Trustworthy Boolean Reasoning 2: BDDs and SAT

Randal E. Bryant

Carnegie Mellon University

June, 2022

Important Ideas for These Lectures

- SAT solvers are useful tools
  - Many practical problems reducible to SAT
  - Need to learn effective encoding techniques
- For many applications, formulas should be unsatisfiable
  - Program should generate a checkable proof
  - There is a well-developed proof infrastructure
- **Binary Decision Diagrams (BDDs) can play important role**
  - **In supplementing current SAT algorithms**
  - **In proof generation**

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, Model Checking, ...

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

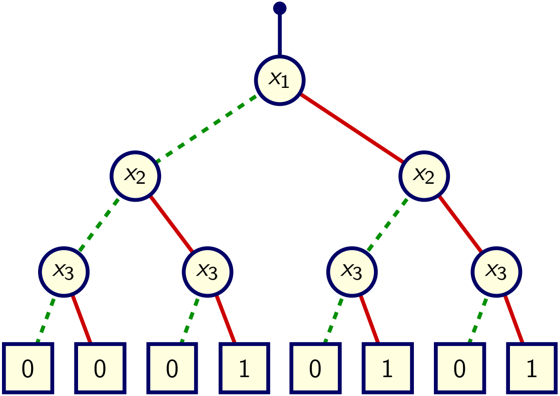
Boolean Function Representations

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- Size $= O(2^n)$

Decision Tree



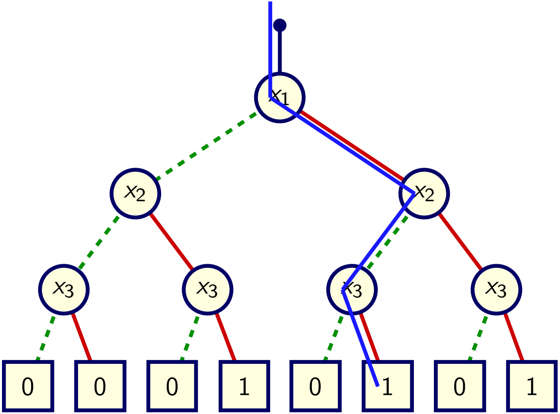
Boolean Function Representations

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- Size $= O(2^n)$

Decision Tree

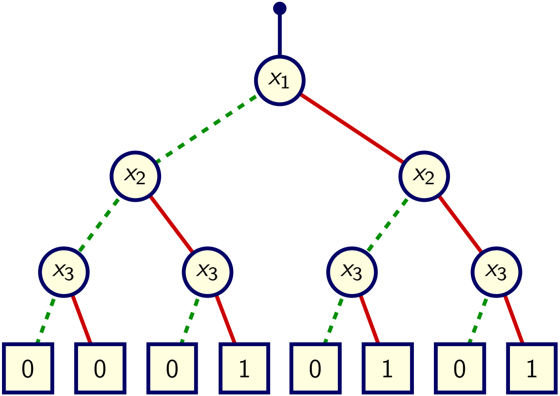


Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Graph Representation



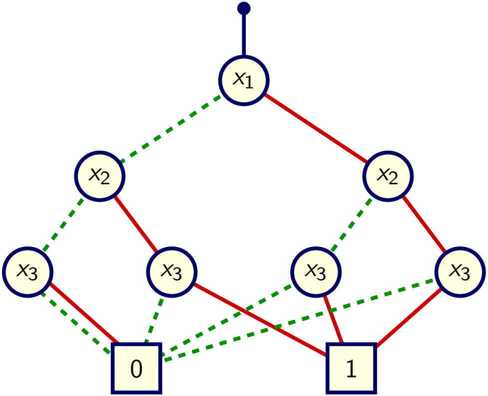
- Merge isomorphic nodes
- Eliminate redundant tests

Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Graph Representation



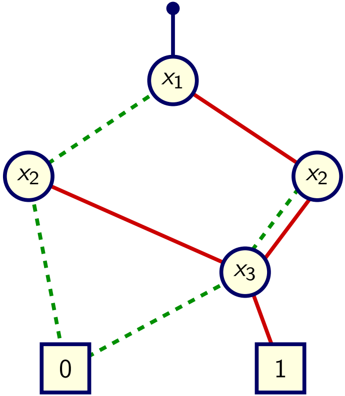
- Merge isomorphic nodes
- Eliminate redundant tests

Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Graph Representation



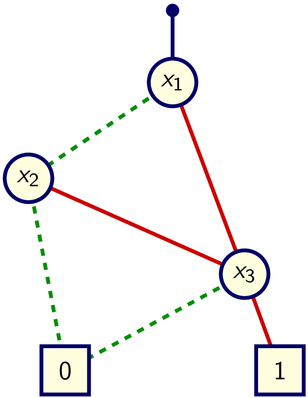
- Merge isomorphic nodes
- Eliminate redundant tests

Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Graph Representation



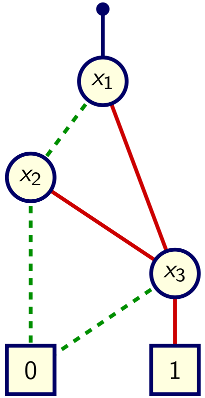
- Merge isomorphic nodes
- Eliminate redundant tests

Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Reduced Ordered Binary Decision Diagram



- Canonical representation of Boolean function
- No further simplifications possible

BDD Representation of Unsatisfiable Formula

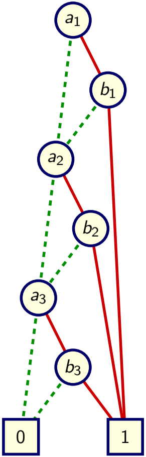
$$\boxed{0}$$

- Refer to this as $\perp$
- Unique
- Converting from CNF to BDD may require exponential number of steps

Effect of Variable Ordering

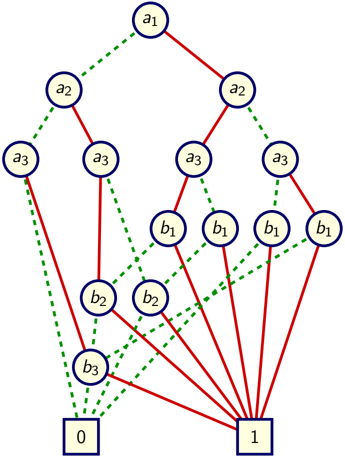
$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$$

Good Ordering



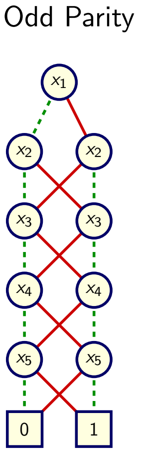
- Linear growth

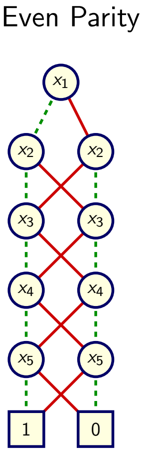
Bad Ordering



- Exponential growth

BDD Representation of Parity Constraints





- 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

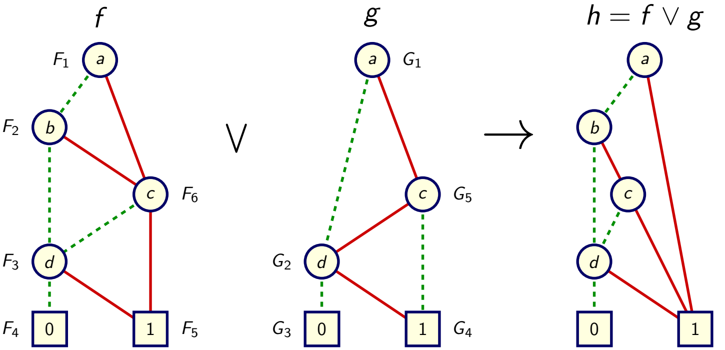
Key Algorithmic Properties

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

Apply Algorithm

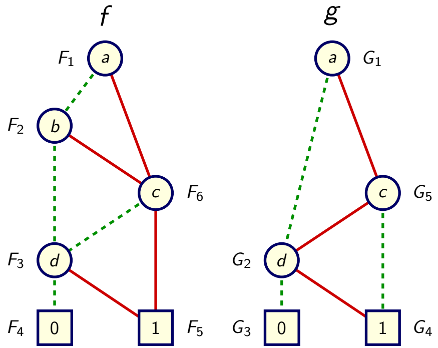
$$h \quad \leftarrow \quad f \odot g$$

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



Apply Algorithm Recursion

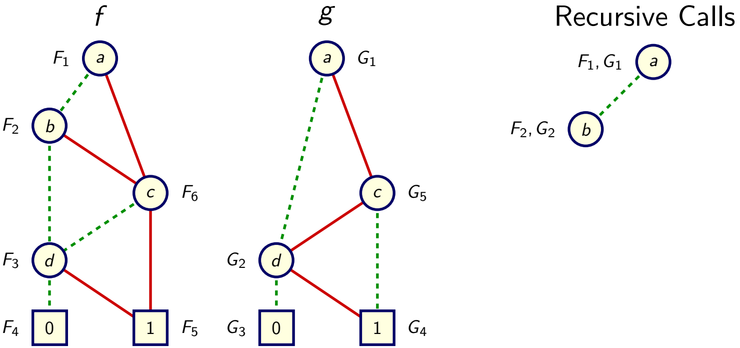
- Recurse through argument graphs
- Stop when hit terminal case
- Save results in cache to reuse when hit same arguments



Recursive Calls

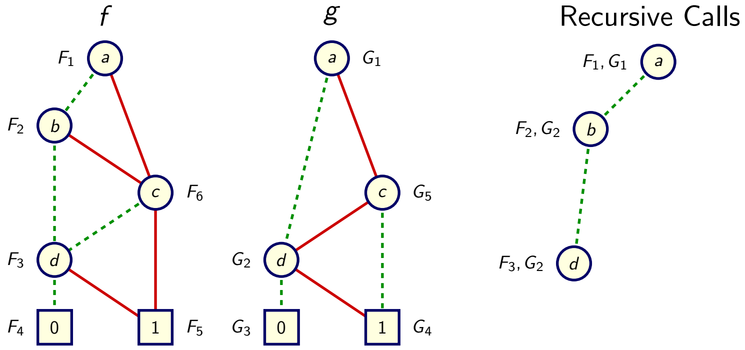
Apply Algorithm Recursion

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



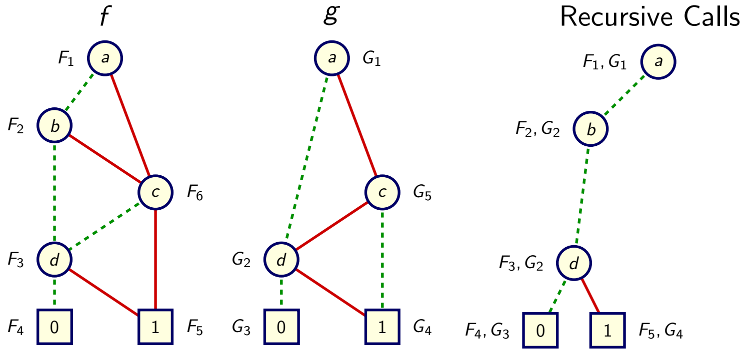
Apply Algorithm Recursion

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



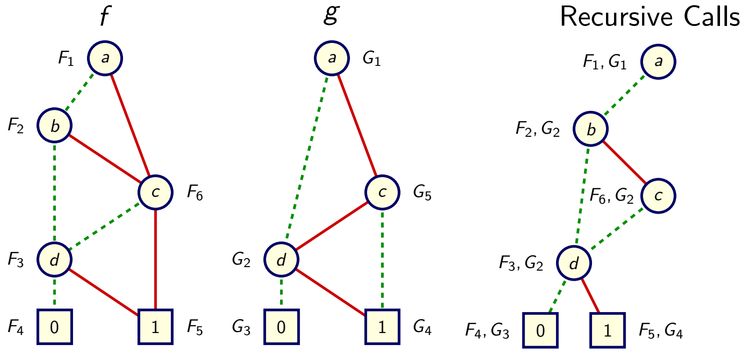
Apply Algorithm Recursion

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



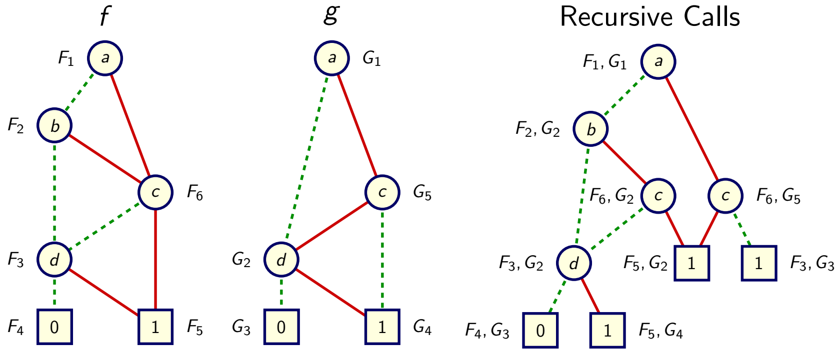
Apply Algorithm Recursion

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



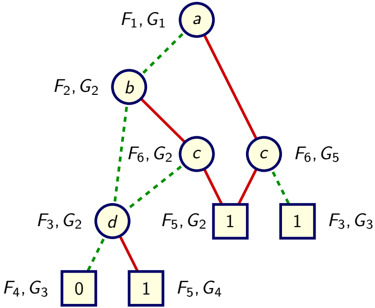
Apply Algorithm Recursion

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

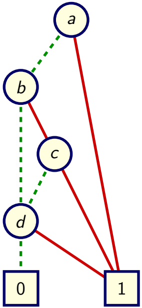


Apply Algorithm Result

Recursive Calls



Reduced Result



BDD-Based SAT Solving: Direct Evaluation

Algorithm

1. Compute BDD t_i for each input clause C_i
2. Form conjunction

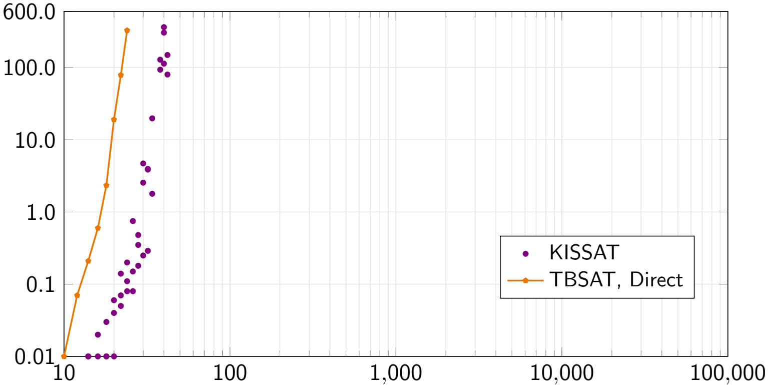
$$s \quad = \quad \bigwedge_{1 \leq i \leq m} t_i$$

   - E.g., with linear or tree evaluation
3. Return UNSAT ($s = \bot$) or SAT ($s \neq \bot$)

Practicality

- Only for small problems
- Resulting BDD s represents *all* solutions

Parity Benchmark Runtime



- TBSAT: BDD-Based SAT Solver
- In direct mode, even worse than KISSAT
- Limited to $n \leq 24$ within 600 seconds

BDD-Based SAT Solving: Bucket Elimination

- Maintain list ("bucket") B_j for each variable x_j
- Each BDD stored in bucket according to root node variable

Algorithm

Initialization:

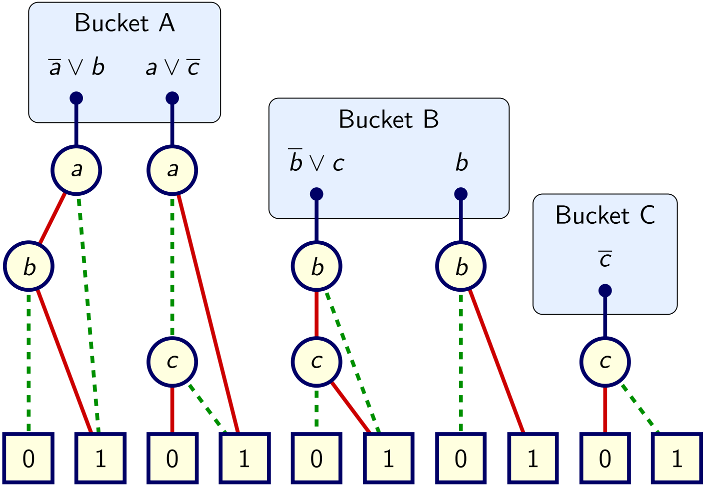
Form BDD t_i for each input clause C_i
Place each t_i in bucket according to $Var(t_i)$

For each bucket B_j:

Form conjunction s_j of all BDDs in bucket B_j
If $s_j = \bot$ then return UNSAT
Compute $r_j = \exists x_j s_j$
Place r_j in bucket according to $Var(r_j)$

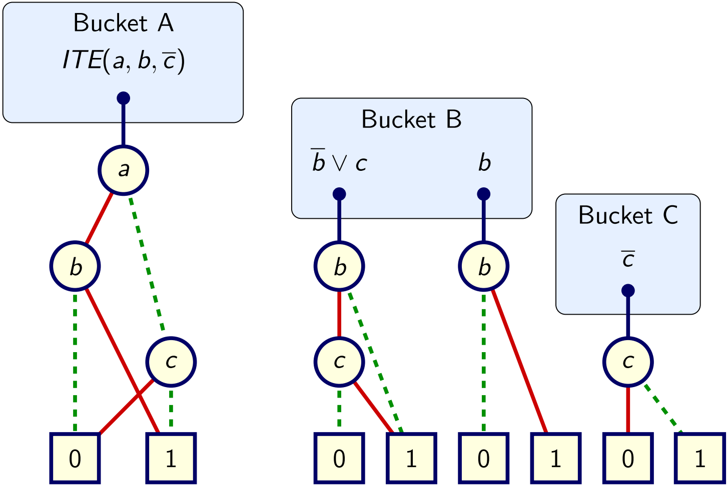
return SAT

Bucket Elimination Example



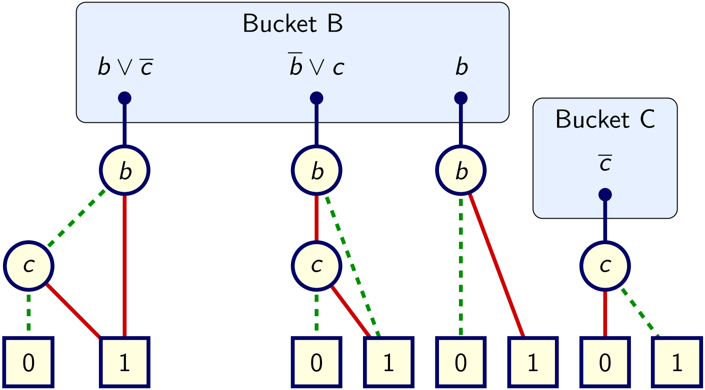
- Initially: BDD for each input clause
- In bucket according to root variable

Bucket Elimination Example



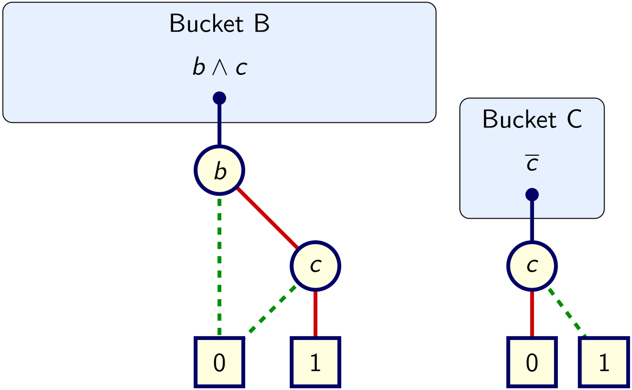
- Conjunct BDDs in topmost bucket A

Bucket Elimination Example



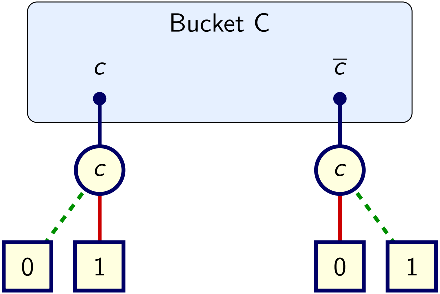
- Existentially quantify variable a
- Place result in appropriate bucket

Bucket Elimination Example



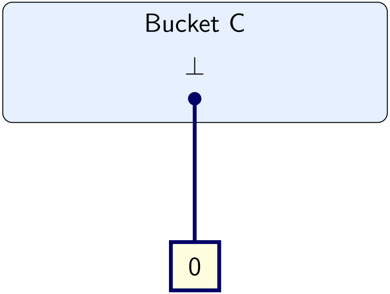
- Conjunct BDDs in bucket B

Bucket Elimination Example



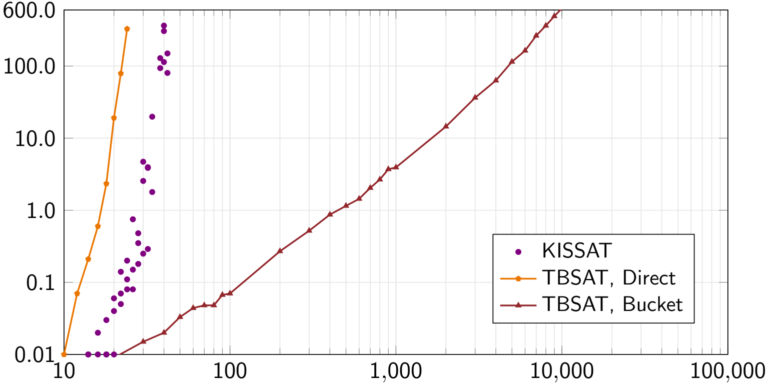
- Existentially quantify variable b
- Place result in appropriate bucket

Bucket Elimination Example



- Conjunct BDDs in bucket C
- Final result will be $\perp$ or $\top$

Parity Benchmark Runtime



- $n = 10{,}000$ in 633 seconds
- Large benefit from quantification
  - abstracts away intermediate variables

What Parity Benchmark Demonstrates

- Binary Decision Diagrams (BDDs) can play important role in SAT
  - In supplementing current SAT algorithms
  - But, *what about proof generation?*

Extended Resolution and BDDs

- Tseitin, 1967

Can introduce extension variables

- Variable z that has not yet occurred in proof
- Must add *defining* clauses
  - Encode constraint of form $z \leftrightarrow F$
  - Boolean formula z over input and earlier extension variables

Extension variable z becomes shorthand for formula F

- Repeated use can yield exponentially smaller proof

Extended Resolution and BDDs

- Tseitin, 1967

Can introduce extension variables

- Variable z that has not yet occurred in proof
- Must add *defining* clauses
  - Encode constraint of form $z \leftrightarrow F$
  - Boolean formula z over input and earlier extension variables

Extension variable z becomes shorthand for formula F

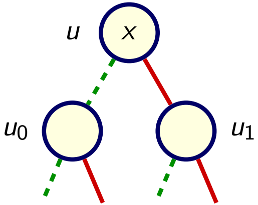
- Repeated use can yield exponentially smaller proof

Generate extension variable for every node in BDD

- Biere, Sinz, Jussila, 2006
- Each recursive step of Apply algorithm justified as proof steps
- Reducing formula to BDD $\perp$ yields UNSAT proof

Generating Extended Resolution Proofs

- Create extension variable for each node in BDD
  - Notation: Same symbol for node and its extension variable

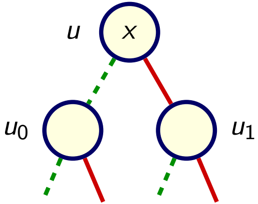


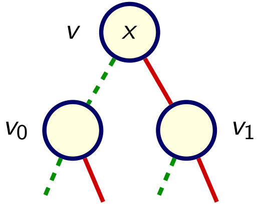
- Defining clauses encode constraint $u \leftrightarrow \text{ITE}(x, u_1, u_0)$

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

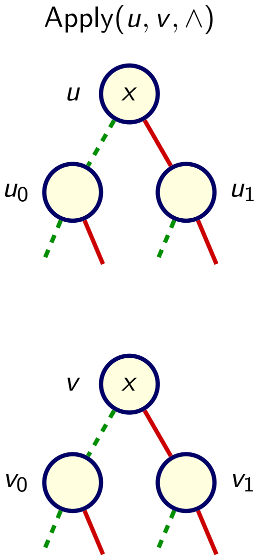
Apply Algorithm Recursion

Apply$(u, v, \wedge)$





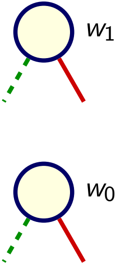
Apply Algorithm Recursion



Recursion

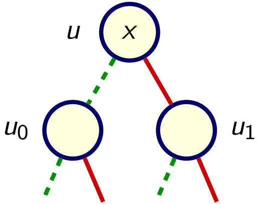
$\mathrm{Apply}(u_1, v_1, \wedge) \rightarrow$

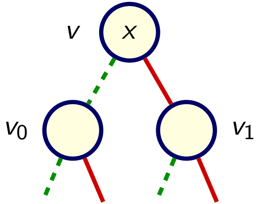
$\mathrm{Apply}(u_0, v_0, \wedge) \rightarrow$



Apply Algorithm Recursion

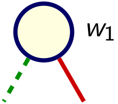
$\text{Apply}(u, v, \wedge)$



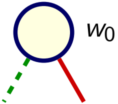


Recursion

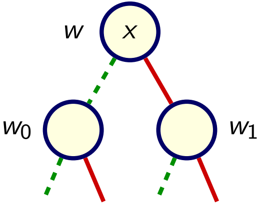
$\text{Apply}(u_1, v_1, \wedge) \;\rightarrow$



$\text{Apply}(u_0, v_0, \wedge) \;\rightarrow$



Result



Proof-Generating Apply Operation

Integrate Proof Generation into Apply Operation

- When Apply$(u, v, \wedge)$ returns 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 u, v, and 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

$$
\dfrac{\dfrac{\begin{array}{l} x \rightarrow (u \rightarrow u_1) \\ x \rightarrow (v \rightarrow v_1) \\ x \rightarrow (w_1 \rightarrow w) \quad \color{blue}{u_1 \wedge v_1 \rightarrow w_1} \end{array}}{x \rightarrow (u \wedge v \rightarrow w)} \qquad \dfrac{\begin{array}{l} \overline{x} \rightarrow (u \rightarrow u_0) \\ \overline{x} \rightarrow (v \rightarrow v_0) \\ \overline{x} \rightarrow (w_0 \rightarrow w) \quad \color{blue}{u_0 \wedge v_0 \rightarrow w_0} \end{array}}{\overline{x} \rightarrow (u \wedge v \rightarrow w)}}{u \wedge v \rightarrow w}
$$

Can perform with 2 RUP steps

Quantification Operations

Operation $\text{EQuant}(f, X)$

- Abstract away details of satisfying (partial) 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
  - $\text{EQuant}(u, X) \rightarrow w$
  - Generate proof $u \rightarrow w$
  - Algorithm similar to proof-generating Apply operation

Overall Proof Task

Input Variables

Input Clauses

- Set of input clauses C_I over the input variables

Completion

- Generate Proof $C_I \models \perp$

Trusted BDDs (TBDD)

Components

- BDD with root node t
- Proof step for unit clause (t)

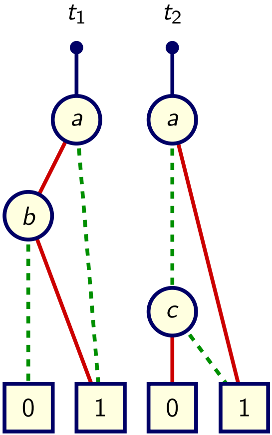
Interpretation

- $C_I \models t$
- Any variable assignment that satisfies input clauses must yield 1 for BDD with root t

TBDD Example

C_1	$\overline{a} \vee b$
C_2	$a \vee \overline{c}$

$$t_1 \longleftarrow \mathit{FromClause}(C_1)$$
$$t_2 \longleftarrow \mathit{FromClause}(C_2)$$

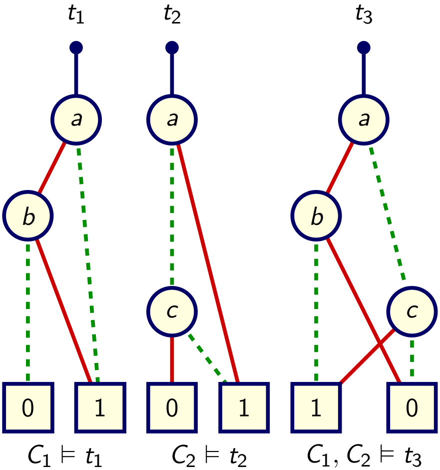


$C_1 \models t_1$ $\quad$ $C_2 \models t_2$

TBDD Example

C_1	$\overline{a} \vee b$
C_2	$a \vee \overline{c}$

$$t_1 \longleftarrow \mathit{FromClause}(C_1)$$
$$t_2 \longleftarrow \mathit{FromClause}(C_2)$$
$$t_3 \longleftarrow \mathit{ApplyAnd}(t_1, t_2)$$



Structure of Overall Proof

Input Variables

- BDD variable for each input variable

Structure of Overall Proof

Input Variables

- BDD variable for each input variable

Input Clauses

- For each input clause $C_i \in C_I$, generate BDD representation t_i
- Generate *validation* proof $C_i \vDash t_i$
  - Sequence of resolution steps based on linear structure of BDD
- Initial set of TBDDs

Structure of Overall Proof

Input Variables

- BDD variable for each input variable

Input Clauses

- For each input clause $C_i \in C_I$, generate BDD representation t_i
- Generate *validation* proof $C_i \vDash t_i$
  - Sequence of resolution steps based on linear structure of BDD
- Initial set of TBDDs

Combine Top-Level BDDs

- Choose TBDDs t_i, t_j. Use to generate TBDD t_k
- $t_k \longleftarrow \text{ApplyAnd}(t_i, t_j)$
  - Combine proofs $C_I \vDash t_i$, $C_I \vDash t_j$ and $t_i \wedge t_j \rightarrow t_k$ to validate $C_I \vDash t_k$
- $t_k \longleftarrow \text{EQuant}(t_i, X)$
  - Combine proofs $C_I \vDash t_i$ and $t_i \rightarrow t_k$ to validate $C_I \vDash t_k$

Structure of Overall Proof

Input Variables

- BDD variable for each input variable

Input Clauses

- For each input clause $C_i \in C_I$, generate BDD representation t_i
- Generate *validation* proof $C_i \models t_i$
  - Sequence of resolution steps based on linear structure of BDD
- Initial set of TBDDs

Combine Top-Level BDDs

- Choose TBDDs t_i, t_j. Use to generate TBDD t_k
- $t_k \longleftarrow \text{ApplyAnd}(t_i, t_j)$
  - Combine proofs $C_I \models t_i$, $C_I \models t_j$ and $t_i \wedge t_j \rightarrow t_k$ to validate $C_I \models t_k$
- $t_k \longleftarrow \text{EQuant}(t_i, X)$
  - Combine proofs $C_I \models t_i$ and $t_i \rightarrow t_k$ to validate $C_I \models t_k$

Completion

- When $t_k = \bot$ have proof $C_I \models \bot$

Comparing Proofs

Generated by CDCL Solver

- Resolution
- Encode conflict clauses
  - Increasingly strong constraints on set of satisfying solutions
- Reach empty clause when detect there is no solution

Generated with BDD-Based Solver

- Extended resolution
- Justify each recursive step of BDD algorithm
- Reach empty clause when reduce formula to BDD leaf $\perp$

Checking

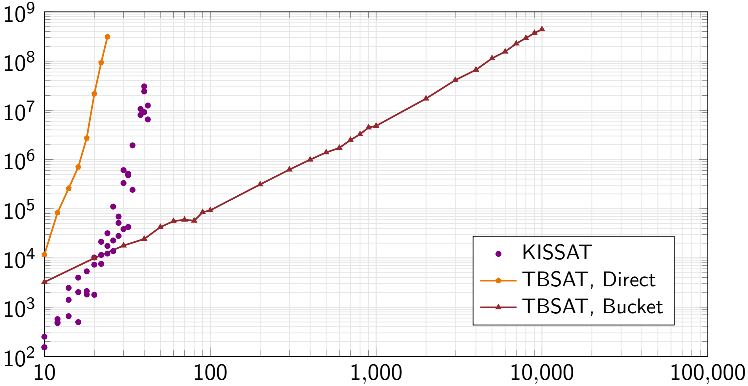
- Both checked with DRAT/LRAT checkers

TBSAT (Trusted BDD Satisfiability solver)

Implementation

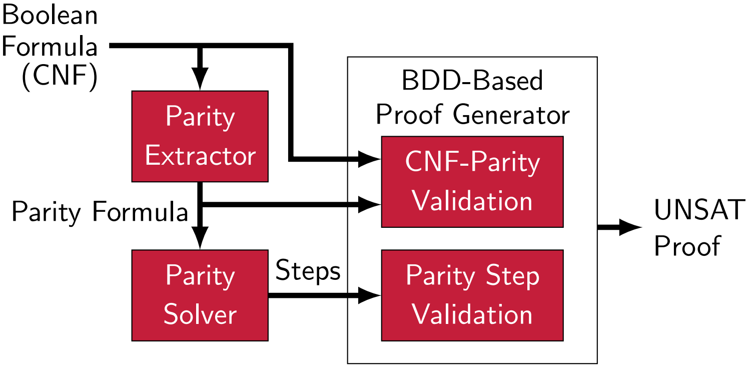
- TBUDDY: Modified version of BuDDy BDD package
  - Lind-Nielsen, ca. 1998
- Support for TBDDs and proof generation
- C/C++
- `https://github.com/rebryant/tbuddy-artifact`

Parity Benchmark Proof Complexity



- Total number of proof steps
  - Defining clauses + RUP clauses
- TBSAT with bucket elimination scales polynomially
  - Checker time ≈ Solver time

Integrating Parity Reasoning into Proof-Generating SAT Solver



- Overall flow same as SAT solver
- Parity solver does all of the reasoning
- BDDs serve only as mechanism for generating clausal proof

Gaussian Elimination Over GF2

System of Equations $E = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_m\}$

$$\mathbf{e}_i : \quad \sum_{j=1,n} a_{i,j} \cdot x_j = b_i$$

Assume

- $a_{i,j}, x_j \in \{0, 1\}$
- $a + b \;\equiv\; a \oplus b$
- $a \cdot b \;\equiv\; a \wedge b$

Capability

- Can determine if there are any solutions for $x_1, x_2, \ldots, x_n$

Gaussian Elimination Over GF2

System of Equations $E = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_m\}$

$$\mathbf{e}_i : \quad \sum_{j=1,n} a_{i,j} \cdot x_j = b_i$$

Elimination Step

1. Choose pivot equation $\mathbf{e}_s$ and variable x_t such that $a_{s,t} = 1$
2. For each $i \neq s$:

$$\mathbf{e}_i \quad \leftarrow \quad \begin{cases} \mathbf{e}_i & a_{i,t} = 0 \\ \mathbf{e}_s + \mathbf{e}_i, & a_{i,t} = 1 \end{cases}$$

   - Guarantees $a_{i,t} = 0$ for all $i \neq s$
3. Remove $\mathbf{e}_s$ from E and repeat until single equation left

Gaussian Elimination Results

Possible Outcomes

1. If encounter degenerate equation
   - Of form $0 = 1$
   - Has no solution
2. Otherwise,
   - Can perform back substitution to find solution

CNF to Parity Constraint Validation

Clauses

- Suppose clauses $C_{i_1}, C_{i_2}, \ldots, C_{i_k}$ encode parity constraint equation $\mathbf{e}$
- Have validated BDD representations $t_{i_1}, t_{i_2}, \ldots, t_{i_k}$

Form conjunction

$$s \;=\; \bigwedge_{1 \leq j \leq k} t_{i_j}$$

- Also yields proof $C_I \models s$

Represent Constraint

- Form BDD representation t_j of $\mathbf{e}$

Validate

- Generate proof $s \rightarrow t_j$
- Use to validate term $C_I \models t_j$

Parity Step Validation

Assume

- Have BDDs t_i and t_j representing equations $\mathbf{e}_i$ and $\mathbf{e}_j$
- Satisfying $C_I \vDash t_i$ and $C_I \vDash t_j$

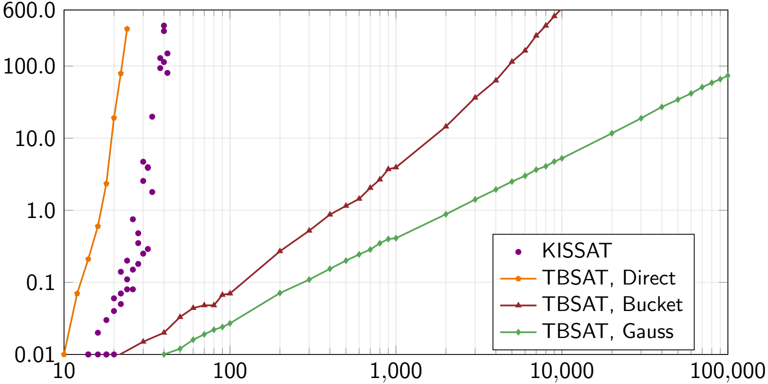
Compute

- $s \longleftarrow \mathit{ApplyAnd}(t_i, t_j)$
  - Gives proof $t_i \wedge t_j \rightarrow s$
- Generate BDD representation t_k of equation $\mathbf{e}_k = \mathbf{e}_i + \mathbf{e}_j$

Validation

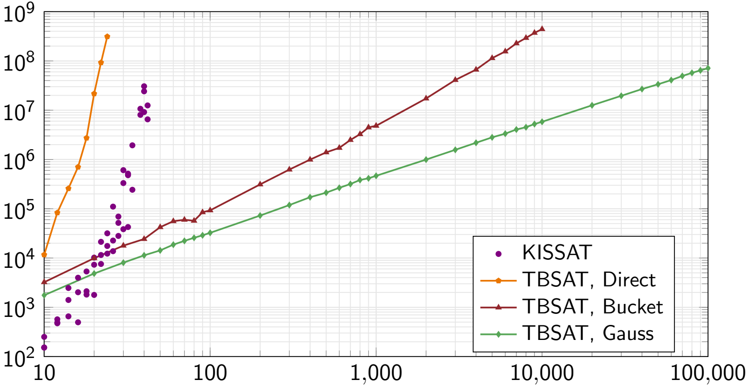
- Generate proof $s \rightarrow t_k$
- Combine with other proofs to validate $C_I \vDash t_k$

Parity Benchmark Runtime



- $n = 100{,}000$ in 74 seconds
- Upper limit: $n = 699{,}051$
  - BuDDy limited to $2^{21} - 1$ BDD variables

Parity Benchmark Proof Complexity



- Total number of proof steps
  - Defining clauses + RUP clauses
- Checker time $\approx$ Solver time

Final Thoughts on SAT Solvers

CDCL is the best overall approach

- Readily generates resolution proofs
- But, very weak for parity and cardinality constraints

BDDs provide complementary strengths

- Can generate extended resolution proofs
- Very strong for parity constraints
- Some success with cardinality constraints

Future solvers should use combination of methods

- With unified proof framework
- Clausal reasoning
- Constraint reasoning
- Boolean reasoning

Final Thoughts on Checkable Proofs

Important capability

- Vital to gain confidence in automated reasoning tools
- Benefits both tool developers and tool users

SAT community handled this especially well

- Started with well-established logical framework (resolution)
- Developed efficient algorithms that integrated well with solvers (RUP)
- Included more general capabilities (extended resolution)
- Formulated file formats, tool chain
- Fostered deployment through competitions

More challenging for other domains

- Beyond Boolean

Some 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, A. Biere, and M. J. H. Heule, "Clausal Proofs from Pseudo-Boolean Reasoning," *TACAS*, 2022
- R. E. Bryant, "TBUDDY: A Proof-Generating BDD Package," *in submission*, 2022