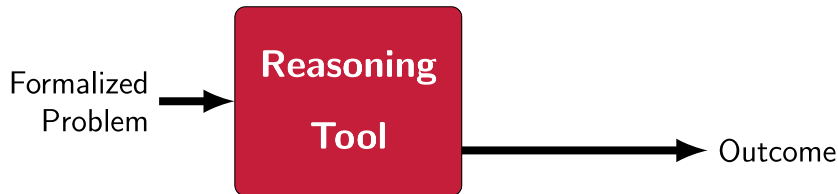
Binary Decision Diagrams Applied to Verifiable Automated Reasoning

Randal E. Bryant

Carnegie Mellon University

2021

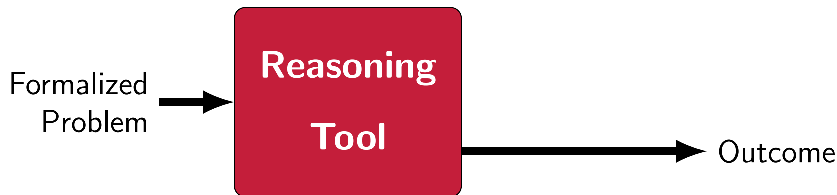
Automated Reasoning Programs



Example Applications

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

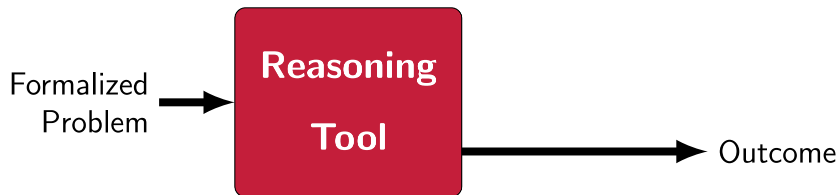
Automated Reasoning Programs



Can We Trust the Results?

- *No!*
- Complex software with many optimizations

Automated Reasoning Programs



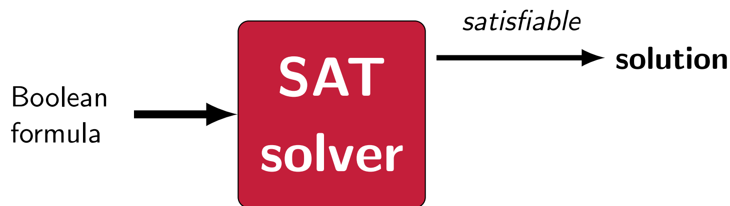
Can We Trust the Results?

- *No!*
- Complex software with many optimizations

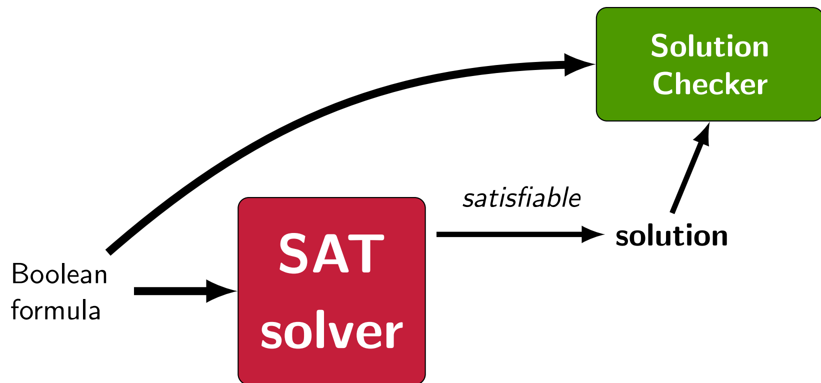
Is This a Problem?

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

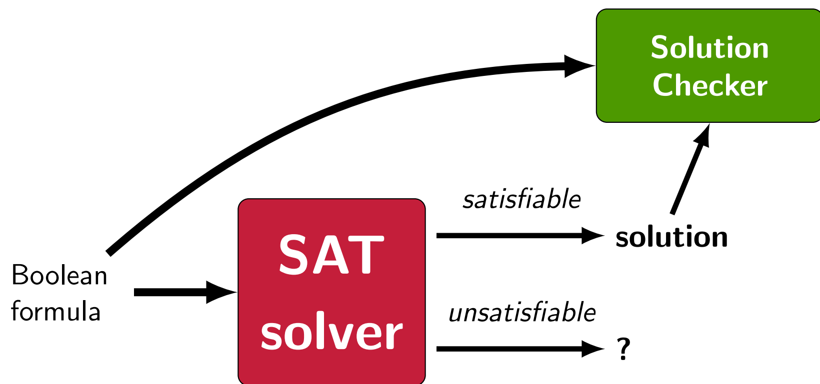
Boolean Satisfiability Solvers



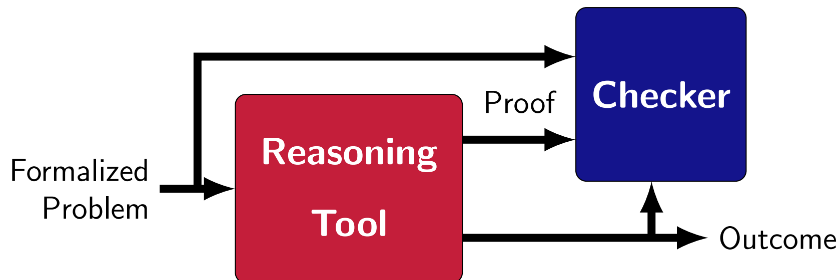
Boolean Satisfiability Solvers



Boolean Satisfiability Solvers



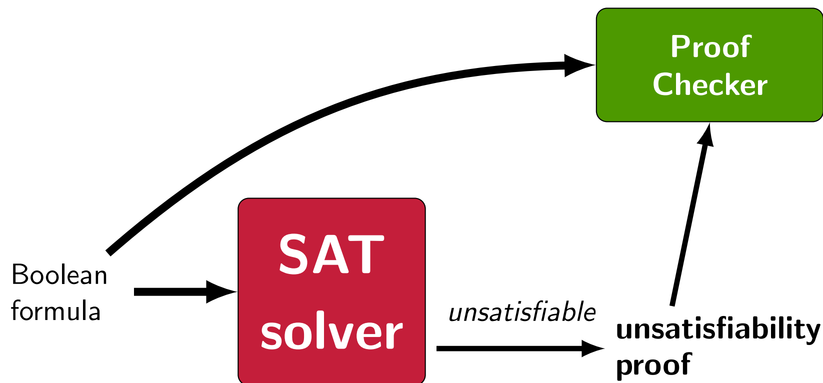
Proof-Generating Automated Reasoning Programs



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

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

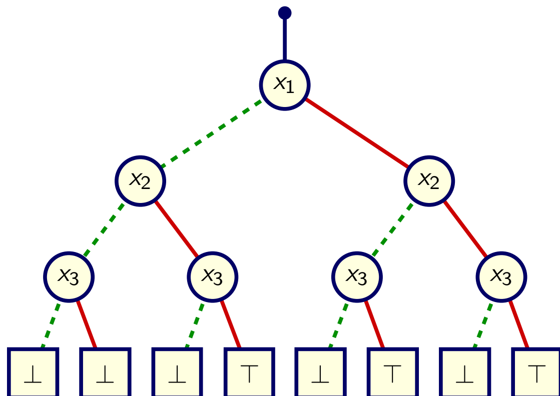
Boolean Function Representations

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	$\top$

- Size $= O(2^n)$

Decision Tree

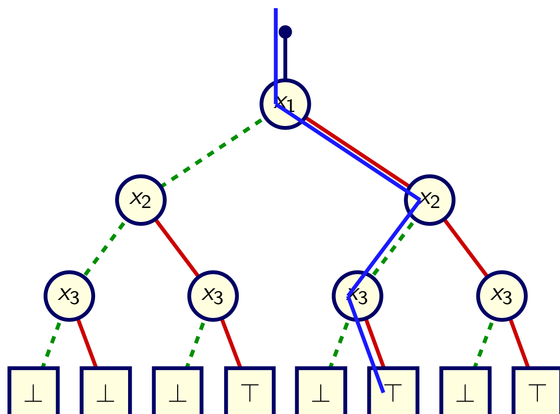


Boolean Function Representations

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



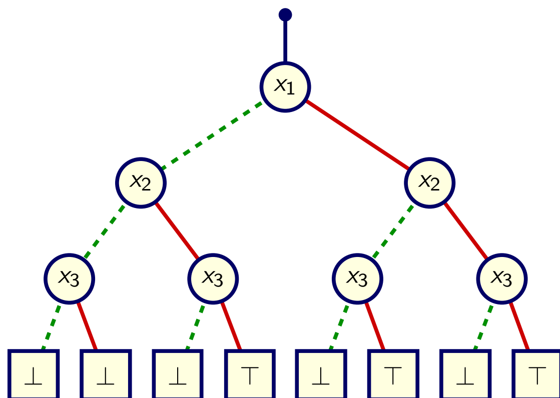
- Size $= O(2^n)$
- Assignment defines path from root to leaf

Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	$\perp$
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



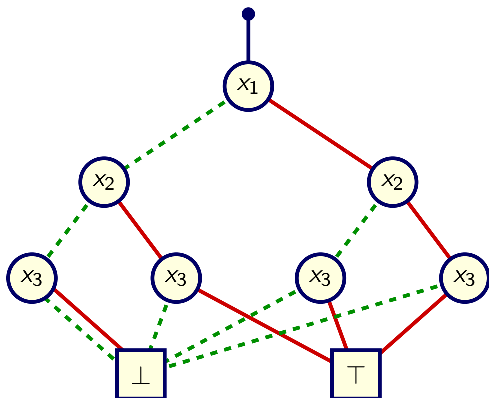
- Merge isomorphic nodes
- Eliminate redundant tests

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



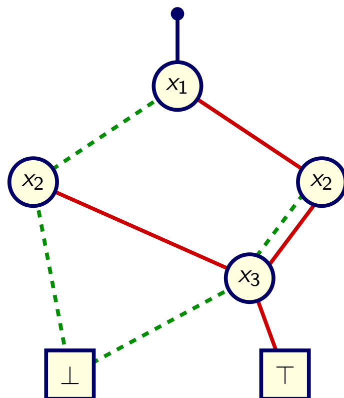
- Merge isomorphic nodes
- Eliminate redundant tests

Reducing to Canonical Form

Truth Table

x_1	x_2	x_3	f
0	0	0	$\bot$
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



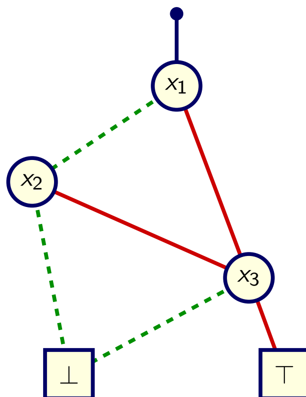
- Merge isomorphic nodes
- Eliminate redundant tests

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



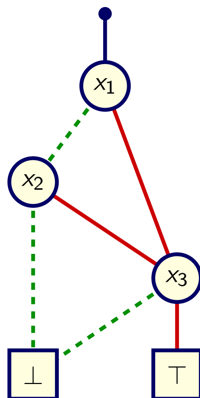
- Merge isomorphic nodes
- Eliminate redundant tests

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

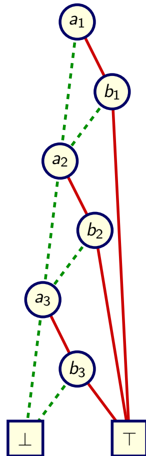


- Canonical representation of Boolean function
- No further simplifications possible

Effect of Variable Ordering

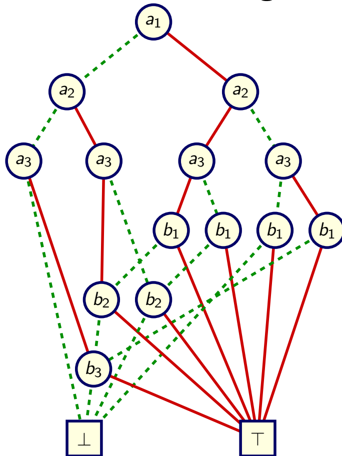
$$(a_1 \land b_1) \lor (a_2 \land b_2) \lor (a_3 \land b_3)$$

Good Ordering



- Linear growth

Bad Ordering



- Exponential growth

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

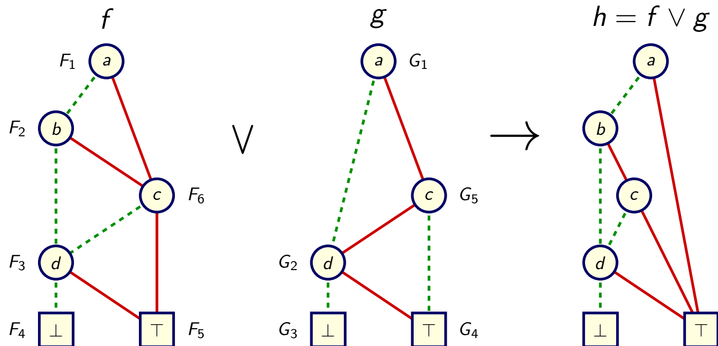
Key Algorithmic Properties

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

Apply Algorithm

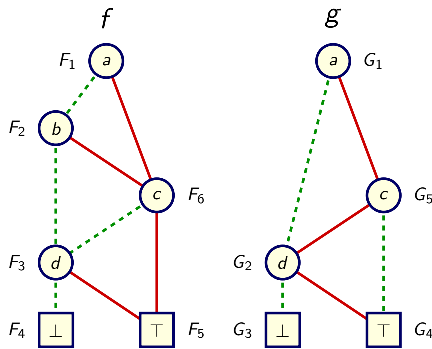
$h \leftarrow f \odot g$

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

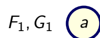


Apply Algorithm Recursion

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

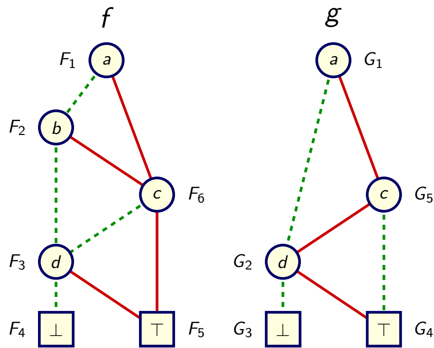


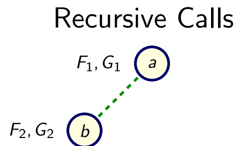
Recursive Calls



Apply Algorithm Recursion

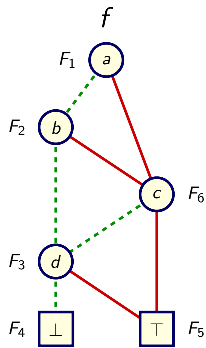
- Recurse through argument graphs
- Stop when hit terminal case
- Save results in cache to reuse when hit same arguments

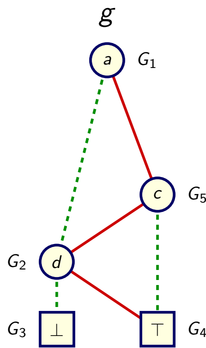


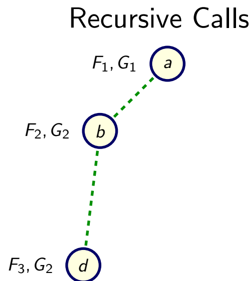


Apply Algorithm Recursion

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

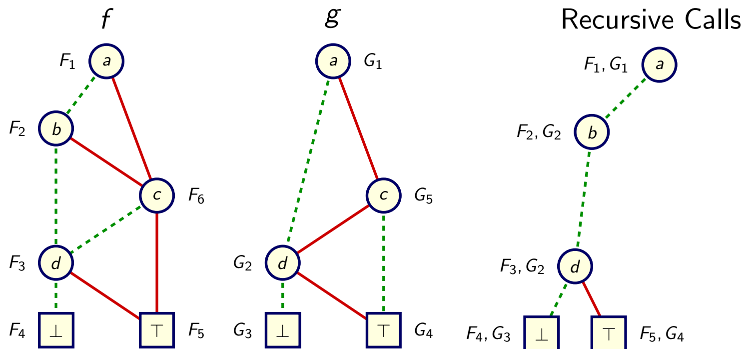






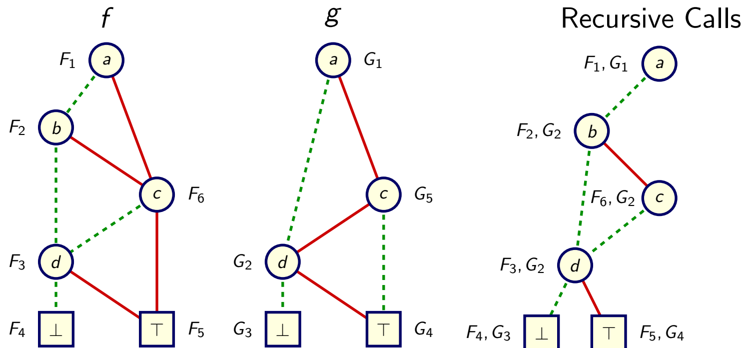
Apply Algorithm Recursion

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



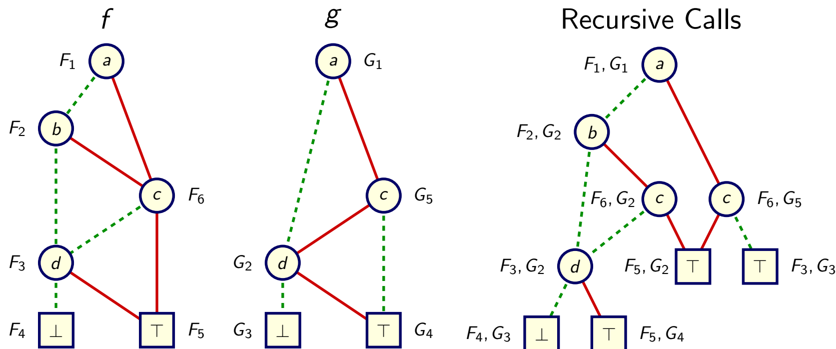
Apply Algorithm Recursion

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



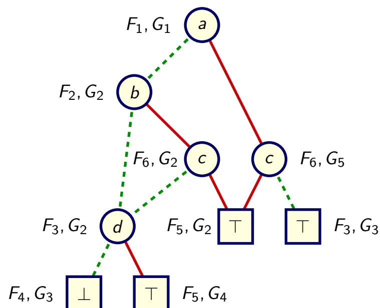
Apply Algorithm Recursion

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

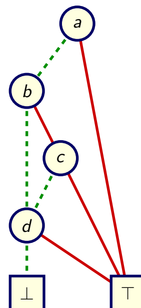


Apply Algorithm Result

Recursive Calls



Reduced Result



Proof-Generating BDD Applications

Enhance reliability of formal reasoning tools

Boolean Satisfiability Solvers

- Pure Boolean approaches
- Adding pseudo-Boolean reasoning

Quantified Boolean Formula Solvers

- Must generate proofs for both true and false formulas

Proof Format Background

Clauses

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

Resolution Principle

$$\frac{\neg u \vee v \vee w \qquad \neg w \vee x \vee \neg z}{(\neg u \vee v) \vee (x \vee \neg z)}$$

- Robinson, 1966
- Generalization of implication
- See https://en.wikipedia.org/wiki/Resolution_(logic)

Clausal Proof

Step	Clause	Antecedents	Formula
1	$\neg v \vee w$		$v \rightarrow w$
2	$\neg v \vee \neg w$		$v \rightarrow \neg w$
3	v		v
4	$\neg v$	1, 2	$\neg v$
5	$\bot$	3, 4	$v \wedge \neg v$

Steps 1–3: Input clauses
Steps 4–5: Derived clauses

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

Extended Resolution (ER)

- Tseitin, 1967

Can introduce extension variables

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

Extension variable e becomes shorthand for formula F

- Repeated use can yield exponentially smaller proof

Extended Resolution Example

Example: Prove following set of constraints unsatisfiable

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

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

Constraint	Clauses
$u \land v \rightarrow e$	$e \lor \neg u \lor \neg v$
$e \rightarrow u$	$\neg e \lor u$
$e \rightarrow v$	$\neg e \lor v$

Extended Resolution Proof

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

Input clauses (steps 1–4)

Defining clauses (steps 5–7)

Derived clauses (steps 8–15)

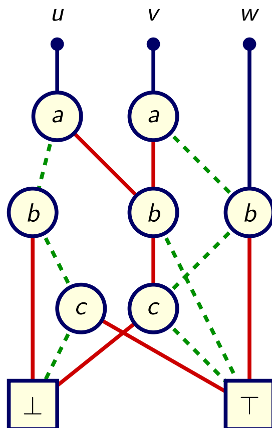
Reduced Ordered Binary Decision Diagrams (BDDs)

Representation

- Canonical representation of set of Boolean functions
- Each root node u, v, w denotes a Boolean function

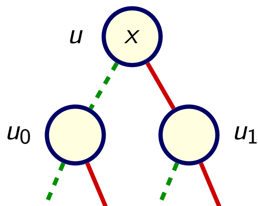
Algorithms

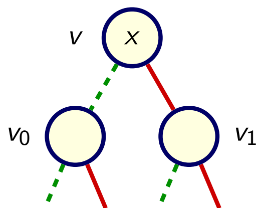
- Apply(u, v, op)
  - Boolean operation op
    - e.g., $\wedge$, $\vee$
  - Generates BDD representation of $u \; op \; v$
- EQuant(u, X)
  - X set of variables
  - BDD representation of $\exists X u$



Apply Algorithm Recursion

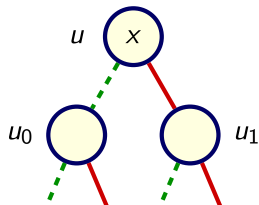
Apply($u, v, \wedge$)

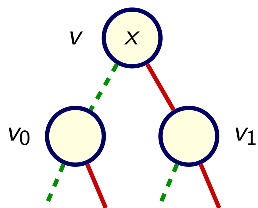




Apply Algorithm Recursion

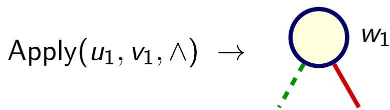
$\text{Apply}(u, v, \wedge)$



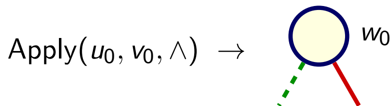


Recursion

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

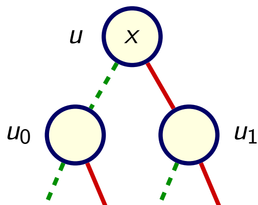


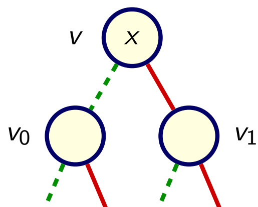
$\text{Apply}(u_0, v_0, \wedge) \rightarrow$



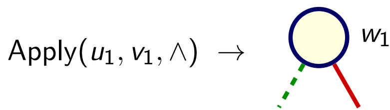
Apply Algorithm Recursion

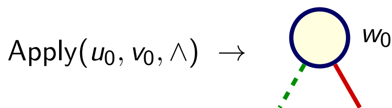
$\text{Apply}(u, v, \wedge)$



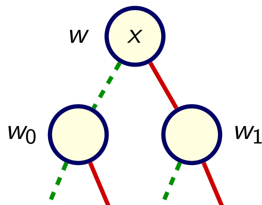


Recursion



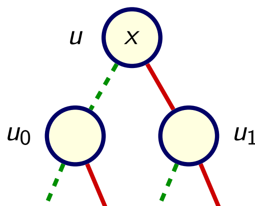


Result



Generating ER Proofs

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



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

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

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)	$\neg x \rightarrow (u \rightarrow u_0)$
HD(v)	$x \rightarrow (v \rightarrow v_1)$	LD(v)	$\neg x \rightarrow (v \rightarrow v_0)$
HU(w)	$x \rightarrow (w_1 \rightarrow w)$	LU(w)	$\neg x \rightarrow (w_0 \rightarrow w)$

Resolution Steps

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

Quantification Operations

Operation 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
  - 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 $C_I \vdash \bot$

Structure of Overall Proof

Input Variables

- Generate BDD variable for each input variable

Input Clauses

- For each input clause $C \in C_I$, generate BDD representation u
  - Using Apply with $\vee$ operation
- Generate proof $C \vdash u$
  - Sequence of resolution steps based on linear structure of BDD

Combine Top-Level BDDs

- $\text{Apply}(u, v, \wedge) \rightarrow w$
  - Combine proofs $C_I \vdash u$, $C_I \vdash v$ and $u \wedge v \rightarrow w$ to get $C_I \vdash w$
- $\text{EQuant}(u, X) \rightarrow w$
  - Combine proofs $C_I \vdash u$ and $u \rightarrow w$ to get $C_I \vdash w$

Completion

- When $\text{Apply}(u, v, \wedge) \rightarrow \bot$ have proof $C_I \vdash \bot$

PGBDD (Proof-Generating BDDs)

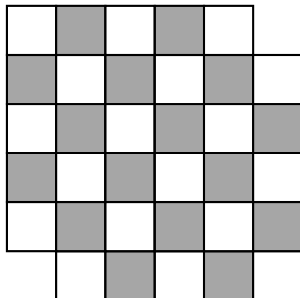
Implementation

- 2000 lines Python code (slow!)
- BDD package + proof generator
- https://github.com/rebryant/pgbdd-artifact

Benchmark Generators

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

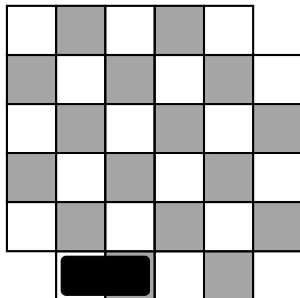
Mutilated Chessboard Problem



Definition

- $N \times N$ chessboard with 2 corners removed
- Cover with tiles, each covering two squares

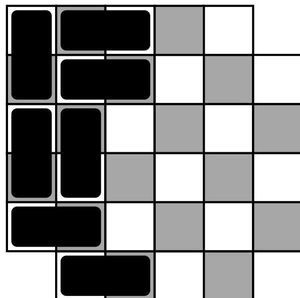
Mutilated Chessboard Problem



Definition

- $N \times N$ chessboard with 2 corners removed
- Cover with tiles, each covering two squares

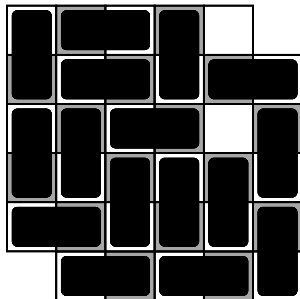
Mutilated Chessboard Problem



Definition

- $N \times N$ chessboard with 2 corners removed
- Cover with tiles, each covering two squares

Mutilated Chessboard Problem



Definition

- $N \times N$ chessboard with 2 corners removed
- Cover with tiles, each covering two squares

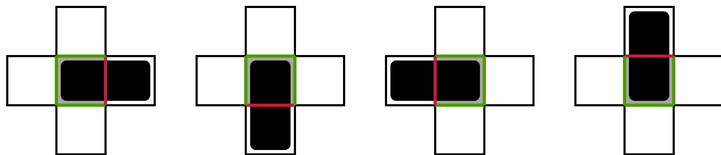
Solutions

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

Proof

- All resolution proofs of exponential size

Encoding as SAT Problem



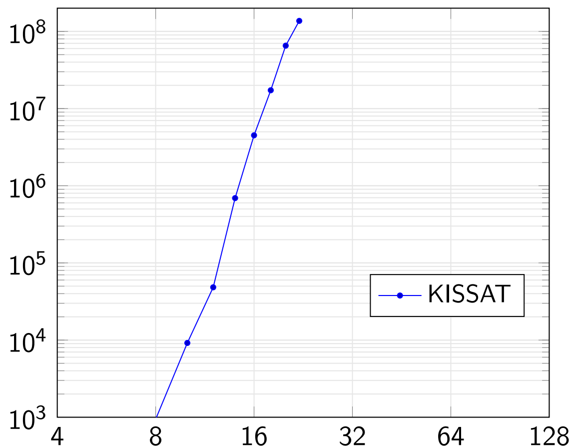
Boolean variable for each boundary between two squares

- $(N-1) \cdot N - 2$ vertical boundaries $x_{i,j}$
- $(N-1) \cdot N - 2$ horizontal boundaries $y_{i,j}$

Constraints

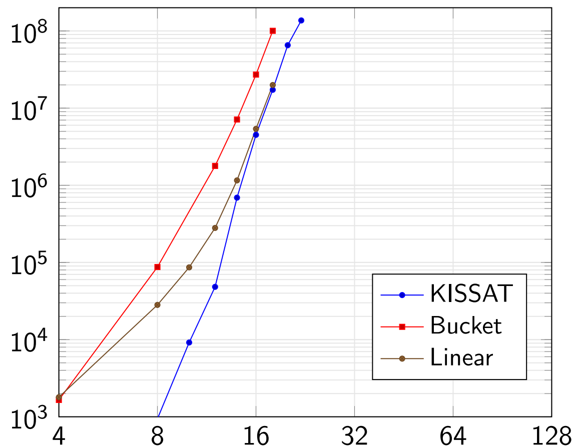
- For each square, exactly one of its boundary variables $= 1$

Chess Proof Complexity: KISSAT



- Winner of 2020 SAT competition
- Requires 12.6 hours for $N = 22$.

Chess Proof Complexity: Earlier BDD-Based Approaches



- Linear: No quantification (Sinz & Biere, 2006)
- Bucket: Eliminate variables from top of BDD downward (Jussila, Sinz, & Biere, 2006)

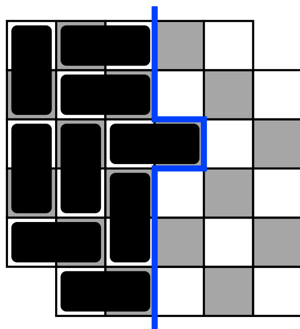
Column Scanning

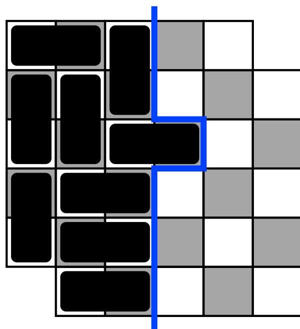
Scanning

- Add tiles for each column from left to right

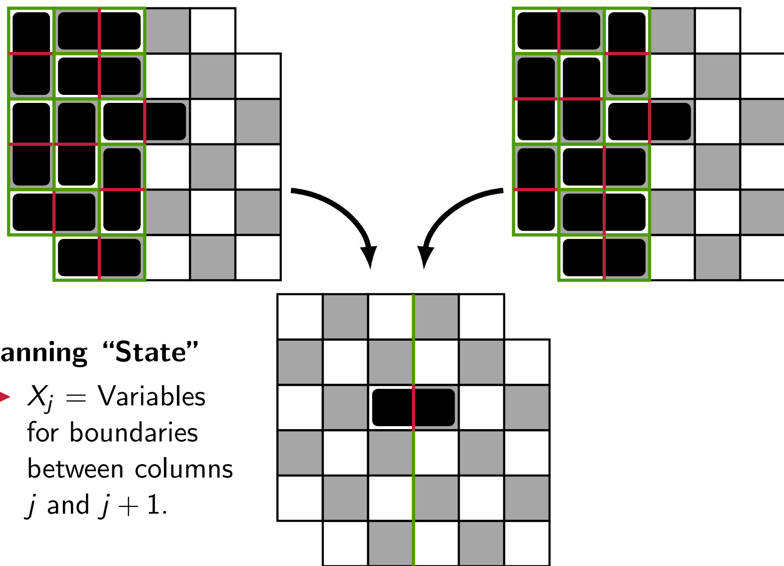
Observation

- When placing tiles in column, only need to know which squares are already occupied





Abstraction Via Quantification



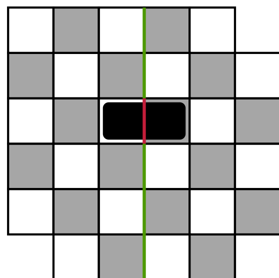
Scanning "State"

- X_j = Variables for boundaries between columns j and $j+1$.

Symbolic Computation of State Sets

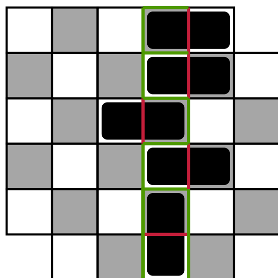
State at column j-1

$\sigma_{j-1}(X_{j-1})$



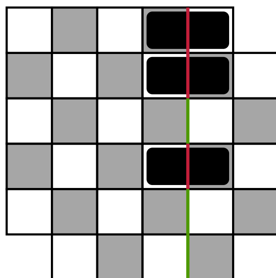
Column j transition

$T_j(X_{j-1}, Y_j, X_j)$



State at column j

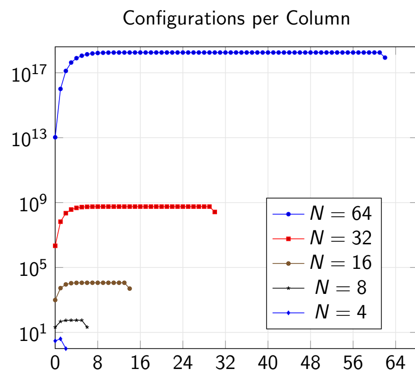
$\sigma_j(X_j)$

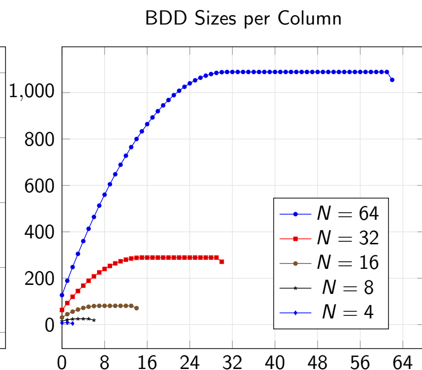


$$\sigma_j(X_j) = \exists X_{j-1}[\sigma_{j-1}(X_{j-1}) \wedge \exists Y_j \, T_j(X_{j-1}, Y_j, X_j)]$$

- Does not redefine underlying problem
- Way to order conjunctions and quantifications
- Requires quantification ordering to differ from BDD variable ordering

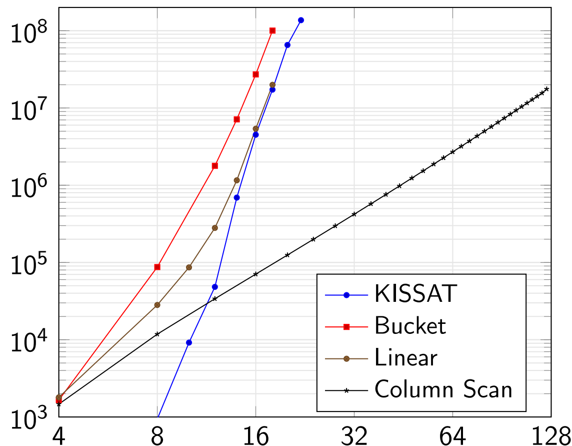
Representing State Sets





- Number of configurations $\sim 2^N$
- BDD representation $\sim N^2$

Chess Proof Complexity: Column Scanning



- Problem size $\sim N^2$
- Proof size $\sim N^{2.7}$

Beyond Standard Boolean Reasoning

PGBDD Operations

- $\vee$ operation to build BDD representations of clauses
- $\wedge$ operation to form conjunctions of clauses
- $\exists$ operation to quantify out variables

Adding Other Reasoning Methods

- Generalize based on *implication redundancy*
- Given existing terms $u_1, u_2, \ldots, u_k$
  - Each represented by BDD root node u_i
  - Have proved that $C_I \vdash u_i$
- Generate new result $w = Op(u_1, u_2, \ldots, u_k)$
  - Generate proof $u_1 \wedge u_2 \wedge \cdots \wedge u_k \rightarrow w$
  - Combine proofs to get $C_I \vdash w$

Pseudo-Boolean Formulas

Forms

- Equations

$$\sum_{1 \leq i \leq n} a_i \, x_i \;\; = \;\; b$$

- Constraints

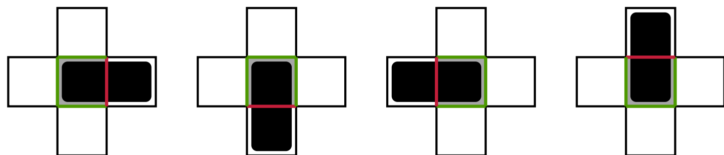
$$\sum_{1 \leq i \leq n} a_i \, x_i \;\; \geq \;\; b$$

- a_i, b integer constants
- x_i 0-1 valued variables

Capabilities

- Compact encoding of problems
- Powerful solution techniques
  - Gaussian elimination
  - Fourier-Motzkin elimination

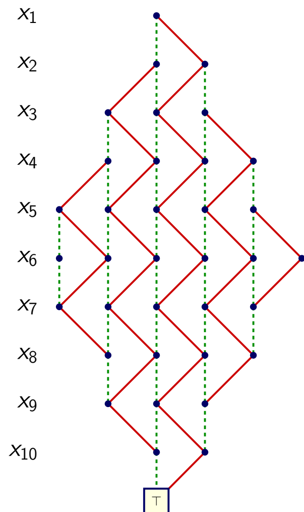
Pseudo-Boolean Encoding of Mutilated Chessboard



For every square i, j:

$$x_{N(i,j)} + x_{E(i,j)} + x_{S(i,j)} + x_{W(i,j)} \quad = \quad 1$$

Representing Pseudo-Boolean Formulas with BDDs



- Example equation:

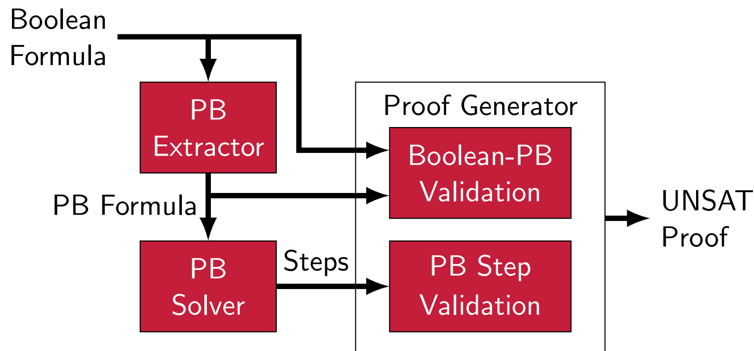
$$\begin{array}{l} +x_1 + x_3 + x_5 + x_7 + x_9 \\ -x_2 - x_4 - x_6 - x_8 - x_{10} \end{array} = 0$$

- BDD size $O(a_{\max}\, n^2)$

$$a_{\max} = \max_{1 \leq i \leq n} |a_i|$$

  - Independent of variable ordering

Integrating Pseudo-Boolean Reasoning into Proof-Generating SAT Solver



Validating Each Step:

- Given formulas F_1 and F_2
- Validate $F_1 \wedge F_2 \rightarrow F_1 + F_2$

PGPBS (Proof-Generating Pseudo-Boolean Solver)

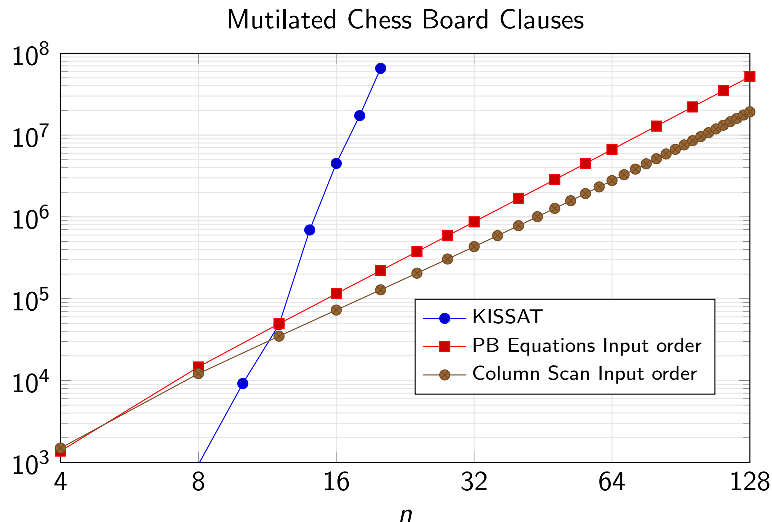
Implementation

- Augmented version of PGBDD
- 1500 lines Python code for solving PB constraints
- `https://github.com/rebryant/pgpbs-artifact`

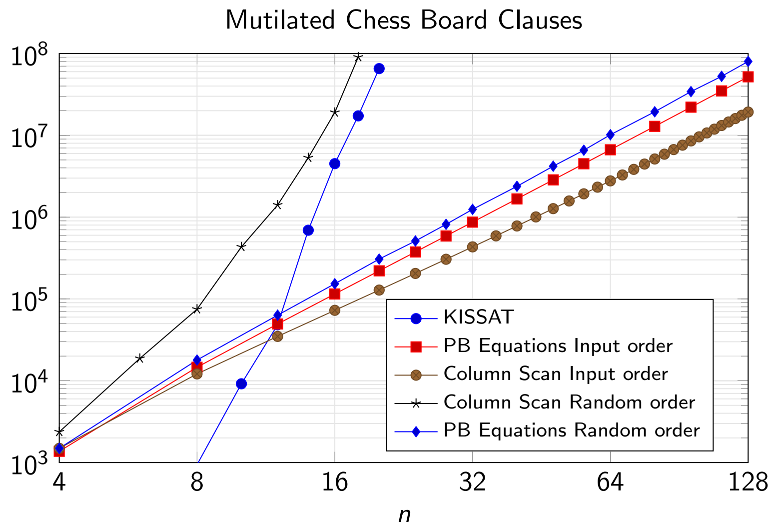
Constraint Extraction

- CNF file input
- Detects XOR, at-most-one, at-least-one, exactly-one constraints
- Heuristic methods
- Generates schedule
  - How clauses can be grouped into constraints

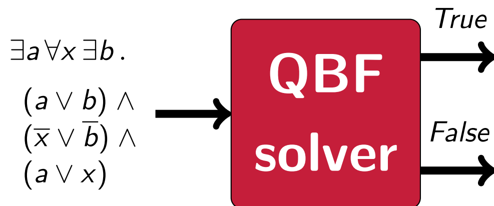
Mutilated Chessboard Revisited



Mutilated Chessboard Revisited



Quantified Boolean Formulas (QBF)



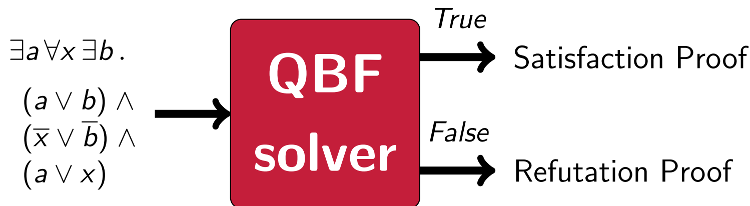
Adding Quantifiers

- Can encode many reasoning problems in compact form
- Much more difficult to solve

Solving

- Assume fully quantified
  - no free variables
- No satisfying assignment
  - Formula is either true or false

Quantified Boolean Formulas (QBF)



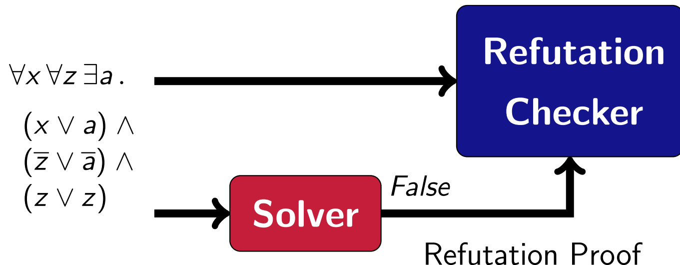
Adding Quantifiers

- Can encode many reasoning problems in compact form
- Much more difficult to solve

Solving

- Assume fully quantified
  - no free variables
- No satisfying assignment
  - Formula is either true or false

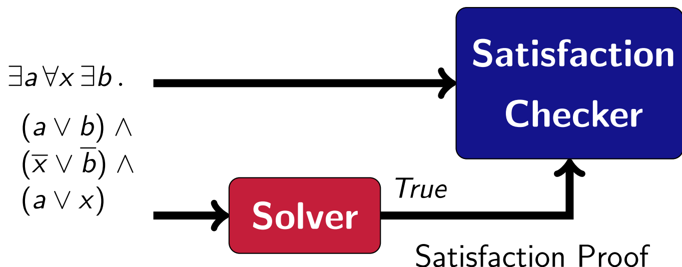
Handling False Formulas



Refutation Proof

- Similar to proofs of unsatisfiability by SAT checker
- Steps leading to empty clause
  - Add new clauses by resolution
  - Eliminate universal variables
- Implemented by *some*, but not all QBF solvers

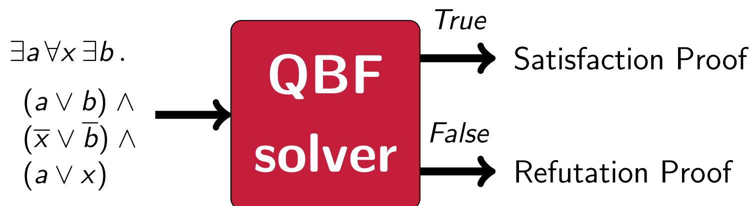
Handling True Formulas



Satisfaction Proof

- No approach in widespread use
- Implemented by very few other solvers

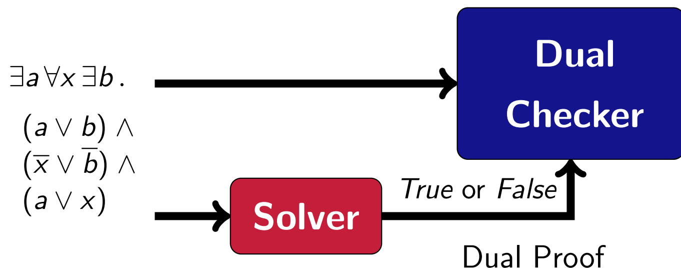
What We Want



Proof-Generating QBF Solver

- Both satisfaction and refutation proofs
- Length of proof linear in runtime of solver
- Checking time near-linear in length of proof

What We *Really* Want



Dual Proof

- Unified framework for satisfaction and refutation proofs
- QBF solver can generate proof as it operates
  - Before it determines whether formula is true or false
- Single checker can handle both cases
  - Single logical framework

QRAT Proof System

- Heule, Seidl, Biere 2014

Clausal Proof System

- Developed to check correctness of QBF preprocessors
- Start with input clauses
- Proof rules to add and delete clauses

Refutation Proof

- Add clauses until generate empty clause
- Logical contradiction

Satisfaction Proof

- Add and delete clauses until have empty set of clauses
- Logical tautology

QRAT Logical Basis

Proof Structure

- Input formula Φ_I
- Each clause addition or deletion step yields modified QBF

$$\Phi_I = \Phi_1, \Phi_2, \ldots, \Phi_t$$

Refutation Proof

- Each step must be *truth preserving*: $\Phi_i \rightarrow \Phi_{i+1}$

$$\Phi_I = \Phi_1 \rightarrow \Phi_2 \rightarrow \cdots \rightarrow \Phi_t = \bot$$

Satisfaction Proof

- Each step must be *falsehood preserving*: $\Phi_i \leftarrow \Phi_{i+1}$

$$\Phi_I = \Phi_1 \leftarrow \Phi_2 \leftarrow \cdots \leftarrow \Phi_t = \top$$

Dual Proof

- Each step must be *equivalence preserving*: $\Phi_i \leftrightarrow \Phi_{i+1}$

$$\Phi_I = \Phi_1 \leftrightarrow \Phi_2 \leftrightarrow \cdots \leftrightarrow \Phi_t \in \{\bot, \top\}$$

PGBDDQ (Proof-Generating BDDs with Quantification)

BDD-Based QBF solver

- Generates dual, refutation, or satisfaction proofs
- https://github.com/rebryant/pgbddq-artifact

Required Proof Capabilities

- Conjunction
  - $w \leftarrow \text{Apply}(u, v, \wedge)$
  - $u \wedge v \leftrightarrow w$
- Existential quantification
  - $w \leftarrow \text{EQuant}(u, x)$
  - $\exists x\, u \leftrightarrow w$
- Existential quantification
  - $w \leftarrow \text{UQuant}(u, x)$
  - $\forall x\, u \leftrightarrow w$

Further Work

Higher Performance Implementation

- Extend existing BDD package

More Automation

- Variable ordering
- Conjunction and quantification scheduling

Apply to Other Problems

- Dependency QBF
- Model checking
- Model counting

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," *in submission*, 2021