Quantified Satisfiability and Its Synthesis & Verification Applications

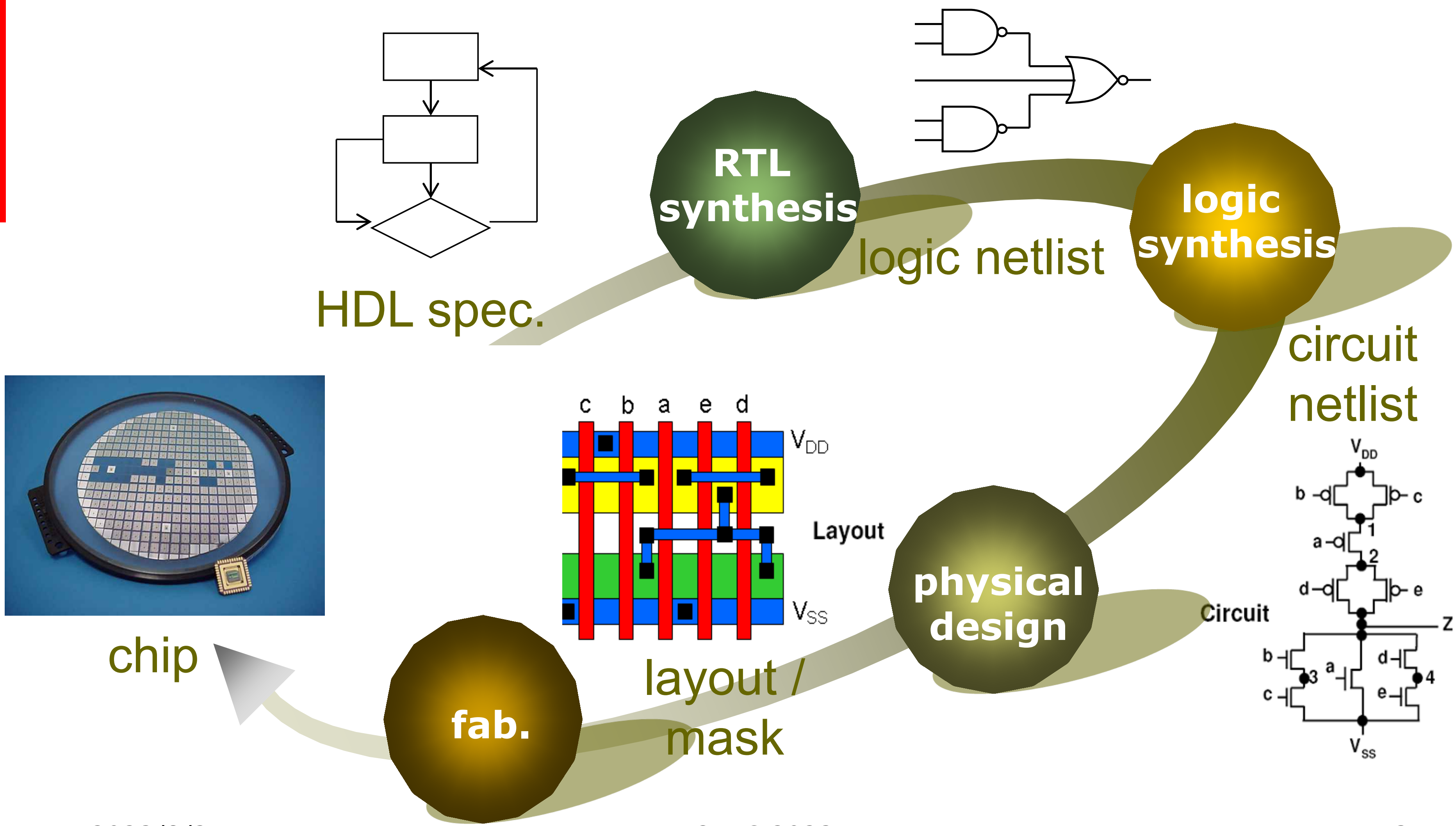
Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

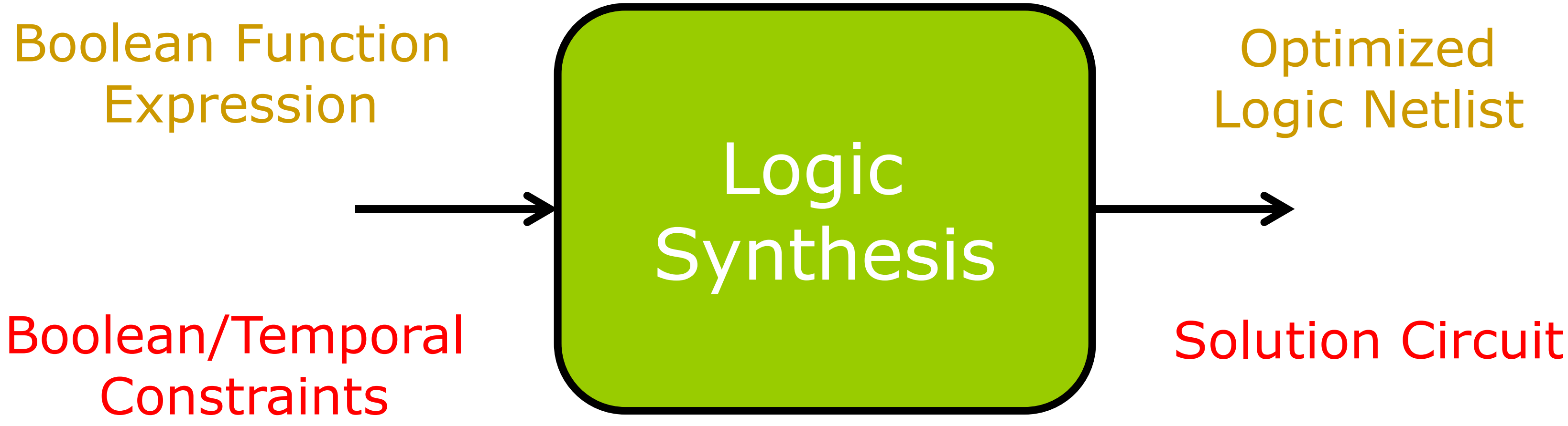
Outline

- Logic synthesis & verification
- Boolean function representation
- Propositional satisfiability & applications
- Quantified satisfiability & applications
- Beyond quantified Boolean satisfiability
  - Dependency quantified Boolean formula
  - Second-order quantified Boolean formula
  - #SAT (model counting)
  - Stochastic Boolean satisfiability
  - Dependency stochastic Boolean satisfiability

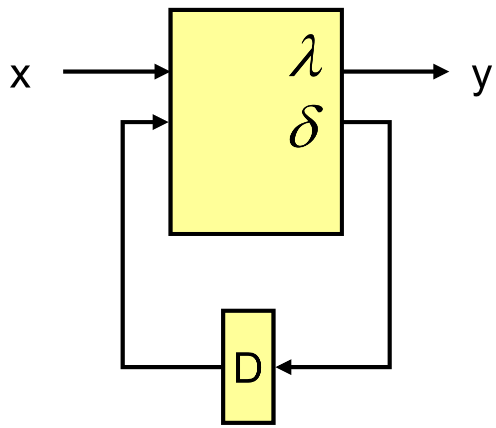
IC Design Flow



Logic Synthesis

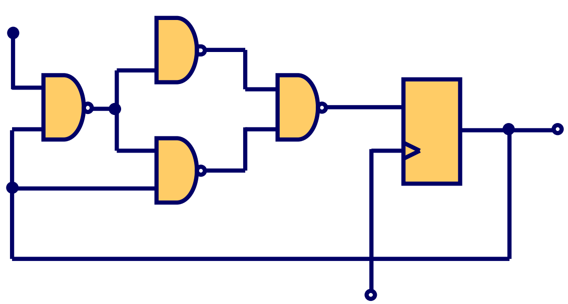


Logic Synthesis



Given: Functional description of finite-state machine F(Q,X,Y,δ,λ) where:

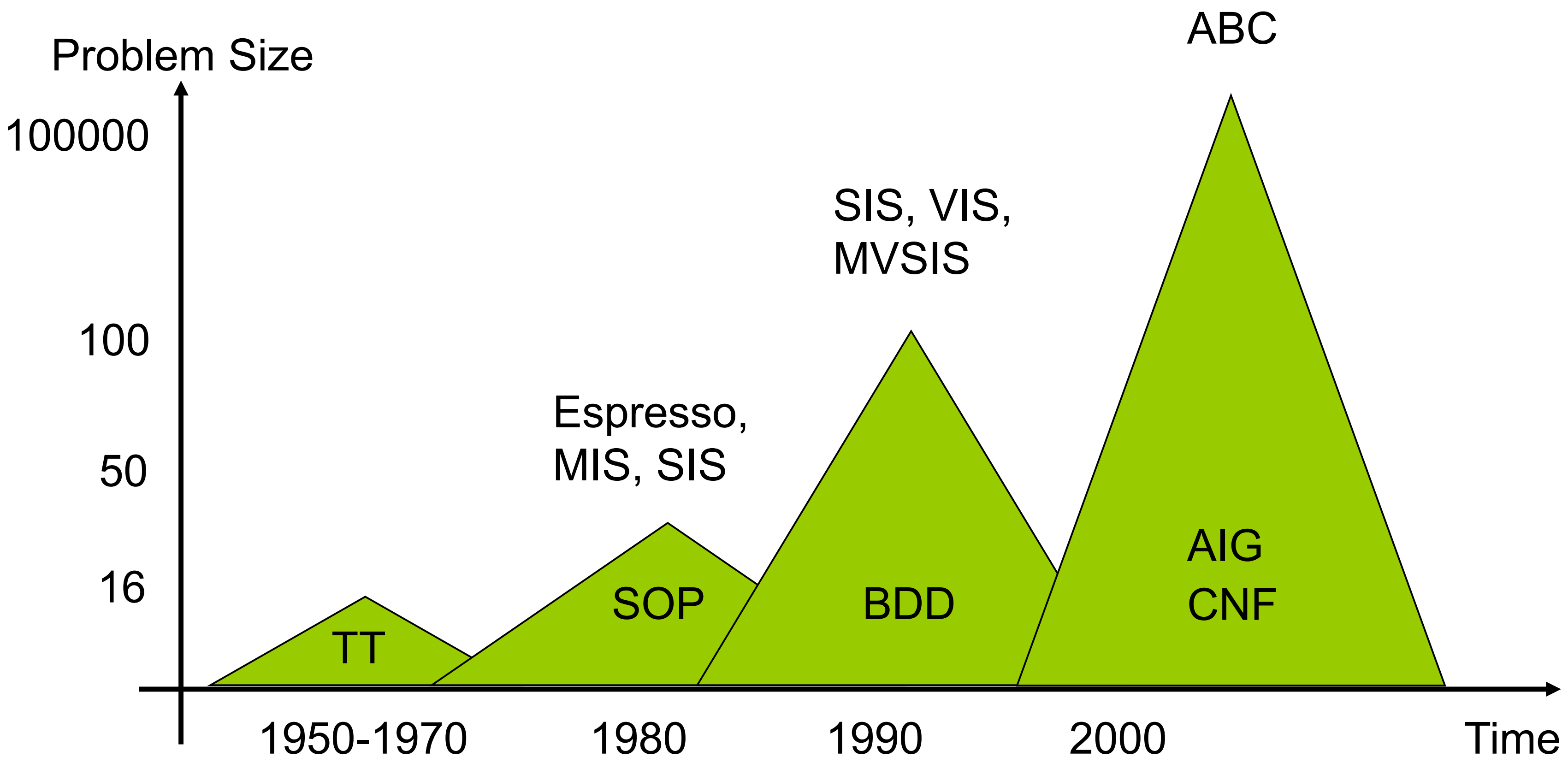
Q: Set of internal states
X: Input alphabet
Y: Output alphabet
δ: $X \times Q \rightarrow Q$ (next state *function*)
λ: $X \times Q \rightarrow Y$ (output *function*)



Target: Circuit C(G, W) where:
G: set of circuit components g $\in$ {gates, FFs, etc.}
W: set of wires connecting G

Backgrounds

- Historic evolution of data structures and tools in logic synthesis and verification



Courtesy of Alan Mishchenko

Boolean Function Representation

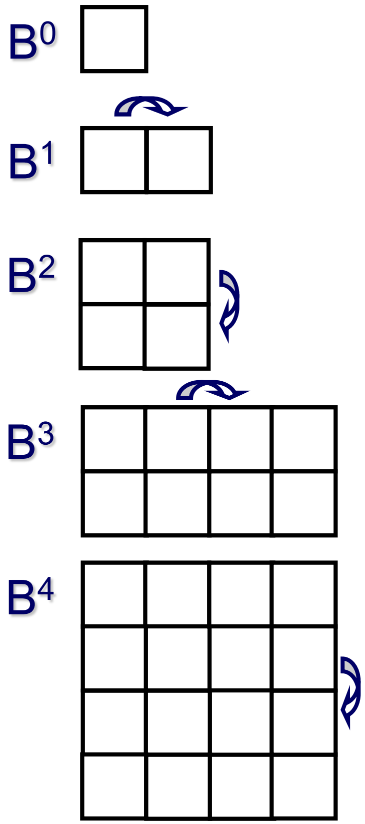
- Logic synthesis translates Boolean functions into circuits

- We need representations of Boolean functions for two reasons:
  - to represent and manipulate the actual circuit that we are implementing
  - to facilitate *Boolean reasoning*

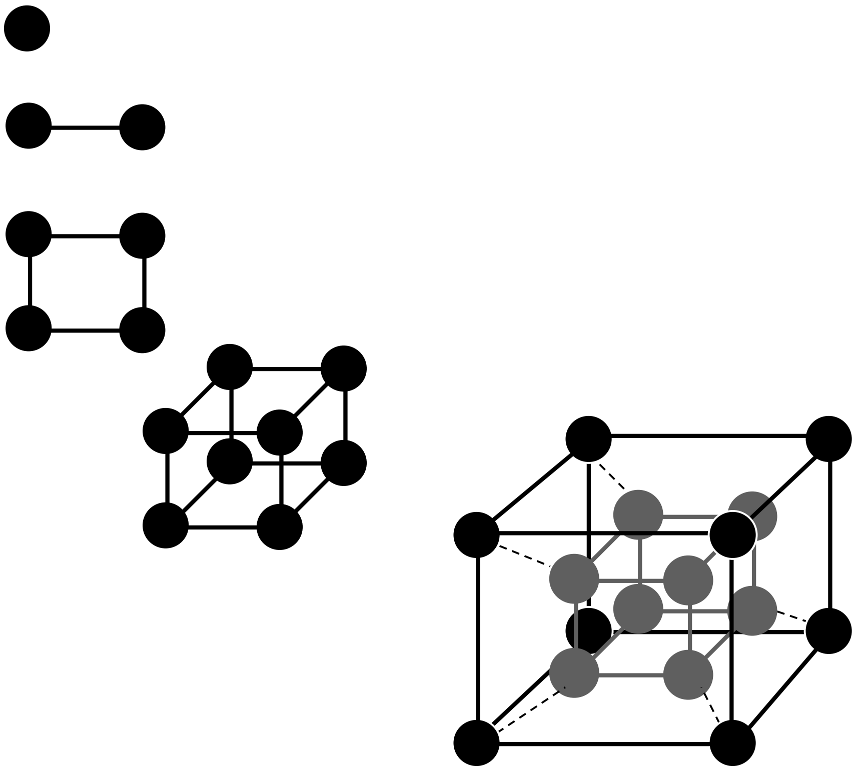
Boolean Space

- B = {0,1}
- B^2 = {0,1}×{0,1} = {00, 01, 10, 11}

Karnaugh Maps:



Boolean Lattices:



Boolean Function

- A Boolean function f over input variables: $x_1, x_2, \ldots, x_m$, is a mapping f: $\mathbf{B}^m \rightarrow Y$, where $\mathbf{B} = \{0,1\}$ and $Y = \{0,1,d\}$
  - E.g.
  - The output value of $f(x_1, x_2, x_3)$, say, partitions $\mathbf{B}^m$ into three sets:
    - on-set ($f=1$)
      - E.g. {010, 011, 110, 111} (characteristic function $f^1 = x_2$)
    - off-set ($f = 0$)
      - E.g. {100, 101} (characteristic function $f^0 = x_1 \neg x_2$)
    - don't-care set ($f = d$)
      - E.g. {000, 001} (characteristic function $f^d = \neg x_1 \neg x_2$)

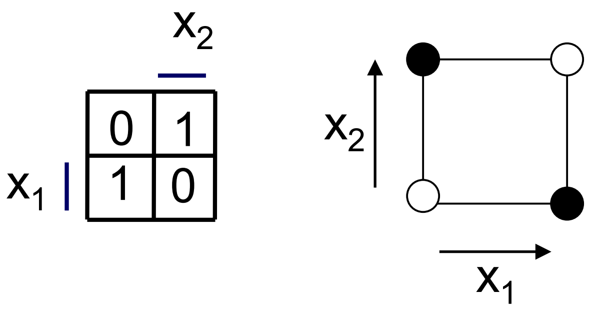
- f is an incompletely specified function if the don't-care set is nonempty. Otherwise, f is a completely specified function
  - Unless otherwise said, a Boolean function is meant to be completely specified

Boolean Function

- A Boolean function f: $\mathbf{B}^n \rightarrow \mathbf{B}$ over variables $x_1,\ldots,x_n$ maps each Boolean valuation (truth assignment) in $\mathbf{B}^n$ to 0 or 1

Example

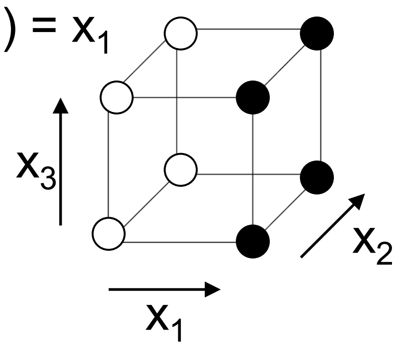
$f(x_1,x_2)$ with $f(0,0) = 0$, $f(0,1) = 1$, $f(1,0) = 1$, $f(1,1) = 0$



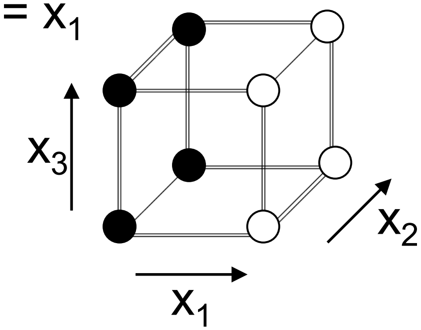
Boolean Function

- Onset of f, denoted as f^1, is $f^1 = \{v \in \mathbf{B}^n \mid f(v)=1\}$
  - If $f^1 = \mathbf{B}^n$, f is a tautology
- Offset of f, denoted as f^0, is $f^0 = \{v \in \mathbf{B}^n \mid f(v)=0\}$
  - If $f^0 = \mathbf{B}^n$, f is unsatisfiable. Otherwise, f is satisfiable.
- f^1 and f^0 are sets, not functions!
- Boolean functions f and g are equivalent if $\forall v \in \mathbf{B}^n.\ f(v) = g(v)$ where v is a truth assignment or Boolean valuation
- A literal is a Boolean variable *x* or its negation x' (or $\overline{x}$, $\neg x$) in a Boolean formula

$f(x_1, x_2, x_3) = x_1$

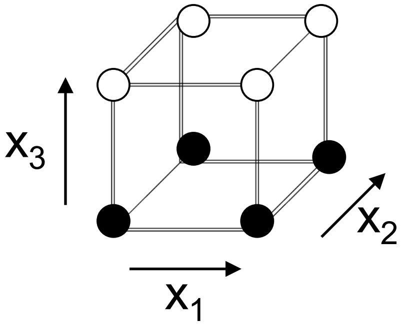


$f(x_1, x_2, x_3) = \overline{x_1}$



Boolean Function

- There are 2^n vertices in $\mathbf{B}^n$
- There are 2^{2^n} distinct Boolean functions
  - Each subset $f^1 \subseteq \mathbf{B}^n$ of vertices in $\mathbf{B}^n$ forms a distinct Boolean function f with onset f^1



$x_1 x_2 x_3$		f
0 0 0		1
0 0 1		0
0 1 0		1
0 1 1		0
1 0 0	$\Rightarrow$	1
1 0 1		0
1 1 0		1
1 1 1		0

Boolean Operations

Given two Boolean functions:

$f : \mathbf{B}^n \to \mathbf{B}$

$g : \mathbf{B}^n \to \mathbf{B}$

- $h = f \wedge g$ from AND operation is defined as
  $h^1 = f^1 \cap g^1$; $h^0 = \mathbf{B}^n \setminus h^1$
- $h = f \vee g$ from OR operation is defined as
  $h^1 = f^1 \cup g^1$; $h^0 = \mathbf{B}^n \setminus h^1$
- $h = \neg f$ from COMPLEMENT operation is defined as
  $h^1 = f^0$; $h^0 = f^1$

Cofactor and Quantification

Given a Boolean function:

$f : \mathbf{B}^n \rightarrow \mathbf{B}$, with the input variable $(x_1, x_2, \ldots, x_i, \ldots, x_n)$

- Positive cofactor on variable x_i
  $h = f_{x_i}$ is defined as $h = f(x_1, x_2, \ldots, 1, \ldots, x_n)$
- Negative cofactor on variable x_i
  $h = f_{\neg x_i}$ is defined as $h = f(x_1, x_2, \ldots, 0, \ldots, x_n)$
- Existential quantification over variable x_i
  $h = \exists x_i.\, f$ is defined as $h = f(x_1, x_2, \ldots, 0, \ldots, x_n) \vee f(x_1, x_2, \ldots, 1, \ldots, x_n)$
- Universal quantification over variable x_i
  $h = \forall x_i.\, f$ is defined as $h = f(x_1, x_2, \ldots, 0, \ldots, x_n) \wedge f(x_1, x_2, \ldots, 1, \ldots, x_n)$
- Boolean difference over variable x_i
  $h = \partial f / \partial x_i$ is defined as $h = f(x_1, x_2, \ldots, 0, \ldots, x_n) \oplus f(x_1, x_2, \ldots, 1, \ldots, x_n)$

Boolean Function Representation

- Some common representations:
  - Truth table
  - Boolean formula
    - SOP (sum-of-products, or called disjunctive normal form, DNF)
    - POS (product-of-sums, or called conjunctive normal form, CNF)
  - BDD (binary decision diagram)
  - Boolean network (consists of nodes and wires)
    - Generic Boolean network
      - Network of nodes with generic functional representations or even subcircuits
    - Specialized Boolean network
      - Network of nodes with SOPs (PLAs)
      - And-Inv Graph (AIG)

- Why different representations?
  - Different representations have their own strengths and weaknesses (no single data structure is best for all applications)

Boolean Function Representation
Truth Table

- Truth table (function table for multi-valued functions):
  The truth table of a function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ is a tabulation of its value at each of the 2^n vertices of $\mathbf{B}^n$.

  In other words the truth table lists all mintems

  Example: $f = a'b'c'd + a'b'cd + a'bc'd + ab'c'd + ab'cd + abc'd + abcd' + abcd$

  The truth table representation is
  - impractical for large n
  - canonical

  If two functions are the equal, then their canonical representations are isomorphic.

	abcd	f
0	0000	0
1	0001	1
2	0010	0
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	0

	abcd	f
8	1000	0
9	1001	1
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	1
15	1111	1

Boolean Function Representation
Boolean Formula

- A Boolean formula is defined inductively as an expression with the following formation rules (syntax):

formula ::=		'(' formula ')'	
	|	Boolean constant	(**true** or **false**)
	|	<Boolean variable>	
	|	formula "+" formula	(OR operator)
	|	formula "·" formula	(AND operator)
	|	$\neg$ formula	(complement)

Example

$f = (x_1 \cdot x_2) + (x_3) + \neg(\neg(x_4 \cdot (\neg x_1)))$

typically "·" is omitted and '(', ')' are omitted when the operator priority is clear, e.g., $f = x_1\ x_2 + x_3 + x_4 \neg x_1$

Boolean Function Representation
Boolean Formula in SOP

- Any function can be represented as a sum-of-products (SOP), also called sum-of-cubes (a cube is a product term), or disjunctive normal form (DNF)

  Example

  φ = ab + a'c + bc

Boolean Function Representation
Boolean Formula in POS

- Any function can be represented as a product-of-sums (POS), also called conjunctive normal form (CNF)
  - Dual of the SOP representation

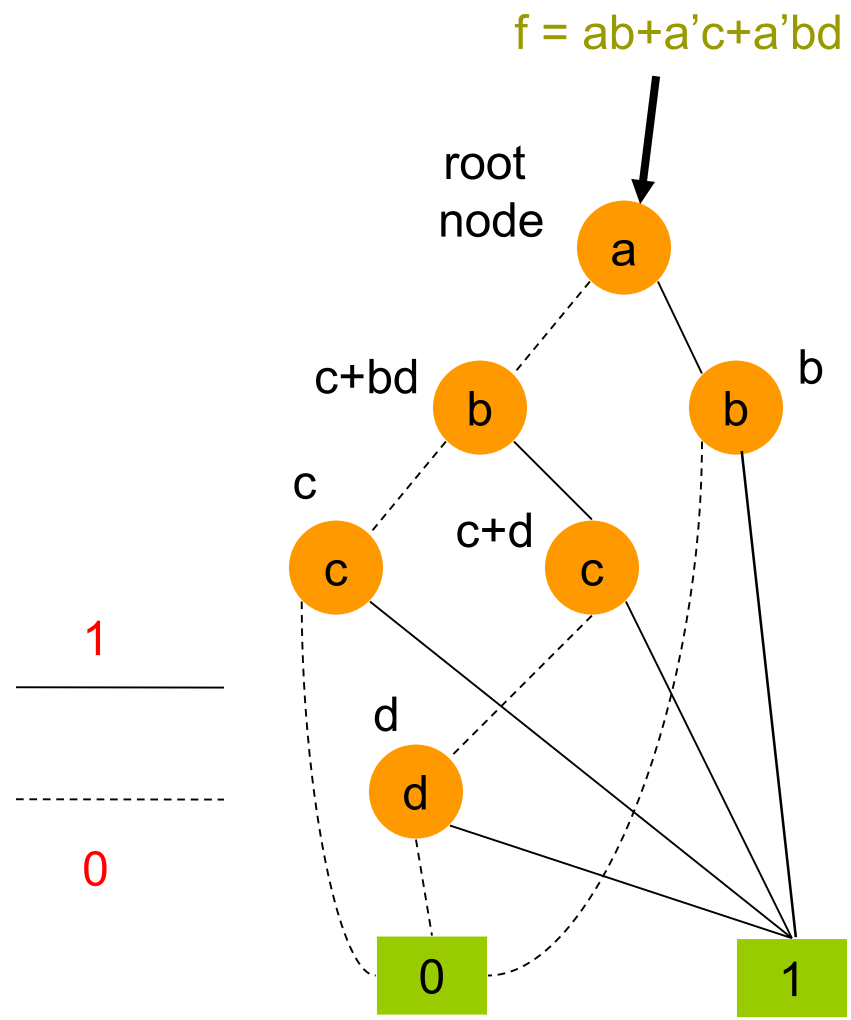
Example

$\varphi =$ (a+b′+c) (a′+b+c) (a+b′+c′) (a+b+c)

- Exercise: Any Boolean function in POS can be converted to SOP using De Morgan's law and the distributive law, and vice versa

Boolean Function Representation
Binary Decision Diagram

- BDD – a graph representation of Boolean functions
  - A leaf node represents constant 0 or 1
  - A non-leaf node represents a decision node (multiplexer) controlled by some variable
  - Can make a BDD representation **canonical** by imposing the variable ordering and reduction criteria (ROBDD)



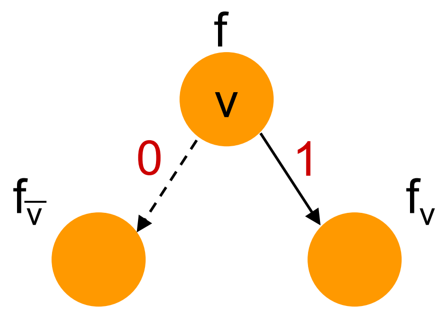
Boolean Function Representation
Binary Decision Diagram

- Any Boolean function f can be written in term of **Shannon expansion**

  $$f = v\, f_v + \neg v\, f_{\neg v}$$

  - Positive cofactor: $f_{xi} = f(x_1,\ldots,x_i=1,\ldots, x_n)$
  - Negative cofactor: $f_{\neg xi} = f(x_1,\ldots,x_i=0,\ldots, x_n)$

- BDD is a compressed Shannon cofactor tree:
  - The two children of a node with function f controlled by variable v represent two sub-functions f_v and $f_{\neg v}$

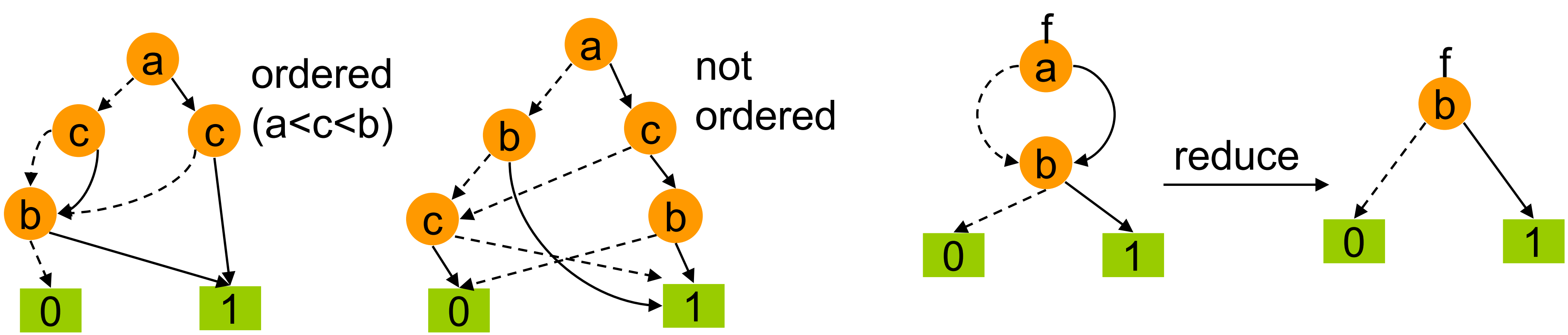


Boolean Function Representation
Binary Decision Diagram

- Reduced and ordered BDD (ROBDD) is a **canonical** Boolean function representation
  - Ordered:
    - cofactor variables are in the same order along all paths

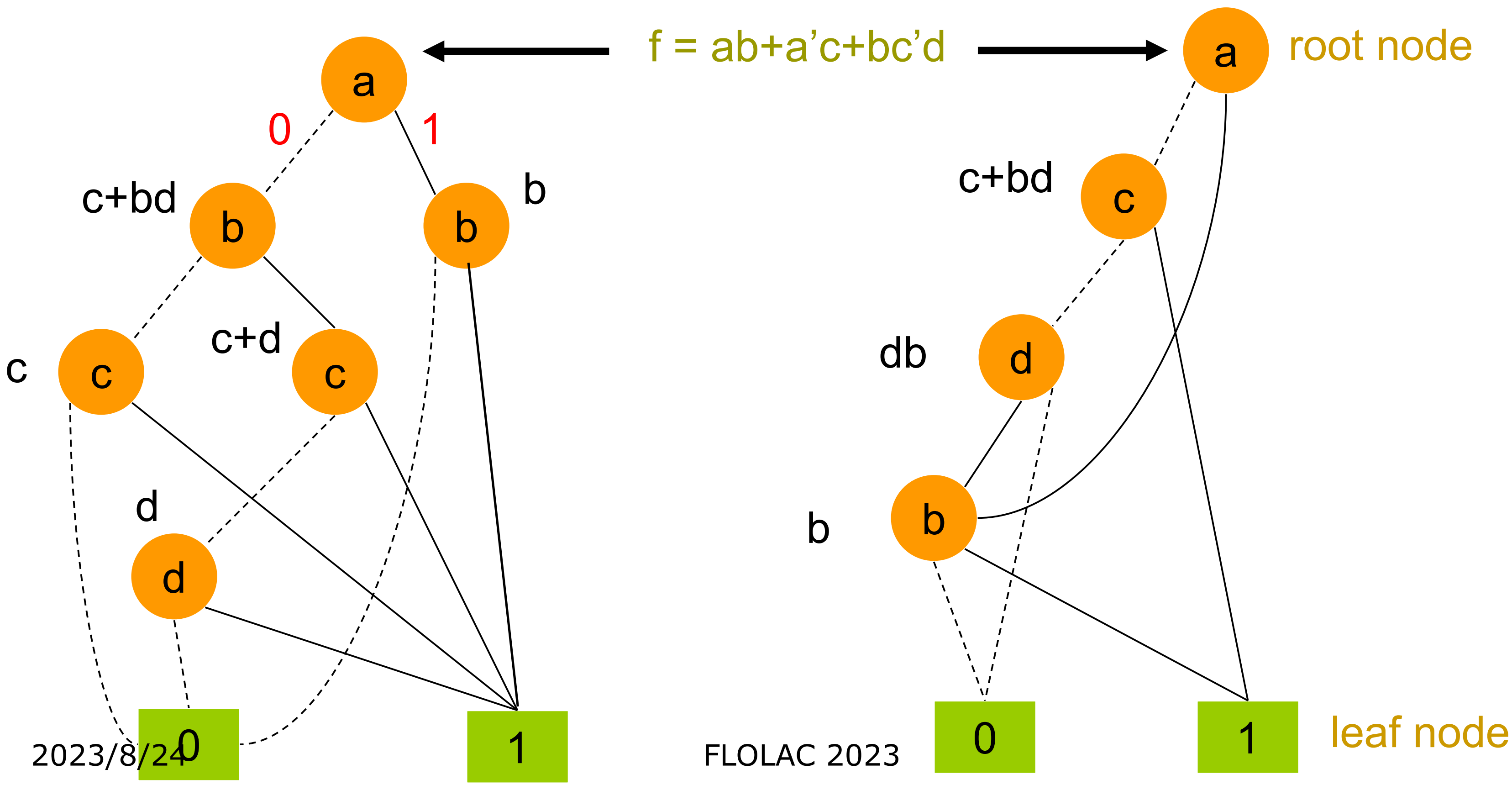
      $x_{i_1} < x_{i_2} < x_{i_3} < \ldots < x_{i_n}$
  - Reduced:
    - any node with two identical children is removed
    - two nodes with isomorphic BDD's are merged

    These two rules make any node in an ROBDD represent a distinct logic function



Boolean Function Representation
Binary Decision Diagram

- For a Boolean function,
  - ROBDD is unique with respect to a given variable ordering
  - Different orderings may result in different ROBDD structures



Boolean Function Representation
Boolean Network

- A Boolean network is a directed graph $C(G,N)$ where G are the gates and $N \subseteq (G \times G)$ are the directed edges (nets) connecting the gates.

  Some of the vertices are designated:
  Inputs: $I \subseteq G$
  Outputs: $O \subseteq G$
  $I \cap O = \emptyset$

  Each gate g is assigned a Boolean function f_g which computes the output of the gate in terms of its inputs.

Boolean Function Representation
Boolean Network

- The fanin FI(g) of a gate g are the predecessor gates of g: FI(g) = {g' | (g',g) $\in$ N} (N: the set of nets)

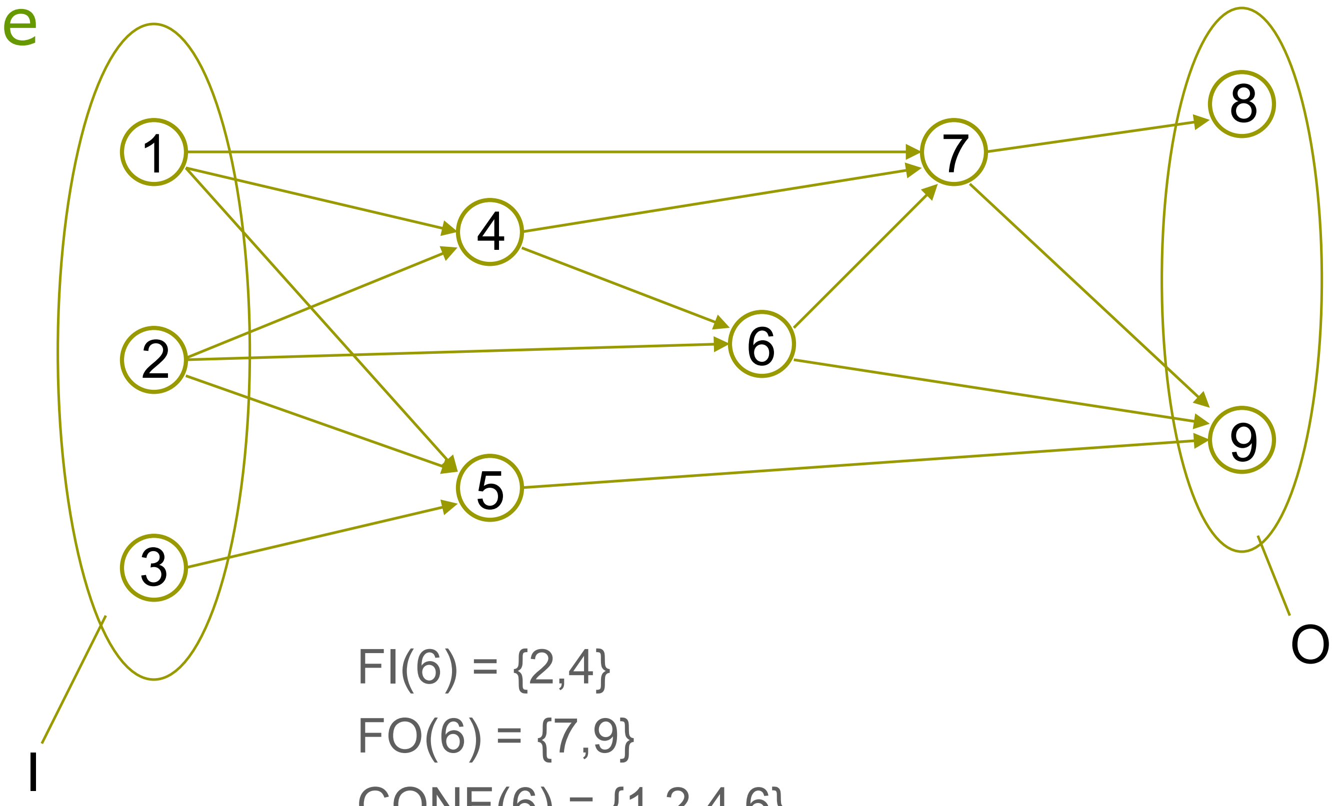
- The fanout FO(g) of a gate g are the successor gates of g: FO(g) = {g' | (g,g') $\in$ N}

- The cone CONE(g) of a gate g is the transitive fanin (TFI) of g and g itself

- The support SUPPORT(g) of a gate g are all inputs in its cone:
  SUPPORT(g) = CONE(g) $\cap$ I

Boolean Function Representation
Boolean Network

Example



FI(6) = {2,4}

FO(6) = {7,9}

CONE(6) = {1,2,4,6}

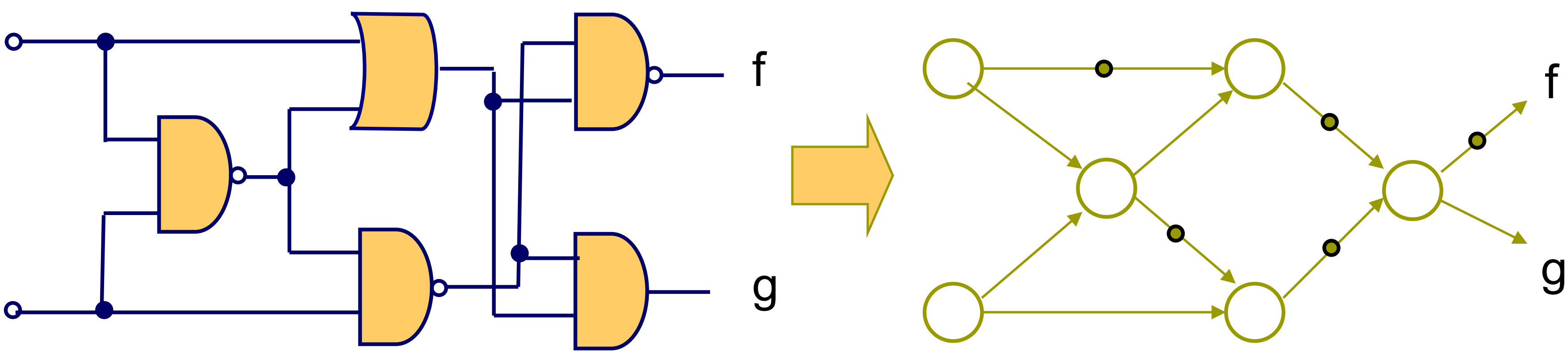
SUPPORT(6) = {1,2}

Every node may have its own function

Boolean Function Representation
And-Inverter Graph

- AND-INVERTER graphs (AIGs)
  vertices: 2-input AND gates
  edges: interconnects with (optional) dots representing INVs

- Hash table to identify and reuse structurally isomorphic circuits

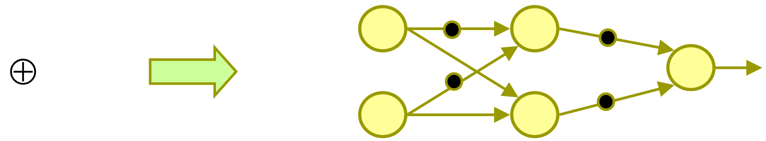


Boolean Function Representation

- Truth table
  - Canonical
  - Useful in representing small functions
- SOP
  - Useful in two-level logic optimization, and in representing local node functions in a Boolean network
- POS
  - Useful in SAT solving and Boolean reasoning
  - Rarely used in circuit synthesis (due to the asymmetric characteristics of NMOS and PMOS)
- ROBDD
  - Canonical
  - Useful in Boolean reasoning
- Boolean network
  - Useful in multi-level logic optimization
- AIG
  - Useful in multi-level logic optimization and Boolean reasoning

Circuit to CNF Conversion

- Naive conversion of circuit to CNF:
  - Multiply out expressions of circuit until two level structure
  - Example: $y = x_1 \oplus x_2 \oplus x_2 \oplus \ldots \oplus x_n$ (Parity function)
    - circuit size is linear in the number of variables

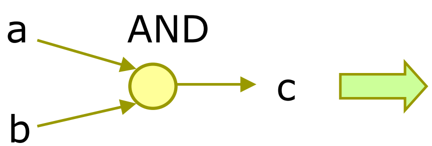


    - generated chess-board Karnaugh map
    - CNF (or DNF) formula has 2^{n-1} terms (exponential in #vars)

- Better approach:
  - Introduce one variable per circuit vertex
  - Formulate the circuit as a conjunction of constraints imposed on the vertex values by the gates
  - Uses more variables but size of formula is linear in the size of the circuit

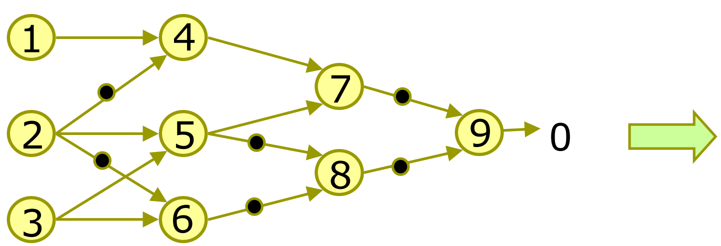
Circuit to CNF Conversion

- Example
  - Single gate:



$(\neg a + \neg b + c)(a + \neg c)(b + \neg c)$

  - Circuit of connected gates:



$(\neg 1 + 2 + 4)(1 + \neg 4)(\neg 2 + \neg 4)$
$(\neg 2 + \neg 3 + 5)(2 + \neg 5)(3 + \neg 5)$
$(2 + \neg 3 + 6)(\neg 2 + \neg 6)(3 + \neg 6)$
$(\neg 4 + \neg 5 + 7)(4 + \neg 7)(5 + \neg 7)$
$(5 + 6 + 8)(\neg 5 + \neg 8)(\neg 6 + \neg 8)$
$(7 + 8 + 9)(\neg 7 + \neg 9)(\neg 8 + \neg 9)$
(9)

Is output always 0 ?

Justify to "1"

Circuit to CNF Conversion

- Circuit to CNF conversion
  - can be done in linear size (with respect to the circuit size) if intermediate variables can be introduced
  - may grow exponentially in size if no intermediate variables are allowed

Propositional Satisfiability

Normal Forms

- A **literal** is a variable or its negation
- A **clause** (**cube**) is a disjunction (conjunction) of literals
- A **conjunctive normal form** (CNF) is a conjunction of clauses; a **disjunctive normal form** (DNF) is a disjunction of cubes
  - E.g.,
    CNF: $(a+\neg b+c)(a+\neg c)(b+d)(\neg a)$
    - $(\neg a)$ is a unit clause, d is a pure literal

    DNF: $a\neg bc + a\neg c + bd + \neg a$

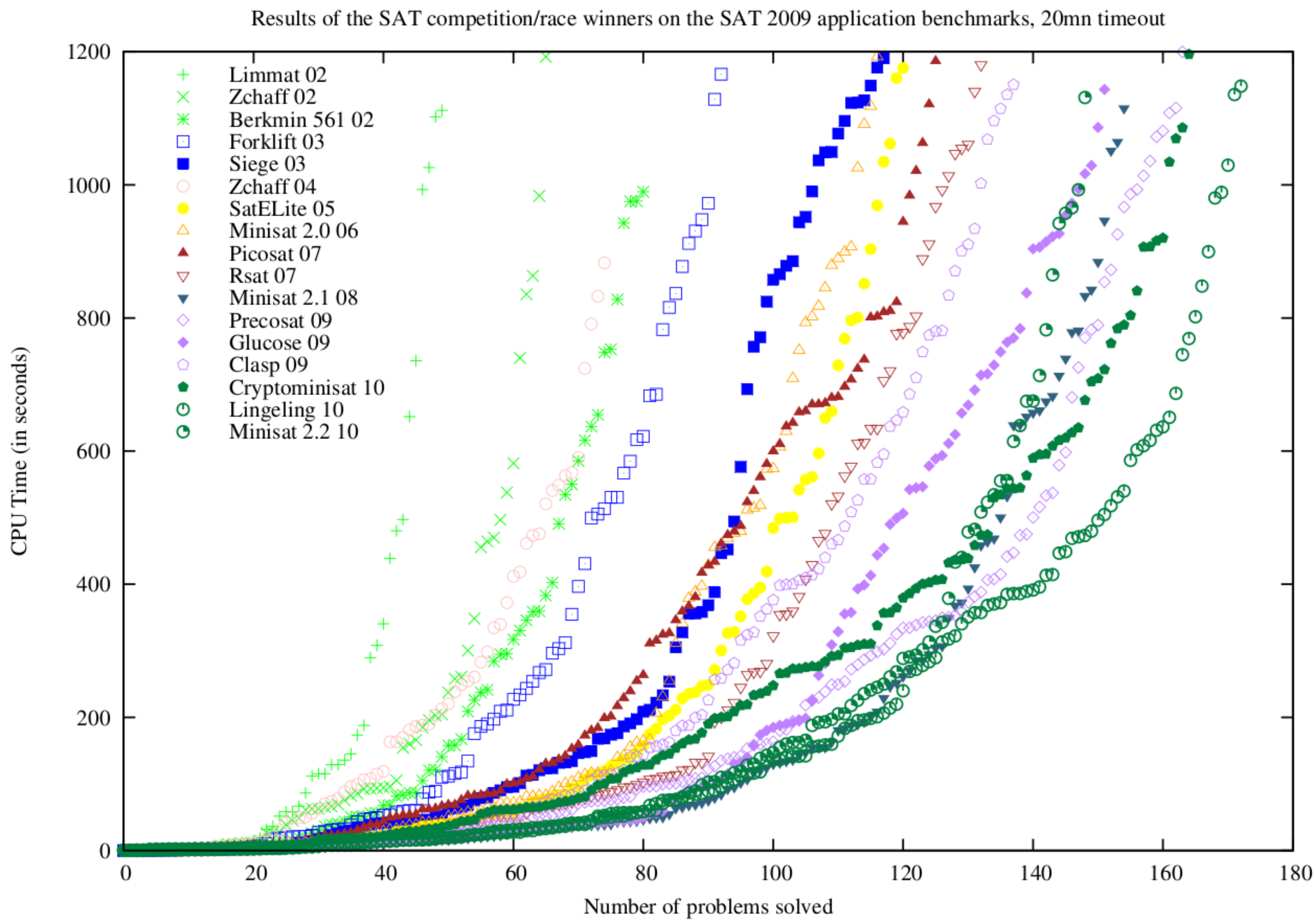
Satisfiability

- The **satisfiability** (SAT) problem asks whether a given CNF formula can be true under some assignment to the variables

- In theory, SAT is intractable
  - The first shown NP-complete problem [Cook, 1971]

- In practice, modern SAT solvers work 'mysteriously' well on application CNFs with ~100,000 variables and ~1,000,000 clauses
  - It enables various applications, and inspires solver development for QBF, SMT (Satisfiability Modulo Theories), DQBF, SSAT, etc.

SAT Competition



http://www.satcompetition.org/PoS11/

SAT Solving

- Ingredients of modern SAT solvers:
  - DPLL-style search
    - [Davis, Putnam, Logemann, Loveland, 1962]
  - Conflict-driven clause learning (CDCL)
    - [Marques-Silva, Sakallah, 1996 (GRASP)]
  - Boolean constraint propagation (BCP) with two-literal watch
    - [Moskewicz, Modigan, Zhao, Zhang, Malik, 2001 (Chaff)]
  - Decision heuristics using variable activity
    - [Moskewicz, Modigan, Zhao, Zhang, Malik, 2001 (Chaff)]
  - Restart
  - Preprocessing
  - Support for incremental solving
    - [Een, Sorensson, 2003 (MiniSat)]

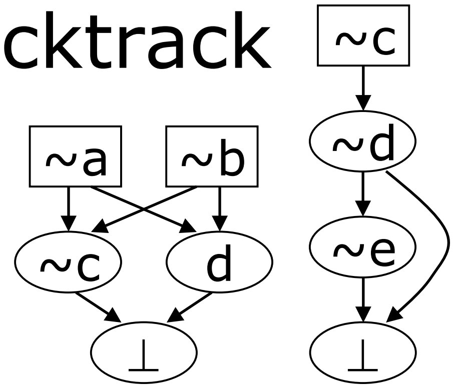
Pre-Modern SAT Procedure

Algorithm `DPLL(Φ)`

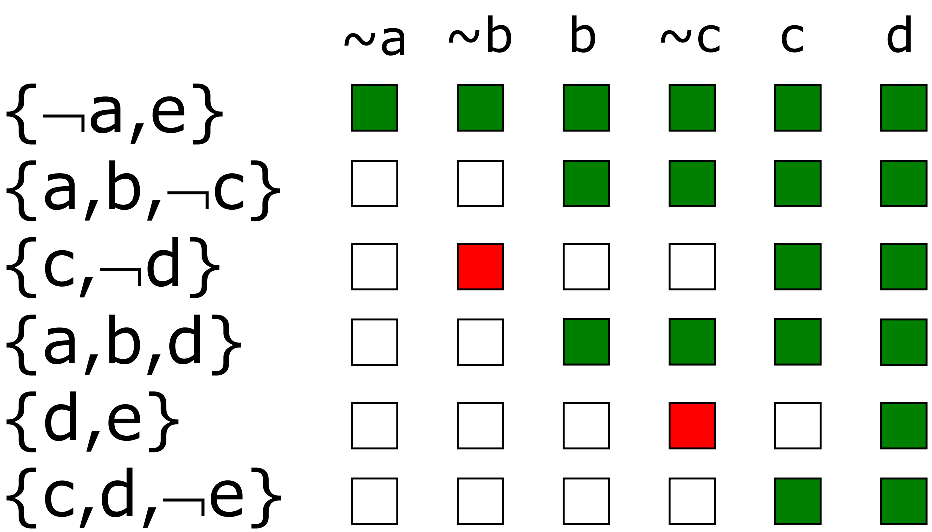
```
{
   while there is a unit clause {l} in Φ
      Φ = BCP(Φ, l);
   while there is a pure literal l in Φ
      Φ = assign(Φ, l);
   if all clauses of Φ satisfied    return true;
   if Φ has a conflicting clause    return false;
   l := choose_literal(Φ);
   return DPLL(assign(Φ,¬l)) ∨ DPLL(assign(Φ,l));
}
```

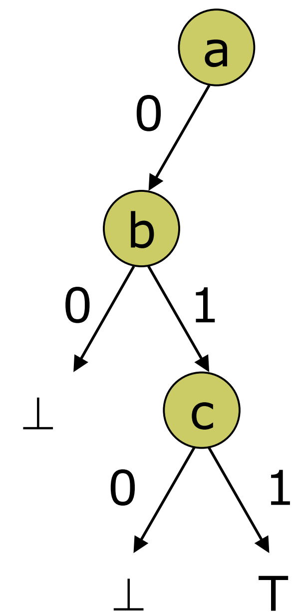
DPLL Procedure

- Chorological backtrack



- E.g.





Modern SAT Procedure

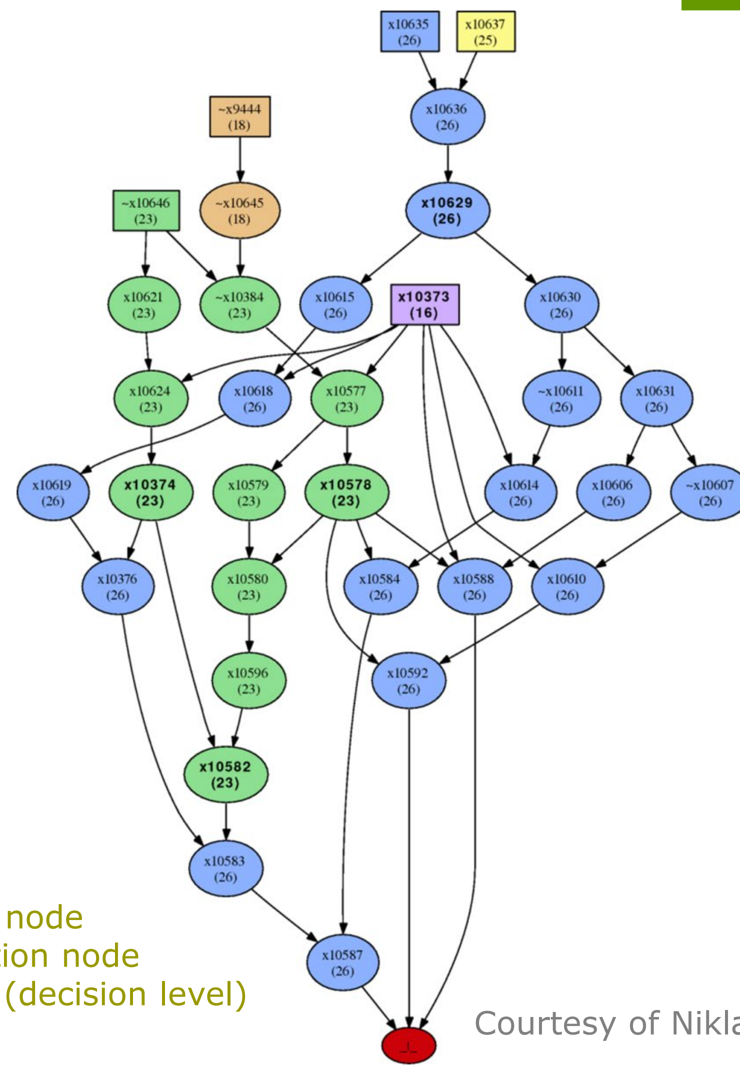
Algorithm `CDCL(Φ)`

```
{
  while(1)
     while there is a unit clause {l} in Φ
         Φ = BCP(Φ, l);
     while there is a pure literal l in Φ
         Φ = assign(Φ, l);
     if Φ contains no conflicting clause
        if all clauses of Φ are satisfied   return true;
        l := choose_literal(Φ);
        assign(Φ,l);
     else
        if conflict at top decision level   return false;
        analyze_conflict();
        undo assignments;
        Φ := add_conflict_clause(Φ);
}
```

Conflict Analysis & Clause Learning

- There can be many learnt clauses from a conflict
- Clause learning admits non-chorological backtrack

- E.g.,
  {¬x10587, ¬x10588, ¬x10592}
  …
  {¬x10374, ¬x10582, ¬x10578, ¬x10373, ¬x10629}
  …
  {x10646, x9444, ¬x10373, ¬x10635, ¬x10637}



Box: decision node
Oval: implication node
Inside: literal (decision level)

Courtesy of Niklas Een

Clause Learning as Resolution

- **Resolution** of two clauses $C_1 \vee x$ and $C_2 \vee \neg x$:

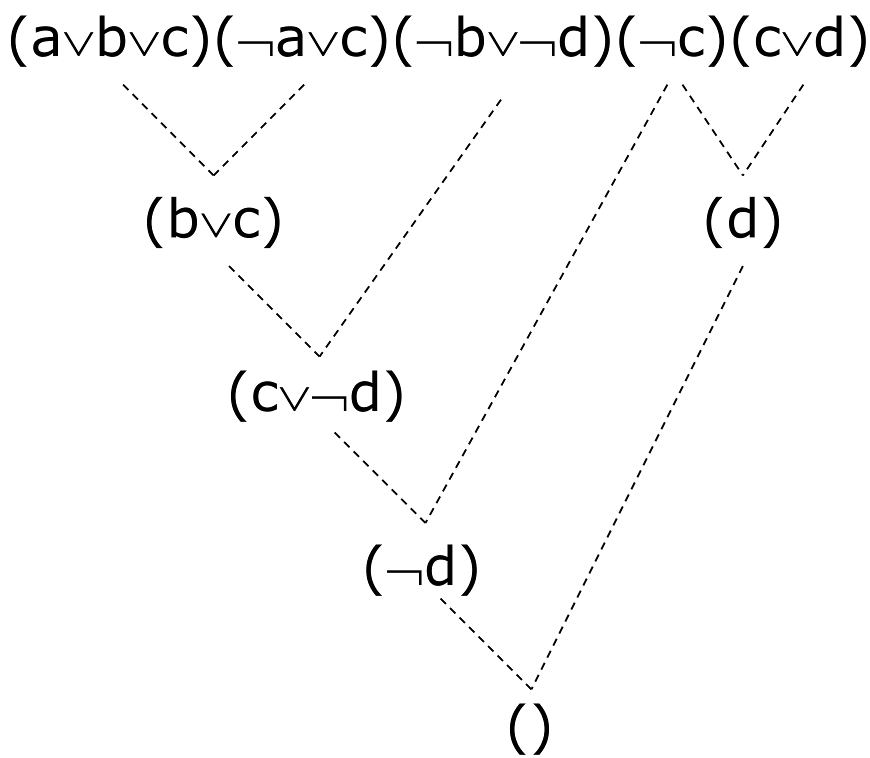
$$\frac{C_1 \vee x \qquad\qquad C_2 \vee \neg x}{C_1 \vee C_2}$$

  where x is the **pivot variable** and $C_1 \vee C_2$ is the **resolvant**, i.e., $C_1 \vee C_2 = \exists x.(C_1 \vee x)(C_2 \vee \neg x)$

- A learnt clause can be obtained from a sequence of resolution steps
  - Exercise:
    Find a resolution sequence leading to the learnt clause
    {¬x10374, ¬x10582, ¬x10578, ¬x10373, ¬x10629}
    in the previous slides

Resolution

- Resolution is complete for SAT solving
  - A CNF formula is unsatisfiable if and only if there exists a resolution sequence leading to the empty clause
  - Example



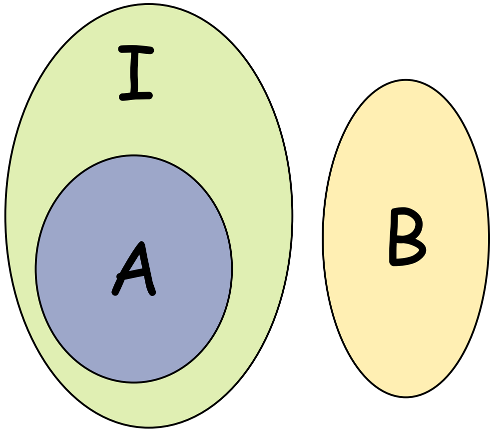
SAT Certification

- True CNF
  - Satisfying assignment (model)
    - Verifiable in linear time
- False CNF
  - Resolution refutation
    - Potentially of exponential size

Craig Interpolation

- [Craig Interpolation Thm, 1957]
  If $A \wedge B$ is UNSAT for formulae A and B, there exists an interpolant I of A such that

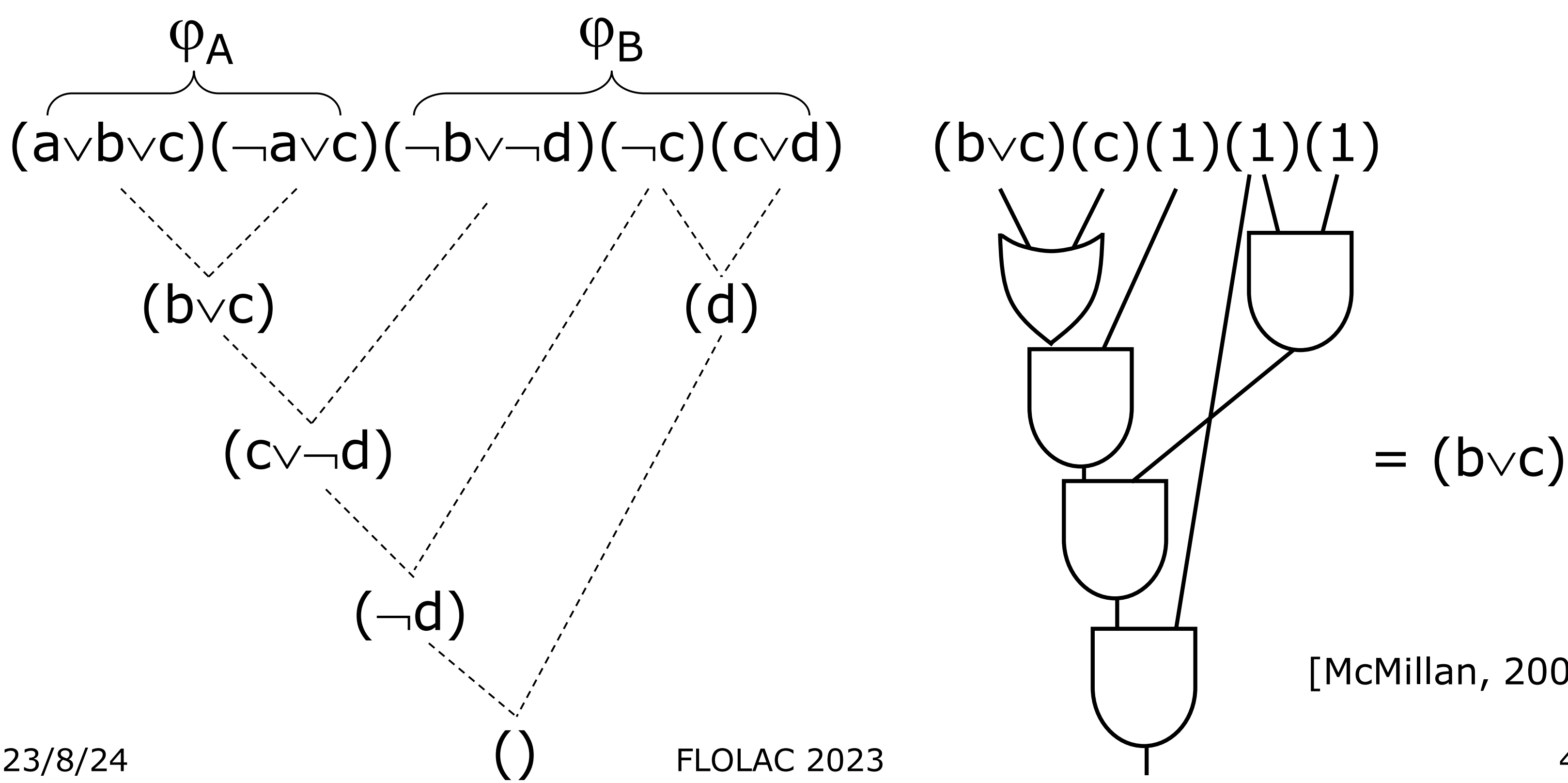
  1. $A \Rightarrow I$
  2. $I \wedge B$ is UNSAT
  3. I refers only to the common variables of A and B



I is an abstraction of A

Interpolant and Resolution Proof

- SAT solver may produce the resolution proof of an UNSAT CNF φ
- For $\varphi = \varphi_A \wedge \varphi_B$ specified, the corresponding interpolant can be obtained in time linear in the resolution proof



[McMillan, 2003]

Incremental SAT Solving

- To solve, in a row, multiple CNF formulae, which are similar except for a few clauses, can we reuse the learnt clauses?
  - What if adding a clause to φ?
  - What if deleting a clause from φ?

Incremental SAT Solving

- MiniSat API
  - void *addClause*(Vec<Lit> clause)
  - bool *solve*(Vec<Lit> assumps)
  - bool *readModel*(Var x) – *for SAT results*
  - bool *assumpUsed*(Lit p) – *for UNSAT results*

  - The method *solve()* treats the literals in assumps as unit clauses to be temporary assumed during the SAT-solving.
  - More clauses can be added after *solve()* returns, then incrementally another SAT-solving executed.

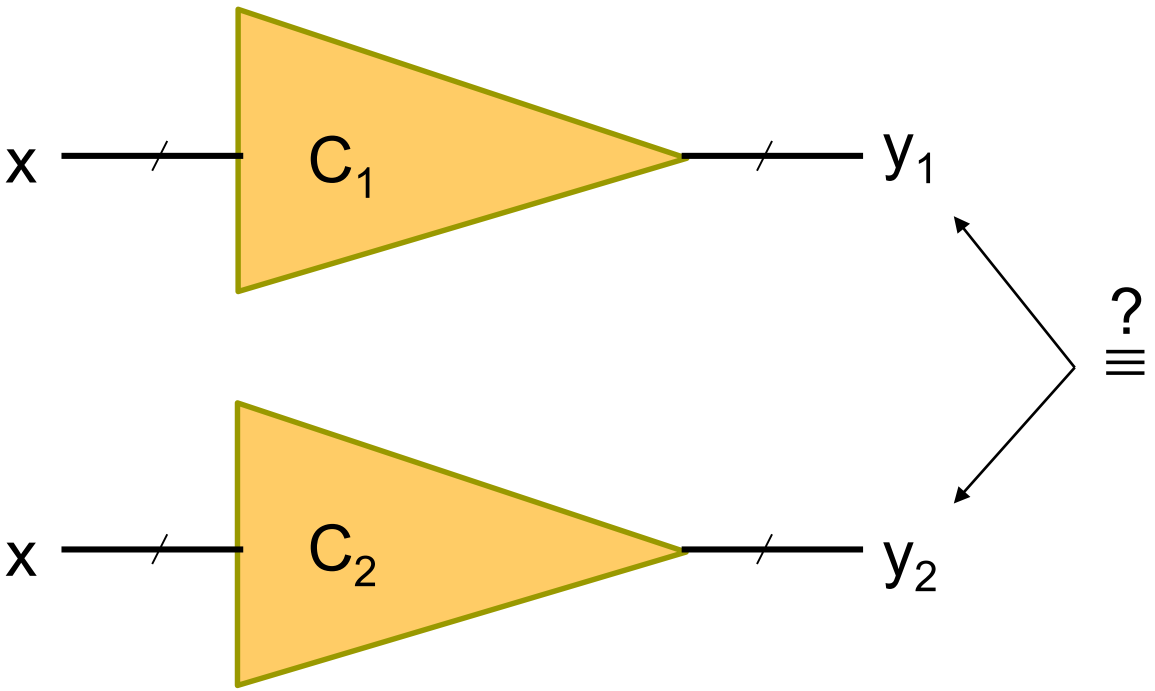
Courtesy of Niklas Een

SAT & Logic Synthesis

Equivalence Checking

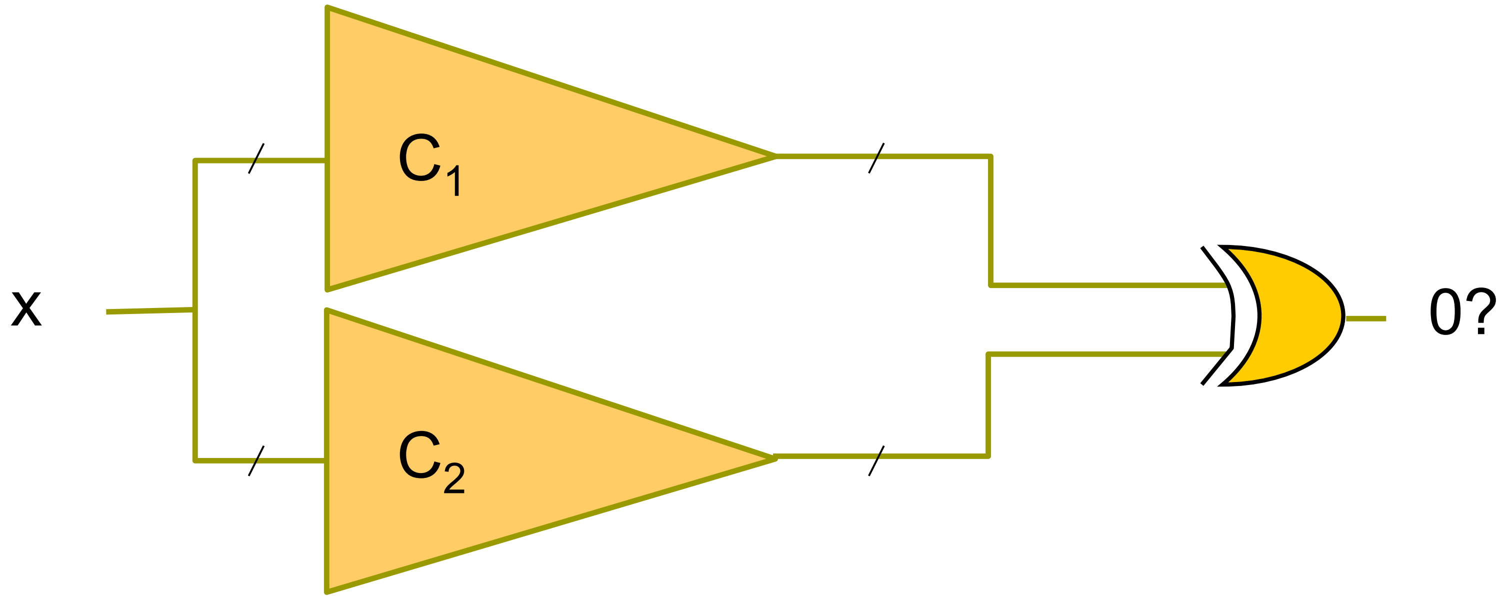
Combinational EC

- Given two combinational circuits C_1 and C_2, are their outputs equivalent under all possible input assignments?



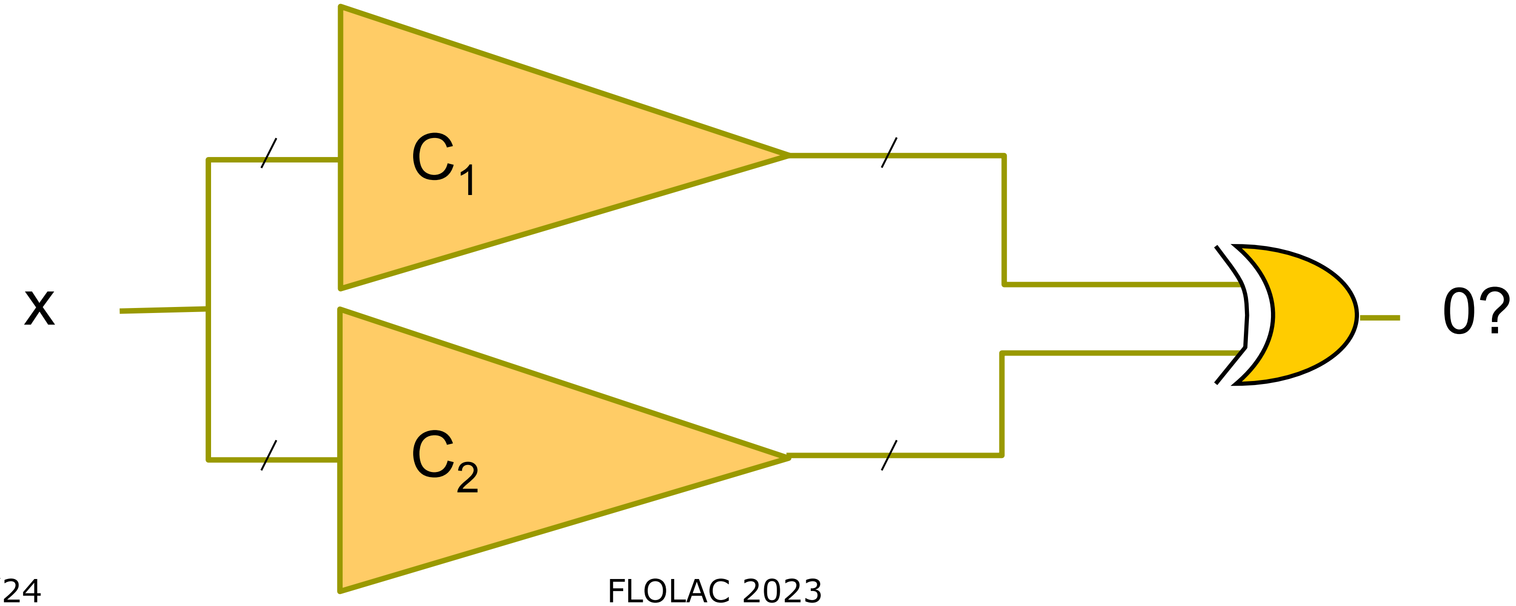
Miter for Combinational EC

- Two combinational circuits C_1 and C_2 are equivalent if and only if the output of their "miter" structure always produces constant 0



Approaches to Combinational EC

- Basic methods:
  - random simulation
    - good at identifying inequivalent signals
  - BDD-based methods
  - structural SAT-based methods



SAT & Logic Synthesis
Functional Dependency

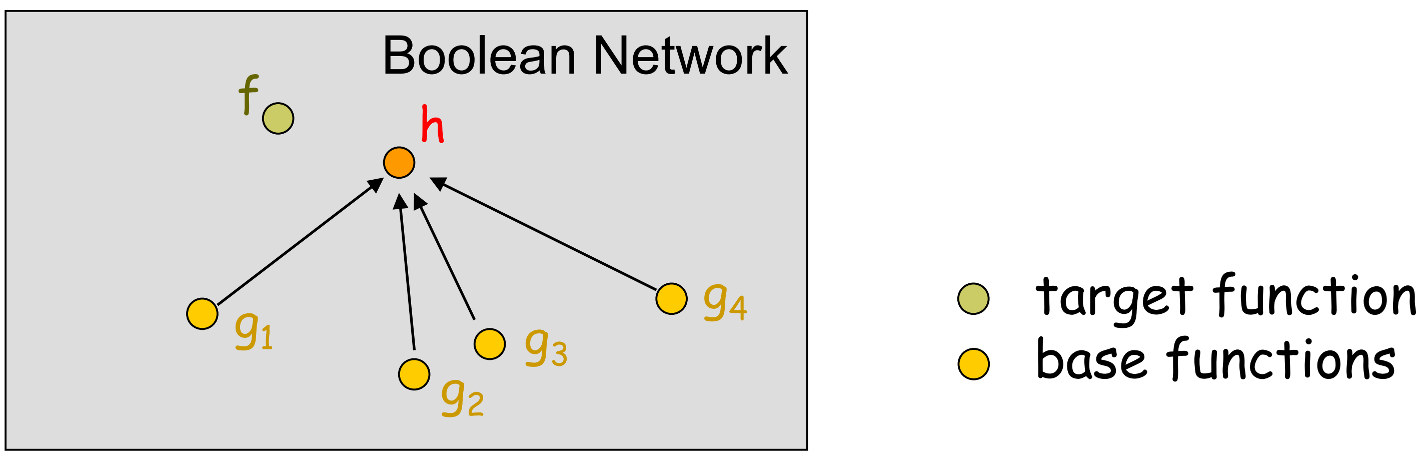
Functional Dependency

- $f(x)$ **functionally depends** on $g_1(x)$, $g_2(x)$, …, $g_m(x)$ if $f(x) = h(g_1(x), g_2(x), \ldots, g_m(x))$, denoted $h(G(x))$
  - Under what condition can function f be expressed as some function h over a set $G=\{g_1,\ldots,g_m\}$ of functions ?
  - h exists $\Leftrightarrow$ $\nexists a,b$ such that $f(a)\neq f(b)$ and $G(a)=G(b)$

i.e., G is more distinguishing than f

Motivation

- Applications of functional dependency
  - Resynthesis/rewiring
  - Redundant register removal
  - BDD minimization
  - Verification reduction
  - …

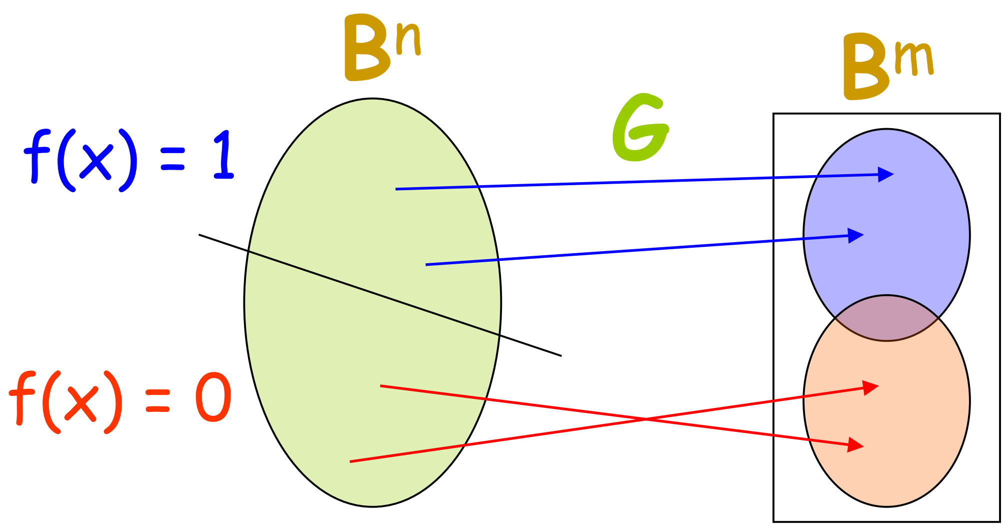


BDD-Based Computation

- BDD-based computation of h

  $h^{on} = \{y \in \mathbf{B}^m : y = G(x) \text{ and } f(x) = 1,\ x \in \mathbf{B}^n\}$

  $h^{off} = \{y \in \mathbf{B}^m : y = G(x) \text{ and } f(x) = 0,\ x \in \mathbf{B}^n\}$



$h^{on} = \exists x.(y \equiv G) \wedge f$

$h^{off} = \exists x.(y \equiv G) \wedge \neg f$

BDD-Based Computation

- Pros
  - Exact computation of h^{on} and h^{off}
  - Better support for don't care minimization
- Cons
  - 2 image computations for every choice of G
  - Inefficient when $|G|$ is large or when there are many choices of G

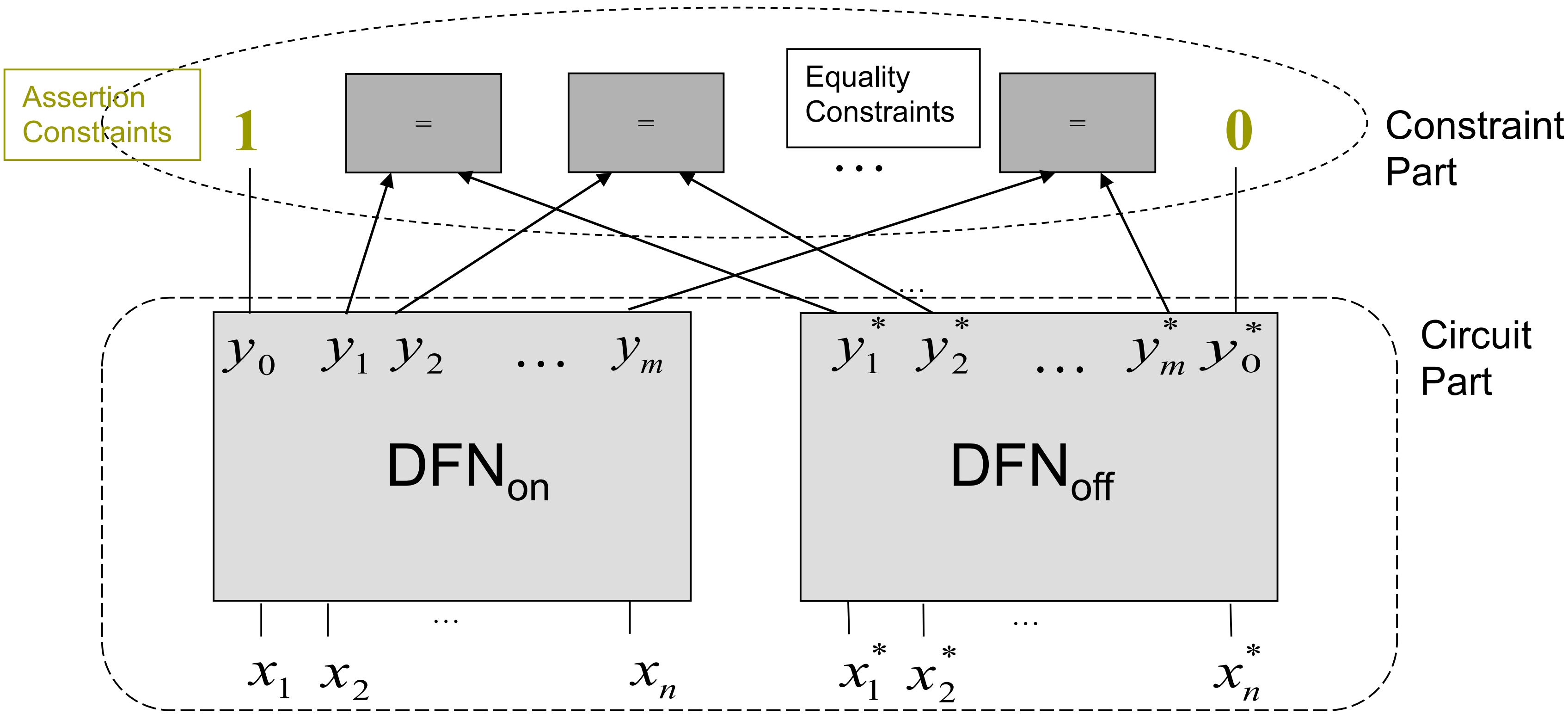
SAT-Based Computation

- h exists $\Leftrightarrow$
  $\nexists a,b$ such that $f(a)\neq f(b)$ and $G(a)=G(b)$,
  i.e., $(f(x)\neq f(x^*))\wedge(G(x)\equiv G(x^*))$ is UNSAT

- How to derive h? How to select G?

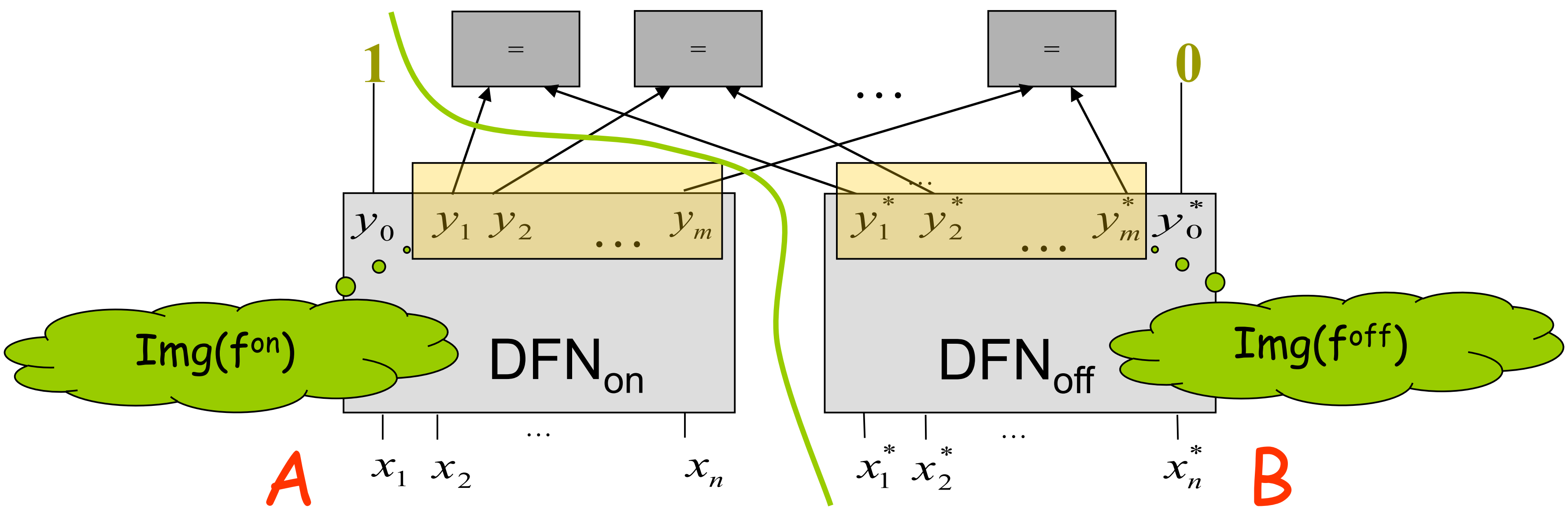
SAT-Based Computation

- $(f(x) \not\equiv f(x^*)) \wedge (G(x) \equiv G(x^*))$ is UNSAT



Deriving *h* with Craig Interpolation

- Clause set A: C_{DFNon}, y_0
- Clause set B: C_{DFNoff}, $\neg y_0^*$, $(y_i \equiv y_i^*)$ for i =1,…,m
- I is an overapproximation of Img(f^{on}) and is disjoint from Img(f^{off})
- I only refers to y_1,…, y_m
- Therefore, I corresponds to a feasible implementation of h



Incremental SAT Solving

- Controlled equality constraints
  $(y_i \equiv y_i^*) \rightarrow (\neg y_i \vee y_i^* \vee \alpha_i)(y_i \vee \neg y_i^* \vee \alpha_i)$
  with auxiliary variables α_i

  α_i = true $\Rightarrow$ i^{th} equality constraint is disabled

  - Fast switch between target and base functions by unit assumptions over control variables
  - Fast enumeration of different base functions
  - Share learned clauses

SAT vs. BDD

- SAT
  - Pros
    - Detect multiple choices of G automatically
    - Scalable to large $|G|$
    - Fast enumeration of different target functions f
    - Fast enumeration of different base functions G
  - Cons
    - Single feasible implementation of h
- BDD
  - Cons
    - Detect one choice of G at a time
    - Limited to small $|G|$
    - Slow enumeration of different target functions f
    - Slow enumeration of different base functions G
  - Pros
    - All possible implementations of h

Quantified Boolean Satisfiability

Quantified Boolean Formula

- A quantified Boolean formula (QBF) is often written in **prenex form** (with quantifiers placed on the left) as

$$\underbrace{Q_1\ x_1, \ldots, Q_n\ x_n}_{\text{prefix}}.\ \underbrace{\varphi}_{\text{matrix}}$$

for $Q_i \in \{\forall, \exists\}$ and φ a quantifier-free formula

  - If φ is further in CNF, the corresponding QBF is in the so-called **prenex CNF** (PCNF), the most popular QBF representation
  - Any QBF can be converted to PCNF

Quantified Boolean Formula

- Quantification order matters in a QBF
- A variable x_i in $(Q_1\ x_1, \ldots, Q_i\ x_i, \ldots, Q_n\ x_n.\ \varphi)$ is of **level** k if there are k quantifier alternations (i.e., changing from $\forall$ to $\exists$ or from $\exists$ to $\forall$) from Q_1 to Q_i.
  - Example

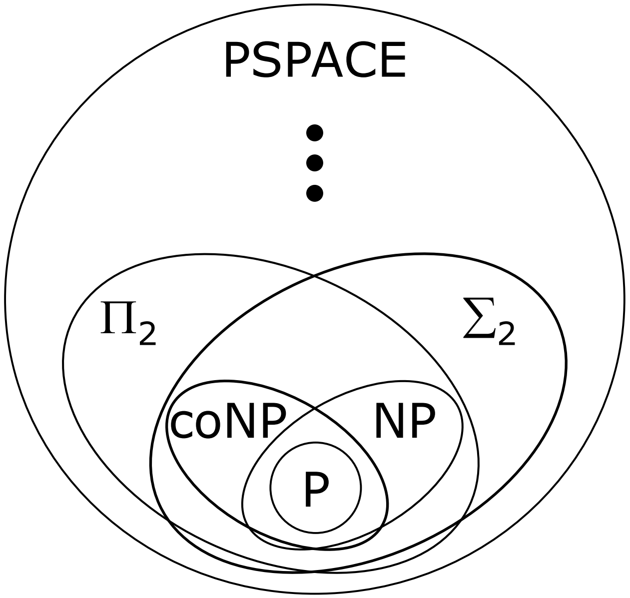
    $\forall a\ \exists b\ \forall c\ \forall d\ \exists e.\ \varphi$

    level(a)=0, level(b)=1, level(c)=2, level(d)=2, level(e)=3

Quantified Boolean Formula

- Many decision problems can be compactly encoded in QBFs

- In theory, QBF solving (QSAT) is PSPACE complete
  - The more the quantifier alternations, the higher the complexity in the Polynomial Hierarchy



- In practice, solvable QBFs are typically of size ~1,000 variables

QBF Solver

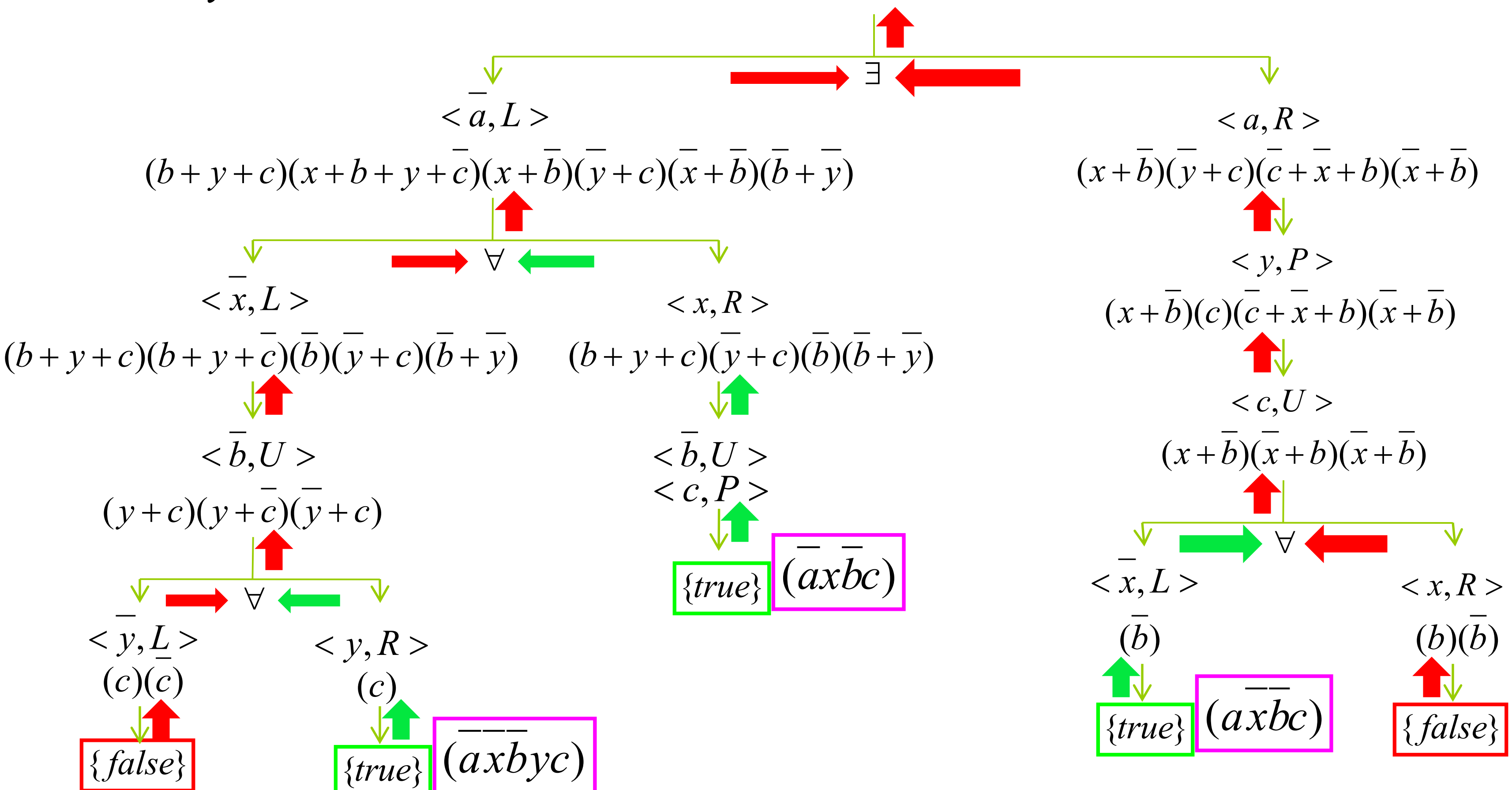
- QBF solver choices
  - Data structures for formula representation
    - **Prenex** vs. non-prenex
    - **Normal form** vs. non-normal form
      - CNF, NNF, BDD, AIG, etc.
  - Solving mechanisms
    - **Search**, Q-resolution, Skolemization, quantifier elimination, etc.
  - Preprocessing techniques

- Standard approach
  - Search-based PCNF formula solving (similar to SAT)
    - Both **clause learning** (from a conflicting assignment) and **cube learning** (from a satisfying assignment) are performed
      - Example
        $\forall a\ \exists b\ \exists c\ \forall d\ \exists e.\ (a+c)(\neg a+\neg c)(b+\neg c+e)(\neg b)(c+d+\neg e)(\neg c+e)(\neg d+e)$
        from 00101, we learn cube $\neg a \neg b c \neg d$ (can be further simplified to $\neg a$)

QBF Solving

- Example

$$\exists a \forall x \exists b \forall y \exists c \quad (a+b+y+c)(a+x+b+y+\bar{c})(x+\bar{b})(\bar{y}+c)(\bar{c}+\bar{a}+\bar{x}+b)(\bar{x}+\bar{b})(a+\bar{b}+\bar{y})$$



Q-Resolution

- **Q-resolution** on PCNF is similar to resolution on CNF, except that the pivots are restricted to existentially quantified variables and the additional rule of ∀**-reduction**

$$\frac{C_1 \lor x \qquad\qquad C_2 \lor \neg x}{\forall\text{-RED}(C_1 \lor C_2)}$$

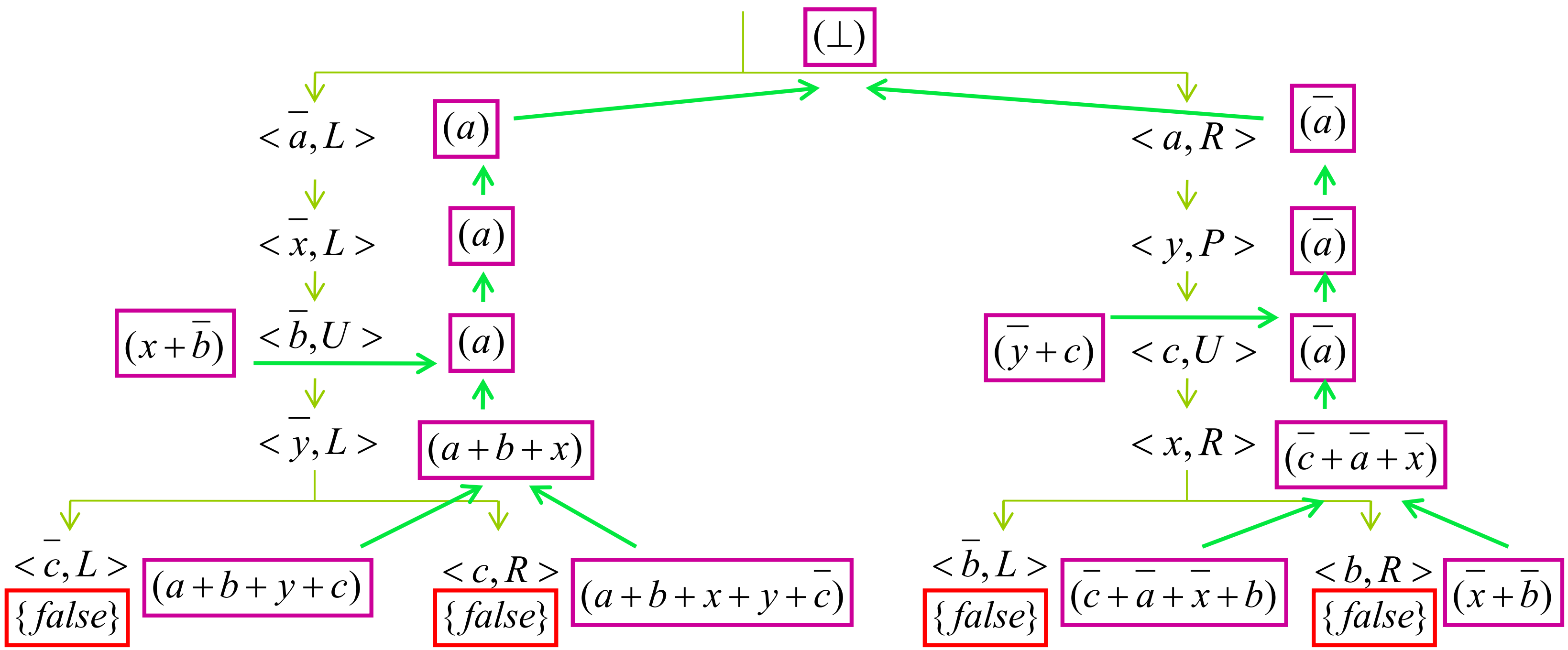
  where operator ∀-RED removes from $C_1 \lor C_2$ the universally (∀) quantified variables whose quantification levels are greater than any of the existentially (∃) quantified variables in $C_1 \lor C_2$
  - E.g.,
    prefix: $\forall a\ \exists b\ \forall c\ \forall d\ \exists e$
    $\forall$-RED$(a+b+c+d) = (a+b)$

- Q-resolution is complete for QBF solving
  - A PCNF formula is unsatisfiable if and only if there exists a Q-resolution sequence leading to the empty clause

Q-Resolution

- Example (cont'd)

$$\exists a \forall x \exists b \forall y \exists c \quad (a+b+y+c)(a+x+b+y+\bar{c})(x+\bar{b})(\bar{y}+c)(\bar{c}+\bar{a}+\bar{x}+b)(\bar{x}+\bar{b})(a+\bar{b}+\bar{y})$$



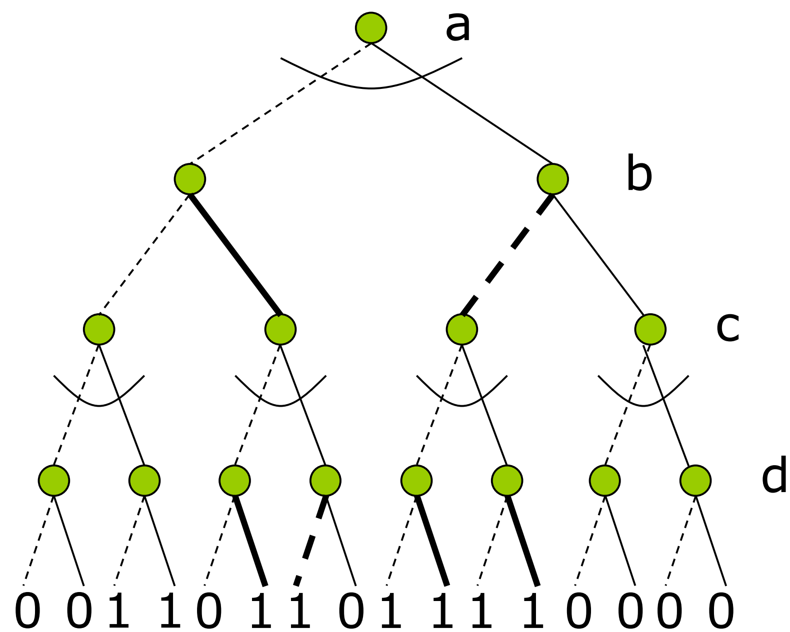
Skolemization

- Skolemization and Skolem normal form
  - Existentially quantified variables are replaced with function symbols
  - QBF prefix contains only two quantification levels
    - $\exists$ function symbols, $\forall$ variables
- Example

$\forall a\ \exists b\ \forall c\ \exists d.$
$(\neg a+\neg b)(\neg b+\neg c+\neg d)(\neg b+c+d)(a+b+c)$

Skolem functions

$\exists F_b(a)\ \exists F_d(a,c)\ \forall a\ \forall c.$
$(\neg a+\neg F_b)(\neg F_b+\neg c+\neg F_d)(\neg F_b+c+F_d)(a+F_b+c)$



QBF Certification

- QBF certification
  - Ensure correctness and, more importantly, provide useful information
  - Certificates
    - True QBF: term-resolution proof / Skolem-function (SF) model
      - SF model is more useful in practical applications
    - False QBF: clause-resolution proof / Herbrand-function (HF) countermodel
      - HF countermodel is more useful in practical applications

QBF Certification

- Unified QBF certification

formula negation

True QBF ⟷ **False QBF**

True QBF	**False QBF**
Cube resolution proof	Clause resolution proof
↓ ResQu	↓ ResQu
Skolem function (model)	Herbrand function (countermodel)

ResQu

- A Skolem-function model (Herbrand-function countermodel) for a true (false) QBF can be derived from its cube (clause) resolution proof

- A **Right-First-And-Or (RFAO) formula** is recursively defined as follows.

  φ := clause | cube | clause $\wedge$ φ | cube $\vee$ φ

  - E.g.,
    (a'+b) $\wedge$ ac $\vee$ (b'+c') $\wedge$ bc
    = ((a'+b) $\wedge$ (ac $\vee$ ((b'+c') $\wedge$ bc)))

ResQu

Countermodel_construct

input: a false QBF Φ and its clause-resolution DAG $G_\Pi(V_\Pi, E_\Pi)$
output: a countermodel in RFAO formulas
begin
  foreach universal variable x of Φ
    `RFAO_node_array[`x`]` $:= \emptyset$;
  foreach vertex v of G_Π in topological order
    if $v.clause$ resulted from $\forall$-reduction on $u.clause$, i.e., $(u, v) \in E_\Pi$
      $v.cube := \neg(v.clause)$;
      foreach universal variable x reduced from $u.clause$ to get $v.clause$
        if x appears as positive literal in $u.clause$
          push $v.clause$ to `RFAO_node_array[`x`]`;
        else if x appears as negative literal in $u.clause$
          push $v.cube$ to `RFAO_node_array[`x`]`;
    if $v.clause$ is the empty clause
      foreach universal variable x of Φ
        simplify `RFAO_node_array[`x`]`;
      return `RFAO_node_array`'s;
end

ResQu

- Example
  - ∃a∀x∃b∀y∃c

$$(a+b+y+c)_1(a+x+b+y+\bar{c})_2(x+\bar{b})_3(\bar{y}+c)_4(\bar{a}+\bar{x}+b+\bar{c})_5(\bar{x}+\bar{b})_6(a+\bar{b}+\bar{y})_7$$

$$(a+\bar{b}+\bar{y})_7 \xrightarrow{(1)} (a+\bar{b})_{7+}$$

$$(a+x+b+y)_8 \xrightarrow{(2)} (a+x+b)_{8+}$$

$$(a+x)_9 \xrightarrow{(3)} (a)_{9+}$$

$$(\bar{a}+\bar{x}+b+\bar{y})_{10} \xrightarrow{(4)} (\bar{a}+\bar{x}+b)_{10+}$$

$$(\bar{a}+\bar{x})_{11} \xrightarrow{(5)} (\bar{a})_{11+}$$

$$(empty)$$

0.	$x : [\,]$	$y : [\,]$
1.	$x : [\,]$	$y : [\,cube(\bar{a}b)\,]$
2.	$x : [\,]$	$y : [\,cube(\bar{a}b),\ clause(a+x+b)\,]$
3.	$x : [\,clause(a)\,]$	$y : [\,cube(\bar{a}b),\ clause(a+x+b)\,]$
4.	$x : [\,clause(a)\,]$	$y : [\,cube(\bar{a}b),\ clause(a+x+b),\ cube(ax\bar{b})\,]$
5.	$x : [\,clause(a),\ cube(a)\,]$	$y : [\,cube(\bar{a}b),\ clause(a+x+b),\ cube(ax\bar{b})\,]$

QBF Certification

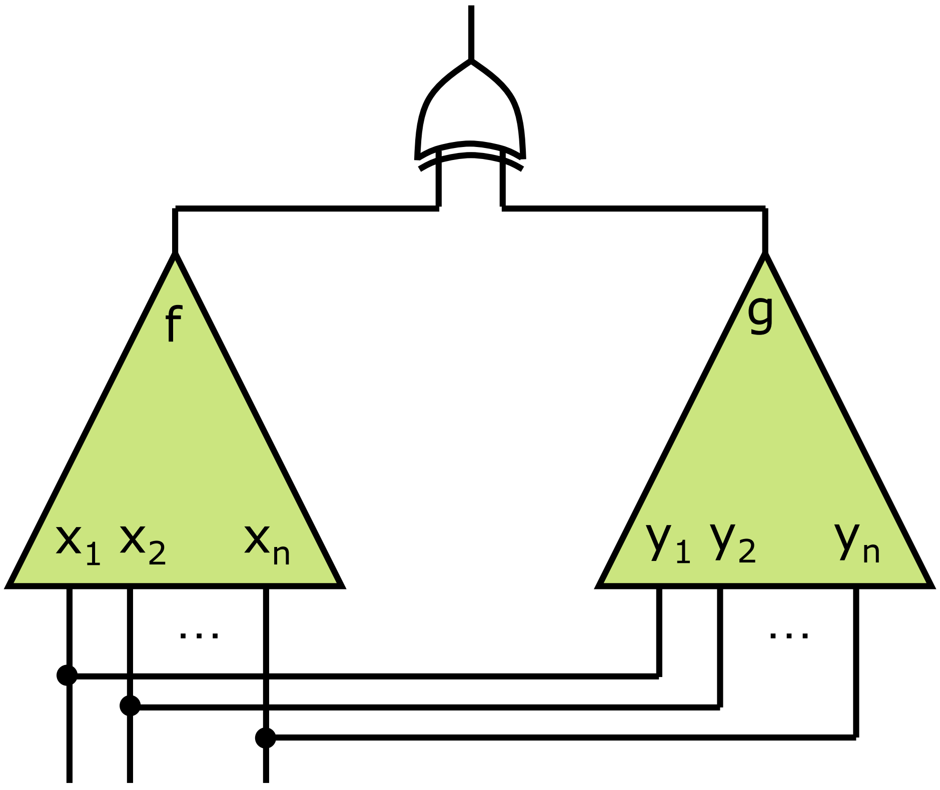
- Applications of Skolem/Herbrand functions
  - Program synthesis
  - Winning strategy synthesis in two player games
  - Plan derivation in AI
  - Logic synthesis
  - …

QSAT & Logic Synthesis

Boolean Matching

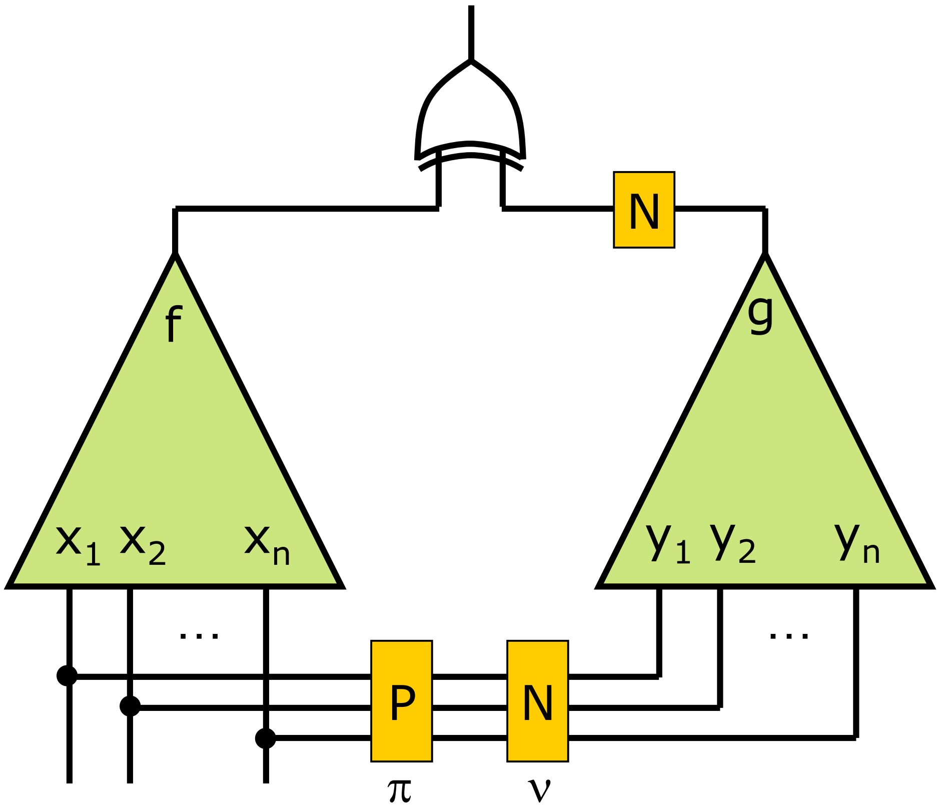
Introduction

- Combinational equivalence checking (CEC)
  - Known input correspondence
  - coNP-complete
  - Well solved in practical applications



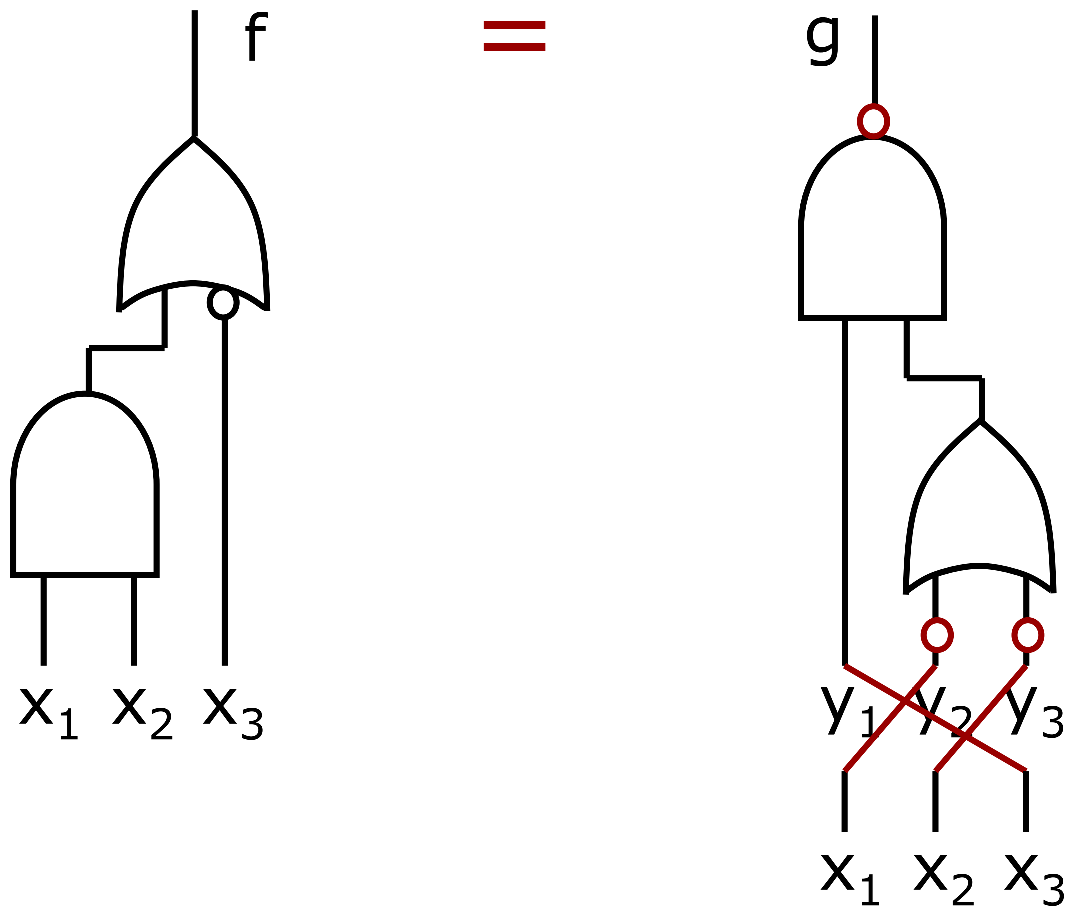
Introduction

- Boolean matching
  - P-equivalence
    - Unknown input permutation
    - O(n!) CEC iterations
  - NP-equivalence
    - Unknown input negation and permutation
    - $O(2^n n!)$ CEC iterations
  - NPN-equivalence
    - Unknown input negation, input permutation, and output negation
    - $O(2^{n+1} n!)$ CEC iterations



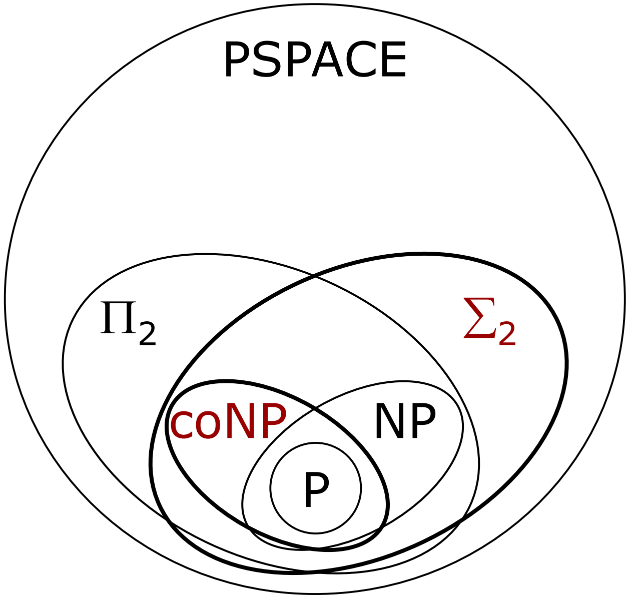
Introduction

- Example



Introduction

- Motivations
  - Theoretically
    - Complexity in between
      coNP (for all …) and
      Σ_2 (there exists … for all …)
      in the Polynomial Hierarchy (PH)
      - Special candidate to test PH collapse
    - Known as Boolean congruence/isomorphism dating back to the 19th century
  - Practically
    - Broad applications
      - Library binding
      - FPGA technology mapping
      - Detection of generalized symmetry
      - Logic verification
      - Design debugging/rectification
      - Functional engineering change order
    - Intensively studied over the last two decades



Introduction

- Prior methods

	Complete ?	Function type	Equivalence type	Solution type	Scalability
Spectral methods	yes	CS	mostly P	one	– –
Signature based methods	no	mostly CS	P/NP	N/A	– ~ ++
Canonical-form based methods	yes	CS	mostly P	one	+
SAT based methods	yes	CS	mostly P	one/all	+
BooM (QBF/SAT-like)	yes	CS / IS	NPN	one/all	++

CS: completely specified
IS: incompletely specified

BooM: A Fast Boolean Matcher

- Features of BooM
  - General computation framework
  - Effective search space reduction techniques
    - **Dynamic learning** and **abstraction**
  - Theoretical SAT-iteration upper-bound:

$\cancel{O(2^n n!)}$ $\quad$ $O(2^{2n})$

Formulation

- Reduce NPN-equiv to 2 NP-equiv checks
  - Matching f and g; matching f and $\neg$g
- 2^{nd} order formula of NP-equivalence

  $\exists \nu \circ \pi, \forall x\ ((f_c(x) \wedge g_c(\nu \circ \pi(x))) \Rightarrow (f(x) \equiv g(\nu \circ \pi(x))))$

  - f_c and g_c are the care conditions of f and g, respectively
- Need 1^{st} order formula instead for SAT solving

Formulation

- 0-1 matrix representation of ν ∘ π

$$
\begin{array}{c c}
 & \begin{array}{cccccccc} x_1 & \neg x_1 & x_2 & \neg x_2 & \cdots & x_n & \neg x_n \end{array} \\
\begin{array}{c} y_1 \\ y_2 \\ \vdots \\ y_n \end{array} &
\begin{pmatrix}
a_{11} & b_{11} & a_{12} & b_{12} & \cdots & a_{1n} & b_{1n} \\
a_{21} & b_{21} & a_{22} & b_{22} & \cdots & a_{2n} & b_{2n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
a_{n1} & b_{n1} & a_{n2} & b_{n2} & \cdots & a_{nn} & b_{nn}
\end{pmatrix}
\end{array}
\quad \sum = 1
$$

$\sum = 1$

$a_{ij} \Rightarrow (x_j \equiv y_i)$

$b_{ij} \Rightarrow (\neg x_j \equiv y_i)$

Formulation

- Quantified Boolean formula (QBF) for NP-equivalence

  $$\exists a, \exists b, \forall x, \forall y\ (\varphi_C \wedge \varphi_A \wedge ((f_c \wedge g_c) \Rightarrow (f \equiv g))$$

  - φ_C: cardinality constraint
  - φ_A: $\bigwedge_{i,j} (a_{ij} \Rightarrow (y_i \equiv x_j))\ (b_{ij} \Rightarrow (y_i \equiv \neg x_j))$

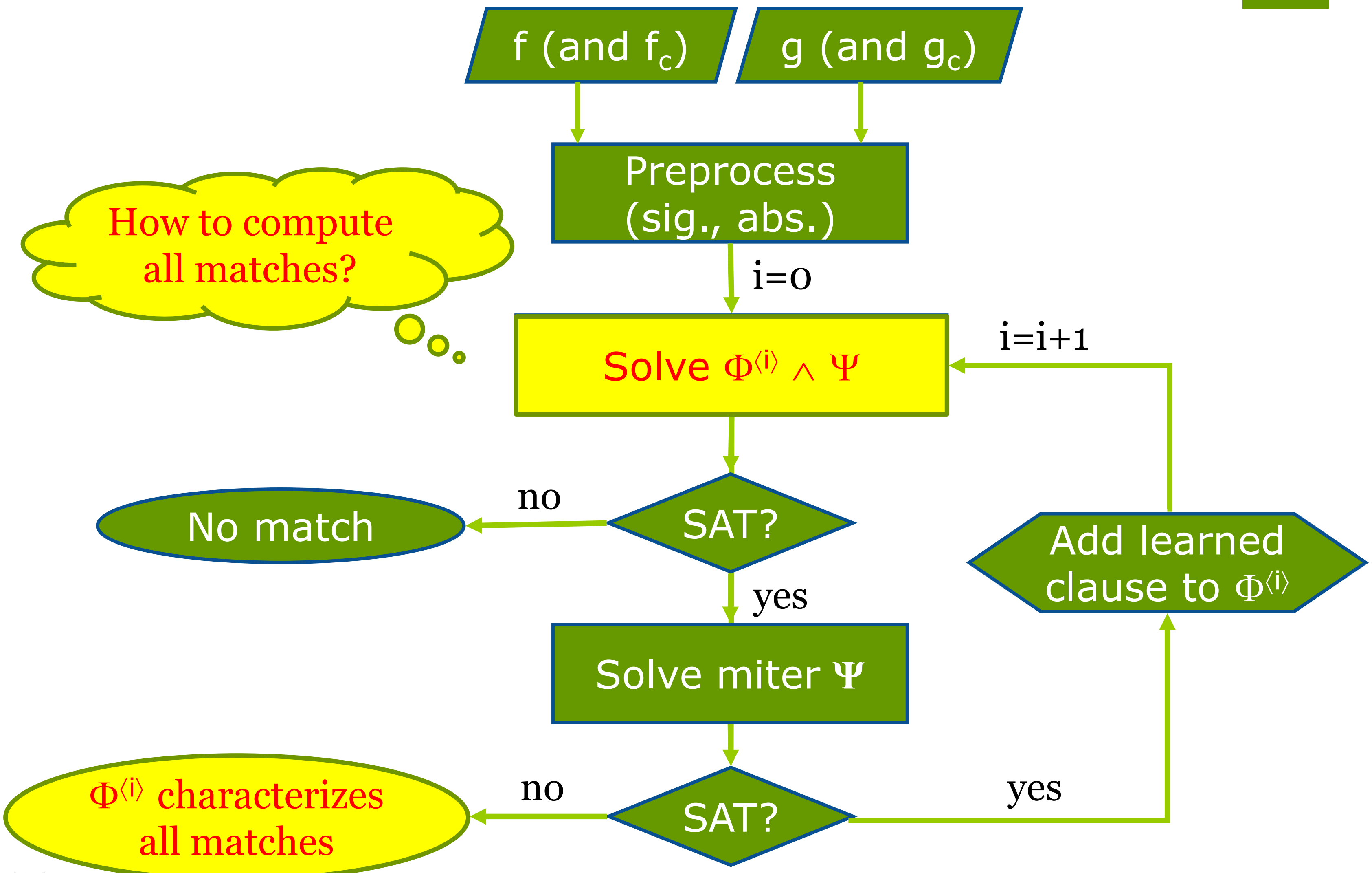
- Look for an assignment to a- and b-variables that satisfies φ_C and makes the **miter constraint**

  $$\Psi = \varphi_A \wedge (f \neq g) \wedge f_c \wedge g_c$$

  unsatisfiable

- Refine φ_C iteratively in a sequence $\Phi^{\langle 0 \rangle}, \Phi^{\langle 1 \rangle}, \ldots, \Phi^{\langle k \rangle}$, for $\Phi^{\langle i+1 \rangle} \Rightarrow \Phi^{\langle i \rangle}$ through **conflict-based learning**

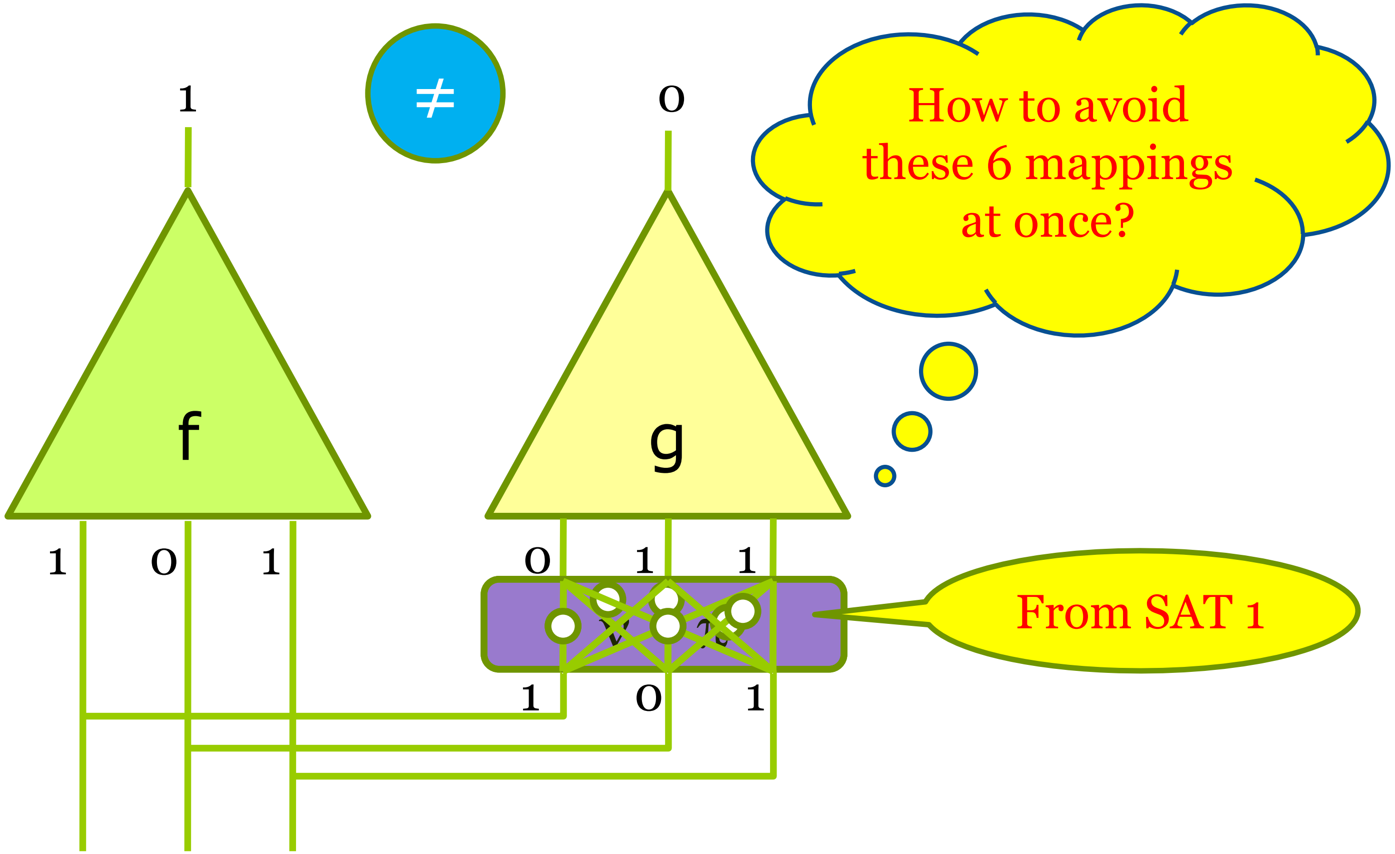
BooM Flow



f (and f_c)
g (and g_c)
Preprocess
(sig., abs.)
i=0
Solve $\Phi^{\langle i \rangle} \wedge \Psi$
i=i+1
How to compute
all matches?
no
No match
SAT?
yes
Add learned
clause to $\Phi^{\langle i \rangle}$
Solve miter Ψ
no
SAT?
yes
$\Phi^{\langle i \rangle}$ characterizes
all matches

NP-Equivalence
Conflict-based Learning

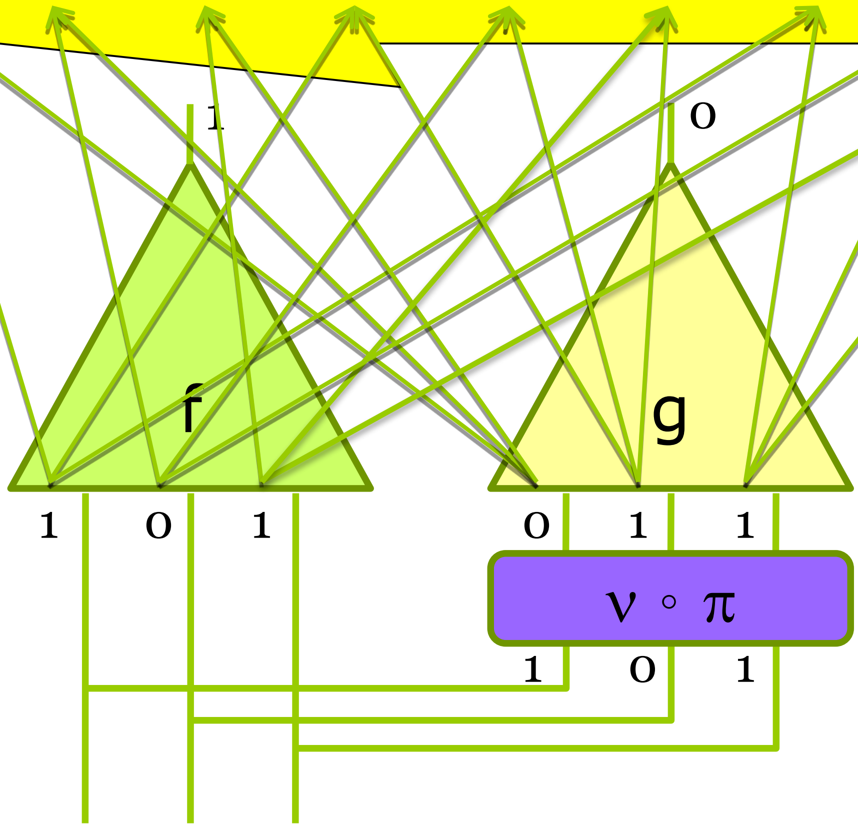
- Observation



NP-Equivalence
Conflict-based Learning

- Learnt clause generation

$(a_{11} \vee b_{12} \vee a_{13} \vee b_{21} \vee a_{22} \vee b_{23} \vee b_{31} \vee a_{32} \vee b_{33})$



NP-Equivalence Conflict-based Learning

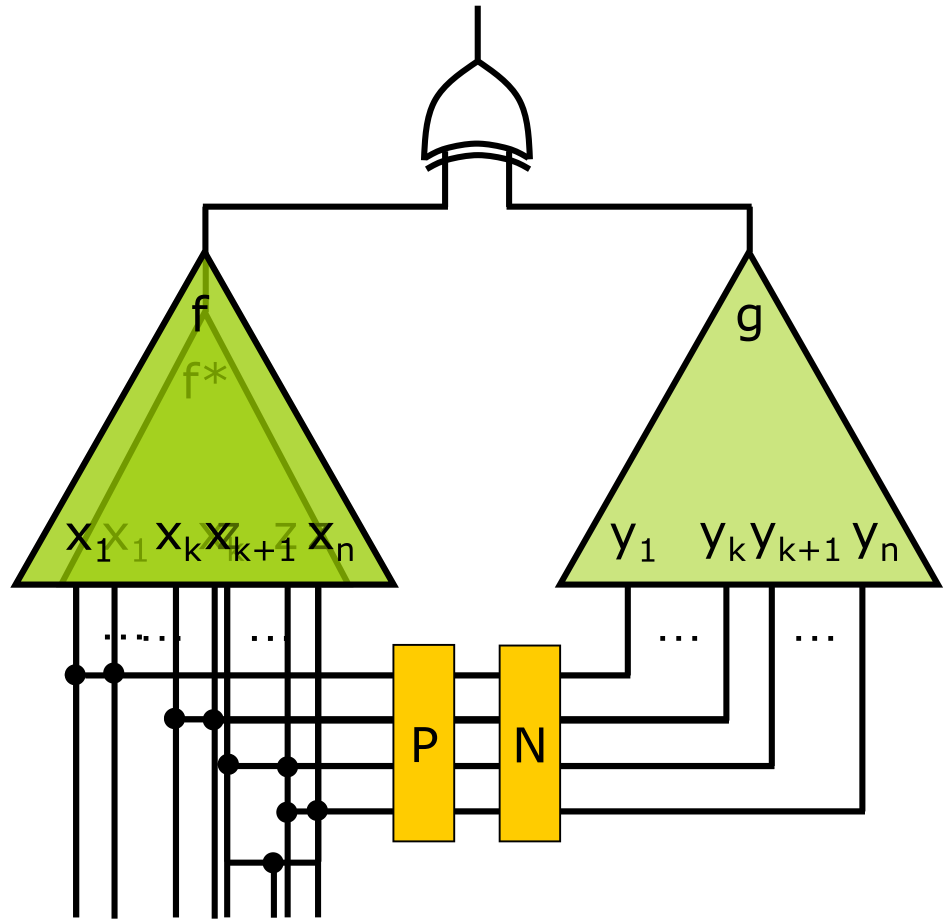
- Proposition:
  If $f(u) \neq g(v)$ with $v = \nu \circ \pi(u)$ for some $\nu \circ \pi$ satisfying $\Phi^{\langle i \rangle}$, then the learned clause $\bigvee_{ij} l_{ij}$ for literals
  $l_{ij} = (v_i \neq u_j)$? a_{ij} : b_{ij}
  excludes from $\Phi^{\langle i \rangle}$ the mappings $\{\nu' \circ \pi' \mid \nu' \circ \pi'(u) = \nu \circ \pi(u)\}$

- Proposition:
  The learned clause prunes n! infeasible mappings

- Proposition:
  The refinement process $\Phi^{\langle 0 \rangle}, \Phi^{\langle 1 \rangle}, \ldots, \Phi^{\langle k \rangle}$ is bounded by 2^{2n} iterations

NP-Equivalence Abstraction

- Abstract Boolean matching
  - Abstract $f(x_1,\ldots,x_k,x_{k+1},\ldots,x_n)$ to $f(x_1,\ldots,x_k,z,\ldots,z) = f^*(x_1,\ldots,x_k,z)$
  - Match $g(y_1,\ldots,y_n)$ against $f^*(x_1,\ldots,x_k,z)$
  - Infeasible matching solutions of f^* and g are also infeasible for f and g



NP-Equivalence Abstraction

- Abstract Boolean matching
  - Similar matrix representation of negation/permutation

$$
\begin{array}{cc}
 & \begin{array}{ccccccc} x_1^* & \neg x_1^* & \cdots & x_k^* & \neg x_k^* & z & \neg z \end{array} \\
\begin{array}{c} y_1 \\ y_2 \\ \vdots \\ y_n \end{array} &
\begin{pmatrix}
a_{11} & b_{11} & \cdots & a_{1k} & b_{1k} & a_{1(k+1)} & b_{1(k+1)} \\
a_{21} & b_{21} & \cdots & a_{2k} & b_{2k} & a_{2(k+1)} & b_{2(k+1)} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\
a_{n1} & b_{n1} & \cdots & a_{nk} & b_{nk} & a_{n(k+1)} & b_{n(k+1)}
\end{pmatrix}
\end{array}
\quad \sum = 1
$$

$$\sum = 1$$

- Similar cardinality constraints, except for allowing multiple y-variables mapped to z

NP-Equivalence Abstraction

- Used for preprocessing
- Information learned for abstract model is valid for concrete model
- Simplified matching in reduced Boolean space

P-Equivalence
Conflict-based Learning

- Proposition:

  If $f(u) \neq g(v)$ with $v = \pi(u)$ for some π satisfying $\Phi^{\langle i \rangle}$, then the learned clause $\bigvee_{ij} l_{ij}$ for literals

  $l_{ij} = (v_i{=}0 \text{ and } u_j{=}1)\ ?\ a_{ij} : \varnothing$

  excludes from $\Phi^{\langle i \rangle}$ the mappings $\{\pi' \mid \pi'(u) = \pi(u)\}$

P-Equivalence Abstraction

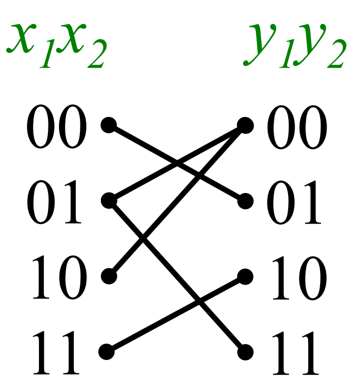
- Abstraction enforces search in biased truth assignments and makes learning strong
  - For f* having k support variables, a learned clause converted back to the concrete model consists of at most (k–1)(n–k+1) literals

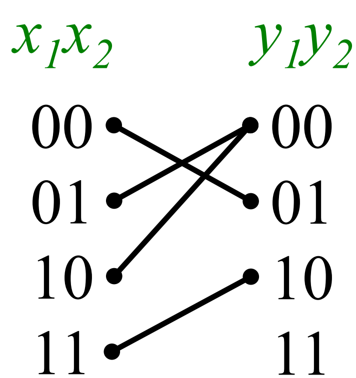
QSAT & Logic Synthesis
Relation Determinization

Relation vs. Function

- Relation *R(X, Y)*
  - Allow one-to-many mappings
    - Can describe non-deterministic behavior
  - More generic than functions

- Function *F(X)*
  - Disallow one-to-many mappings
    - Can only describe deterministic behavior
  - A special case of relation

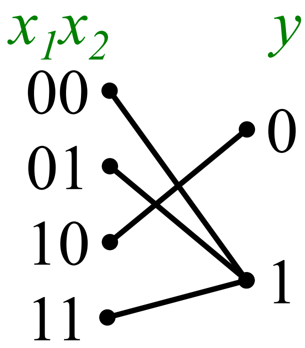




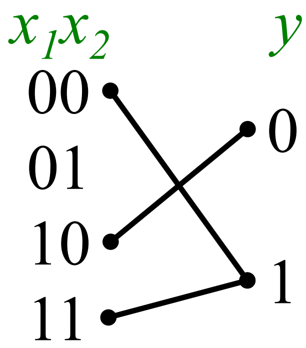
$f_1 = x_1 x_2$
$f_2 = \neg x_1 \neg x_2$

Relation

- Total relation
  - Every input element is mapped to at least one output element

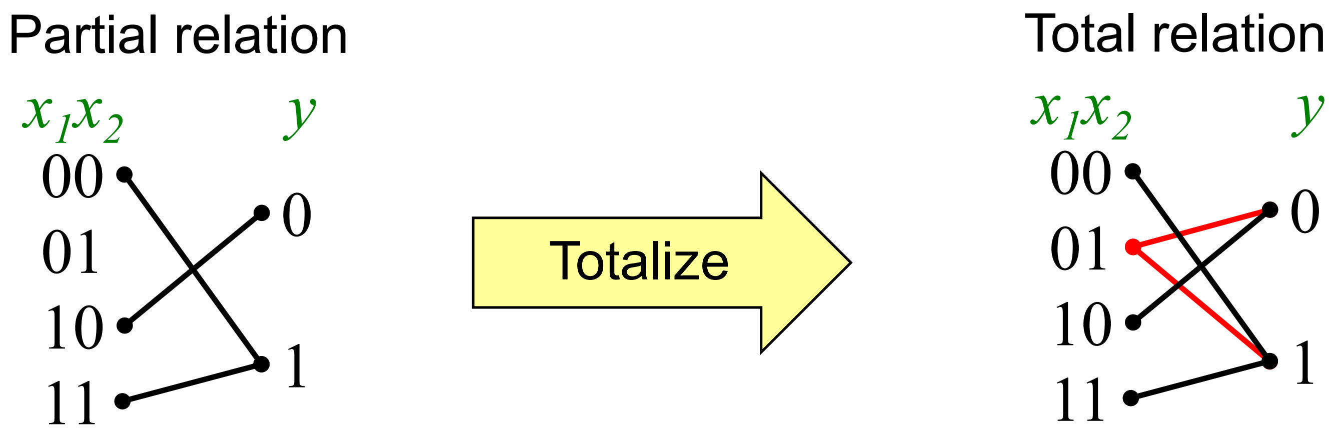


- Partial relation
  - Some input element is not mapped to any output element



Relation

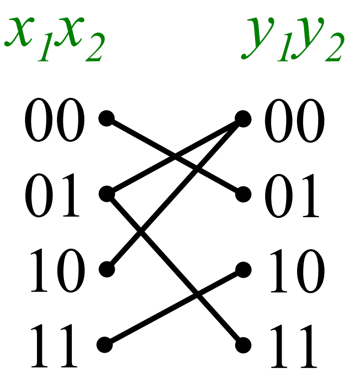
- A partial relation can be **totalized**
  - Assume that the input element not mapped to any output element is a don't care



$$T(X, y) = R(X, y) \vee \forall y.\ \neg R(X, y)$$

Motivation

- Applications of Boolean relation
  - In high-level design, Boolean relations can be used to describe (nondeterministic) specifications
  - In gate-level design, Boolean relations can be used to characterize the flexibility of sub-circuits
    - Boolean relations are more powerful than traditional don't-care representations

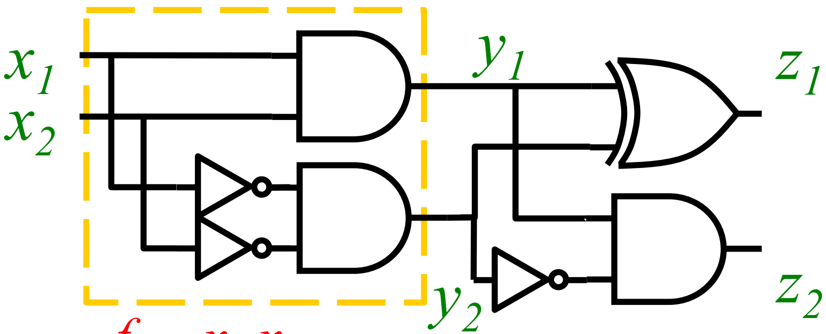


Motivation

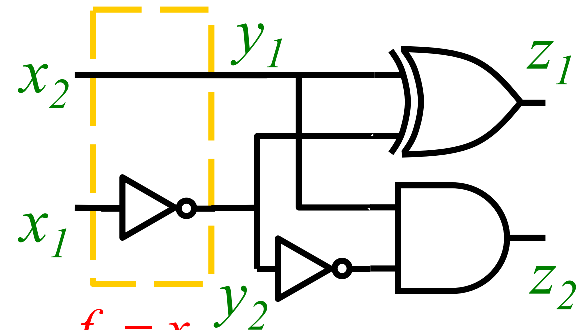
- Relation determinization
  - For hardware implement of a system, we need functions rather than relations
    - Physical realization are deterministic by nature
    - One input stimulus results in one output response
  - To simplify implementation, we can explore the flexibilities described by a relation for optimization

Motivation

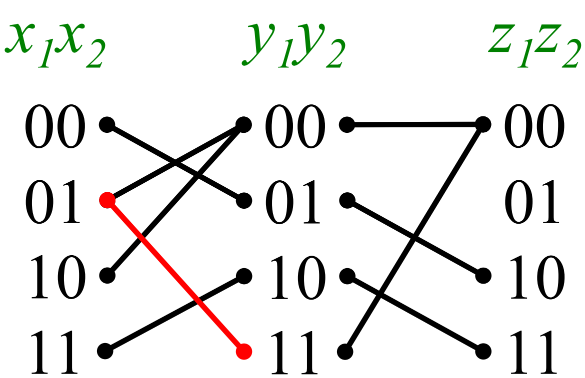
- Example



$f_1 = x_1\, x_2$
$f_2 = \neg x_1 \neg x_2$



$f_1 = x_2$
$f_2 = \neg x_1$



Relation Determinization

- Given a *nondeterministic* Boolean relation $R(X, Y)$, how to determinize and extract functions from it?

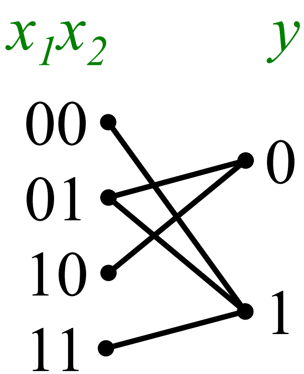
- For a deterministic total relation, we can uniquely extract the corresponding functions

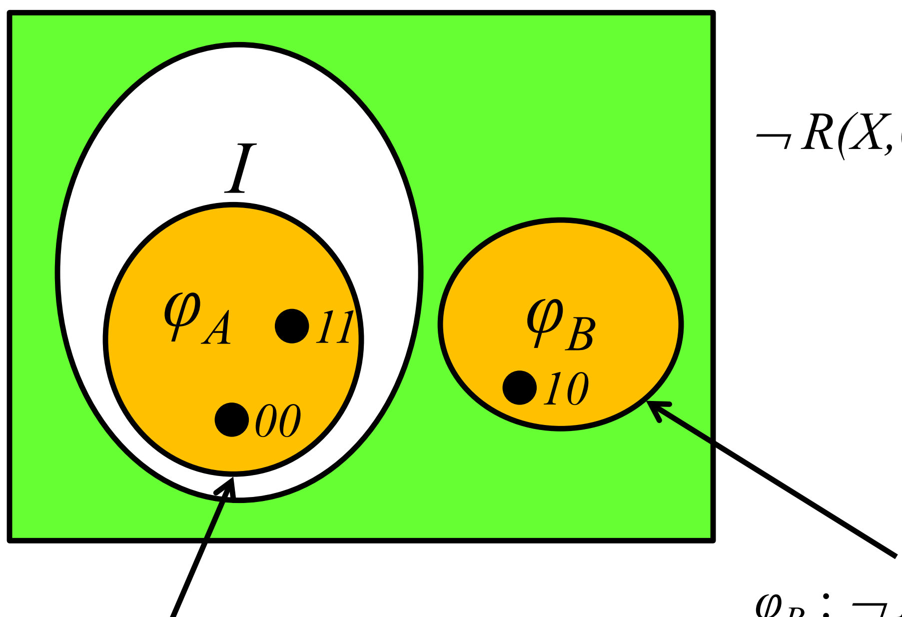
Relation Determinization

- Approaches to relation determinization
  - Iterative method (determinize one output at a time)
    - BDD- or SOP-based representation
      - Not scalable
      - Better optimization
    - AIG representation
      - Focus on scalability with reasonable optimization quality
  - Non-iterative method (determinize all ouputs at once)
    - QBF solving

Iterative Relation Determinization

- Single-output relation
  - For a single-output total relation $R(X, y)$, we derive a function f for variable y using interpolation





$\neg R(X,0) \wedge \neg R(X,1)$ UNSAT

$\varphi_A : \neg R(X,0)$
Minimal care onset of f

$\varphi_B : \neg R(X,1)$
Minimal care offset of f

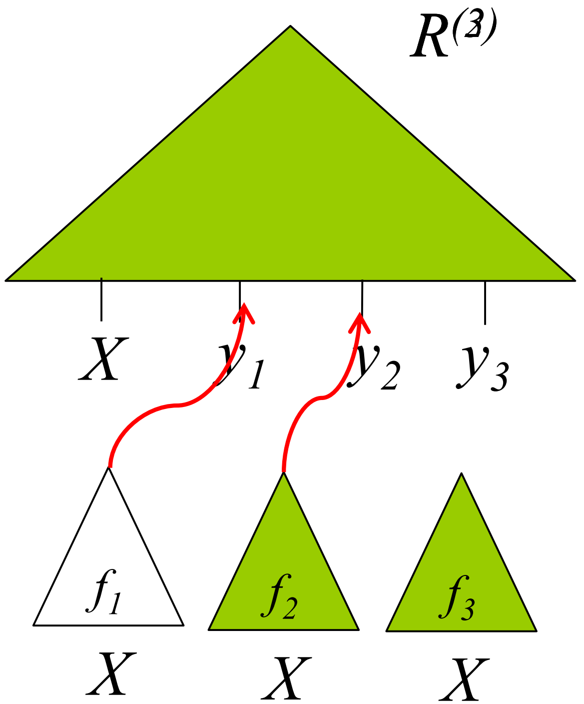
Iterative Relation Determinization

- Multi-output relation
  - Two-phase computation:
    1. Backward reduction
       - Reduce to single-output case

         $R(X, y_1, \ldots, y_n) \to \exists y_2, \ldots, \exists y_n.\, R(X, y_1, \ldots, y_n)$
    2. Forward substitution
       - Extract functions

Iterative Relation Determinization

- Example



Phase1: (expansion reduction)

$$\exists y_3.R(X, y_1, y_2, y_3) \rightarrow R^{(3)}(X, y_1, y_2)$$
$$\exists y_2.R^{(3)}(X, y_1, y_2) \rightarrow R^{(2)}(X, y_1)$$

Phase2:

$$R^{(2)}(X, y_1) \rightarrow y_1 = f_1(X)$$
$$R^{(3)}(X, y_1, y_2) \rightarrow R^{(3)}(X, f_1(X), y_2) \rightarrow y_2 = f_2(X)$$
$$R(X, y_1, y_2, y_3) \rightarrow R(X, f_1(X), f_2(X), y_2) \rightarrow y_3 = f_3(X)$$

Non-Iterative Relation Determinization

- Solve QBF

  $\forall x_1, \ldots, \forall x_m, \exists y_1, \ldots, \exists y_n.\ R(x_1, \ldots, x_m, y_1, \ldots, y_n)$

  - The Skolem functions of variables $y_1, \ldots, y_n$ correspond to the functions we want

Dependency Quantified Boolean Satisfiability

Dependency Quantified Boolean Formula

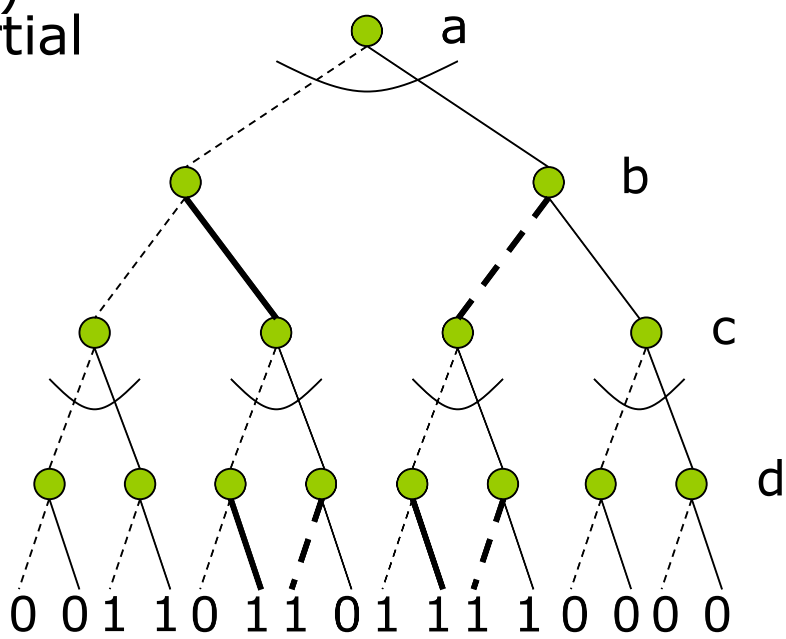
- A dependency quantified Boolean formula (DQBF) is commonly written in a **prenex form** as

$$\Phi = \underbrace{\forall X, \exists y_1(D_1), \dots, \exists y_m(D_m)}_{\text{prefix}}.\ \underbrace{\varphi}_{\text{matrix}}$$

  for $D_i \subseteq X$ being the *dependency set* of y_i and φ a quantifier-free formula
- Φ is true if and only if there exist Skolem functions $f_i(D_i)$ for y_i such that $\varphi|_{f_1(D_1)/y_1,\dots,f_m(D_m)/y_m}$ is a tautology

Dependency Quantified Boolean Formula

- A game interpretation of DQBF
  - Multi-player game played between ∀-player (to falsity the formula) and multiple ∃-players with partial information (to satisfy the formula)



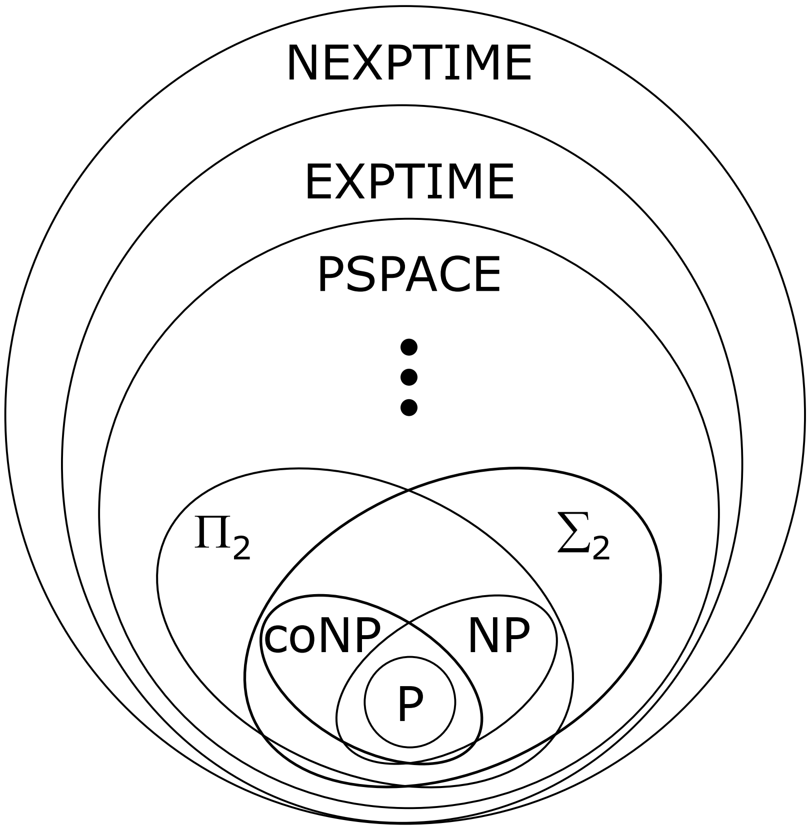
$\forall a\ \forall c\ \exists b(a)\ \exists d(c).$
$(\neg a+\neg b)(\neg b+\neg c+\neg d)(\neg b+c+d)(a+b+c)$

Skolem functions

$\exists F_b(a)\ \exists F_d(c)\ \forall a\ \forall c.$
$(\neg a+\neg F_b)(\neg F_b+\neg c+\neg F_d)(\neg F_b+c+F_d)(a+F_b+c)$

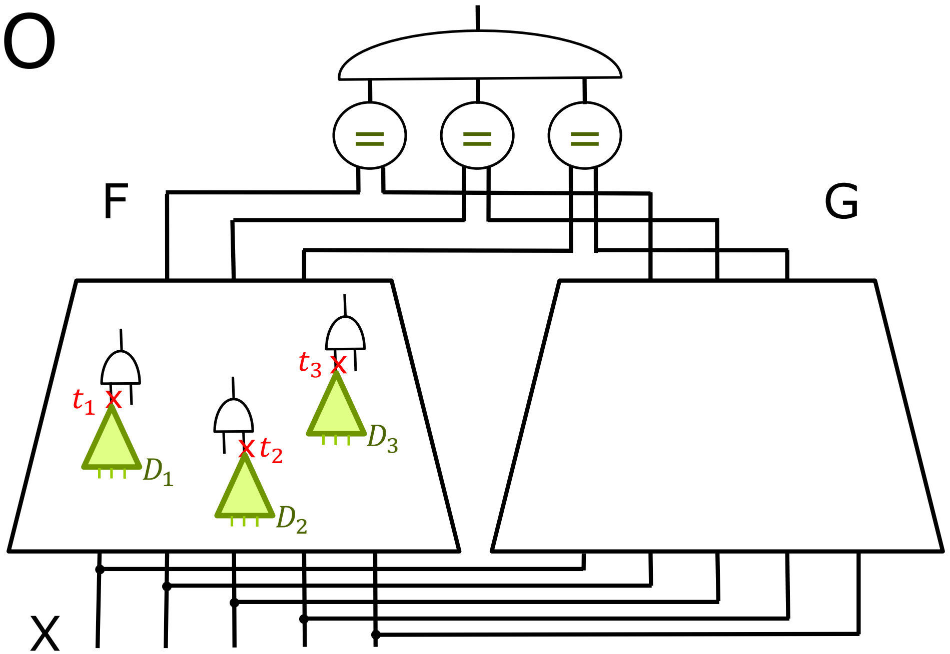
Dependency Quantified Boolean Formula

- Deciding DQBF satisfiability is NEXPTIME-complete
- DQBF solvers and preprocessors have been significantly advanced in recent years
- More applications have been identified



Application: Combinational ECO

- Combinational ECO



$$\forall X, Y, \exists T(D). \big(Y = E(X)\big) \rightarrow \big(F(X,T) = G(X)\big)$$

where Y are internal signals referred to by D_i, and E are functions of Y signals

Application: Sequential ECO

Sequential ECO

$$
\begin{aligned}
&\forall X, Y, S_1, S_2, S_1', S_2', \exists T(D), Q(S_1 \cup S_2), Q'(S_1' \cup S_2'). \\
&(I(S_1, S_2) \to Q) \land \\
&\big(Q \land \big(Y = E(X, S_1)\big) \land R(X, S_1, S_2, S_1', S_2') \to Q'\big) \land \\
&\Big(Q \to \big(F(X, S_1, T) = G(X, S_2)\big)\Big) \land \\
&((S_1, S_2) = (S_1', S_2')) \to (Q = Q')
\end{aligned}
$$

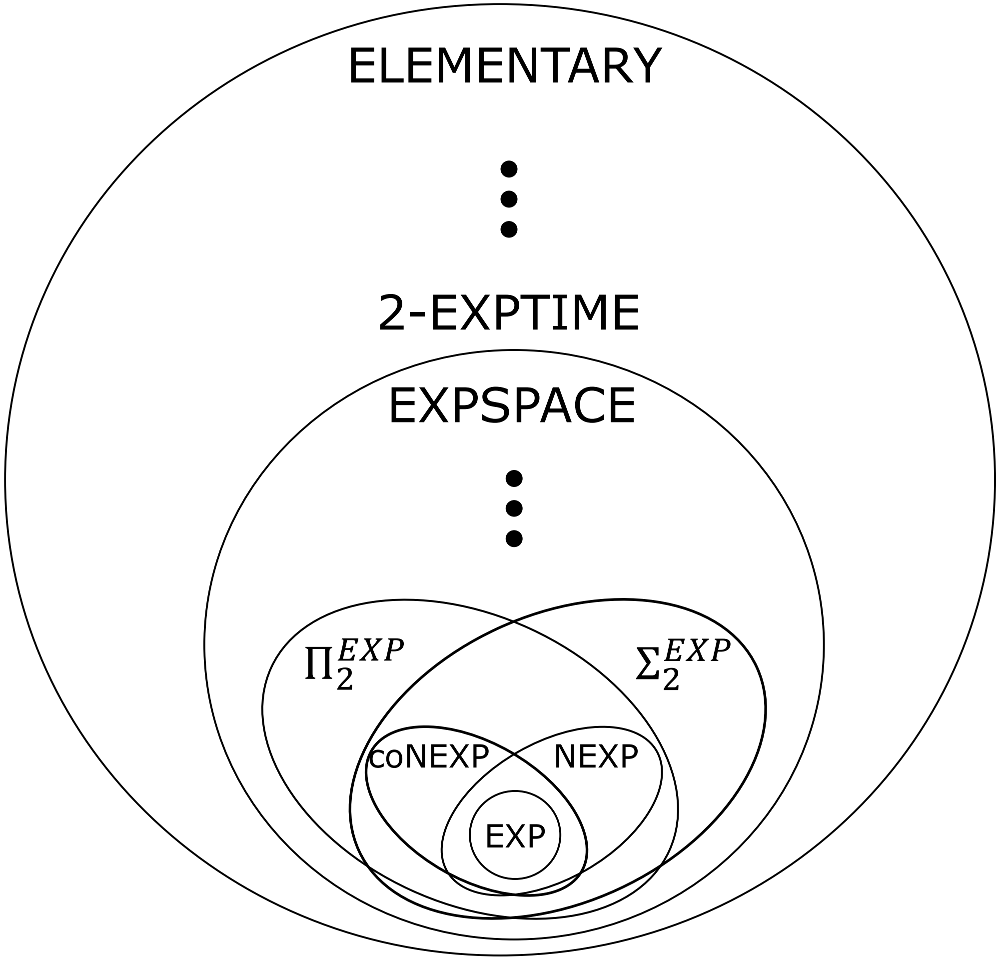
where S_1 and S_2 (S_1' and S_2') are current-state (next-state) variables of circuits F and G, respectively,
$D = \{D_i\}$ with $D_i \subseteq X \cup Y \cup S_1$, and
$R = (S_1' = \Delta_1(X, S_1, T)) \land (S_2' = \Delta_2(X, S_2))$ with Δ_1 and Δ_2 being the transition functions of circuits F and G, respectively

Second-Order Quantified Boolean Satisfiability

Motivation

- The great success of SAT-solving technology has motivated building solvers for more complex problems
  - E.g., from SAT (NP-complete) to QBF (PSPACE-complete), further to DQBF (S-form: NEXP-complete, H-form: coNEXP-complete)
- Second-order quantified Boolean formula (SOQBF) extends DQBF to the entire *Exponential Time Hierarchy* (EXPH)
  - Σ_1^{EXP}: $\exists F_1, \forall X. \varphi$ (S-form DQBF); Π_1^{EXP}: $\forall F_1, \exists X. \varphi$ (H-form DQBF)
  - Σ_2^{EXP}: $\exists F_1, \forall F_2, \exists X. \varphi$; Π_2^{EXP}: $\forall F_1, \exists F_2, \forall X. \varphi$
  - Σ_3^{EXP}: $\exists F_1, \forall F_2, \exists F_3, \forall X. \varphi$; Π_3^{EXP}: $\forall F_1, \exists F_2, \forall F_3, \exists X. \varphi$
  - …
  - SOQBF_k is Σ_k^{EXP}-complete (Π_k^{EXP}-complete) if starting with $\exists$ ($\forall$)

Complexity Classes



Although SOQBF_i well corresponds to the Exponential Hierarchy (EXPH), SOQBF is unlikely to be EXPSPACE-complete!

Syntax of SOQBF

- General form

  $$\Phi ::= 0 \mid 1 \mid x \mid f \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\Phi \mid \exists f.\Phi$$

  - x: proposition (atomic) variable, f: function variable
  - $\exists x$: first-order quantifier, $\exists f$: second-order quantifier
  - Assume each function variable f has a fixed support set, denoted $\mathbf{S}(f)$, of atomic variables
    - Convertible by Ackermann's expansion for functions with unfixed arguments
      - E.g., $f(f(x,y),z)$ can be rewritten as $\exists w.(f_1 \wedge (w \leftrightarrow f_2)) \wedge \forall x,y,z,w.((x \leftrightarrow w)(y \leftrightarrow z)) \rightarrow (f_1 \leftrightarrow f_2))$ for $\mathbf{S}(f_1) = \{w,z\}$, $\mathbf{S}(f_2) = \{x,y\}$,
- General form can be converted to prenex form via variable renaming

Syntax of SOQBF

- Prenex form

  $Q_1 F_1, Q_2 F_2, \dots, Q_n F_n, Q_{n+1} X_1, \dots, Q_{n+m} X_m. \varphi$

  - $Q_i = \{\forall, \exists\}, Q_i \neq Q_{i+1}$ for $i \in [1, n-1]$ and $i \in [n+1, n+m-1]$
  - F_i and X_j are sets of function and atomic variables, respectively
  - Each $f \in F_i$ is associated with a support set $\mathbf{S}(f) \subseteq X_1 \cup \cdots \cup X_m$
  - φ: a quantifier-free formula over variables $F_1 \cup \cdots \cup F_n \cup X_1 \cup \cdots \cup X_m$
  - SO-quantification level $lev(f) = i$ for $f \in F_i$; FO-quantification level $lev(x) = j$ for $x \in X_j$
  - Assume all valuables in an SOQBF are quantified (with no free variables)
- Prenex form with multiple levels of atomic quantifiers can be converted to prenex form with a single level of atomic quantifiers

Syntax of SOQBF

- Prenex form with a single atomic quantification level

  $Q_1 F_1, Q_2 F_2, \ldots, Q_n F_n, Q_{n+1} X. \varphi$

  - $Q_i = \{\forall, \exists\}$ for $i \in 1, \ldots, n$+1, and $Q_j \neq Q_{j+1}$ for $j \in [1, n]$
- Collapsing atomic quantifiers into one level may incur level increase in second-order quantifiers
  - E.g.,

    $Q_1 F_1, Q_2 F_2, \ldots, Q_n F_n, \forall X_1, \exists y, \forall X_2. \varphi$

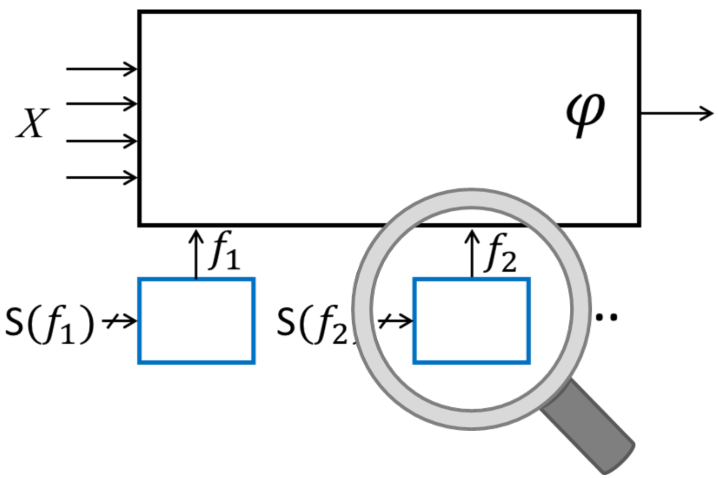
    can be converted to

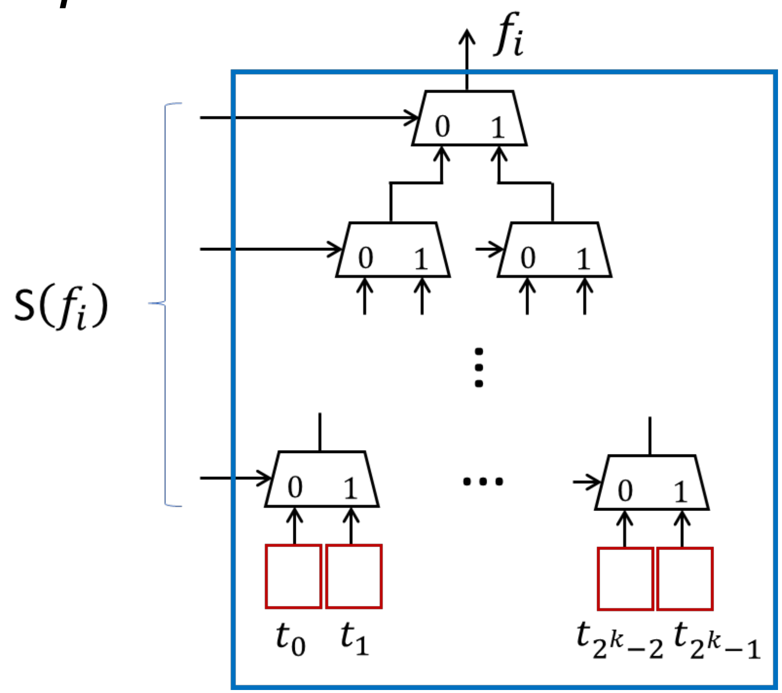
    $Q_1 F_1, Q_2 F_2, \ldots, Q_n F_n, \exists f_y, \forall X_1, \forall y, \forall X_2. \left(y \leftrightarrow f_y\right) \rightarrow \varphi$

    for $\mathbf{S}\left(f_y\right) = X_1$

Semantics of SOQBF

- Circuit representation of the matrix of $Q_1F_1, Q_2F_2, \ldots, Q_nF_n, Q_{n+1}X.\varphi$





Semantics of SOQBF

- In evaluating an SOQBF, an assignment to a function variable f_i with $|\boldsymbol{S}(f_i)| = k$ corresponds to determining the truth-table values $t_0, t_1, \ldots, t_{2^k-1}$
- Given an assignment α to all function variables $\bigcup_i F_i$, the SOQBF $\Phi = Q_1 F_1, Q_2 F_2, \ldots, Q_n F_n, Q_{n+1} X. \varphi$ under assignment α is true if the QBF $Q_{n+1} X. \varphi|_\alpha$ induced under α is true

Semantics of SOQBF

- $Q_1 F_1, Q_2 F_2, \dots, Q_n F_n, Q_{n+1} X. \varphi$ can be evaluated by a series of QBF evaluations with respect to function variable assignments that follow the prefix of the second-order quantifiers $Q_1 F_1, Q_2 F_2, \dots, Q_n F_n$
- Game-theoretic semantics
  - A two-player game interpretation: The ∃-player (∀-player) assigns existential (universal) function variables to satisfy (falsify) the formula. The prefix of the SOQBF determines the order of the players' moves. The SOQBF is true (false) iff the ∃-player (∀-player) has a winning strategy.
- An SOQBF is true if there exists a model (∃-player's winning strategy), i.e., a set of *Skolem functionals* for the existential function variables such that substituting each existential function variable with its corresponding Skolem functional makes the induced formula a tautology

Converting SOQBF to QBF

- An SOQBF can be converted to a model-equivalent QBF via *ground instantiation*, where every function variable is instantiated with respect to a full assignment over its support set
  - Iteratively eliminating the innermost atomic variable through formula expansion until no more atomic variable is left
  - Specifically,

$Q_1 F_1, Q_2 F_2, \ldots, Q_n F_n, QX, \forall y.\varphi$ is converted to $Q_1 F_1^{y} \cup F_1^{\neg y}, \ldots, F_1^{y} \cup F_1^{\neg y}, QX.\varphi|_{y} \wedge \varphi|_{\neg y}$

$Q_1 F_1, Q_2 F_2, \ldots, Q_n F_n, QX, \exists y.\varphi$ is converted to $Q_1 F_1^{y} \cup F_1^{\neg y}, \ldots, F_1^{y} \cup F_1^{\neg y}, QX.\varphi|_{y} \vee \varphi|_{\neg y}$

where $F_i^{y} = \{f^{\alpha \wedge y} \mid f^{\alpha} \in F_i, y \in \mathbf{S}(f^{\alpha})\} \cup \{f^{\alpha} \mid f^{\alpha} \in F_i, y \notin \mathbf{S}(f^{\alpha})\}$ and

$$F_i^{\neg y} = \{f^{\alpha \wedge \neg y} \mid f^{\alpha} \in F_i, y \in \mathbf{S}(f^{\alpha})\} \cup \{f^{\alpha} \mid f^{\alpha} \in F_i, y \notin \mathbf{S}(f^{\alpha})\}$$

Converting SOQBF to QBF

- Example
  - $\forall g(x_1, x_2), \exists f(x_1, x_3), \forall x_1, \exists x_2, \forall x_3. (g + f + \neg x_1 + \neg x_2 + x_3)(g + \neg f)$

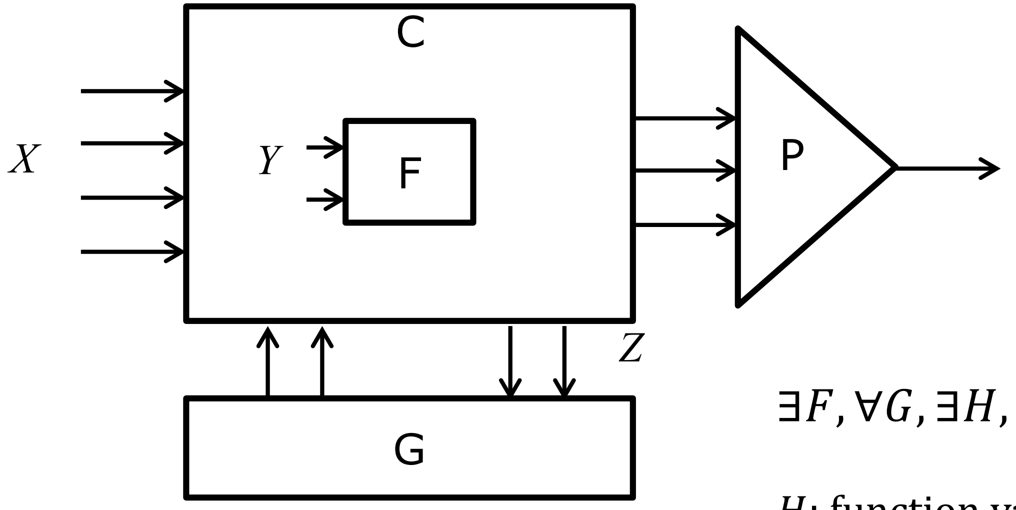
$$= \forall g(x_1, x_2), \exists f^{x_3}(x_1), f^{\neg x_3}(x_1), \forall x_1, \exists x_2.$$
$$(g + f^{\neg x_3} + \neg x_1 + \neg x_2)(g + \neg f^{\neg x_3})(g + \neg f^{x_3})$$

$$= \forall g^{x_2}(x_1), g^{\neg x_2}(x_1), \exists f^{x_3}(x_1), f^{\neg x_3}(x_1), \forall x_1.$$
$$(g^{x_2} + f^{\neg x_3} + \neg x_1)(g^{x_2} + \neg f^{\neg x_3})(g^{x_2} + \neg f^{x_3}) + (g^{\neg x_2} + \neg f^{\neg x_3})(g^{\neg x_2} + \neg f^{x_3})$$

$$= \forall g^{x_1 x_2}, g^{x_1 \neg x_2}, g^{x_1 x_2}, g^{x_1 \neg x_2}, \exists f^{x_1 x_3}, f^{x_1 \neg x_3}, f^{\neg x_1 x_3}, f^{\neg x_1 \neg x_3}.$$
$$((g^{x_1 x_2} + f^{x_1 \neg x_3})(g^{x_1 x_2} + \neg f^{x_1 \neg x_3})(g^{x_1 x_2} + \neg f^{x_1 x_3}) + (g^{x_1 \neg x_2} + \neg f^{x_1 \neg x_3})(g^{x_1 \neg x_2} + \neg f^{x_1 x_3}))$$
$$((g^{\neg x_1 x_2} + \neg f^{\neg x_1 \neg x_3})(g^{\neg x_1 x_2} + \neg f^{\neg x_1 x_3}) + (g^{\neg x_1 \neg x_2} + \neg f^{\neg x_1 \neg x_3})(g^{\neg x_1 \neg x_2} + \neg f^{\neg x_1 x_3}))$$

Application: Secure Unknown Function Synthesis

- Synthesize an unknown function F, its composition with the context C satisfies property P regardless of the operation of G



$\exists F, \forall G, \exists H, \forall X, Y, Z, W. \varphi$

H: function variables for normal form conversion
W: atomic variables for normal form conversion
$\mathbf{S}(F) = Y, \mathbf{S}(G) = Z, \mathbf{S}(H) = X \cup Y \cup Z \cup W$

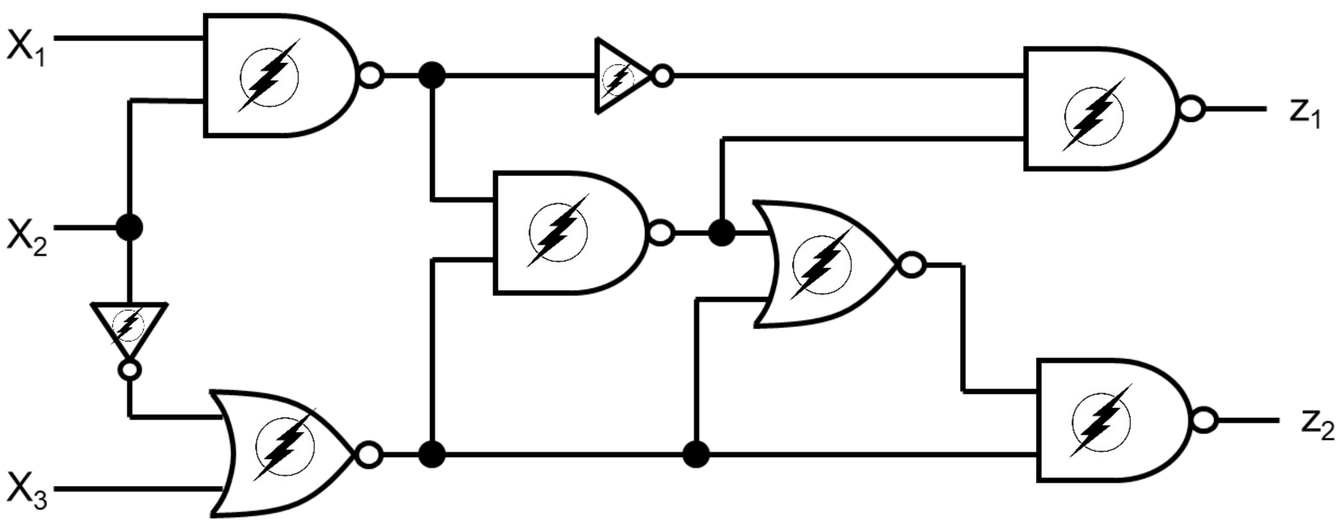
Other Applications

- Quantified bit-vector formulas of SMT
- Memory consistency checking
- Planning for agents with opposing goals

Stochastic Boolean Satisfiability

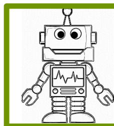
Decision under Uncertainty (Example 1)

- Evaluation of probabilistic circuits [Lee, J 14]
  - Each gate produces correct value under a certain probability
  - Query about the average output error rate, the maximum error rate under some input assignment, etc.



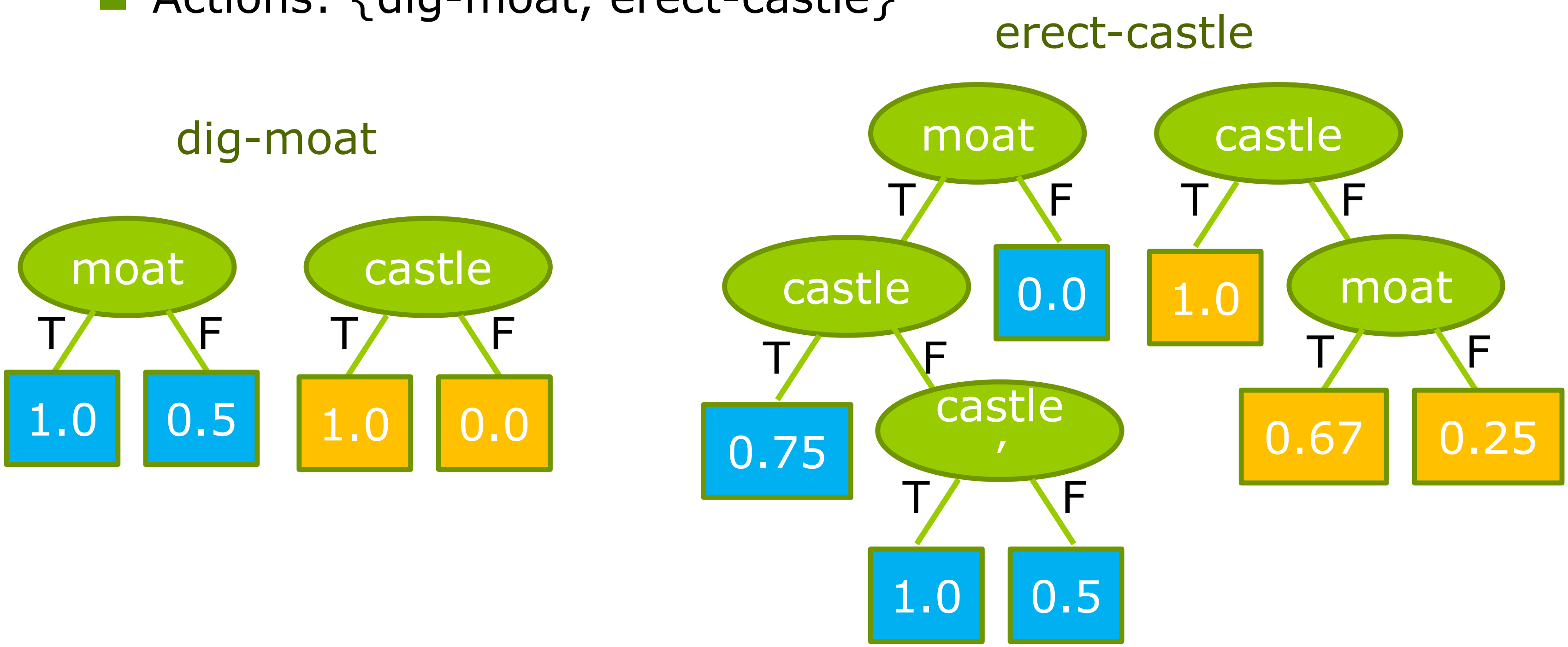
Decision under Uncertainty (Example 2)

- Probabilistic planning: Robot charge [Huang 06]
  - States: $\{S_0, \ldots, S_{15}\}$
    - Initial state: S_0; goal state: S_{15}
  - Actions: $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$
    - Succeed with prob. 0,8
    - Proceed to its right w.r.t. the intended direction with prob. 0,2

	S_1	S_2	S_3
S_4	S_5	S_6	S_7
S_8	S_9	S_{10}	S_{11}
S_{12}	S_{13}	S_{14}	

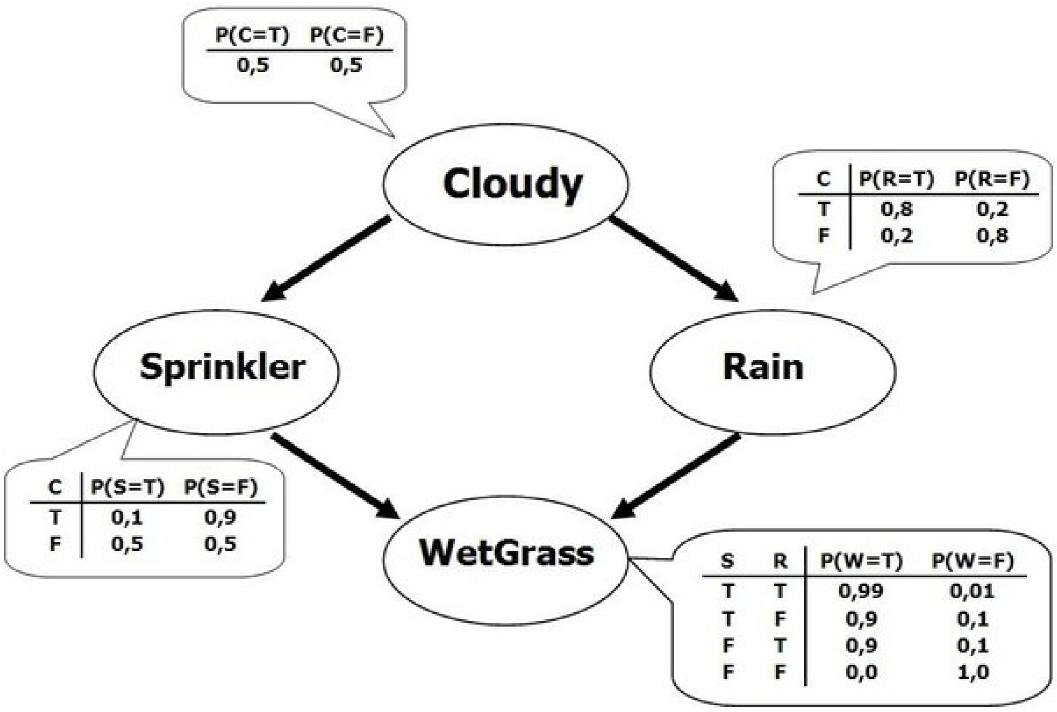
Decision under Uncertainty (Example 3)

- Probabilistic planning: Sand-Castle-67 [Majercik, Littman 98]
  - States: (moat, castle) = {(0,0), (0,1), (1,0), (1,1)}
    - Initial state: (0,0); goal states: (0,1), (1,1)
  - Actions: {dig-moat, erect-castle}



Decision under Uncertainty (Example 4)

- Belief network inference [Dechter 96, Peot 98]
  - BN queries, e.g., belief assessment, most probable explanation, maximum *a posteriori* hypothesis, maximum expected utility



From SAT to #SAT
#SAT – A Counting Problem

- The **#**SAT problem asks how many satisfying solutions are there for a given CNF formula
  - E.g., (a+¬b+c)(a+¬c)(b+d)(¬a+b) has 5 solutions, (a,b,c,d) = (0,0,0,1), (1,1,-,-)
  - A #P-complete problem
  - A.k.a. model counting
    - Exact vs. approximate model counting
    - Weighted model counting: variables are weighted under a function w:$var(\phi)$→[0,1]
      - Compute the sum of weights of satisfying assignments of ϕ

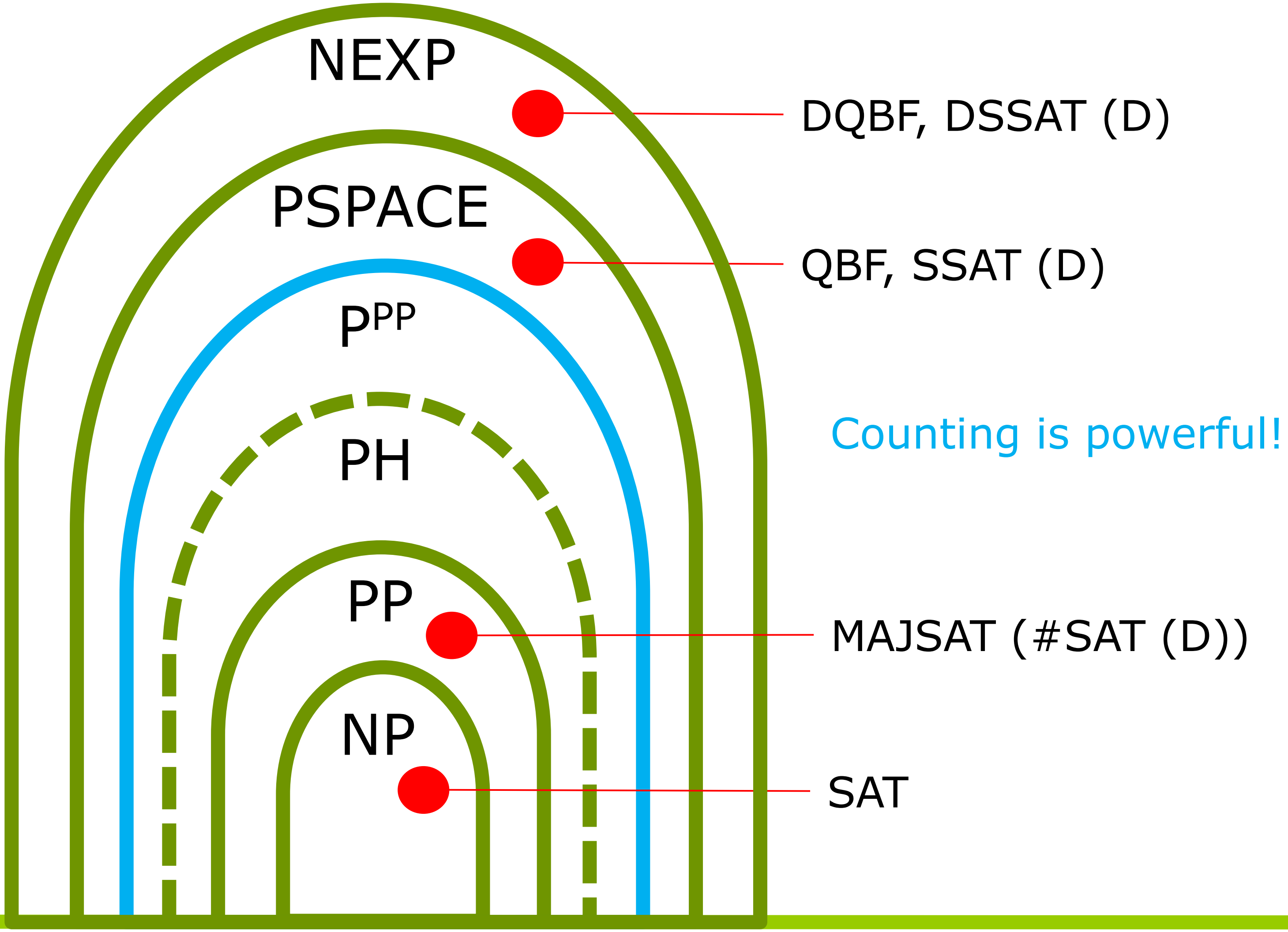
Motivation

- Decision vs. counting problems
  - SAT vs. #SAT
  - HAMILTON PATH vs. #HAMILTON PATH
  - MATCHING vs. PERMANET
  - GRAPH REACHABILITY vs. GRAPH RELIABILITY
- From correctness verification to quantitative verification
  - System reliability
  - AI planning under uncertainty

Concerned Problems in a Nutshell

- SAT: Given a CNF Boolean formula, decide its satisfiability
- #SAT: Given a CNF Boolean formula, count its number of solutions
- QBF: Given a PCNF quantified Boolean formula, decide its satisfiability
- SSAT: Given a PCNF quantified Boolean formula, maximize its satisfying probability
  - SSAT (D): decide whether its maximum satisfying probability $\geq \theta$
- DQBF: Given a PCNF dependency quantified Boolean formula, decide its satisfiability
- DSSAT: Given a PCNF dependency quantified Boolean formula, maximize its satisfying probability
  - DSSAT (D): decide whether its maximum satisfying probability $\geq \theta$

Related Complexity Classes



From QBF to SSAT
Stochastic Boolean Satisfiability

- A stochastic Boolean satisfiability (SSAT) formula is commonly written in a **prenex form** as

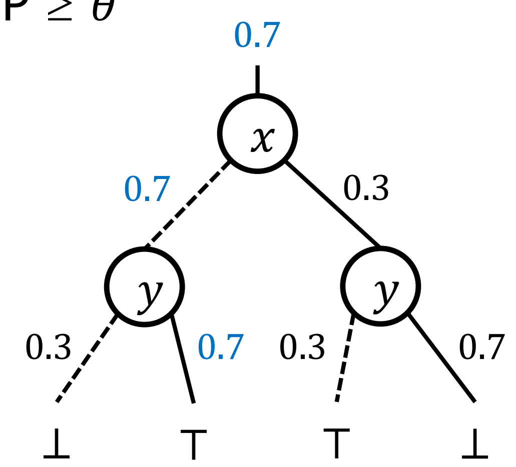
$$\Phi = \underbrace{Q_1 X_1, Q_2 X_2, \ldots, Q_n X_n}_{\text{prefix}}.\ \underbrace{\varphi}_{\text{matrix}}$$

  for $Q_i \in \{\mathcal{R}^p, \exists\}$, $Q_i \neq Q_{i+1}$, and φ a quantifier-free formula often in CNF
  - Randomized quantification $\mathcal{R}^p x$: variable x valuates to TRUE with probability p (different variables can have different probabilities)
  - A variable $x \in X_k$ is of (quantification) **level** k

From QBF to SSAT
Stochastic Boolean Satisfiability

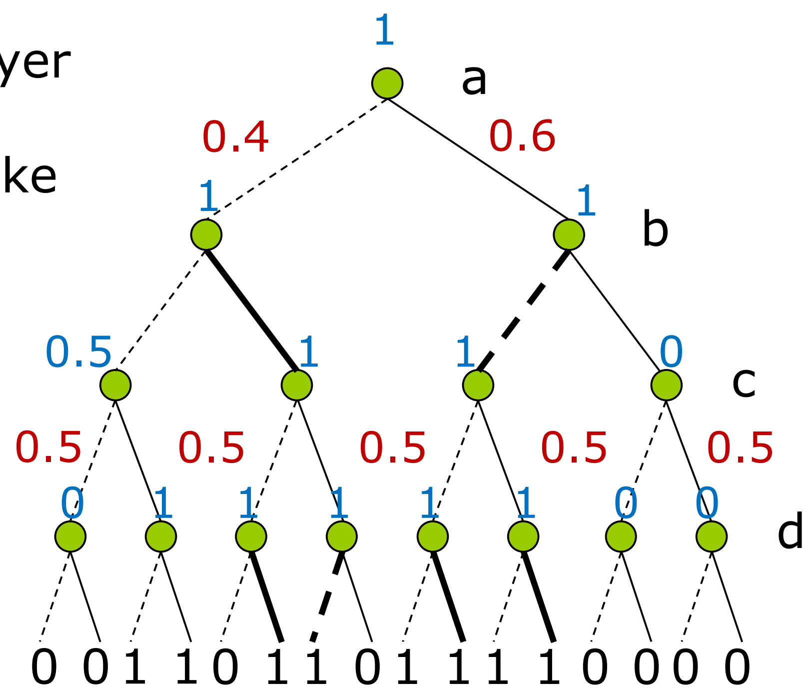
- Semantics of SSAT formula $\Phi = Q_1 v_1 \ldots Q_n v_n . \varphi(v_1, \ldots, v_n)$
  - **Satisfying probability (SP):** Expectation of satisfying φ w.r.t. the prefix structure
    - $\Pr[\top] = 1; \Pr[\bot] = 0$
    - $\Pr[\Phi] = \max\{\Pr[\Phi|_{\neg v}], \Pr[\Phi|_{v}]\}$, for outermost quantification $\exists v$
    - $\Pr[\Phi] = (1-p)\Pr[\Phi|_{\neg v}] + p\Pr[\Phi|_{v}]$, for outermost quantification $\mathcal{R}^p v$
  - Optimization version: Find the SP maximum among all assignments of existential variables
  - Decision version: Determine whether SP $\geq \theta$
  - E.g., $\Phi = \exists x, \mathcal{R}^{0.7} y. (x \vee y)(\neg x \vee \neg y)$

$\Pr[\Phi] = 0.7$



From QBF to SSAT
Stochastic Boolean Satisfiability

- A game (against nature) interpretation of SSAT
  - Two-player game played by ∃-player (to maximize the expectation of satisfaction) and $\mathcal{R}$ -player (to make random moves)



$\mathcal{R}^{0.6}a\ \exists b\ \mathcal{R}^{0.5}c\ \exists d.$
$(\neg a+\neg b)(\neg b+\neg c+\neg d)(\neg b+c+d)(a+b+c)$

Skolem functions

$\exists F_b(a)\ \exists F_d(a,c)\ \mathcal{R}^{0.6}a\ \mathcal{R}^{0.5}c.$
$(\neg a+\neg F_b)(\neg F_b+\neg c+\neg F_d)(\neg F_b+c+F_d)(a+F_b+c)$

Recent SSAT Solvers

- `ClauSSat` [CHJ22]
  - Combining QBF clause selection techniques and model counting
  - Allowing both exact and approximate solution search
- `ElimSSat` [WTJS22]
  - Solving based on quantifier elimination
- `SharpSSat` [FJ23]
  - Solving based on component analysis

Applications

- AI planning under uncertainty [Littman et al. 2001]
- Belief network inference [Littman et al. 2001]
- Trust management [Freudenthal et al. 2003]
- Equivalence verification of probabilistic circuits [Lee et al. 2018]

Dependency Stochastic Boolean Satisfiability

From DQBF to DSSAT
Dependency SSAT

- A dependency SSAT (DSSAT) formula is commonly written in a **prenex form** as

$$\Phi = \underbrace{\mathcal{R}X, \exists y_1(D_1), \ldots, \exists y_m(D_m)}_{\text{prefix}}.\ \underbrace{\varphi}_{\text{matrix}}$$

  for $D_i \subseteq X$ being the *dependency set* of y_i and φ a quantifier-free formula
- SP of Φ w.r.t. Skolem functions $f_1, \ldots, f_m$ is $\Pr[\mathcal{R}X.\, \varphi|_{f_1(D_1)/y_1, \ldots, f_m(D_m)/y_m}]$
- Optimization version: Find the maximum SP
- Decision version: Determine whether SP $\geq \theta$

[Lee, J., AAAI 2021]

From DQBF to DSSAT
Dependency SSAT

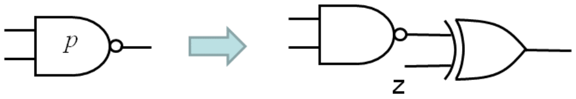
- DSSAT (D) is NEXP-complete
  - By the fact that DSSAT (D) is in NEXP and polynomial-time reducible from DQBF

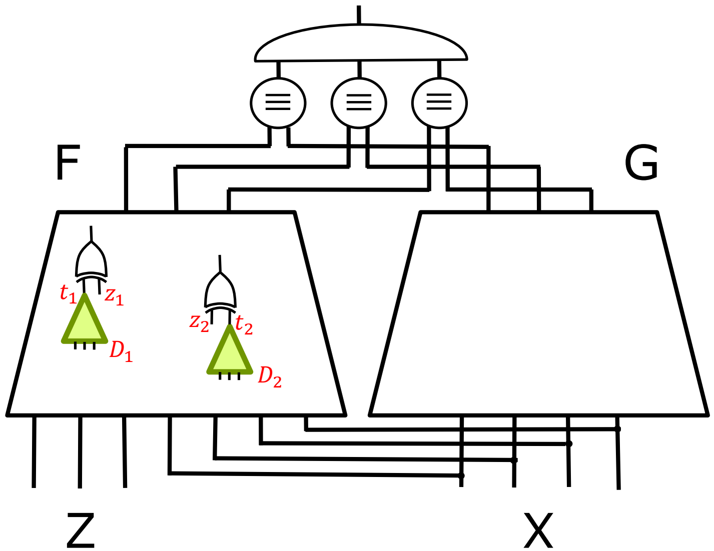
DSSAT Solver

- DSSATpre [CJ23]
  - A preprocessing-based solver converting a DSSAT instance to an SSAT instance

Application: Probabilistic Partial Design

- Probabilistic design is a new paradigm in VLSI design, which allows logic gates to have probabilistic errors
- Black-box synthesis for probabilistic circuit design
  - Black-box outputs $t_1, t_2, \dots$ with their respective inputs $D_1, D_2, \dots$
  - X: primary inputs, Z: error-source pseudo-inputs, Y: intermediate variables





$$\mathcal{R}X, \mathcal{R}Z, \forall Y, \exists T(D). \big(Y = E(X)\big) \to \big(F(X, Z, T) = G(X)\big)$$

Application: Dec-POMDP

- Decentralized Partially Observable Markov Decision Process (Dec-POMDP) generalizes POMDP from single agent to multiple agents
  - $M = (I, S, \{A_i\}, T, \rho, \{O_i\}, \Omega, \Delta_0, h)$
    - Agents $I = \{1, \dots, n\}$
    - States S
    - Actions $\{A_i\}, i \in I$
    - Transition distribution $T: S \times (A_1 \times \cdots \times A_n) \times S \to [0,1]$
    - Reward $\rho: S \times (A_1 \times \cdots \times A_n) \to \mathbb{R}$
    - Observations $\{O_i\}, i \in I$
    - Observation distribution $\Omega: S \times (A_1 \times \cdots \times A_n) \times (O_1 \times \cdots \times O_n) \to [0,1]$
    - Initial state distribution $\Delta_0: S \to [0,1]$
    - Horizon h

Application: Dec-POMDP

- Goal: Find optimal joint policy to maximize the expected total reward $E[\sum_{t=0}^{h-1} \rho(s^t, \vec{a}^t)]$
- Dec-POMDP is NEXP-complete and polynomial-time reducible to DSSAT

Summary and Outlook

- Subjects covered
  - Logic synthesis in a nutshell
  - Boolean satisfiability
  - Quantified Boolean satisfiability
  - Beyond QBF
    - DQBF, SOQBF
    - #SAT, SSAT, DSSAT
- Satisfiability and counting are fundamental in computation
  - Crucial in applications such as EDA, AI, software engineering, etc.
  - New formalisms, solvers, and applications await further exploration

Thanks for Your Attention!