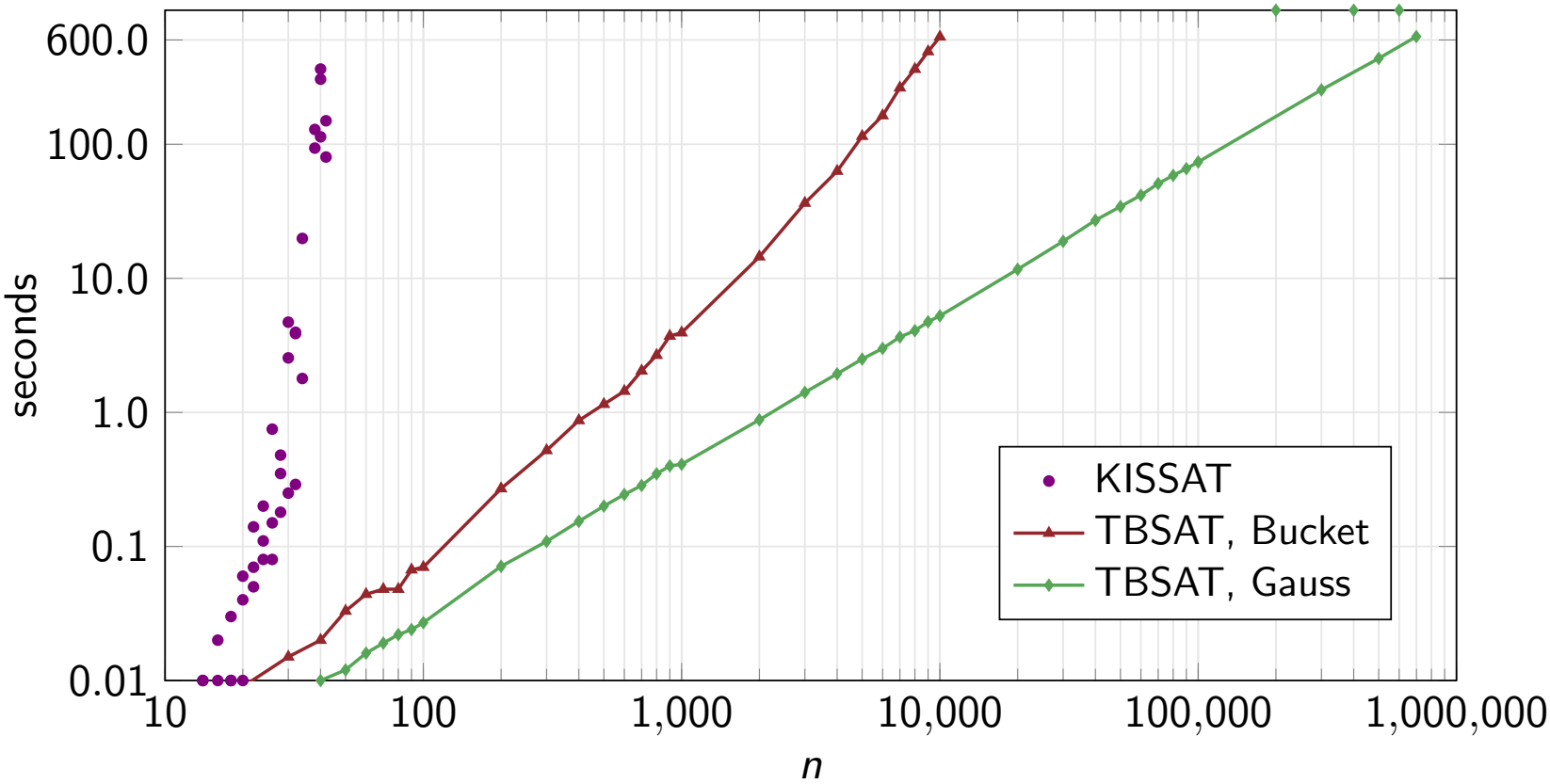
TBDD-Based Parity Reasoning Example

$\dot{w} \longleftarrow$ tbdd_and($\dot{v}_s$, $\dot{v}_j$)

$\dot{v}'_j \longleftarrow$ tbdd_validate(v'_j, $\dot{w}$)

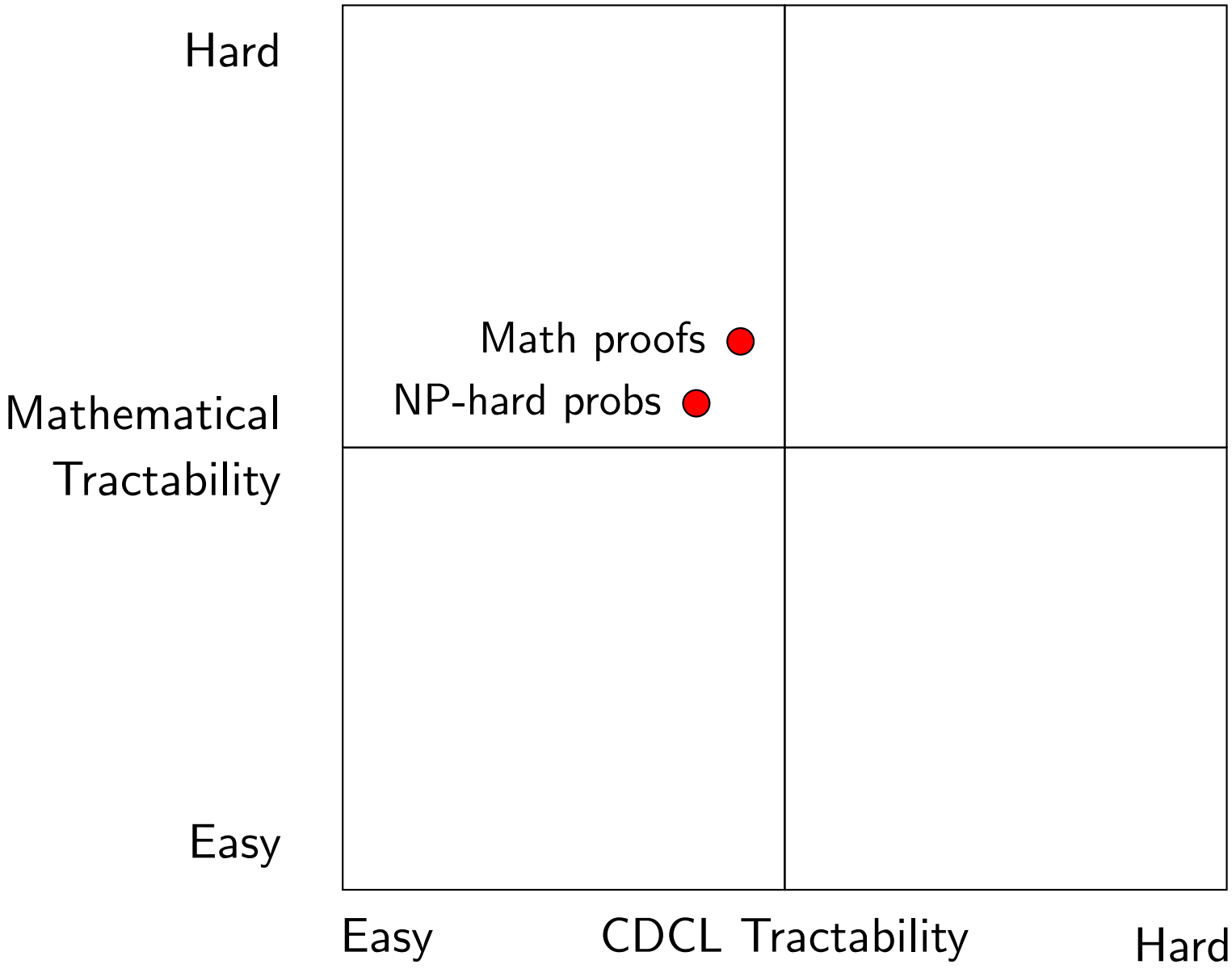


Parity Benchmark Runtime

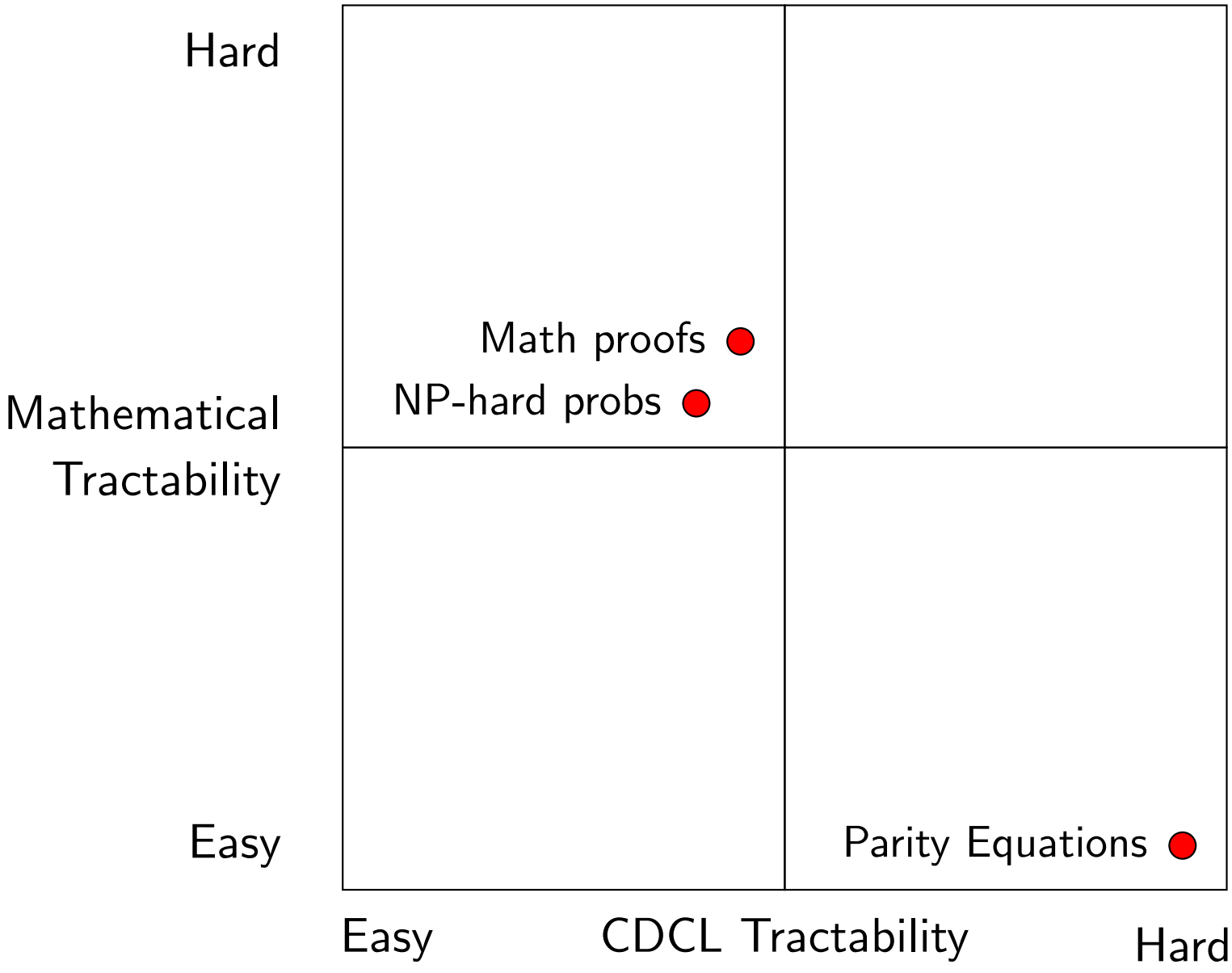


- Upper limit: $n = 699{,}051$
  - BuDDy limited to $2^{21} - 1$ BDD variables
  - CNF file has 2,097,147 variables and 5,592,392 clauses
- Some failures for large values of n due to poor pivot selection

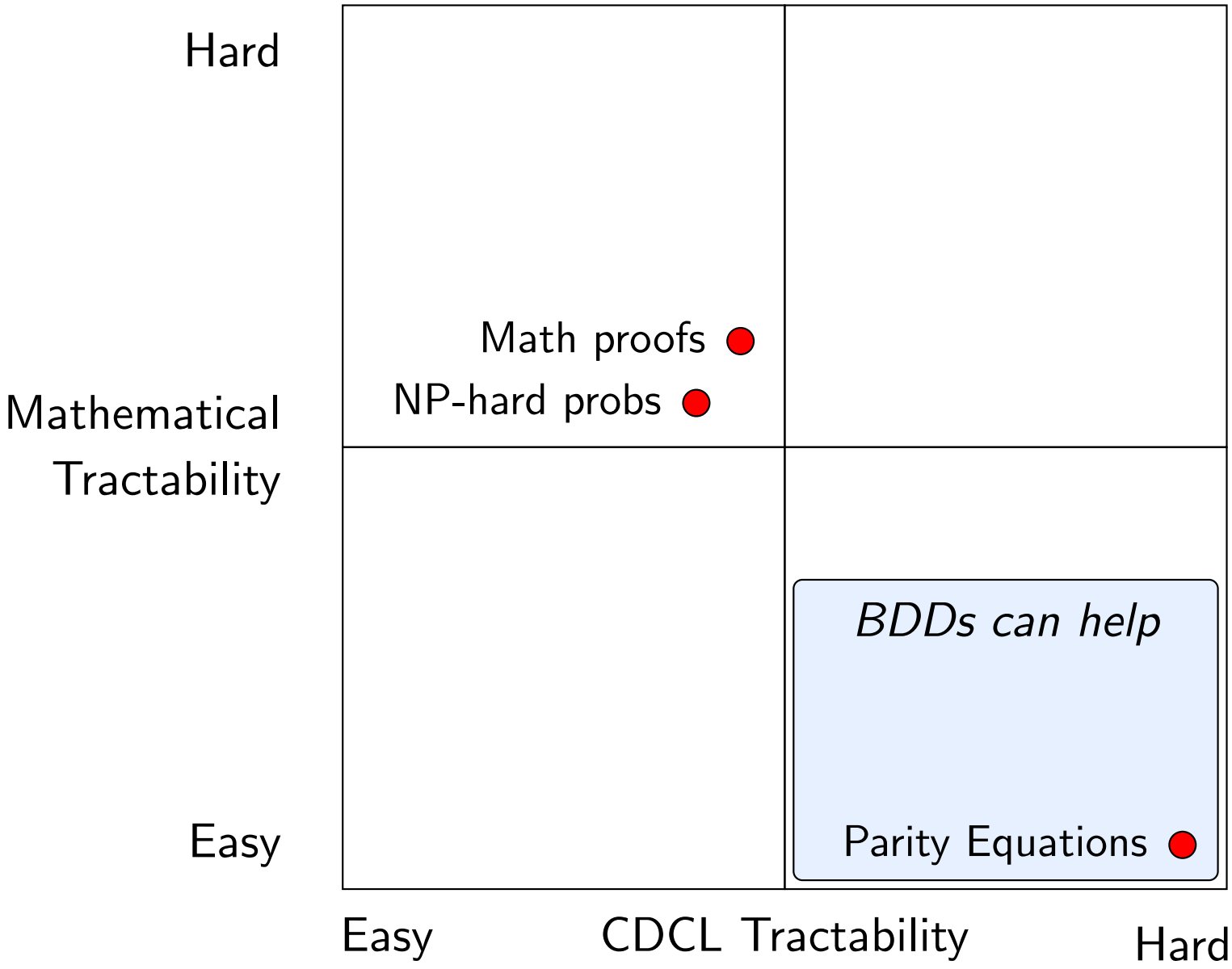
A Perspective on the State of SAT Solving



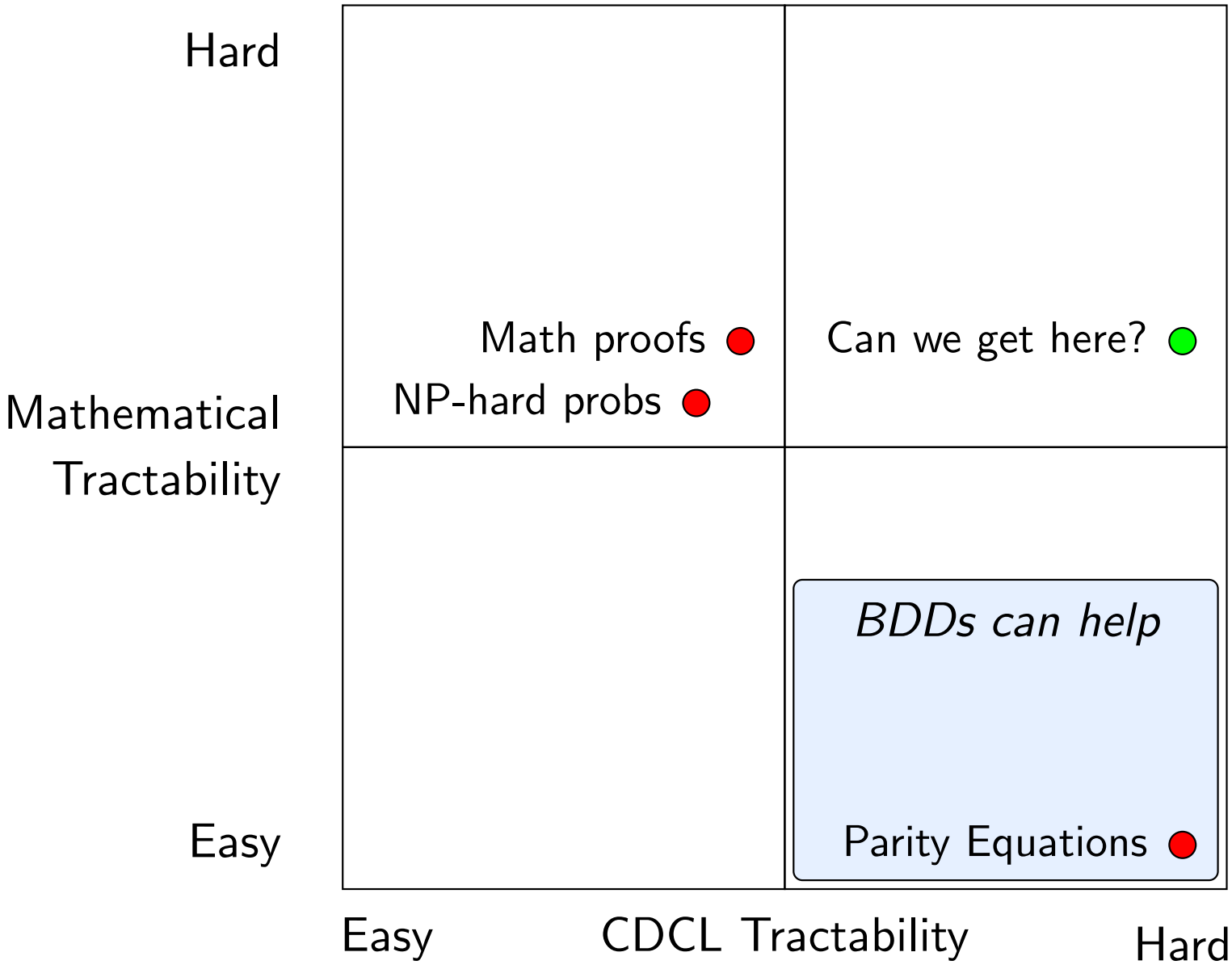
A Perspective on the State of SAT Solving



A Perspective on the State of SAT Solving



A Perspective on the State of SAT Solving



References

BDDs

- R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, 1986
- R. E. Bryant, "Binary Decision Diagrams," *Handbook of Model Checking*, 2018

Proof Generation with BDDs

- R. E. Bryant and M. J. H. Heule, "Generating Extended Resolution Proofs with a BDD-Based SAT Solver," *TACAS*, 2021
- R. E. Bryant and M. J. H. Heule, "Dual Proof Generation for Quantified Boolean Formulas with a BDD-Based Solver," *CADE*, 2021
- R. E. Bryant, A. Biere, and M. J. H. Heule, "Clausal Proofs from Pseudo-Boolean Reasoning," *TACAS*, 2022
- R. E. Bryant, "TBUDDY: A Proof-Generating BDD Package," *FMCAD*, 2022

Supporting other Reasoning Tools

CryptoMiniSAT Solver

- Mate Soos
- State-of-the-art CDCL solver
- Integrated Gauss-Jordan (G-J) solver for parity reasoning
- Solvers exchange unit propagations and conflicts

Previous Limitation

- Could not generate UNSAT proof when G-J enabled

Supporting other Reasoning Tools

CryptoMiniSAT Solver

- Mate Soos
- State-of-the-art CDCL solver
- Integrated Gauss-Jordan (G-J) solver for parity reasoning
- Solvers exchange unit propagations and conflicts

Previous Limitation

- Could not generate UNSAT proof when G-J enabled

Previous Limitation Integrating Tbuddy

- Bryant & Soos, 2022
- Represent parity constraints with TBDDs
- Use TBDD operations to justify unit propagation steps

Possible Areas to Explore

Observations

- BDDs form useful supplement to CDCL
- TBUDDY provides efficient and reliable implementation

Enhancing SAT Solving

- Integrate other forms of reasoning
- Create collaboration between CDCL and BDD engines

Beyond SAT

- Quantified Boolean Formulas (QBF)
- Model counting
- Checking proofs in other proof systems