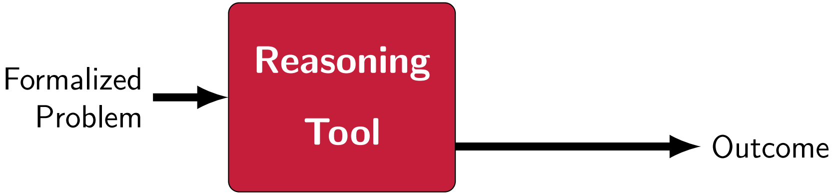
Binary Decision Diagrams Applied to Verifiable Automated Reasoning

Randal E. Bryant

Carnegie Mellon University

2024

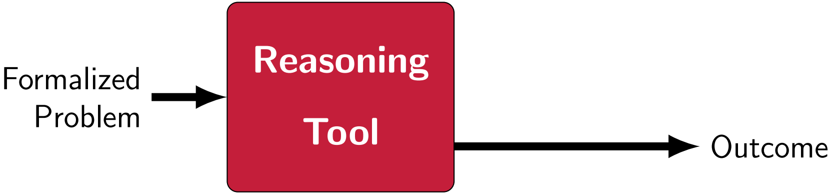
Automated Reasoning Programs



Example Applications

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

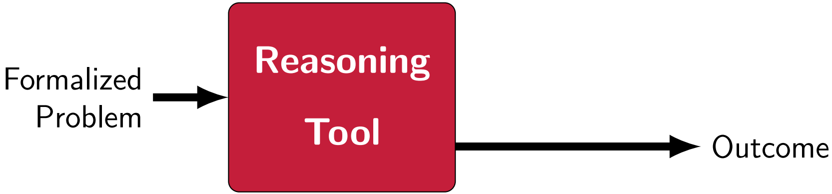
Automated Reasoning Programs



Can We Trust the Results?

- *No!*
- Complex software with many optimizations

Automated Reasoning Programs



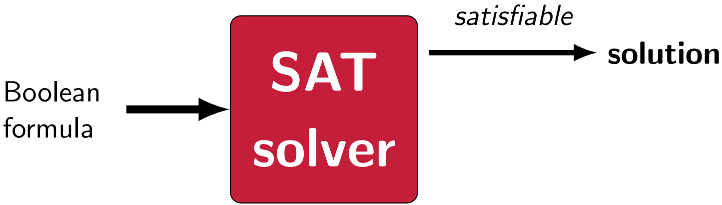
Can We Trust the Results?

- *No!*
- Complex software with many optimizations

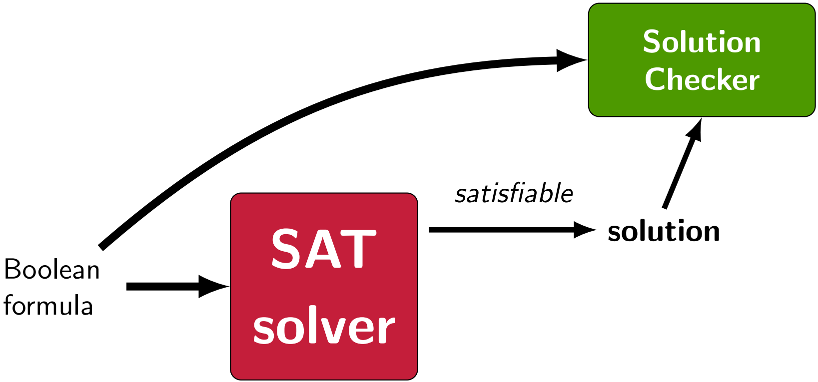
Is This a Problem?

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

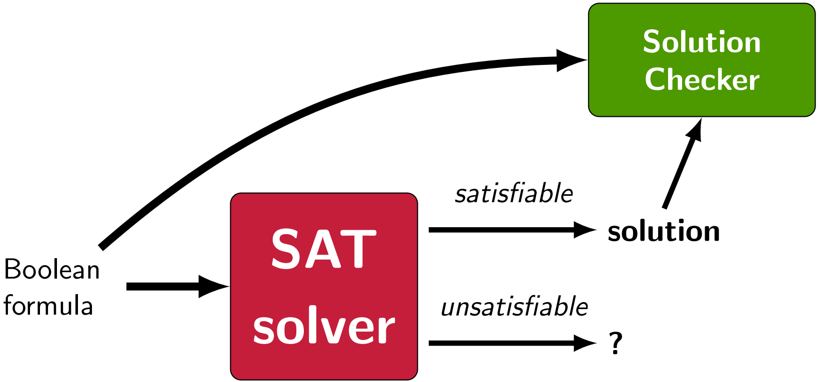
Boolean Satisfiability Solvers



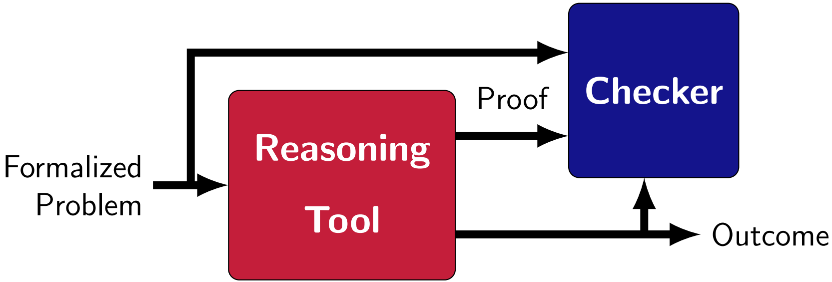
Boolean Satisfiability Solvers



Boolean Satisfiability Solvers



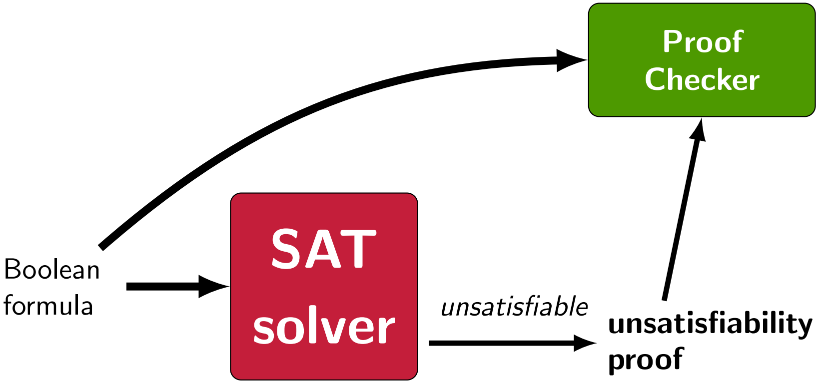
Proof-Generating Automated Reasoning Programs



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

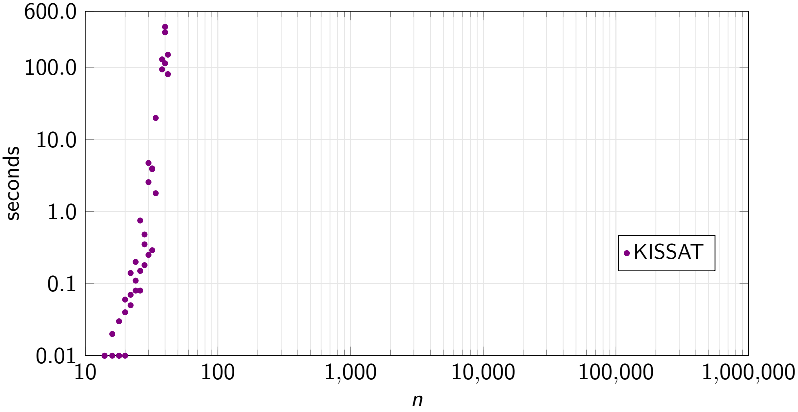
- Chew and Heule, SAT 2020

For n Boolean variables and random permutation π:

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

- Conjunction unsatisfiable

Parity Benchmark Runtime



- KISSAT: State-of-the-art CDCL solver
- 3 different random permutations 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

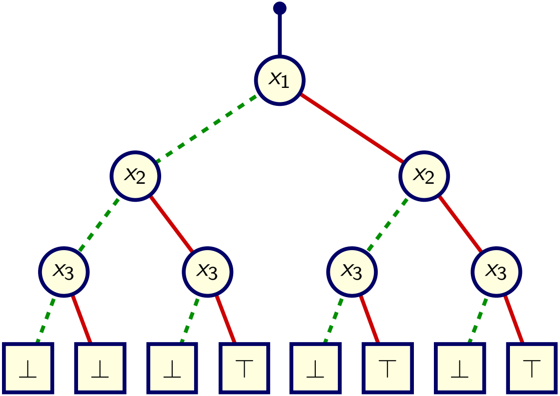
Boolean Function Representations

Truth Table

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

- Size $= O(2^n)$

Decision Tree

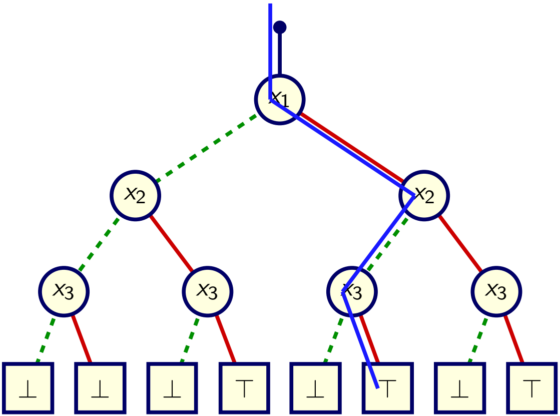


Boolean Function Representations

Truth Table

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

Decision Tree



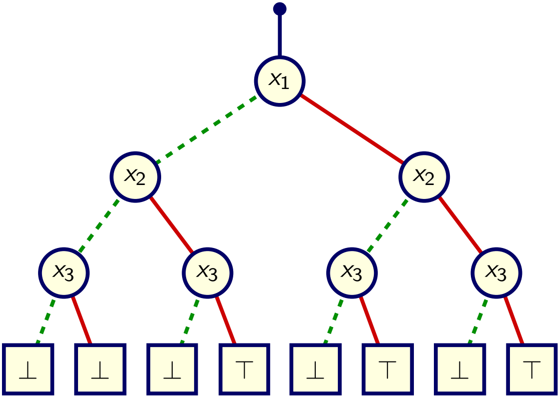
- Size $= O(2^n)$
- Assignment defines path from root to leaf

Reducing to Canonical Form

Truth Table

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

Graph Representation



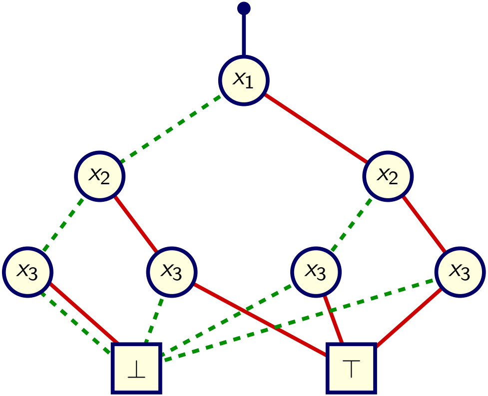
- Merge isomorphic nodes
- Eliminate redundant tests

Reducing to Canonical Form

Truth Table

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

Graph Representation



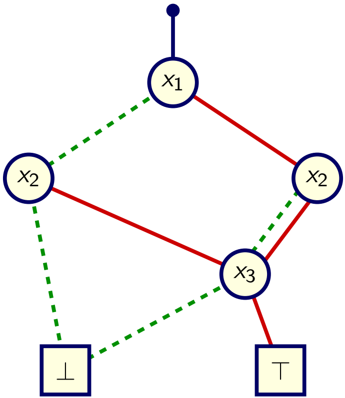
- Merge isomorphic nodes
- Eliminate redundant tests

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



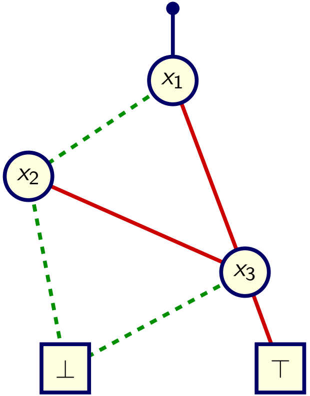
- Merge isomorphic nodes
- Eliminate redundant tests

Reducing to Canonical Form

Truth Table

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

Graph Representation



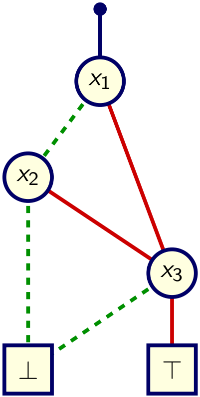
- Merge isomorphic nodes
- Eliminate redundant tests

Canonical Form

Truth Table

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

Reduced Ordered Binary Decision Diagram

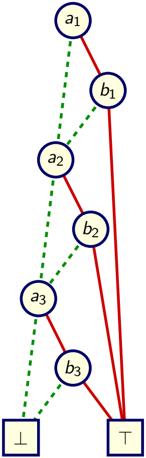


- Canonical representation of Boolean function
- No further simplifications possible

Effect of Variable Ordering

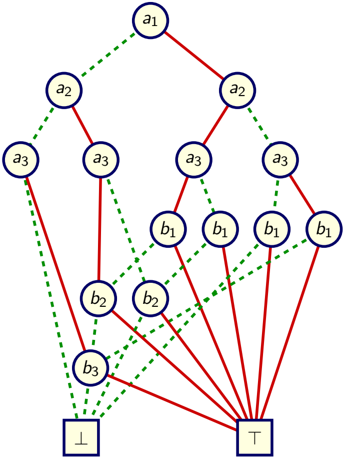
$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$$

Good Ordering



- Linear growth

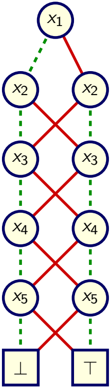
Bad Ordering



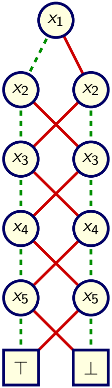
- Exponential growth

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

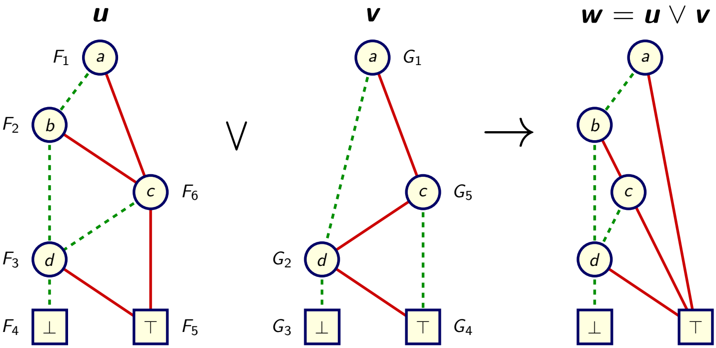
Key Algorithmic Properties

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

Apply Algorithm

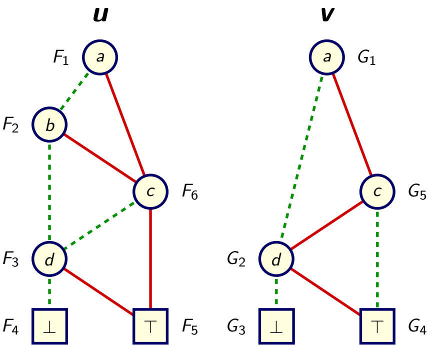
$\boldsymbol{w} \quad \leftarrow \quad \boldsymbol{u} \odot \boldsymbol{v}$

- $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ functions represented as BDDs
- $\odot$ binary Boolean operator
  - E.g., $\wedge$, $\vee$, $\oplus$



Apply Algorithm Recursion

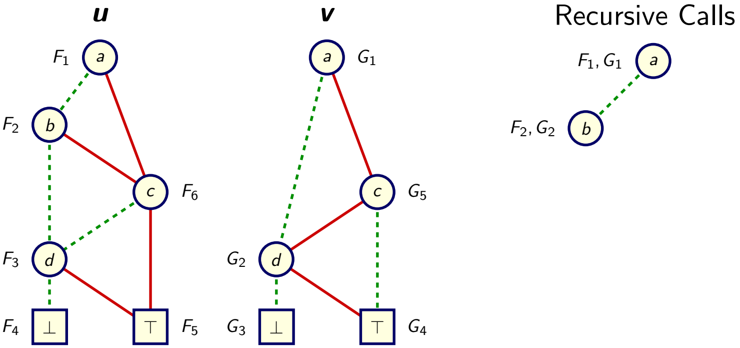
- Recurse through argument graphs
- Stop when hit terminal case
- Save results in cache to reuse when hit same arguments



Recursive Calls

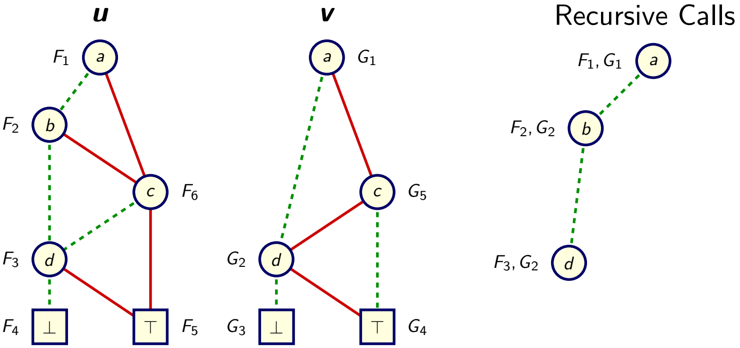
Apply Algorithm Recursion

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



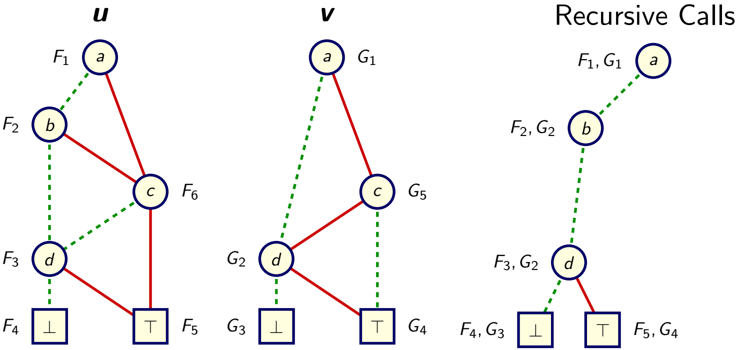
Apply Algorithm Recursion

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



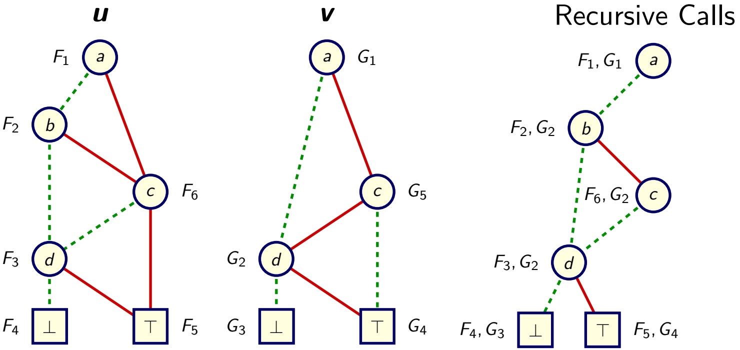
Apply Algorithm Recursion

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



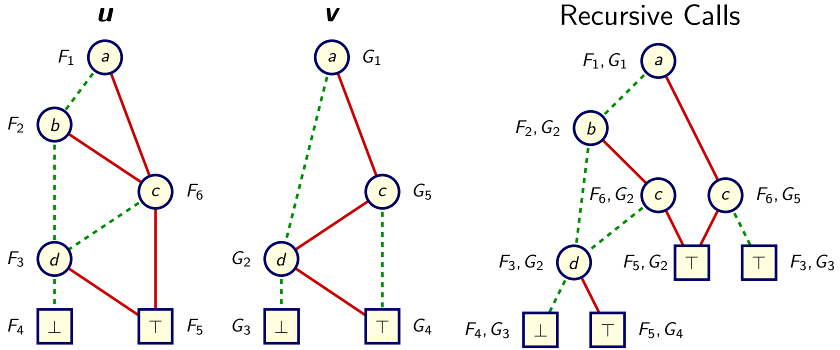
Apply Algorithm Recursion

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



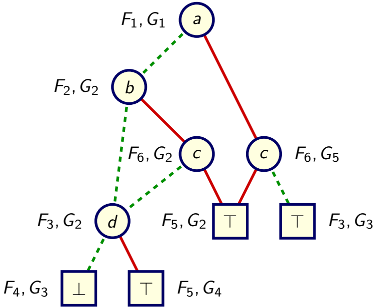
Apply Algorithm Recursion

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

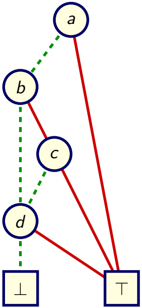


Apply Algorithm Result

Recursive Calls

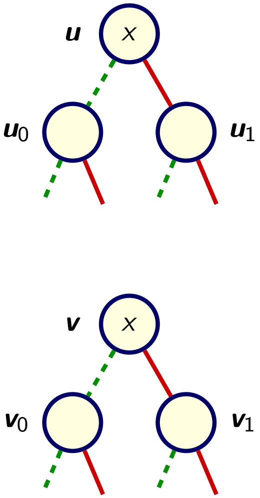


Reduced Result $\boldsymbol{w}$



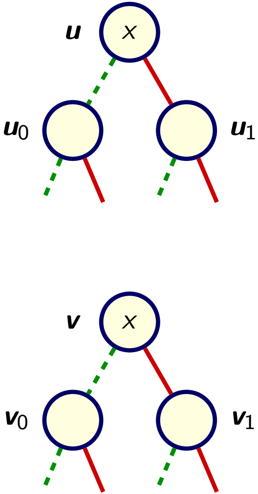
Apply Algorithm Recursion

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



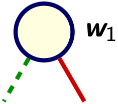
Apply Algorithm Recursion

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

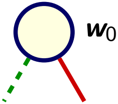


Recursion

$\text{Apply}(\boldsymbol{u}_1, \boldsymbol{v}_1, \wedge) \;\rightarrow$

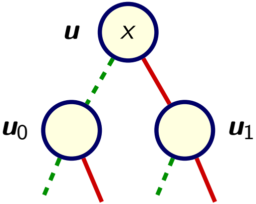


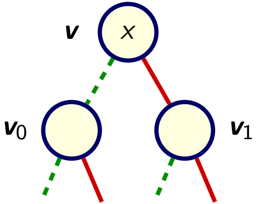
$\text{Apply}(\boldsymbol{u}_0, \boldsymbol{v}_0, \wedge) \;\rightarrow$



Apply Algorithm Recursion

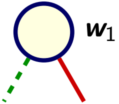
$\text{Apply}(\boldsymbol{u}, \boldsymbol{v}, \wedge)$



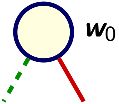


Recursion

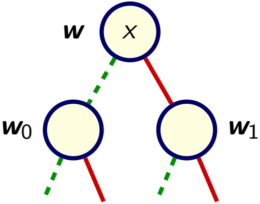
$\text{Apply}(\boldsymbol{u}_1, \boldsymbol{v}_1, \wedge) \;\rightarrow$



$\text{Apply}(\boldsymbol{u}_0, \boldsymbol{v}_0, \wedge) \;\rightarrow$



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
  - $\Rightarrow$ Original formula 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, Biere, Heule, 2022
- TBUDDY: Bryant, 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 $\perp$.

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

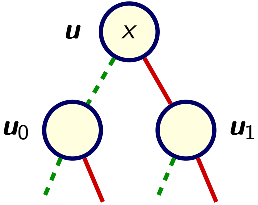
- Implement standalone solver TBSAT
- Incorporate into other solvers

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 $u \leftrightarrow ITE(x, u_1, u_0)$

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($\boldsymbol{u}$)	$\overline{x} \rightarrow (u_0 \rightarrow u)$	$[x \vee \overline{u}_0 \vee u]$

Proof-Generating Apply Operation

Integrate Proof Generation into Apply Operation

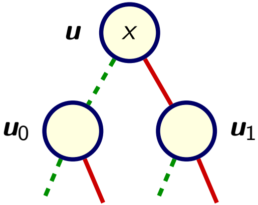
- Apply($\boldsymbol{u}, \boldsymbol{v}, \wedge$) returns $\boldsymbol{w}$
- Also generate proof $u \wedge v \rightarrow w$

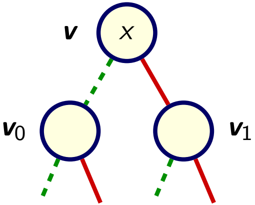
Key Idea:

Proof follows recursion of the Apply *algorithm*

Apply Algorithm Recursion

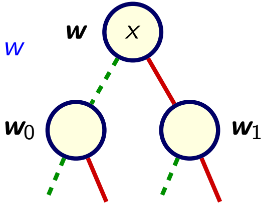
Apply($\boldsymbol{u}, \boldsymbol{v}, \wedge$)





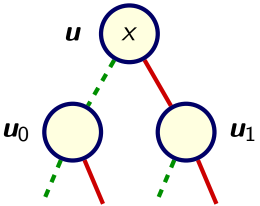
Result

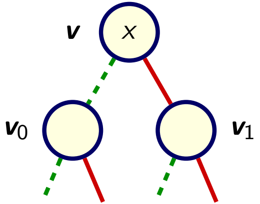
$u \wedge v \rightarrow w$



Apply Algorithm Recursion

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

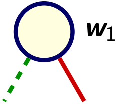




Recursion

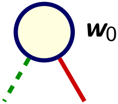
$\text{Apply}(\boldsymbol{u}_1, \boldsymbol{v}_1, \wedge) \rightarrow$

$u_1 \wedge v_1 \rightarrow w_1$

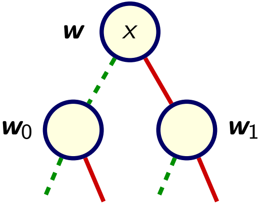


$\text{Apply}(\boldsymbol{u}_0, \boldsymbol{v}_0, \wedge) \rightarrow$

$u_0 \wedge v_0 \rightarrow w_0$



Result



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

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

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

- EQuant($\boldsymbol{u}, x$) $\rightarrow \boldsymbol{v}$
- Separately run ProveImplication($\boldsymbol{u}, \boldsymbol{v}$)
  - Generates proof $u \rightarrow v$
  - Algorithm similar to proof-generating Apply operation

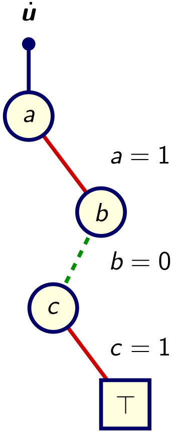
Trusted BDDs (TBDDs)

Components of TBDD $\dot{\boldsymbol{u}}$

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

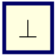
Interpretation. For input formula ϕ:

- $\phi \models u$
- Any variable assignment that satisfies ϕ traces path in BDD from $\boldsymbol{u}$ to leaf node $\top$



Terminal Case

- $\dot{\boldsymbol{u}} = \dot{\bot}$
- ϕ has no satisfying assignments



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` $\dot{\boldsymbol{v}}$`);`

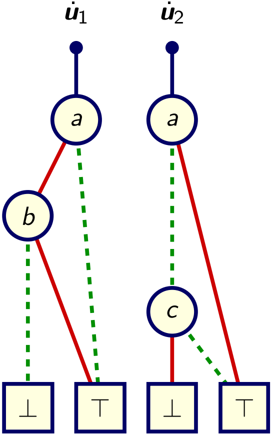
- Form conjunction $\dot{\boldsymbol{w}}$ of TBDDs $\dot{\boldsymbol{u}}$ and $\dot{\boldsymbol{v}}$.
  - Apply operation generates proof $u \wedge v \rightarrow w$
  - Resolution with unit clauses $[u]$ and $[v]$ yields unit clause $[w]$

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

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

TBDD Execution Example

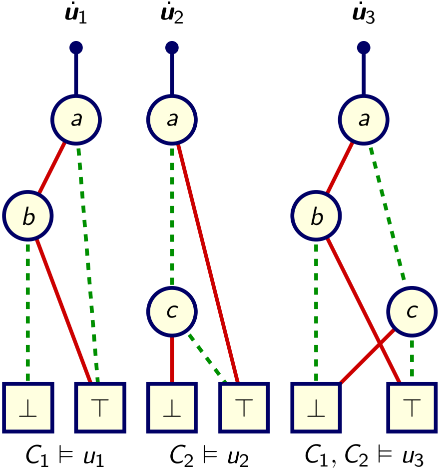
$\dot{\boldsymbol{u}}_1 \longleftarrow$ `tbdd_from_clause`(C_1)
$\dot{\boldsymbol{u}}_2 \longleftarrow$ `tbdd_from_clause`(C_2)



$C_1 \models u_1$ $C_2 \models u_2$

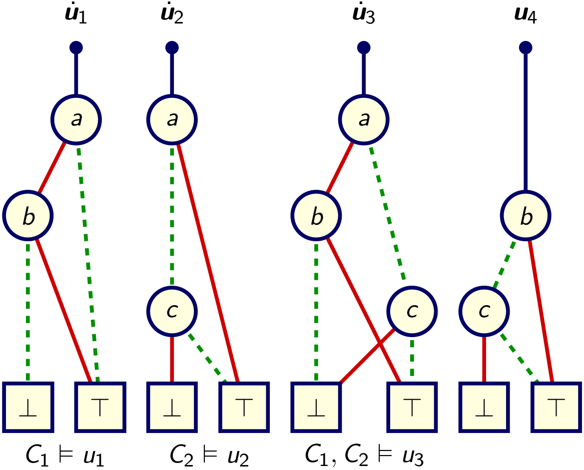
TBDD Execution Example

$\dot{u}_3 \longleftarrow \texttt{tbdd\_and}(\dot{u}_1,\ \dot{u}_2)$



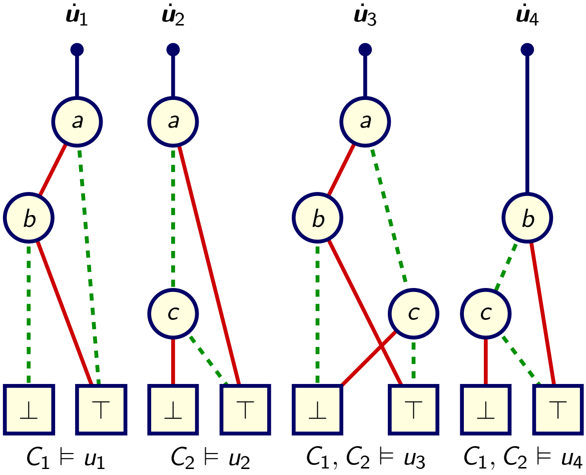
TBDD Execution Example

$\boldsymbol{u}_4 \longleftarrow \texttt{bdd\_exists}(\boldsymbol{u}_3, a)$



TBDD Execution Example

$$\boldsymbol{u}_4 \longleftarrow \texttt{bdd\_exists}(\boldsymbol{u}_3,\, a)$$
$$\dot{\boldsymbol{u}}_4 \longleftarrow \texttt{tbdd\_validate}(\boldsymbol{u}_4,\, \dot{\boldsymbol{u}}_3)$$



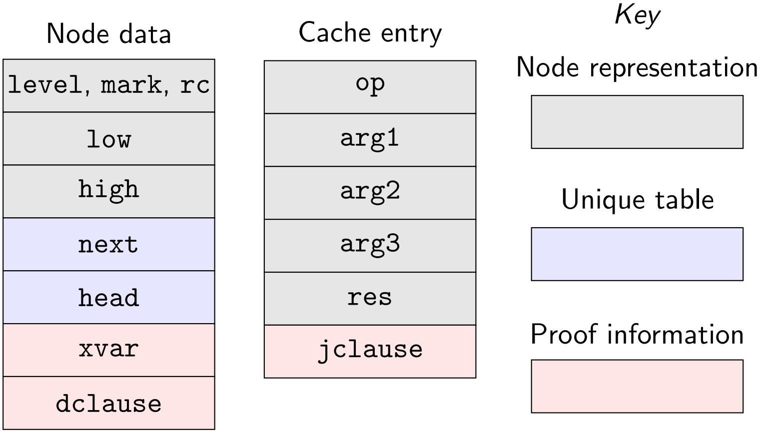
BuDDy BDD Package

BuDDy: Binary Decision Diagram package
Release 2.2

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

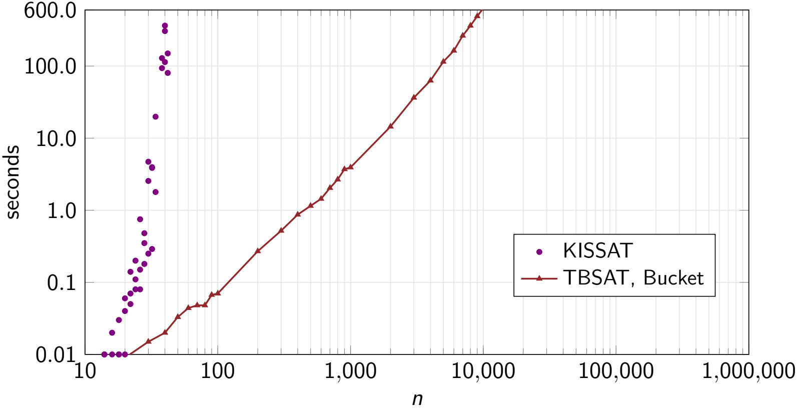
- ~12K lines of code
- Clean, robust, and well documented
- Benchmark comparisons demonstrate good performance
- Node identified by 32-bit index into table
  - Rather than as 64-bit pointer

TBUDDY Data Structures



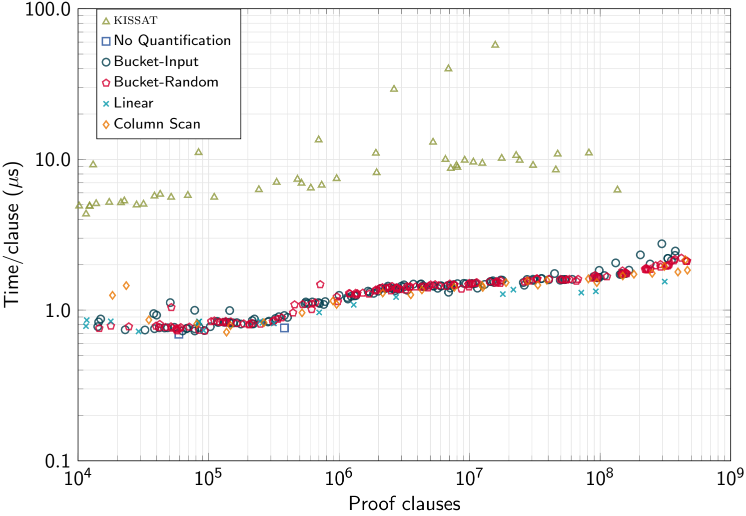
- Node entries: 20 bytes $\rightarrow$ 28 to store proof information
- Cache entry: Existing 24 bytes can also hold proof information
- Total memory overhead $1.35\times$

Parity Benchmark Runtime



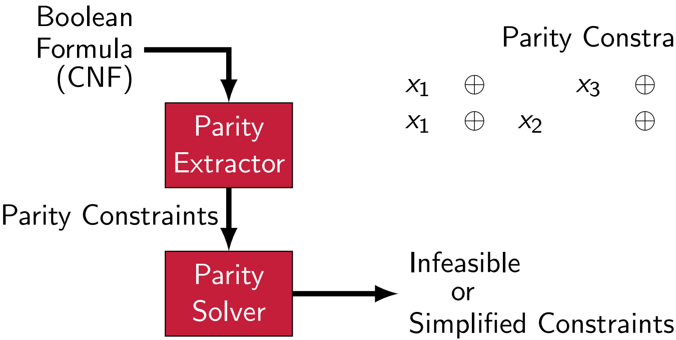
- Bucket elimination
  - Systematic way to perform conjunctions and quantifications
- Random variable ordering
- No guidance from user

CDCL Proofs vs. BDD Proofs



- CDCL proof step indicates reduction in search space
- BDD proof steps justify algorithmic steps

Exploiting Parity Reasoning

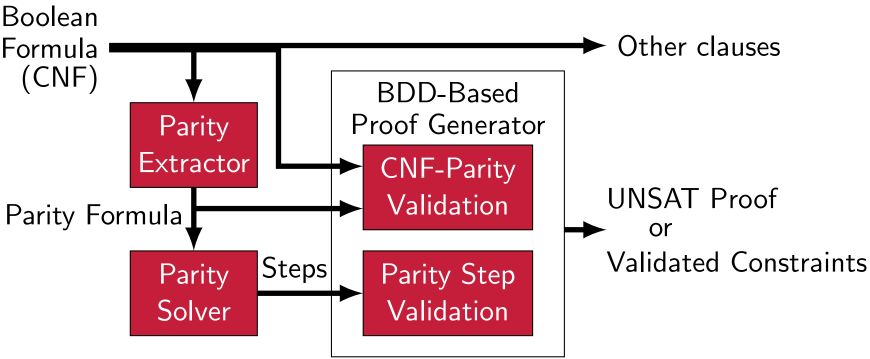


Parity Constraints

$$
\begin{array}{ccccccccc}
x_1 & \oplus & & & x_3 & \oplus & x_4 & = & 0 \\
x_1 & \oplus & x_2 & & & \oplus & x_4 & = & 1
\end{array}
$$

- View parity constraints as system of linear equations
- Reduce via Gaussian elimination
  - If get constraint $0 = 1$, then infeasible
  - Otherwise, can easily generate solutions
- **Challenge:** Certifying results

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

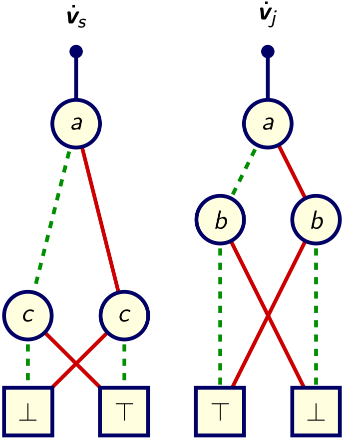
- For pivot P_s, replace each constraint P_j by $P'_j = P_j \oplus P_s$

	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

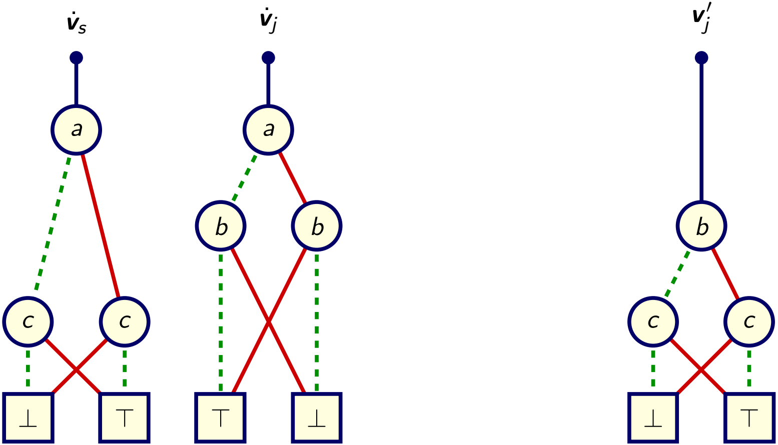
Goal: Compute $P'_j \longleftarrow P_s \oplus P_j$



$P_s : a \oplus c = 1$ $P_j : a \oplus b = 0$

TBDD-Based Parity Reasoning Example

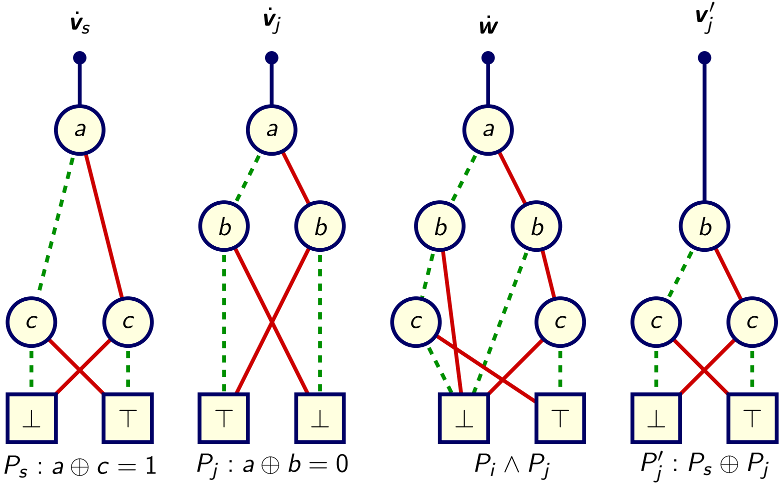
$\boldsymbol{v}_j' \longleftarrow \texttt{bdd\_xnor}(\boldsymbol{v}_s,\ \boldsymbol{v}_j)$



$P_s : a \oplus c = 1$ $P_j : a \oplus b = 0$ $P_j' : P_s \oplus P_j$

TBDD-Based Parity Reasoning Example

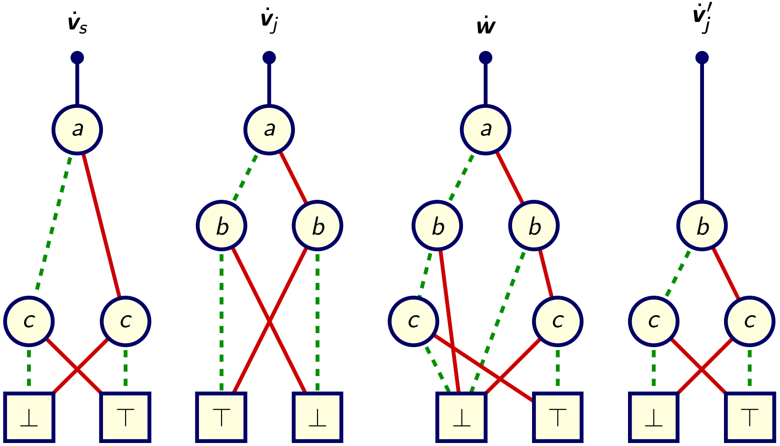
$\dot{\boldsymbol{w}} \longleftarrow \texttt{tbdd\_and}(\dot{\boldsymbol{v}}_s,\ \dot{\boldsymbol{v}}_j)$



TBDD-Based Parity Reasoning Example

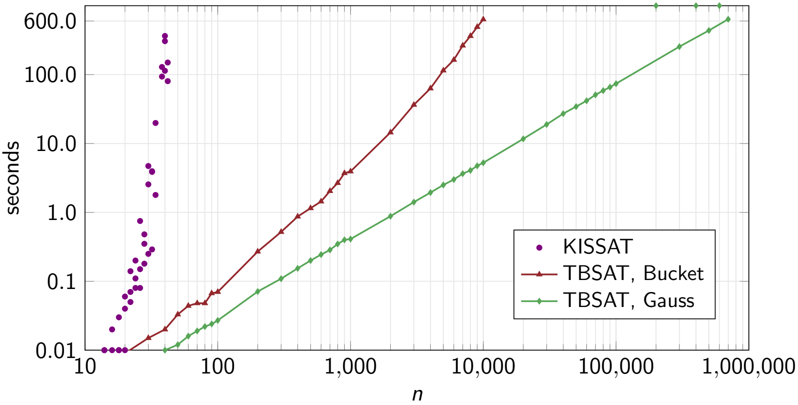
$\dot{\boldsymbol{w}} \longleftarrow \texttt{tbdd\_and}(\dot{\boldsymbol{v}}_s,\ \dot{\boldsymbol{v}}_j)$

$\dot{\boldsymbol{v}}'_j \longleftarrow \texttt{tbdd\_validate}(\boldsymbol{v}'_j,\ \dot{\boldsymbol{w}})$



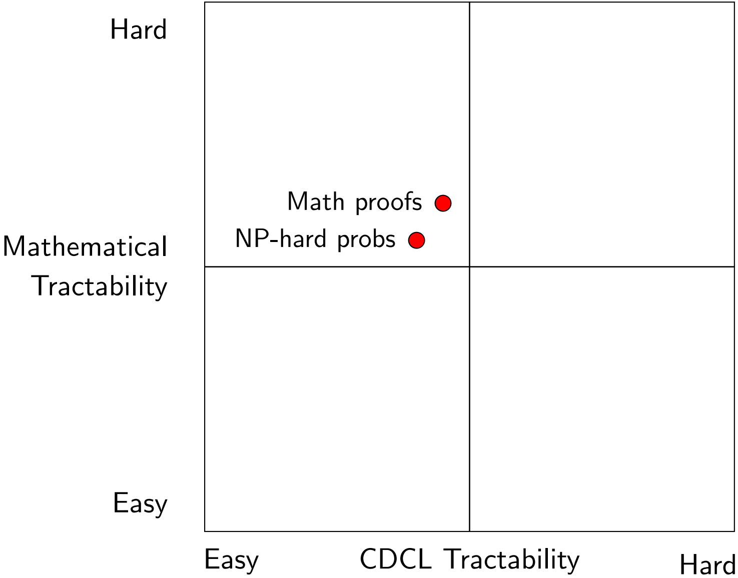
$P_s : a \oplus c = 1$ $\quad P_j : a \oplus b = 0$ $\quad P_i \wedge P_j$ $\quad P'_j : P_s \oplus P_j$

Parity Benchmark Runtime

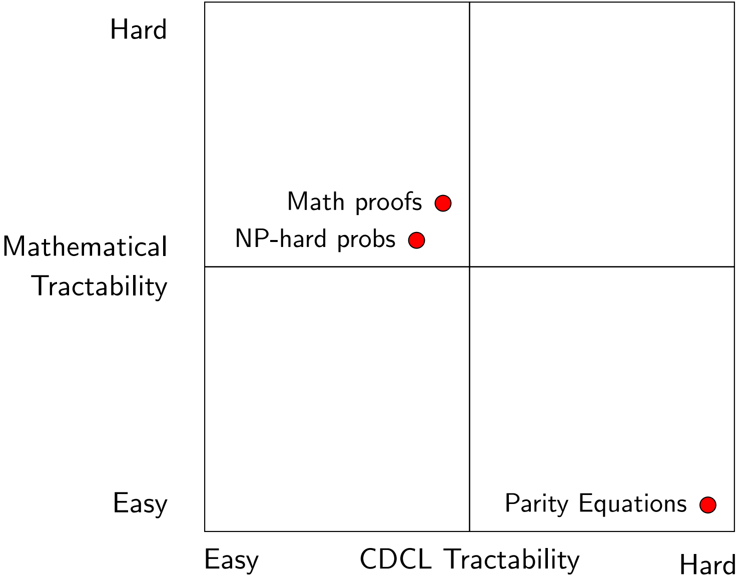


- Upper limit: $n = 699{,}051$
  - BuDDy limited to $2^{21} - 1$ BDD variables
  - CNF file has 2,097,147 variables and 5,592,392 clauses
- Some failures for large values of n due to poor pivot selection

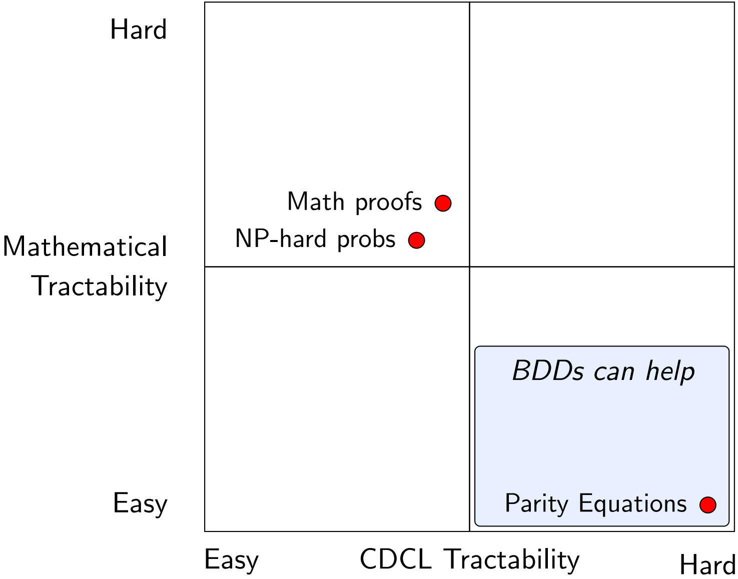
A Perspective on the State of SAT Solving



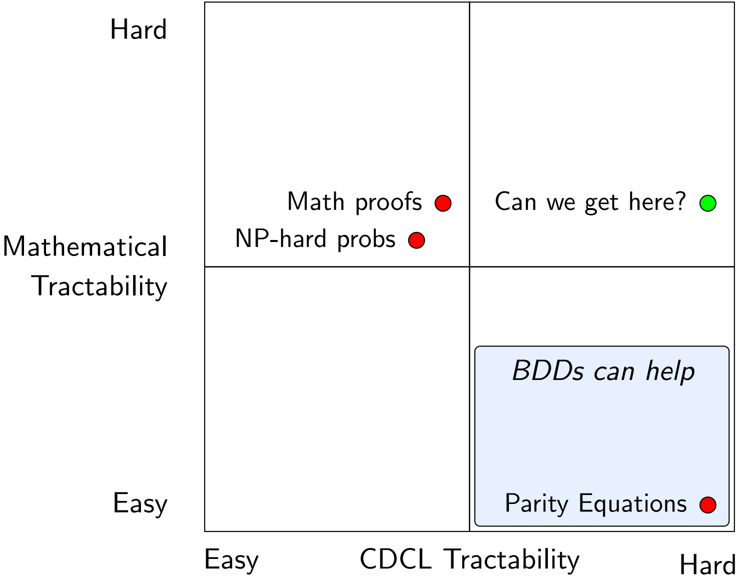
A Perspective on the State of SAT Solving



A Perspective on the State of SAT Solving



A Perspective on the State of SAT Solving

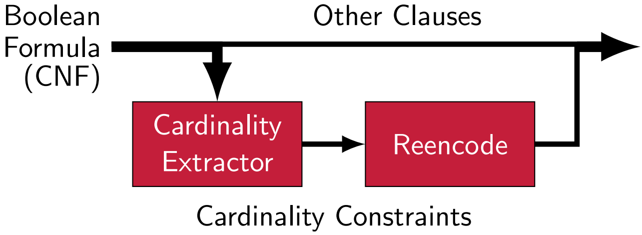


Exploiting Cardinality Constraints

Reeves, Heule, and Bryant, 2024

- *At-most-one*: Constrain set of variables to be one-hot
  $$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} \quad = \quad 1$$
- *Optimization*: Place bound on resource constraint
  $$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} \quad \leq \quad 5$$
- Best encoding depends on many factors
  - Relation to other clauses in formula
  - Whether formula is satisfiable
- In practice, many formulas have suboptimal encodings

Exploiting Cardinality Constraints



- Detect cardinality constraints in formula
- Reencode into better form
  - More sophisticated encodings
  - Possibly alter dynamically during solver operation

Detecting Cardinality Constraints

Clauses

```
1 -11 0          6 19 -20 0       6 27 -28 0       11 -16 0        21 -26 0
2 11 -12 0       7 20 0           7 28 -29 0       12 -17 0        22 -27 0
3 12 -13 0       3 -21 0          8 29 -30 0       13 -18 0        23 -28 0
4 13 -14 0       4 21 -22 0       9 30 0           14 -19 0        24 -29 0
5 14 -15 0       5 22 -23 0       5 -31 0          15 -20 0        25 -30 0
6 15 0           6 23 -24 0       6 31 -32 0       16 -21 0        26 -31 0
2 -16 0          7 24 -25 0       7 32 -33 0       17 -22 0        27 -32 0
3 16 -17 0       8 25 0           8 33 -34 0       18 -23 0        28 -33 0
4 17 -18 0       4 -26 0          9 34 -35 0       19 -24 0        29 -34 0
5 18 -19 0       5 26 -27 0       10 35 0          20 -25 0        30 -35 0
```

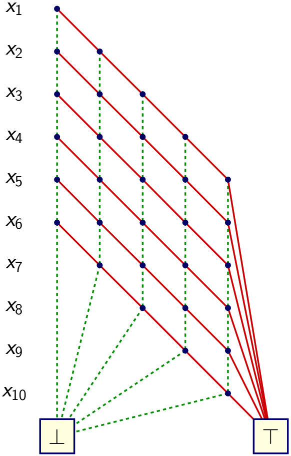
Constraint

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} \quad \geq \quad 5$$

- Often use additional *encoding* variables
- Many possible encoding methods
- Single formula can contain many constraints + other clauses

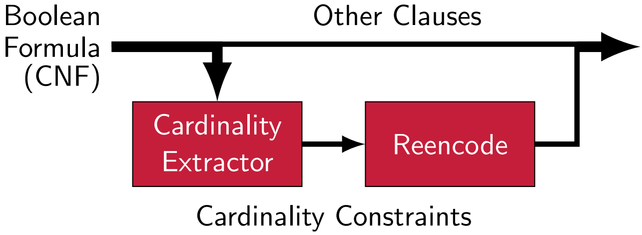
BDD Representation of Cardinality Constraints

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} \quad \geq \quad 5$$



- $O(n^2)$ nodes
  - Independent of variable ordering
- From BDD, easy to determine parameters of constraint
  - Size bounds
  - Phases of variables

Cardinality Constraint Extraction



Guess-and-Verify

- Detect patterns of data and encoding variables that resemble cardinality constraints
- Build BDD by forming conjunction of clauses and quantifying encoding variables
- Detect whether BDD represents valid constraint

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
- J. R. Reeves, M. J. H. Heule, and R. E. Bryant "From Clauses to Klauses," *CAV*, 2024

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

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