Binary Decision Diagrams and Extended Resolution Proof Generation

Randal E. Bryant
Carnegie Mellon University

http://www.cs.cmu.edu/~bryant

Binary Decision Diagrams

Restricted Form of Branching Program

- **Graph representation of Boolean function**
- **Canonical form**
- **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

Summary: Time Line

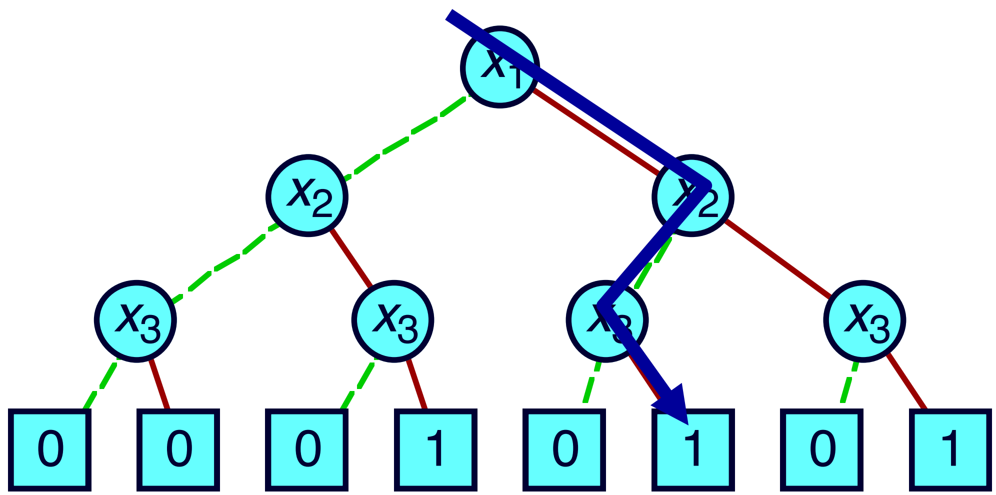
Year	Contribution
1965	Robinson formulates resolution proof framework
1967	Tseitin formulates *extended resolution* (ER)
1986	Bryant introduces Binary Decision Diagrams (BDDs)
2003	Zhang & Malik extend SAT solver to generate UNSAT proofs
2006	Sinz, Biere, (and Jussila) (SBJ) generate ER proofs with BDD-based SAT solver
2020	Bryant & Heule refine and extend SBJ

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

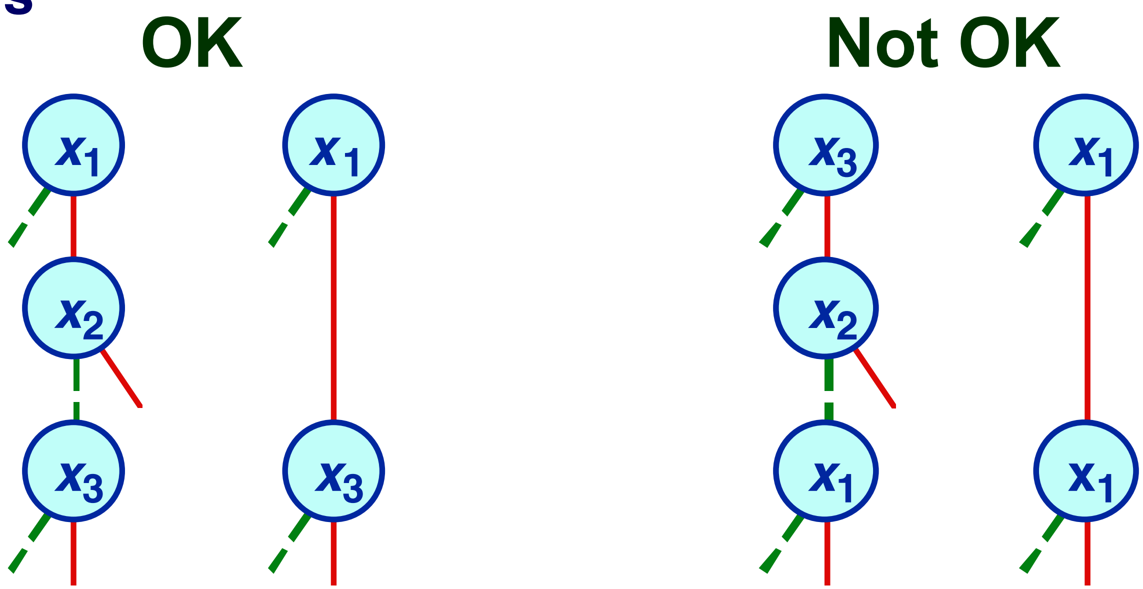
Decision Tree



- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.

Variable Ordering

- **Assign arbitrary total ordering to variables**
  - e.g., $x_1 < x_2 < x_3$
- **Variables must appear in ascending order along all paths**

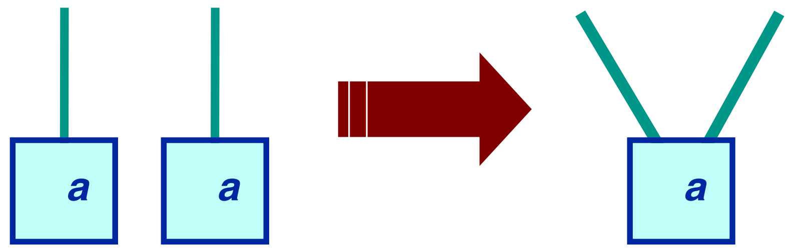


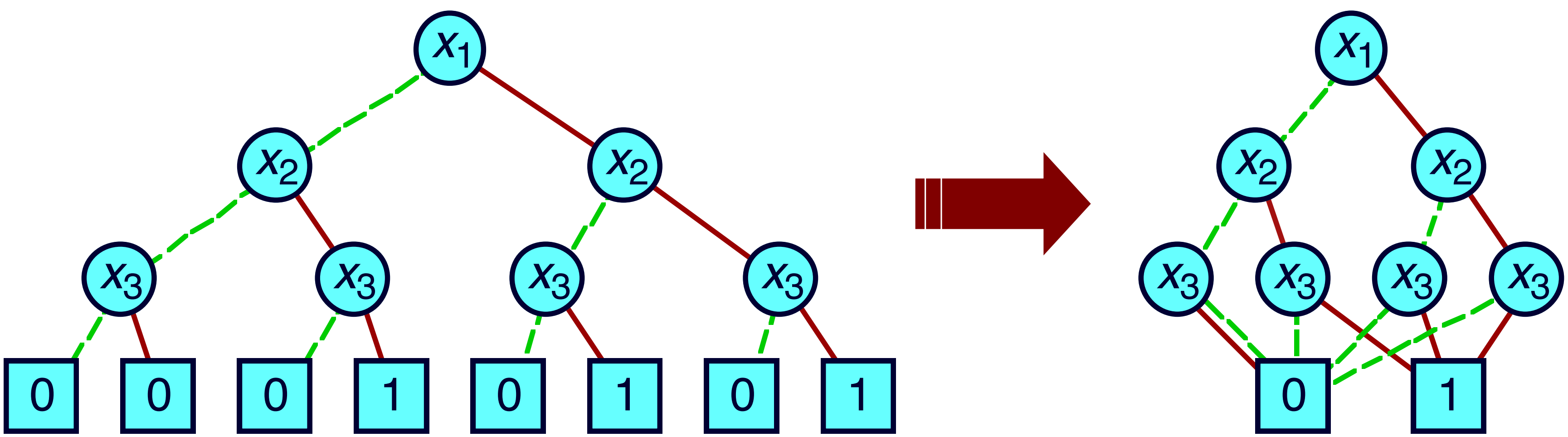
Properties

- **No conflicting variable assignments along path**
- **Simplifies manipulation**

Reduction Rule #1

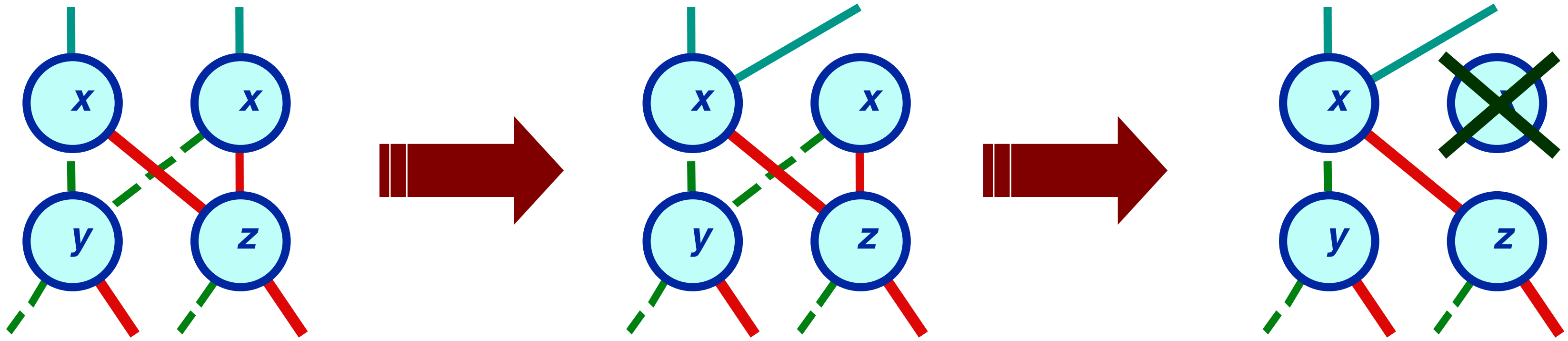
Merge equivalent leaves

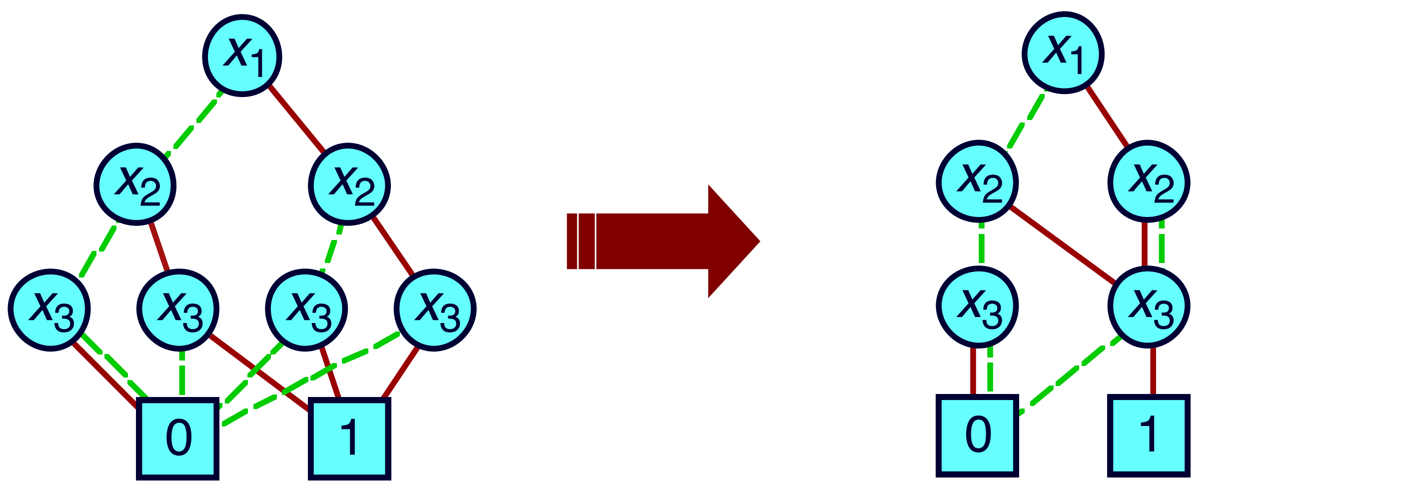




Reduction Rule #2

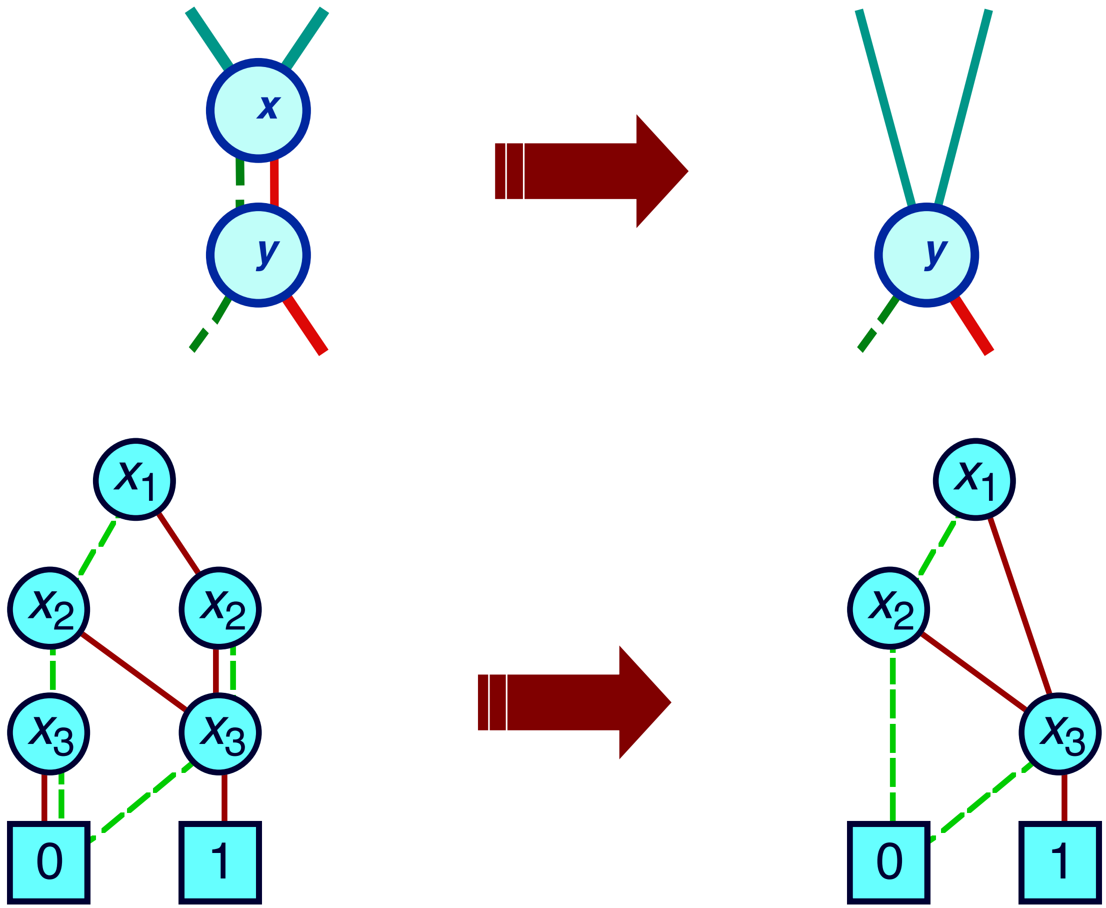
Merge isomorphic nodes





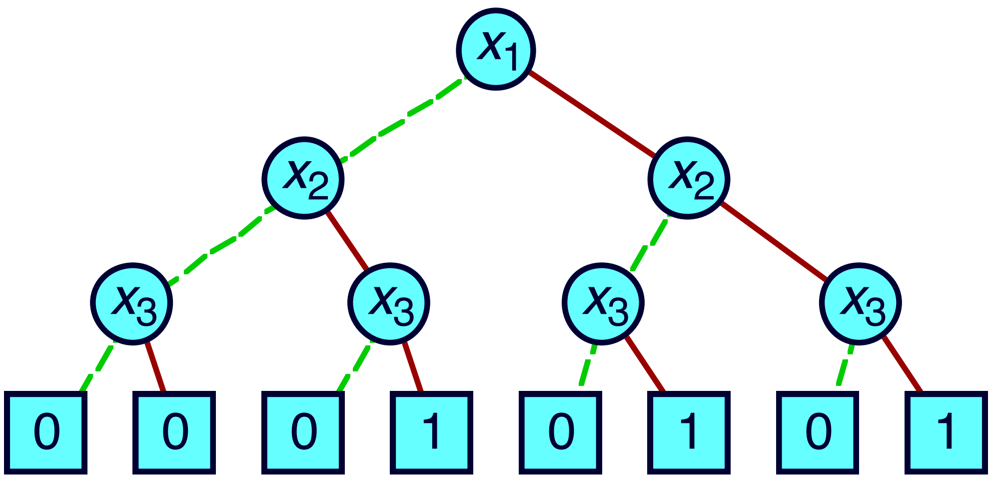
Reduction Rule #3

Eliminate Redundant Tests

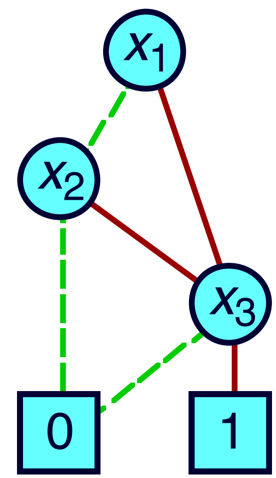


Example OBDD

Initial Graph



Reduced Graph



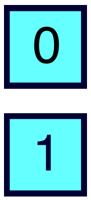
$(x_1 \vee x_2) \wedge x_3$

Canonical representation of Boolean function

- ❑ For given variable ordering
- **Two functions equivalent if and only if graphs isomorphic**
  - Can be tested in linear time
- **Desirable property: *simplest form is canonical.***

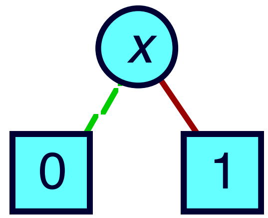
Example Functions

Constants



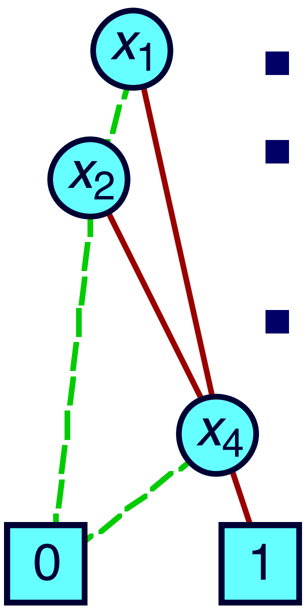
- 0: Unique unsatisfiable function
- 1: Unique tautology

Variable



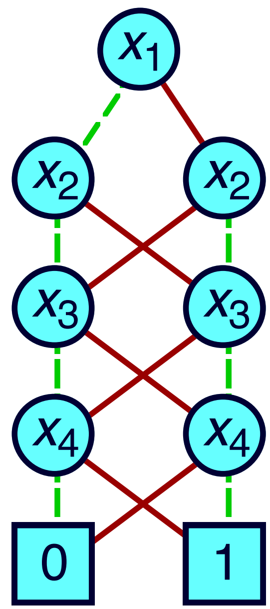
Treat variable as function

Typical Function



- $(x_1 \vee x_2) \wedge x_4$
- No vertex labeled x_3
  - independent of x_3
- Many subgraphs shared

Odd Parity



Linear representation

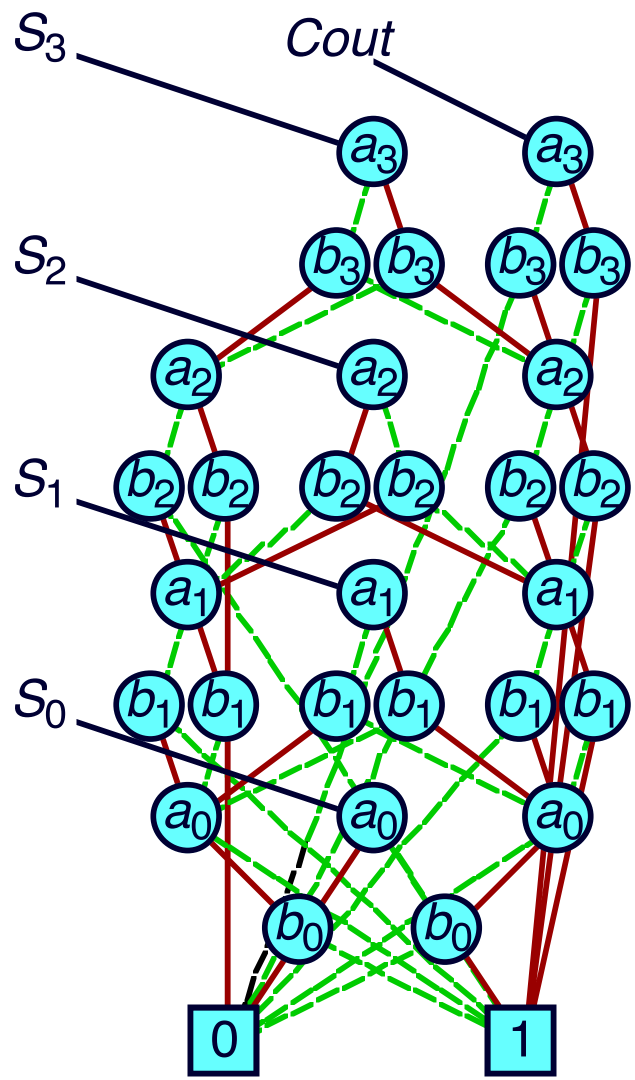
More Complex Functions

Functions

- Add 4-bit words `a` and `b`
- Get 4-bit sum `S`
- Carry output bit `Cout`

Shared Representation

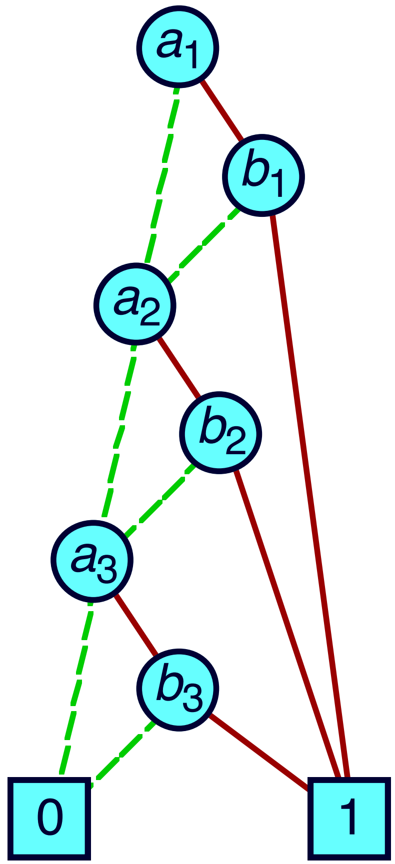
- Graph with multiple roots
- 31 nodes for 4-bit adder
- 571 nodes for 64-bit adder
- Linear growth!



Effect of Variable Ordering

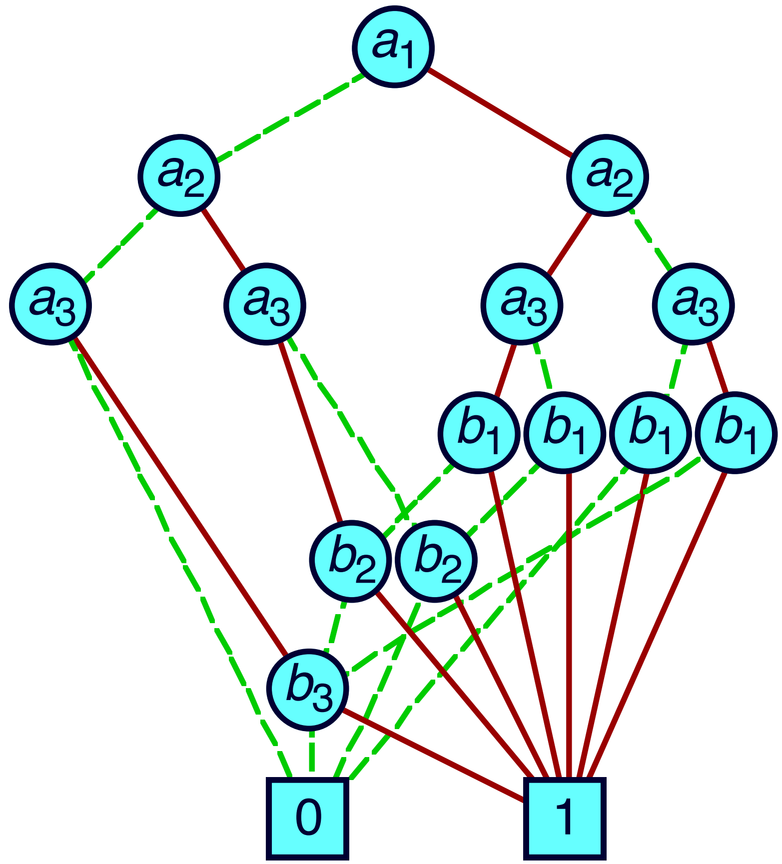
$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$$

Good Ordering



Linear Growth

Bad Ordering



Exponential Growth

Sample Function Classes

Function Class	Best	Worst	Ordering Sensitivity
ALU (Add/Sub)	linear	exponential	High
Symmetric	linear	quadratic	None
Multiplication	exponential	exponential	Low

General Experience

- **Many tasks have reasonable OBDD representations**
- **Algorithms remain practical for up to 500,000 node OBDDs**
- **Heuristic ordering methods generally satisfactory**

Symbolic Manipulation with OBDDs

Strategy

- **Represent data as set of OBDDs**
  - Identical variable orderings
- **Express solution method as sequence of symbolic operations**
  - Sequence of constructor & query operations
  - Similar style to on-line algorithm
- **Implement each operation by OBDD manipulation**
  - Do all the work in the constructor operations

Key Algorithmic Properties

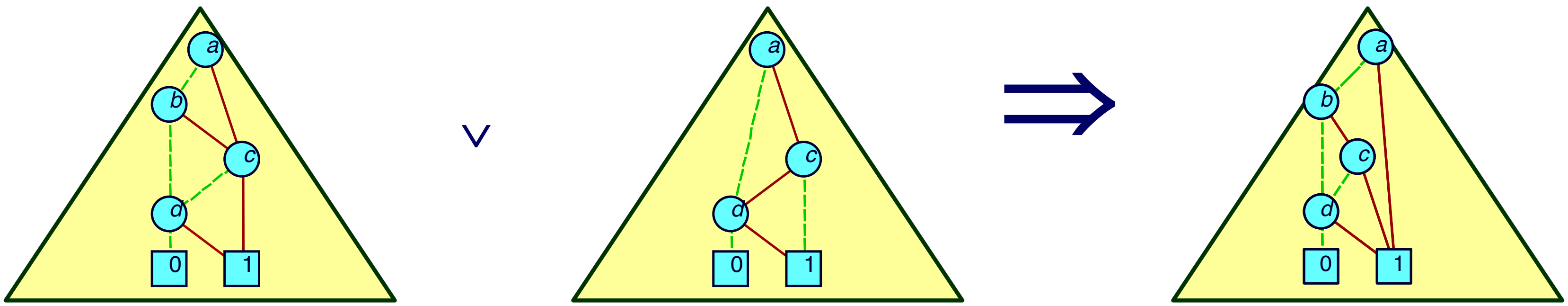
- **Arguments are OBDDs with identical variable orderings**
- **Result is OBDD with same ordering**
- **Each step polynomial complexity**

Apply Operation

Concept

- Basic technique for building OBDD from Boolean formula.

A *op* *B* *A op B*



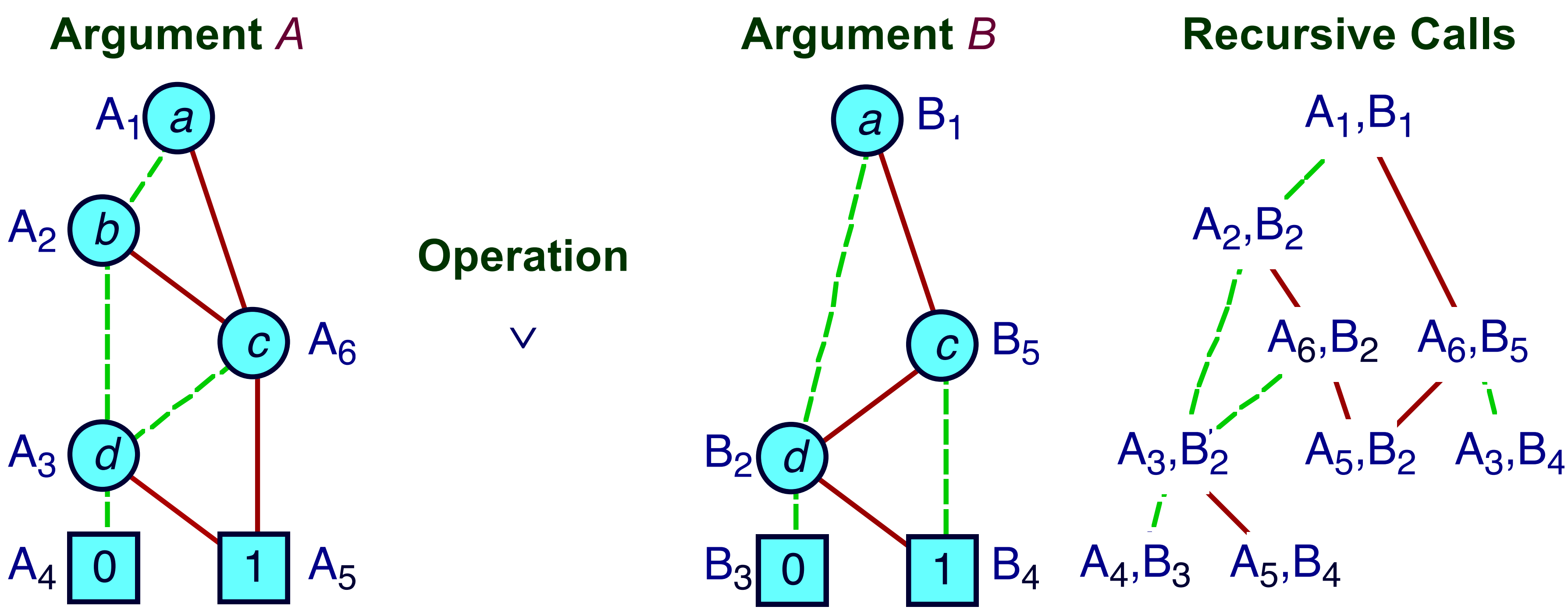
Arguments *A*, *B*, *op*

- *A* and *B*: Boolean Functions
  - Represented as OBDDs
- op: Boolean Operation (e.g., ^, &, |)

Result

- OBDD representing composite function
- *A op B*

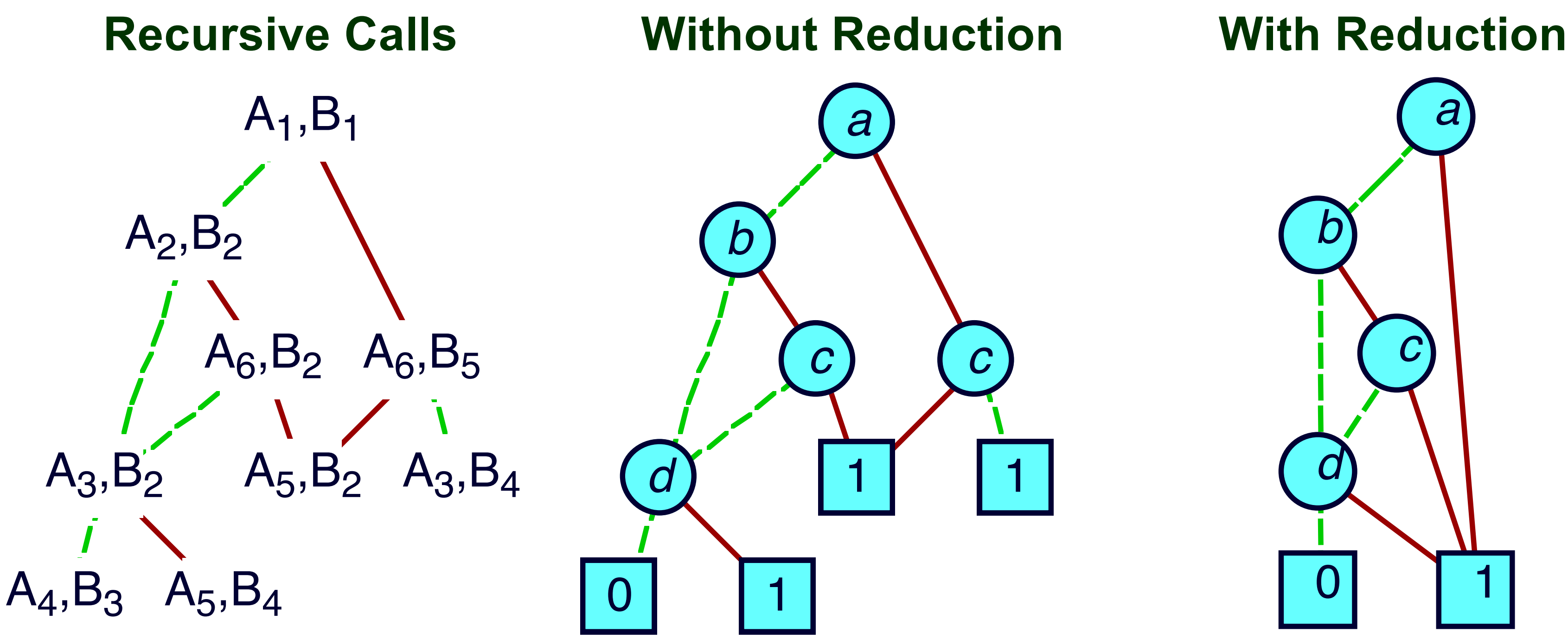
Apply Execution Example



Optimizations

- Dynamic programming
- Early termination rules

Apply Result Generation

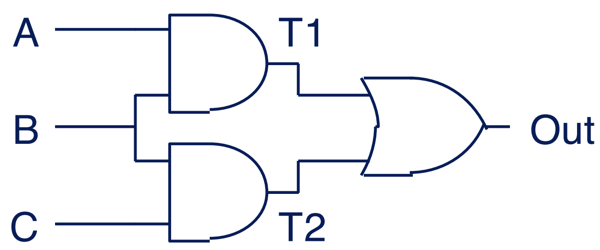


- Recursive calling structure implicitly defines unreduced BDD
- Apply reduction rules bottom-up as return from recursive calls

Generating OBDD from Network

Task: **Represent output functions of gate network as OBDDs.**

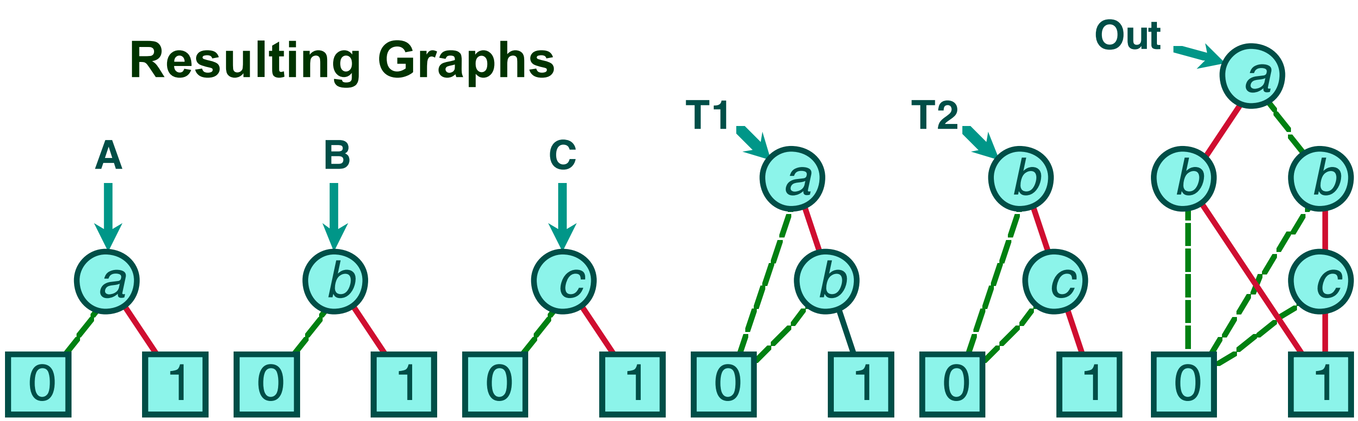
Network



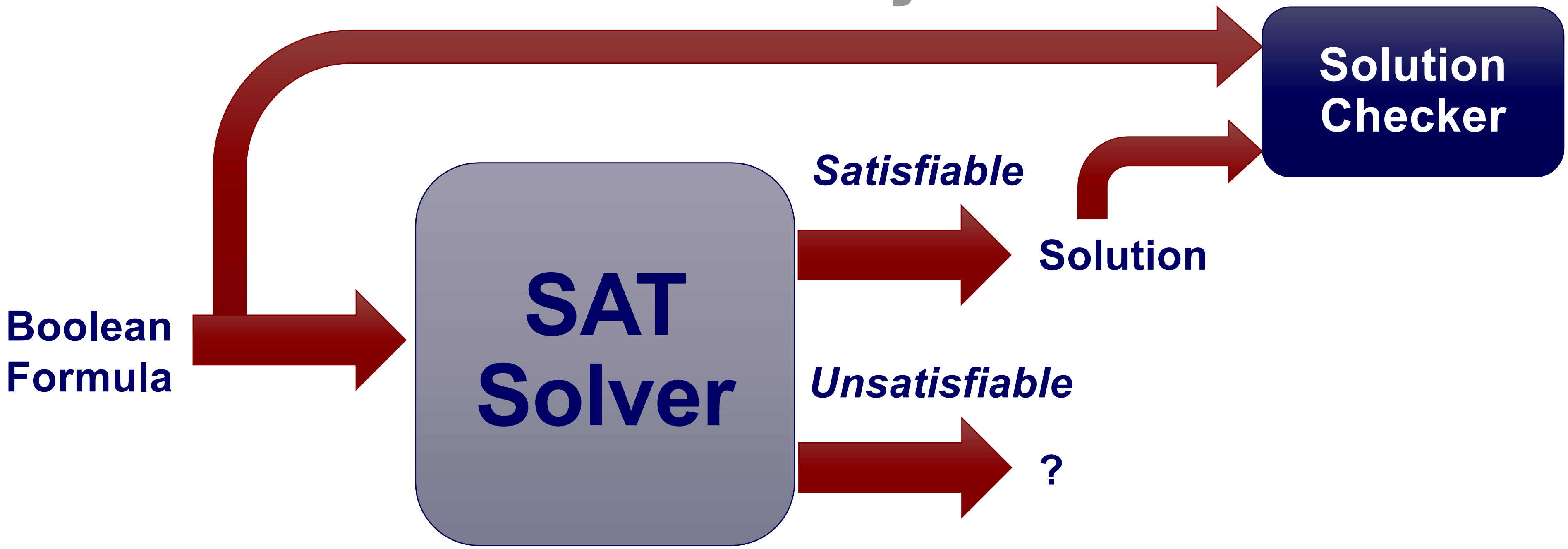
Evaluation

A ← new_var ("a");
B ← new_var ("b");
C ← new_var ("c");
T1 ← And (A, 0, B);
T2 ← And (B, C);
Out ← Or (T1, T2);

Resulting Graphs



Boolean Satisfiability Solvers



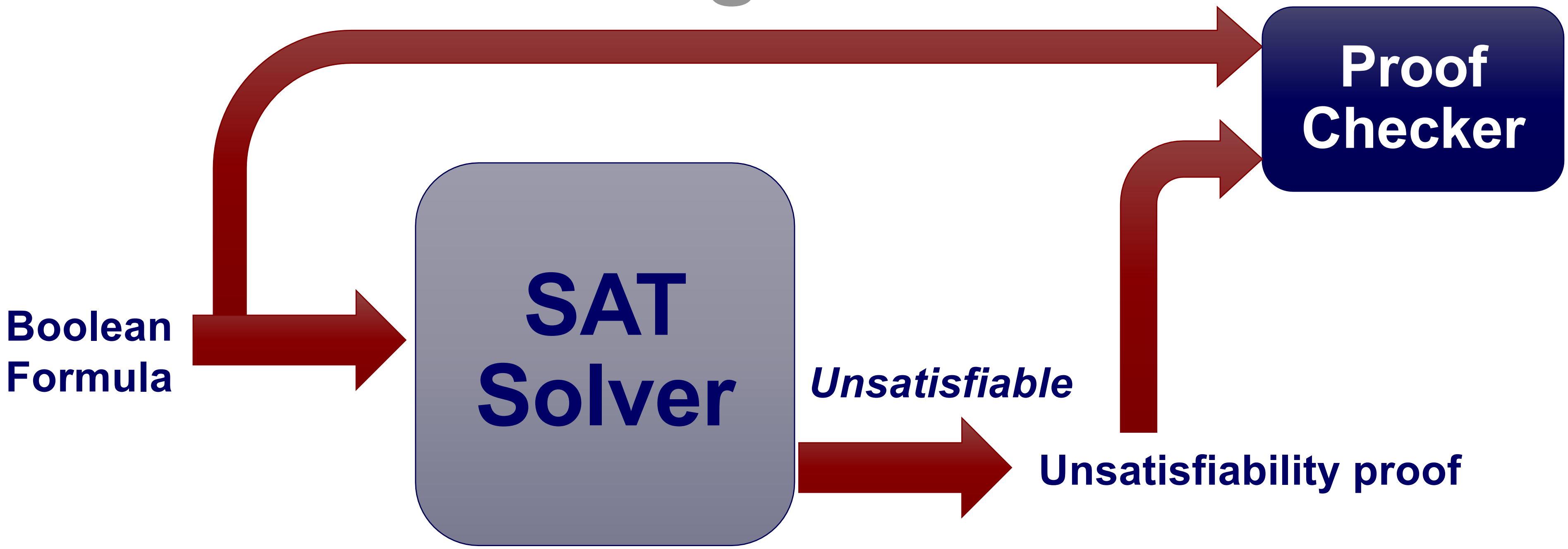
SAT Solvers Useful & Powerful

- Formal verification
- Security verification
- Optimization

Can We Trust Them?

- No!
- Complex software with lots of optimizations

Proof Generating Solvers



Unsatisfiability Proof

- Step-by-step proof in some logical framework

Proof Checker

- Simple program
- May be formally verified

Basics

Clauses

- $\neg u \vee v \vee w$ — Disjunction of literals
- $\bot$ — Empty clause (False)

Resolution Principle

$$P \longrightarrow w \qquad \frac{\overbrace{\neg u \vee v}^{\neg P} \vee w \qquad \neg w \vee \overbrace{x \vee \neg z}^{Q}}{\neg u \vee v \vee x \vee \neg z} \qquad w \longrightarrow Q$$

$$P \longrightarrow Q$$

Clausal Resolution Proof

Step	Clause	Antecedents	Formula
1	$\neg v \lor w$		$v \longrightarrow w$
2	$\neg v \lor \neg w$		$v \longrightarrow \neg w$
3	v		v
4	$\neg v$	1, 2	$\neg v$
5	$\bot$	3, 4	$v \land \neg v$

Input clauses (steps 1–3)

Derived clauses (steps 4–5)

- **Prove conjunction of input clauses unsatisfiable**
- **Add derived clauses**
  - Each provides list of antecedent clauses that resolve to new clause
- **Finish with empty clause**
  - Proof is series of inferences leading to contradiction

Extended Resolution

Can introduce *extension variables*

- **Variable e that has not yet occurred in proof**
- **Must introduce *defining clauses***
  - Clauses creating constraint of form $e \leftrightarrow F$
  - Boolean formula F over input and earlier extension variables
- **Example: Prove following set of constraints unsatisfiable**

Constraint	Clauses	
$u \land v \longrightarrow w$	$\neg u \lor \neg v \lor w$	
$u \land v \longrightarrow \neg w$	$\neg u \lor \neg v \lor \neg w$	
$u \land v$	u	v

- **Strategy: Introduce extension variable e such that $e \leftrightarrow u \land v$**

ER Proof

Step	Clause	Antecedents	Formula
1	$\neg u \lor \neg v \lor w$		$u \land v \longrightarrow w$
2	$\neg u \lor \neg v \lor \neg w$		$u \land v \longrightarrow \neg w$
3	u		u
4	v		v
5	$e \lor \neg u \lor \neg v$		$u \land v \longrightarrow e$
6	$\neg e \lor u$		$e \longrightarrow u$
7	$\neg e \lor v$		$e \longrightarrow v$
8	$\neg e \lor \neg v \lor w$	1, 6	$e \land v \longrightarrow w$
9	$\neg e \lor w$	7, 8	$e \longrightarrow w$
10	$\neg e \lor \neg v \lor \neg w$	2, 6	$e \land v \longrightarrow \neg w$
11	$\neg e \lor \neg w$	7, 10	$e \longrightarrow \neg w$
12	$e \lor \neg v$	3, 5	$v \longrightarrow e$
13	e	4, 12	e
14	$\neg e$	9, 11	$\neg e$
15	$\bot$	13, 14	$e \land \neg e$

Input clauses (steps 1–4)

Defining clauses (steps 5–7)

Derived clauses (steps 8–15)

$u \land v$ replaced with e (steps 9, 11, 13)

Reduced, Ordered Binary Decision Diagrams (BDDs)

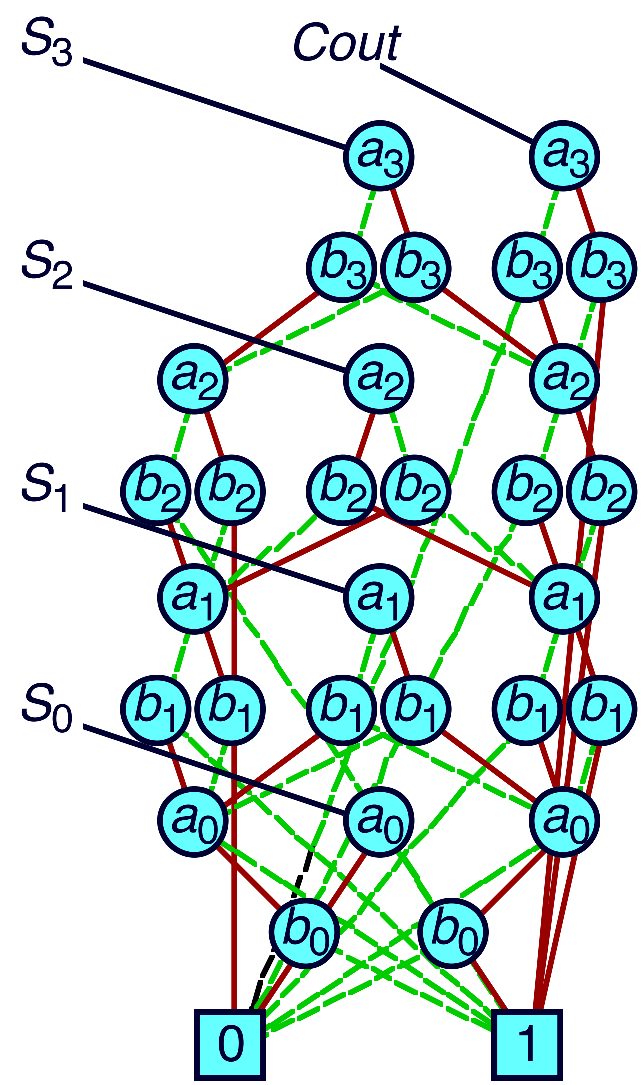
- **Bryant, 1986**

Representation

- **Canonical representation of Boolean function**
- **Compact for many useful cases**

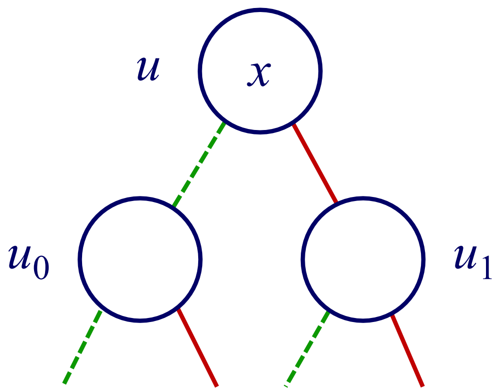
Algorithms

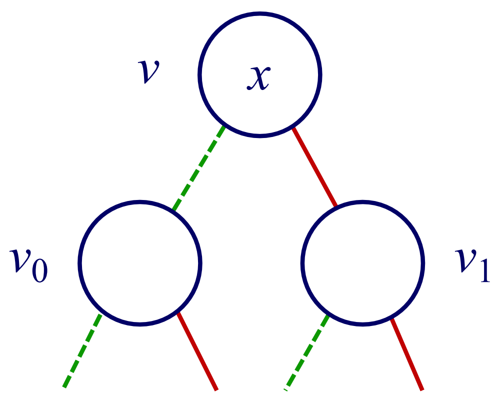
- **Apply(*f*, *g*, *op*)**
  - *op* is Boolean operation (e.g., ∧, ∨, ⊕)
  - BDD representation of *f op g*
- **EQuant(*f*, *V*)**
  - *V* set of variables
  - BDD representation of ∃*V f*



Apply Algorithm Recursion

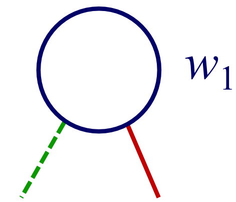
Apply(u, v, $\wedge$)



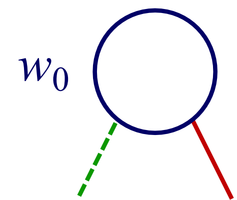


Recursion

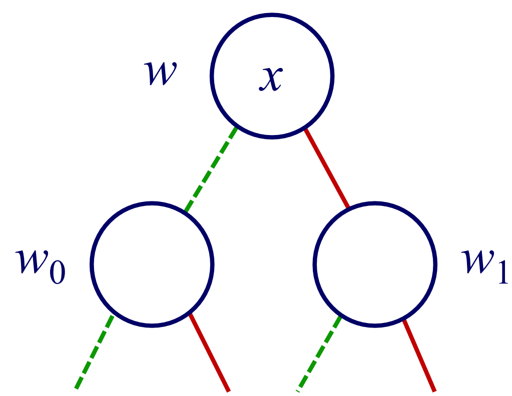
Apply(u_1, v_1, $\wedge$) ➞



Apply(u_0, v_0, $\wedge$) ➞



Result



Apply Algorithm Nuances

Stop recursion when hit terminal case

- $f \wedge 0 \rightarrow 0$ $0 \wedge g \rightarrow 0$
- $f \wedge 1 \rightarrow f$ $1 \wedge g \rightarrow g$
- $f \wedge f \rightarrow f$

Unique Table contains all generated nodes

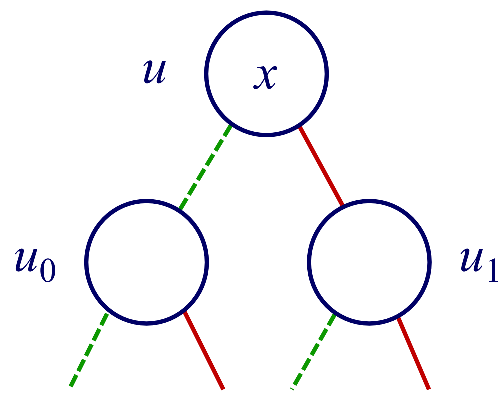
- $[x, u_1, u_0] \rightarrow u$
- Guarantees canonical form of result

Operation Cache contains previously computed results

- $[u, v, \wedge] \rightarrow w$
- Guarantees polynomial performance

Generating ER Proofs

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



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

Clause name	Formula	Clausal form
HD(u)	$x \longrightarrow (u \longrightarrow u_1)$	$\neg x \vee \neg u \vee u_1$
LD(u)	$\neg x \longrightarrow (u \longrightarrow u_0)$	$x \vee \neg u \vee u_0$
HU(u)	$x \longrightarrow (u_1 \longrightarrow u)$	$\neg x \vee \neg u_1 \vee u$
LU(u)	$\neg x \longrightarrow (u_0 \longrightarrow u)$	$x \vee \neg u_0 \vee u$

Proof-Generating Apply Operation

Integrate Proof Generation into Apply Operation

- **When** Apply(u, v, $\land$) returns w, **also generate proof** $u \land v \longrightarrow w$
- **Store step number in operation cache**

Proof Structure

- **Assume recursive calls generate proofs**
  - $u_1 \land v_1 \longrightarrow w_1$
  - $u_0 \land v_0 \longrightarrow w_0$
- **Combine with defining clauses for nodes** u, v, **and** w

Apply Proof Structure

$$\frac{\begin{array}{ll}\textbf{HD}(u):\ x \longrightarrow (u \longrightarrow u_1) & \\ \textbf{HD}(v):\ x \longrightarrow (v \longrightarrow v_1) & \\ \textbf{HU}(w):\ x \longrightarrow (w_1 \longrightarrow w) & u_1 \wedge v_1 \longrightarrow w_1\end{array}}{x \longrightarrow (u \wedge v \longrightarrow w)}$$

$$\frac{\begin{array}{ll}\textbf{LD}(u):\ \neg x \longrightarrow (u \longrightarrow u_0) & \\ \textbf{LD}(v):\ \neg x \longrightarrow (v \longrightarrow v_0) & \\ \textbf{LU}(w):\ \neg x \longrightarrow (w_0 \longrightarrow w) & u_0 \wedge v_0 \longrightarrow w_0\end{array}}{\neg x \longrightarrow (u \wedge v \longrightarrow w)}$$

$$\frac{x \longrightarrow (u \wedge v \longrightarrow w) \qquad \neg x \longrightarrow (u \wedge v \longrightarrow w)}{u \wedge v \longrightarrow w}$$

Nuances

- Many special cases when recursive arguments and results contain equivalences, 0s, and 1s.

Quantification Operations

Operation EQuant(f, V)

- **Critical for obtaining good performance**
- **Abstract away details of satisfying (partial) solutions**

Proof Generation

- **Don't follow recursive structure of algorithms**
- **Instead, follow with implication test**
  - EQuant(u, V) ➞ v
  - Generate proof $u \longrightarrow v$
  - Algorithm similar to proof-generating Apply operation

Overall Structure of Proof

Input Variables

- **Generate BDD variable for each input variable**

Input Clauses

- **Set of input clauses C_I**
- **For each input clause C, generate BDD representation u**
- **Generate proof $C \vdash u$**
  - Sequence of resolution steps based on linear structure of BDD

Combine Top-Level BDDs

- **Apply(u, v, ∧) → w**
  - Combine proofs $C_I \vdash u$, $C_I \vdash v$, and $u \land v \longrightarrow w$ to get $C_I \vdash w$
- **EQuant(u, V) → v**
  - Combine proofs $C_I \vdash u$ and $u \longrightarrow v$ to get $C_I \vdash v$

Completion

- When Apply(u, v, ∧) → 0 have proof $C_I \vdash \perp$

Implementation

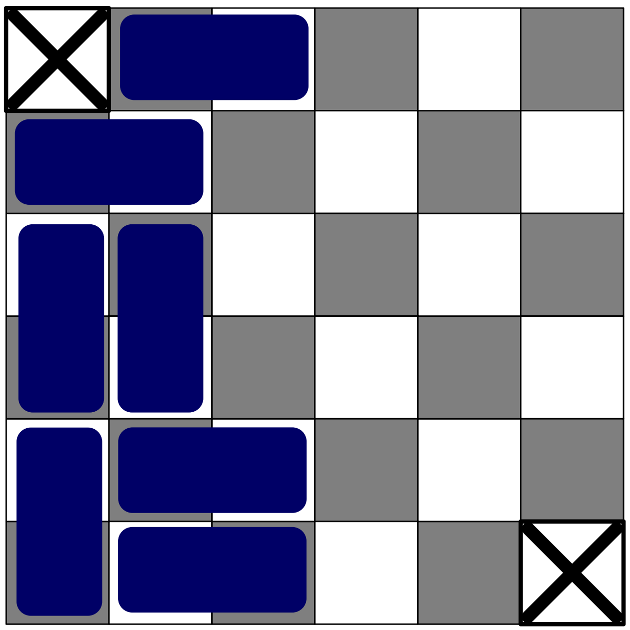
Package

- 2000 lines Python code (slow!)
- BDD package + proof generator

Benchmark Generators

- CNF file
- File specifying ordering of variables
- File specifying schedule:
  - Defines sequence of conjunctions and quantifications

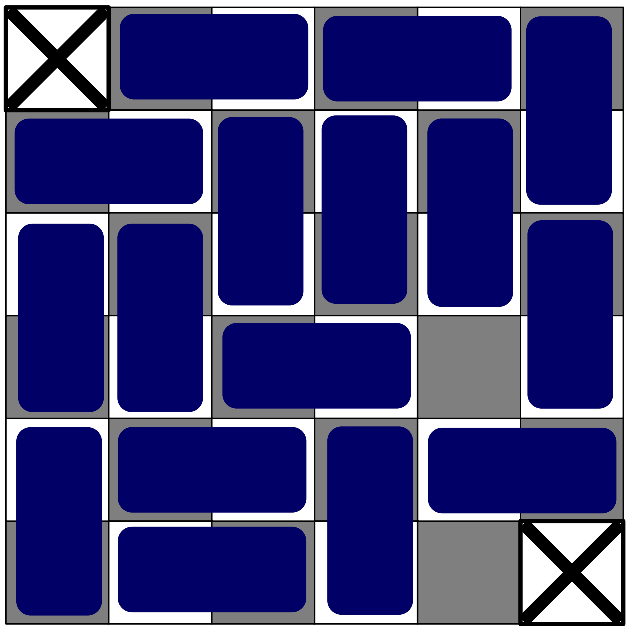
Mutilated Chessboard Problem



Definition

- $N \times N$ chessboard with 2 corners removed
- Cover with tiles, each covering one square

Mutilated Chessboard Problem



Definition

- $N \times N$ chessboard with 2 corners removed
- Cover with tiles, each covering one square

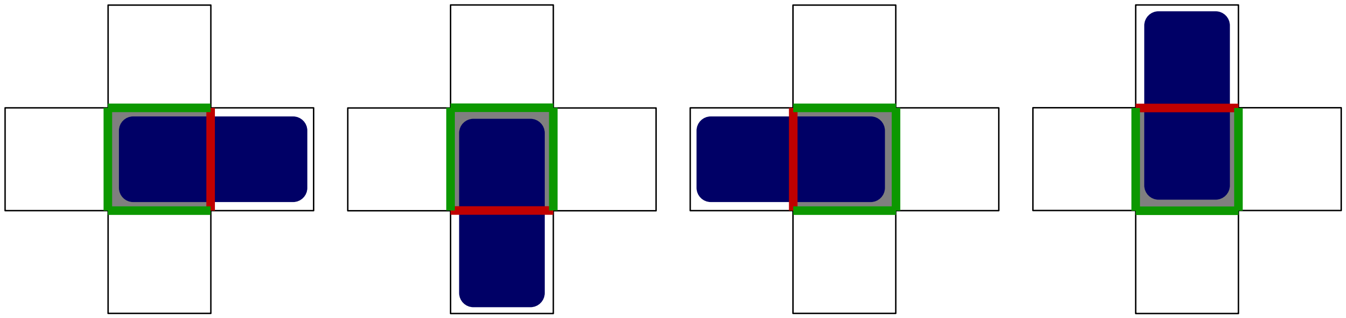
Solutions

- None
- More black squares than white
- Each tile covers one white and one black square

Proof

- All resolution proofs of exponential size

Encoding as SAT Problem



- **Boolean variable for each boundary between two squares**
  - $(N-1) \cdot N - 2$ vertical boundaries $x_{i,j}$
  - $(N-1) \cdot N - 2$ horizontal boundaries $y_{i,j}$
- **Constraints**
  - For each square, exactly one of its boundary variables = 1

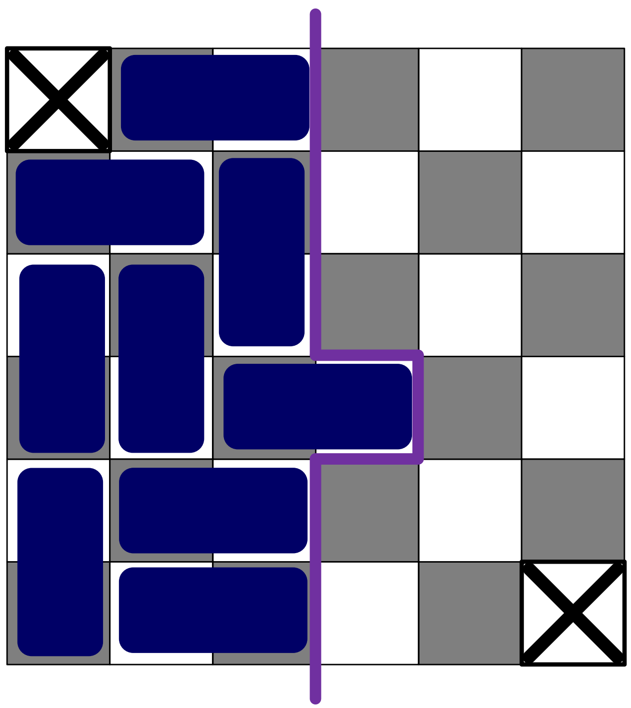
Column Scanning

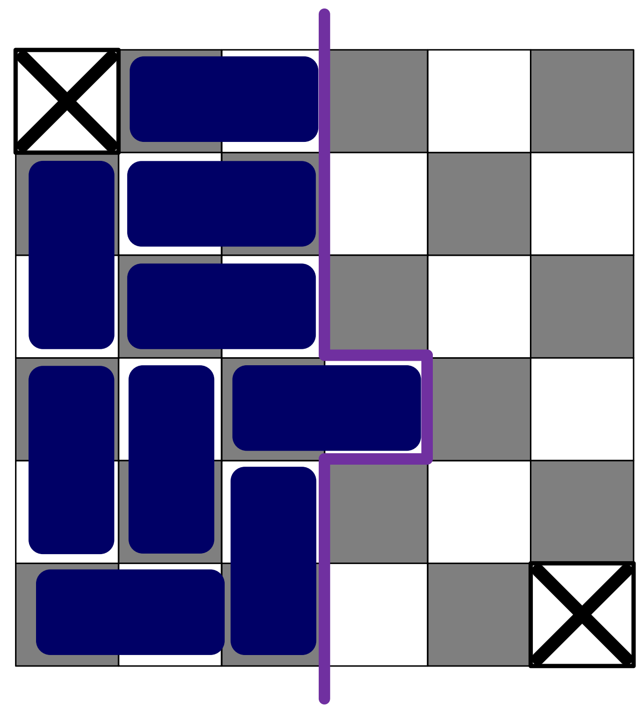
Scanning

- Add tiles for each column from left to right

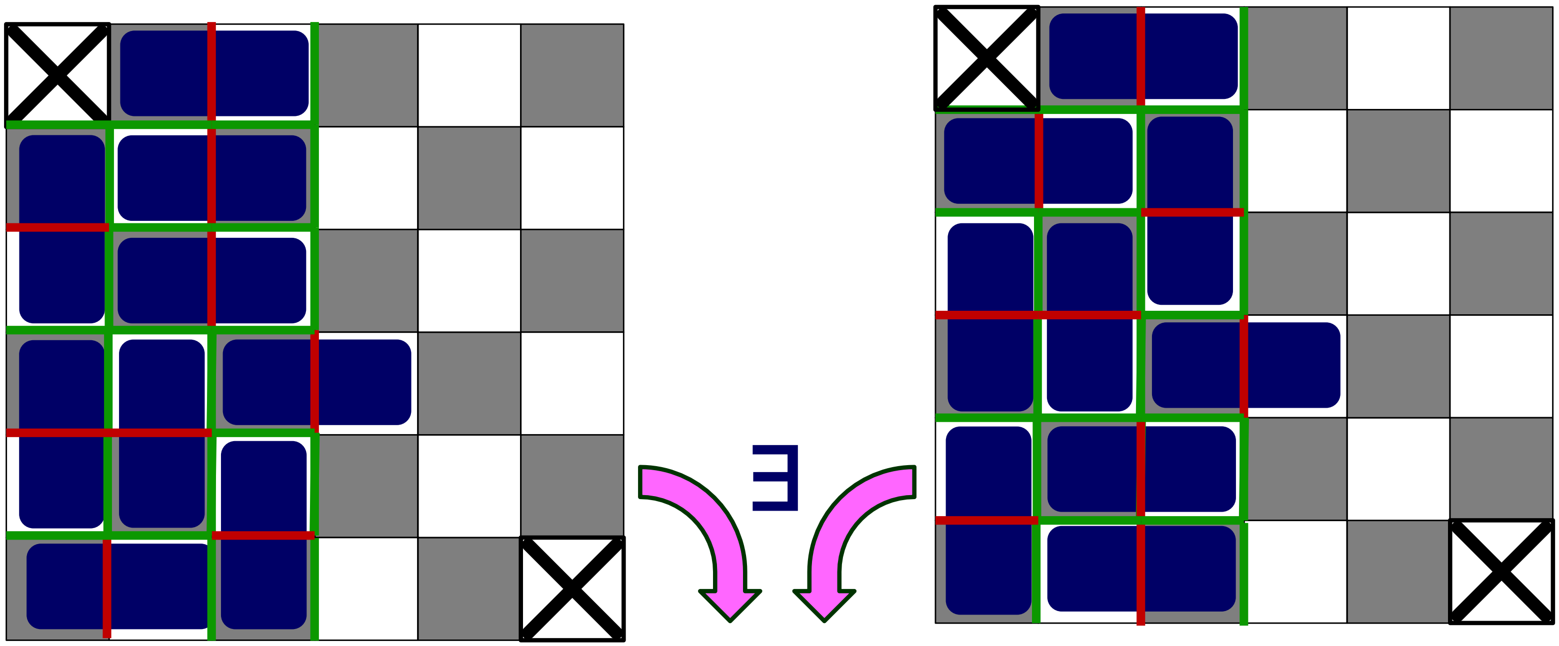
Observation

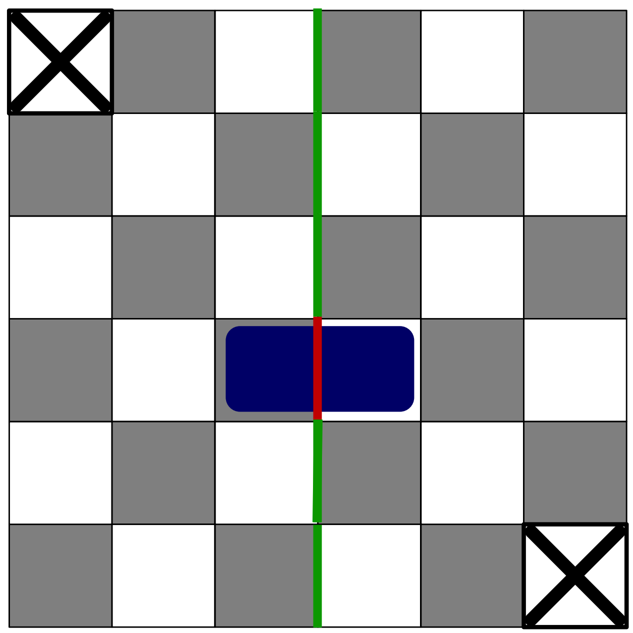
- When tiling column, only need to know which rows have tiles jutting in from left





Abstraction Via Quantification





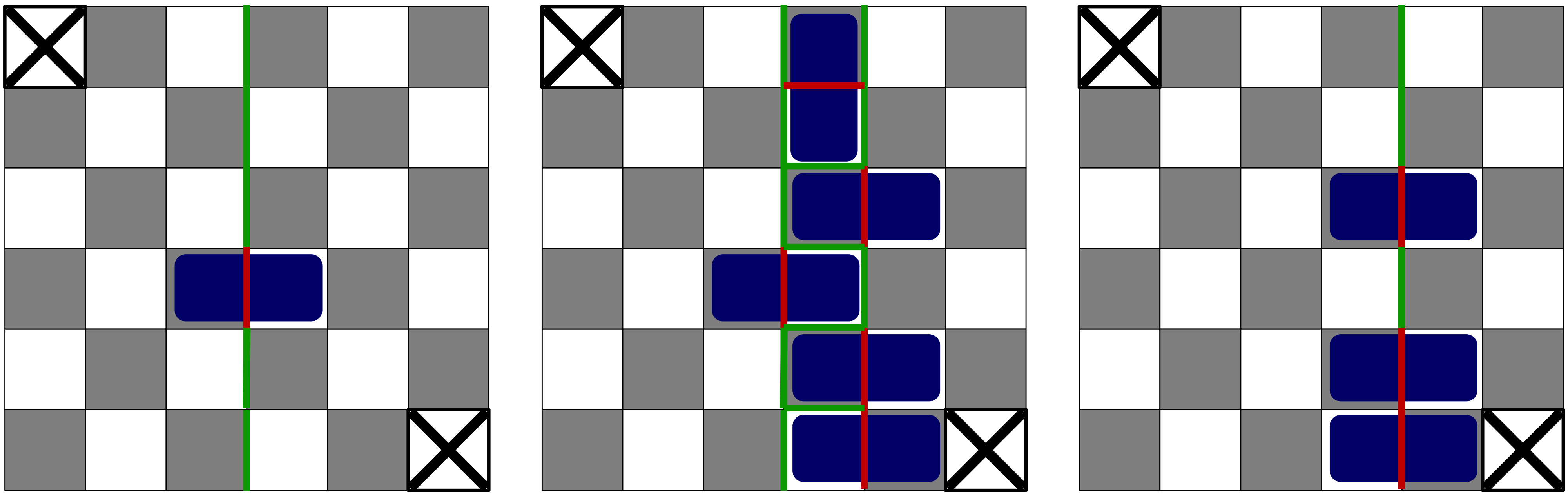
Scanning "State"

- Existentially quantify variables defining earlier boundaries in scan
- X_i = Value of vertical variables to right of column i

Symbolic Computation of State Sets

State at column $j-1$ | **Column j transition** | **State at column j**

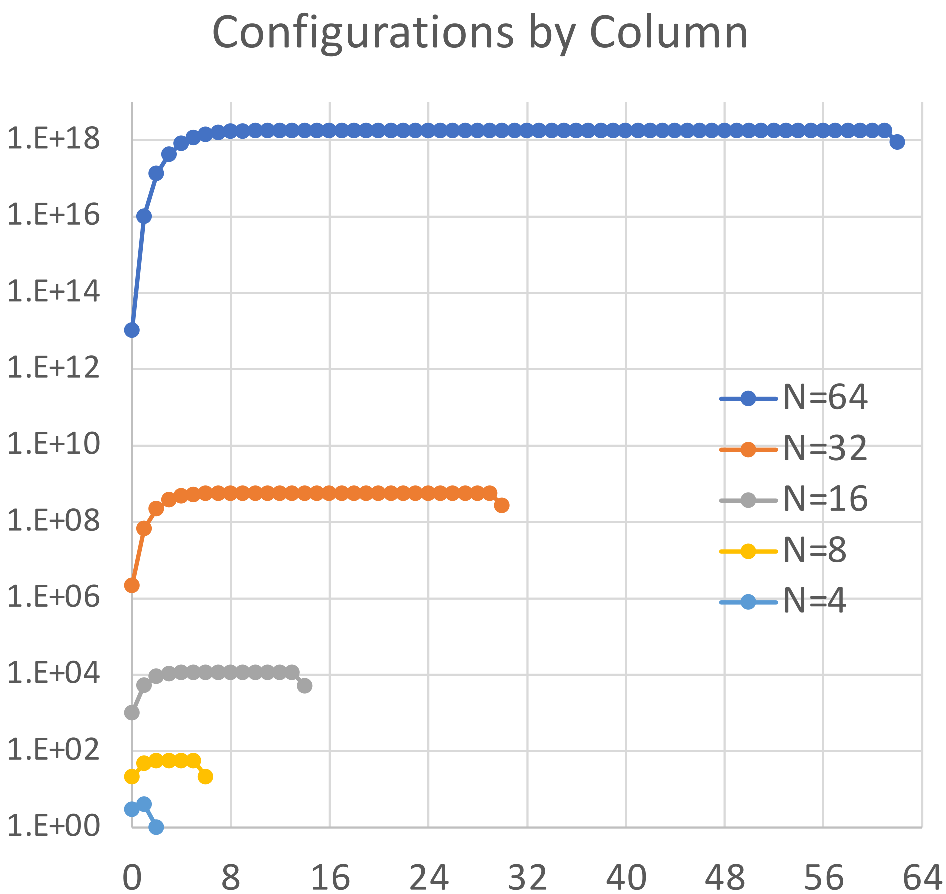
$\sigma_{j-1}(X_{j-1})$ | $T_j(X_{j-1}, Y_j, X_j)$ | $\sigma_j(X_j)$

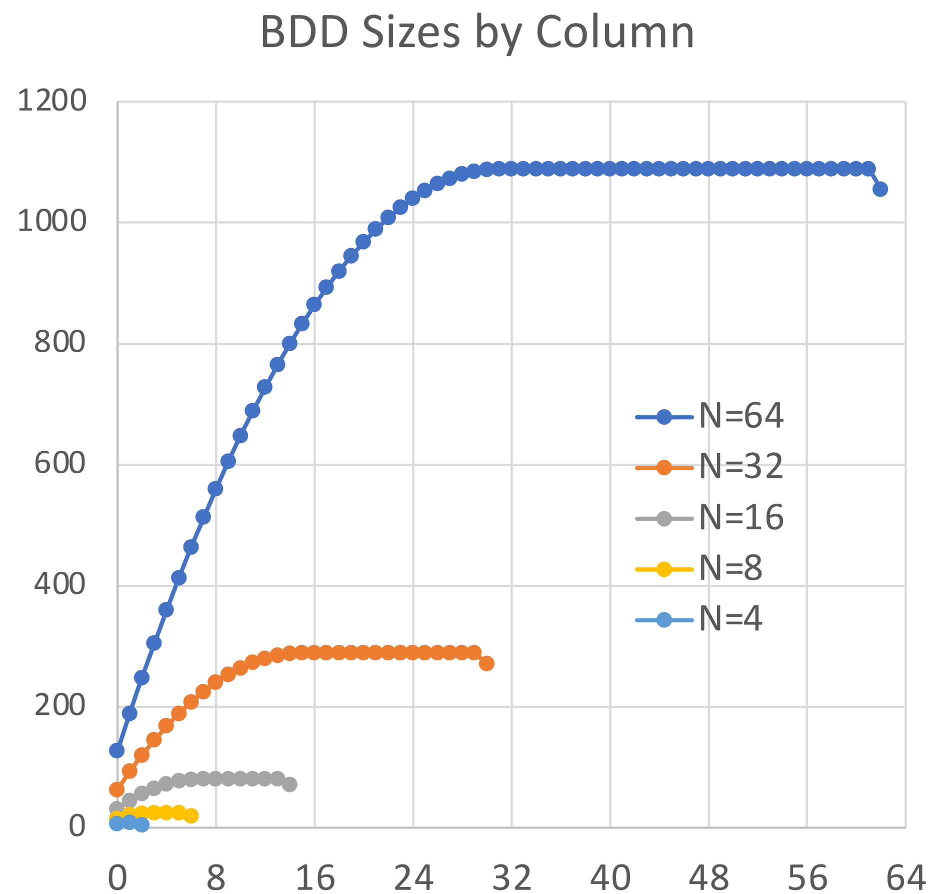


$$\sigma_j(X_j) \;=\; \exists X_{j-1}\left[\sigma_{j-1}(X_{j-1}) \wedge \exists Y_j\, T_j(X_{j-1}, Y_j, X_j)\right]$$

- **Does not redefine underlying problem**
- **Way to order conjunctions and quantifications**

Representing State Sets





- **Number of configurations ~ 2^N**
- **BDD representation ~ N^2**
- **Reaches fixed point after column $N / 2$**

Chess Proof Complexity

- **Problem size ~ N^2**
- **Proof size ~ $N^{2.68}$**

Observations

Key Insight

- **Sinz, Biere, and Jussila**
- **Capture underlying logic of BDD algorithms as ER proofs**

Our Contributions

- **Integrate proof generation with Apply operations**
- **Handle arbitrary existential quantification**
- **Demonstrate on variety of benchmarks**
  - Mutilated chessboard
  - Pigeonhole principle
  - Parity formulas
  - Urquhart formulas

Further Work

Higher Performance Implementation

- **Integrate into existing BDD package**

More Automation

- **Variable ordering**
- **Conjunction and quantification scheduling**

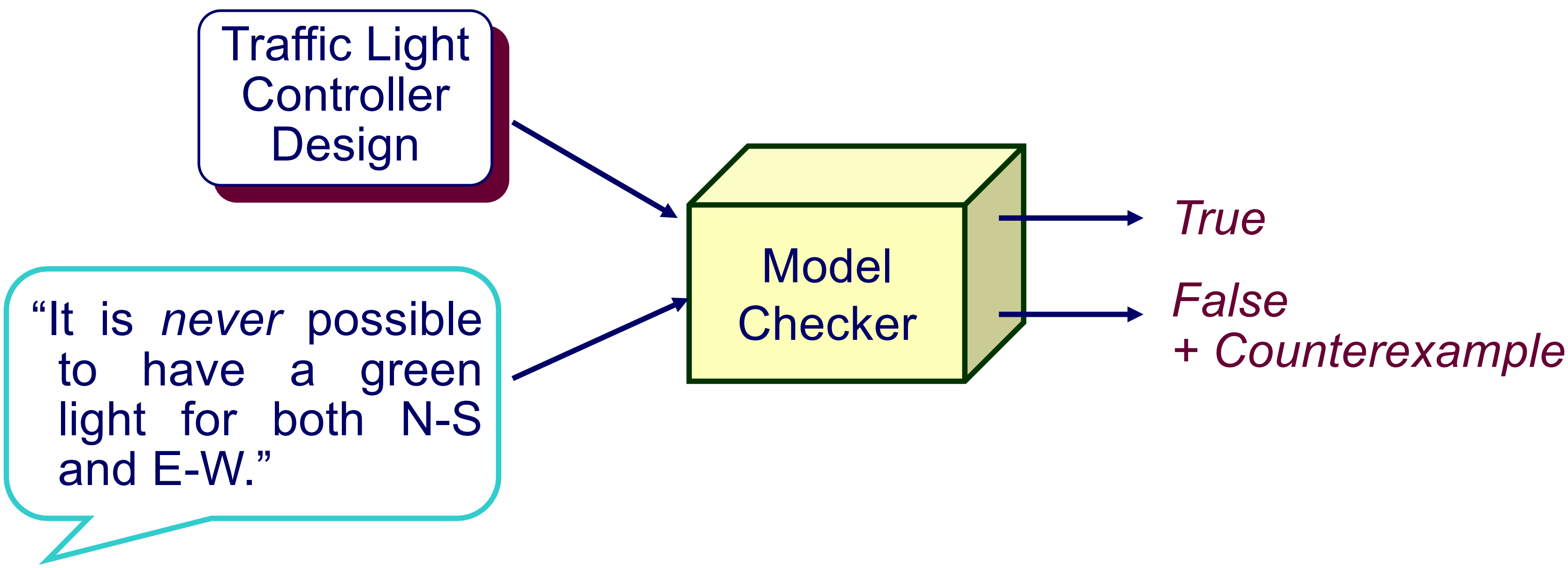
Apply to Other Problems

- **Quantified Boolean Formulas**
  - Extend Boolean formulas with existential and universal quantifiers
  - Have formulated approach
- **Model checking**
- **Model counting**

Temporal Logic Model Checking

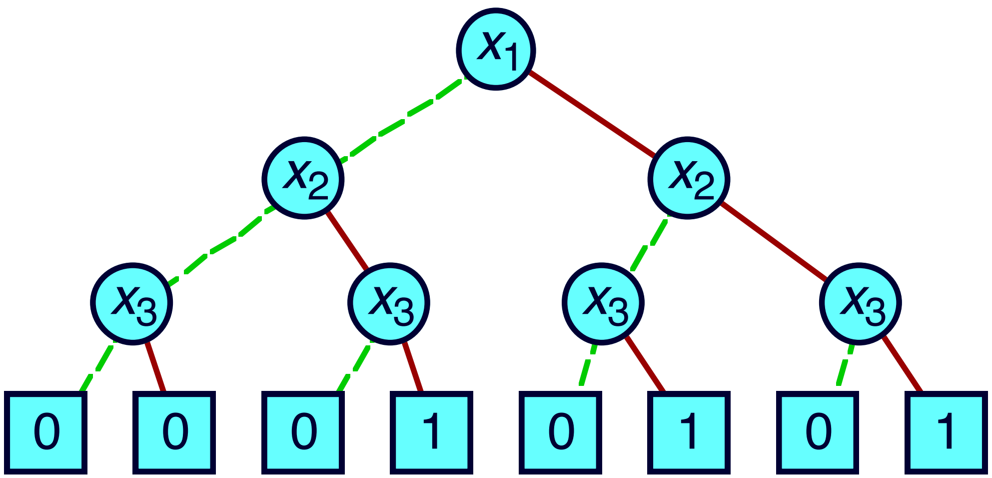
Verify Reactive Systems

- **Construct state machine representation of reactive system**
  - Nondeterminism expresses range of possible behaviors
  - “Product” of component state machines
- **Express desired behavior as formula in temporal logic**
- **Determine whether or not property holds**

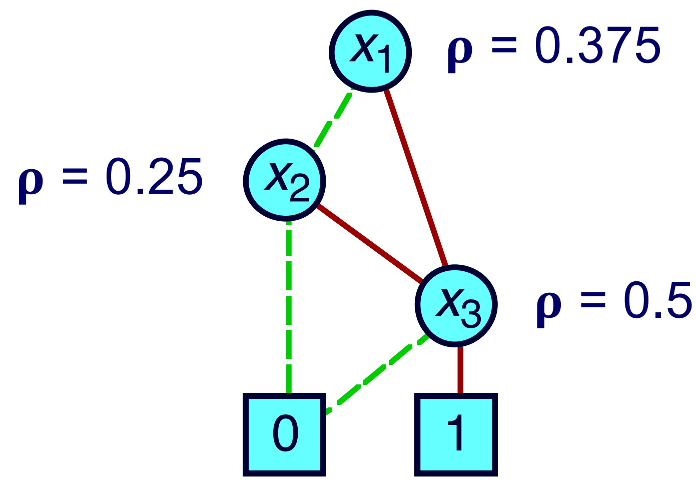


Model Counting with BDDs

Initial Graph



Reduced Graph



Compute *density* of function

- Fraction of paths leading to leaf 1
- Average of densities of children

But, how to generate a proof?