Boolean Satisfiability and Its Applications

FLOLAC 2015

Chung-Yang (Ric) Huang/NTU

2015.07.08/09

What can/should be covered in this topic?

- Fundamentals of Boolean Satisfiability (SAT)
- Techniques to improve SAT solving
- Circuit-based SAT algorithms
- SAT-based (hardware) verification
  - Bounded model checking (BMC)
  - Inductive proof
  - SAT-based abstraction and refinement
  - Interpolation-based method
  - Property-directed reachability

What can/should be covered in this topic?

- SAT-based logic synthesis
  - Redundancy addition and removal
  - Functional dependency
  - SAT-based re-synthesis techniques
  - Engineering Change Order (ECO)
- From SAT to optimization problems
  - Pseudo Boolean satisfiability/optimization problems
- General SAT-based model checking algorithms
- Quantified Boolean Formula (QBF)
- Bit-vector/Arithmetic solver
- Satisfiability Modulo Theories (SMT)

What can/should be covered in this topic?

- Fundamentals of Boolean Satisfiability (SAT)
- Techniques to improve SAT solving
- ~~Circuit-based SAT algorithms~~
- SAT-based (hardware) verification
  - Bounded model checking (BMC)
  - Inductive proof
  - ~~SAT-based abstraction and refinement~~
  - ~~Interpolation-based method~~
  - ~~Property directed reachability~~

What can/should be covered in this topic?

- SAT-based logic synthesis
  - Redundancy addition and removal
  - ~~Functional dependency~~
  - ~~SAT based re-synthesis techniques~~
  - ~~Engineering Change Order (ECO)~~
- From SAT to optimization problems
  - Pseudo Boolean satisfiability/optimization problems
- ~~General SAT-based model checking algorithms~~
- ~~Quantified Boolean Formula (QBF)~~
- ~~Bit-vector/Arithmetic solver~~
- ~~Satisfiability Modulo Theories (SMT)~~

Introduction to Boolean Satisfiability (SAT)

A fundamental problem in computer science

- Given a Boolean network F: B^n B,
  where B = { 0, 1 }, and
  n is the number of inputs I = { x_1, x_2,... , x_n }

- Boolean Satisfiability
  ➔ Finding an input assignment

  A: { $x_1 = a_1$, $x_2 = a_2$,... , $x_n = a_n$ | $a_i \in B$ }

  such that F = 1.

- Exponential complexity...?

Complexity of SAT solver

- Boolean Satisfiability (SAT) was the first proven NP-complete problem by Dr. S. Cook in 1971
  - Given n variables, the number of decisions can be as many as 2^n...
  - If there is a non-deterministic machine, we can construct a polynomial-time algorithm that can guarantee to prove/disprove the SAT problem

  [Pitfall?] Unless there is a non-deterministic machine, we cannot construct a polynomial-time SAT algorithm

➔ How can SAT be useable for million-gate designs?

Boolean Satisfiability Solvers

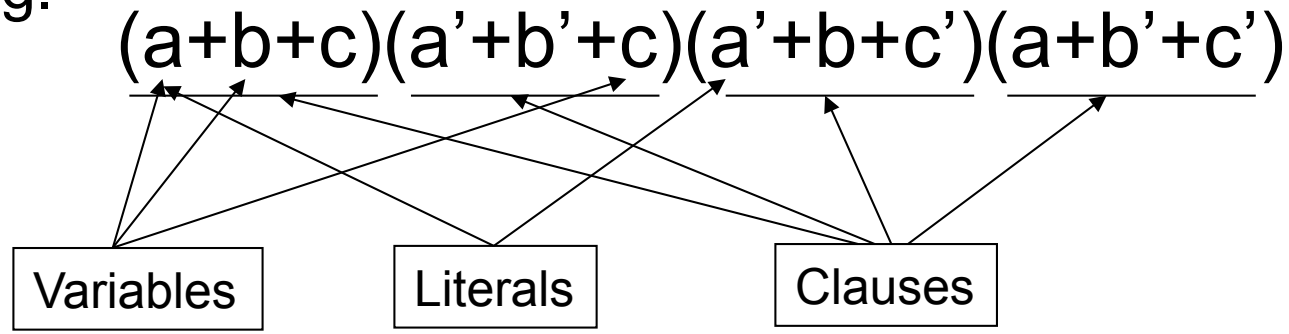
- Boolean SAT solvers have been very successful recent years in the verification area
  - More research / popular than BDDs
  - Applications
    - Equivalence checking, property checking, synthesis, etc
  - Applicable even on million-gate designs
  - For both combinational and sequential problems

  ➔ However, SAT is intrinsically a "combinational" (propositional) solver

- There are many advanced Boolean SAT algorithms
  - We will cover them gradually in the following lecture notes
- Many many SAT solvers
  - glucose, precosat, miniSat, zChaff, BerkMin, Csat, Grasp, SATO,... etc.
  - http://www.satcompetition.org/

Types of Boolean Satisfiability Solvers

1. Conjunctive Normal Form (CNF) Based
   - Boolean function is represented as a CNF (i.e. Product of Sum, POS format)
   - e.g.



   - To be satisfied, all the clauses should be '1'

2. Circuit-Based
   - Boolean function is represented as a circuit netlist
   - SAT algorithm is directly operated on the netlist

CNF vs. Circuit SAT

- Although CNF and circuit SAT solvers look quite different, their algorithms can be very similar

- CNF SAT
  - Simpler data structure; easier to implement
- Circuit SAT
  - Structural information; extensible to word-level

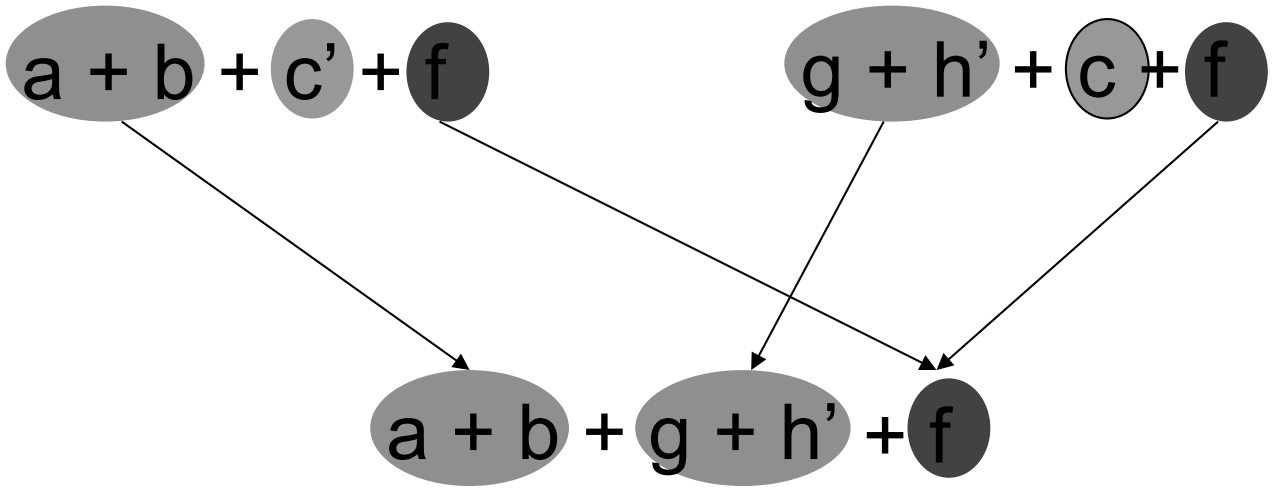
➔ In the following slides, we will focus on the easier-to-implement solver, CNF SAT, only.

CNF-Based SAT Algorithm

1. Davis, Putnam, 1960
   - Explicit resolution based
   - May explode in memory
2. Davis, Logemann, Loveland, (DLL) 1962
   - Search based.
   - Most successful, basis for almost all modern SAT solvers
   - Learning and non-chronological backtracking, 1996
3. St Imarcks algorithm, 1980s
   - Proprietary algorithm. Patented.
   - Commercial versions available
4. Stochastic Methods, 1992
   - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
   - Local search and hill climbing

Resolution

- Resolution of a pair of clauses with exactly ONE incompatible variable
  - Two clauses are said to have distance 1
  - $C_1 \wedge C_2 \rightarrow C_3$ or $C_3 \rightarrow C_1 \wedge C_2$?
  - Existential quantification?

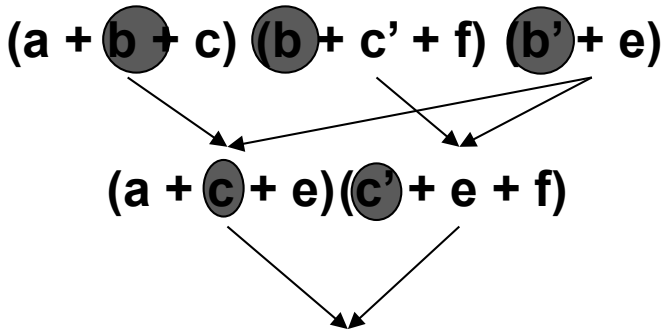


Souce: Prof. Sharad Malik, "The Quest for Efficient Boolean Satisfiability Solvers"

Davis Putnam Algorithm

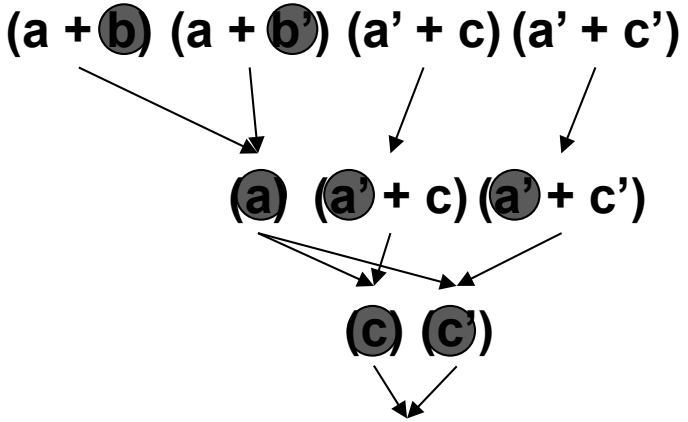
M .Davis, H. Putnam, “A computing procedure for quantification theory", *ACM*, Vol. 7, pp. 201-214, 1960 (citeseer)

- Existential abstraction using resolution
- Iteratively select a variable for resolution till no more variables are left.



SAT

Sol: {a=1, e=1, f=1}



UNSAT

Potential memory explosion problem!

Souce: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Boolean Satisfiability (SAT) Algorithm

1. Davis, Putnam, 1960
   - Explicit resolution based
   - May explode in memory
2. Davis, (Putnam), Logemann, Loveland, (D(P)LL) 1962
   - Search based.
   - Most successful, basis for almost all modern SAT solvers
   - Learning and non-chronological backtracking, 1996
3. St Imarcks algorithm, 1980s
   - Proprietary algorithm. Patented.
   - Commercial versions available
4. Stochastic Methods, 1992
   - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
   - Local search and hill climbing

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

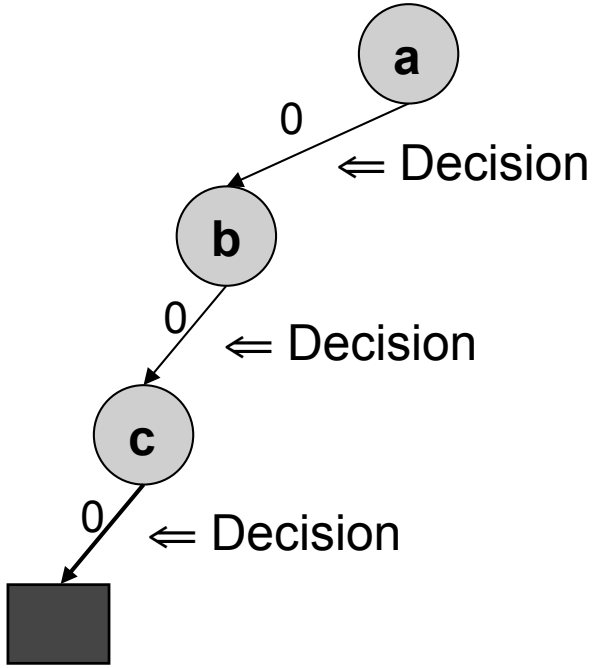
(a + c’ + d)

(a + c’ + d’)

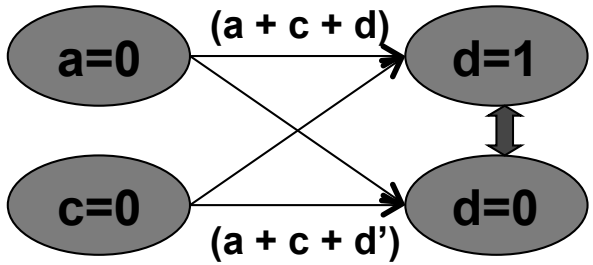
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

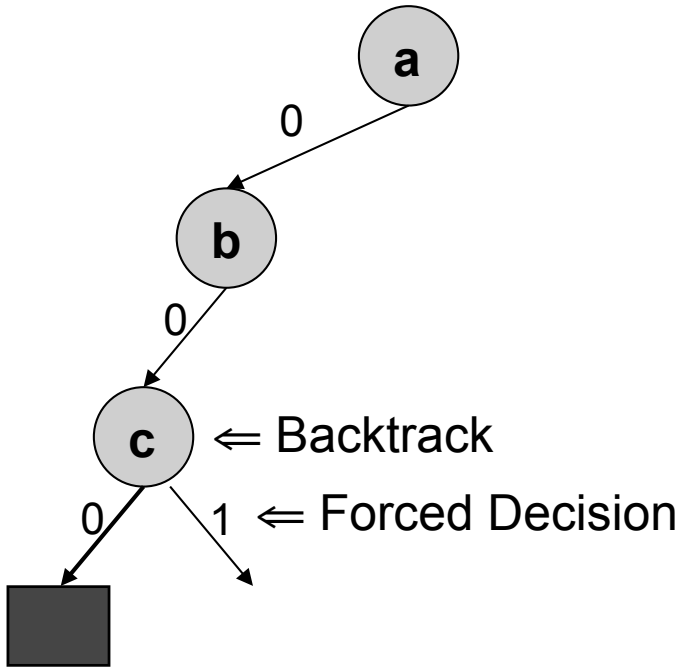
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

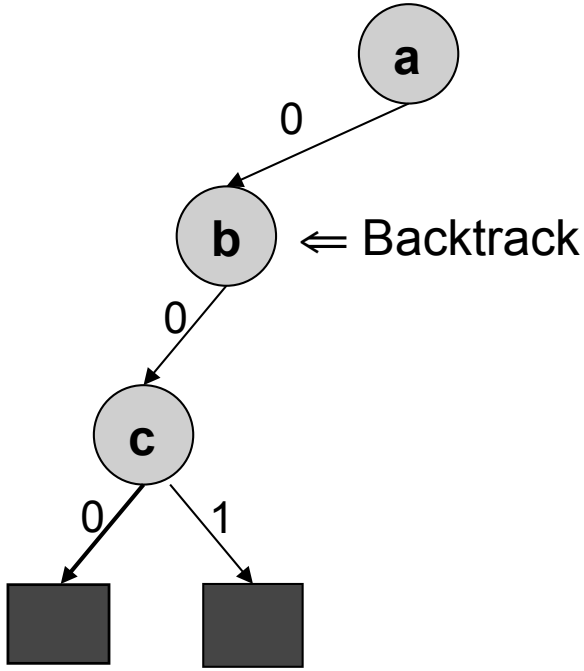
(a + c’ + d)

(a + c’ + d’)

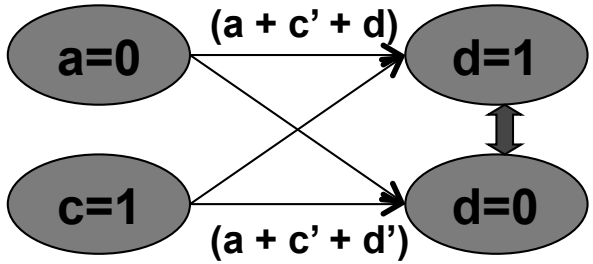
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

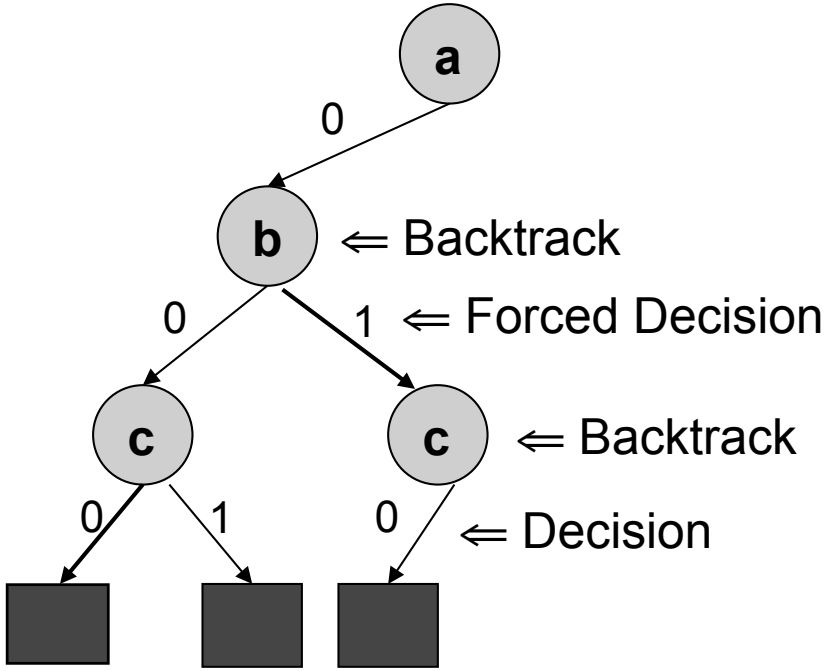
(a + c’ + d)

(a + c’ + d’)

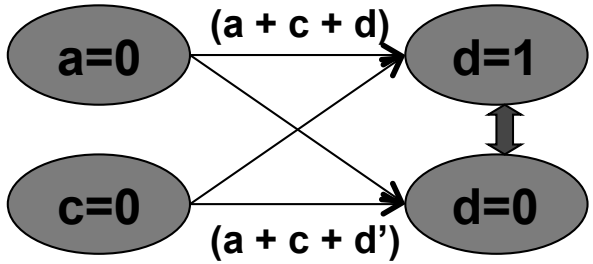
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

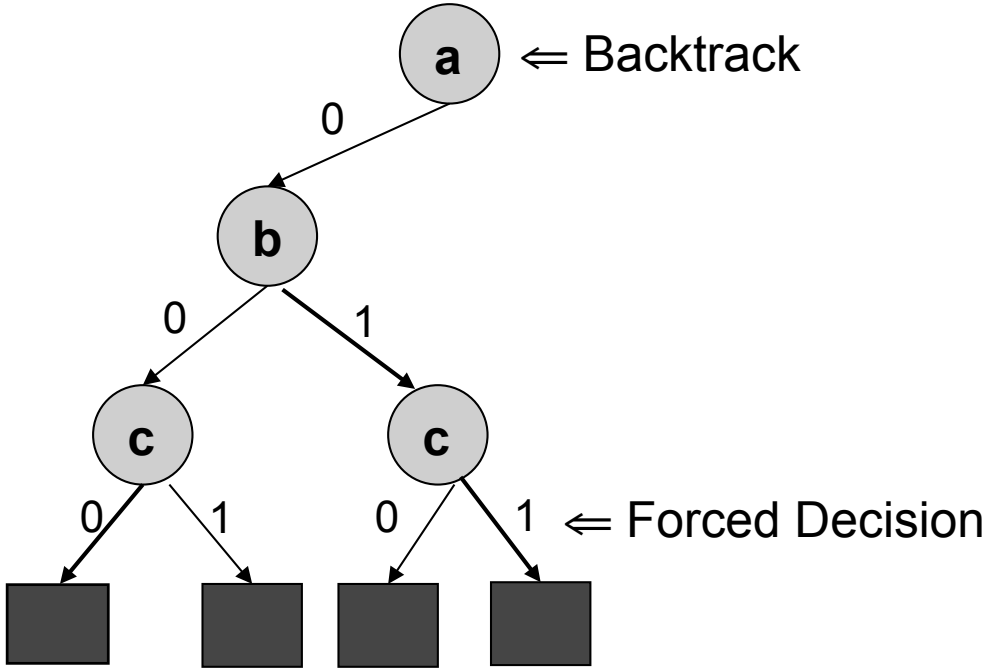
(a + c’ + d)

(a + c’ + d’)

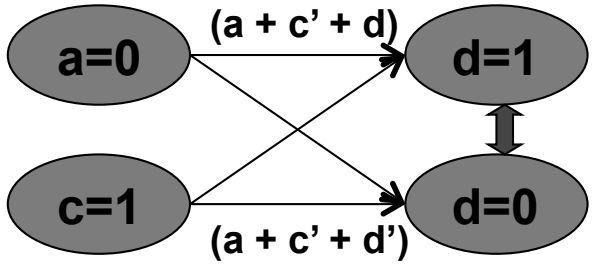
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a' + b + c)

(a + c + d)

(a + c + d')

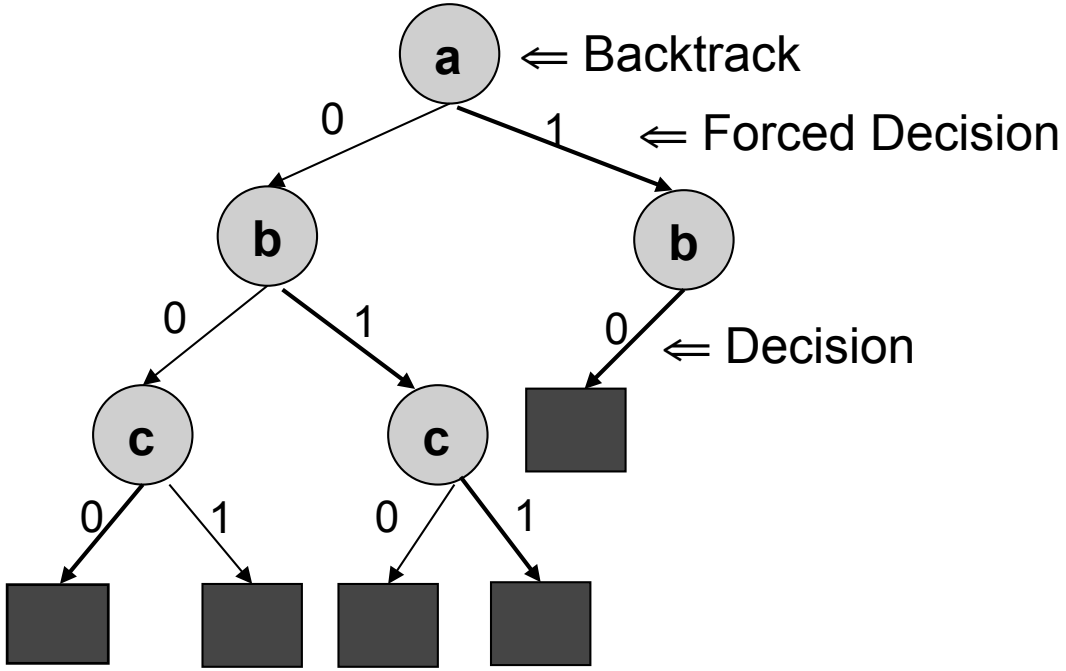
(a + c' + d)

(a + c' + d')

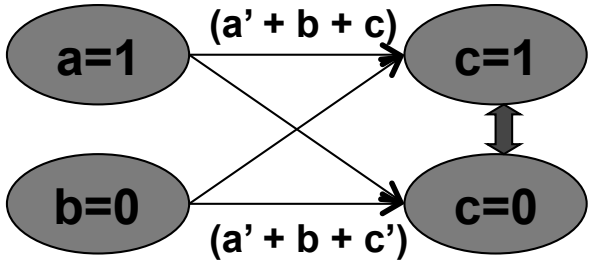
(b' + c' + d)

(a' + b + c')

(a' + b' + c)



Implication Graph



Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

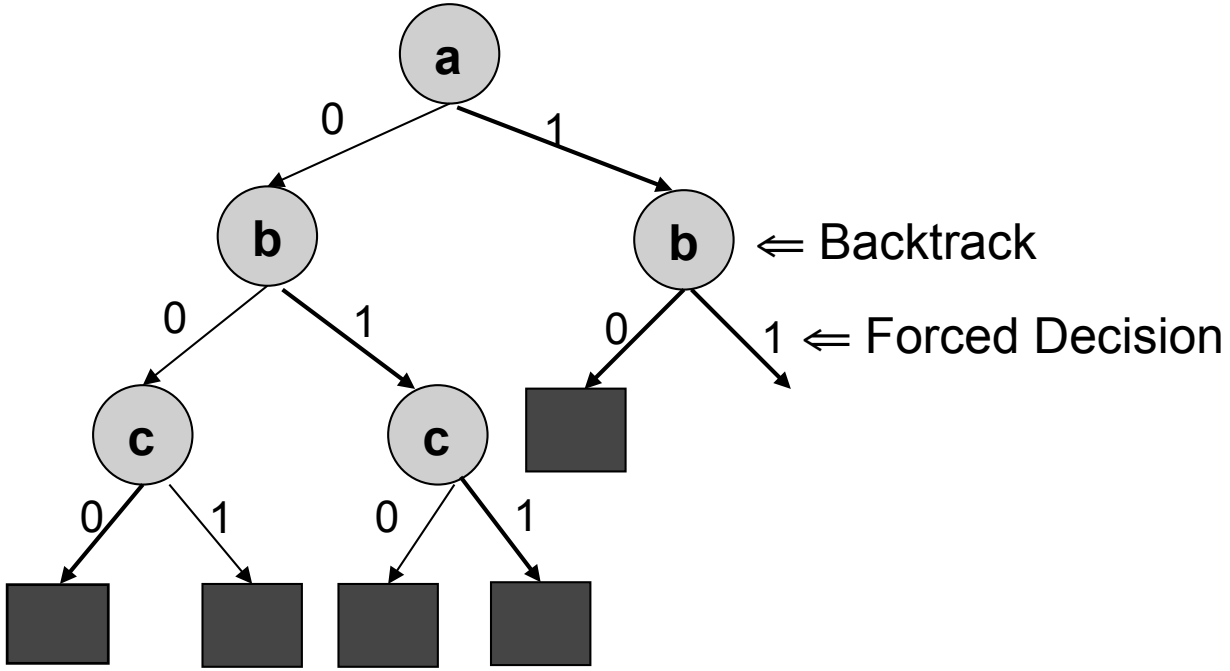
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

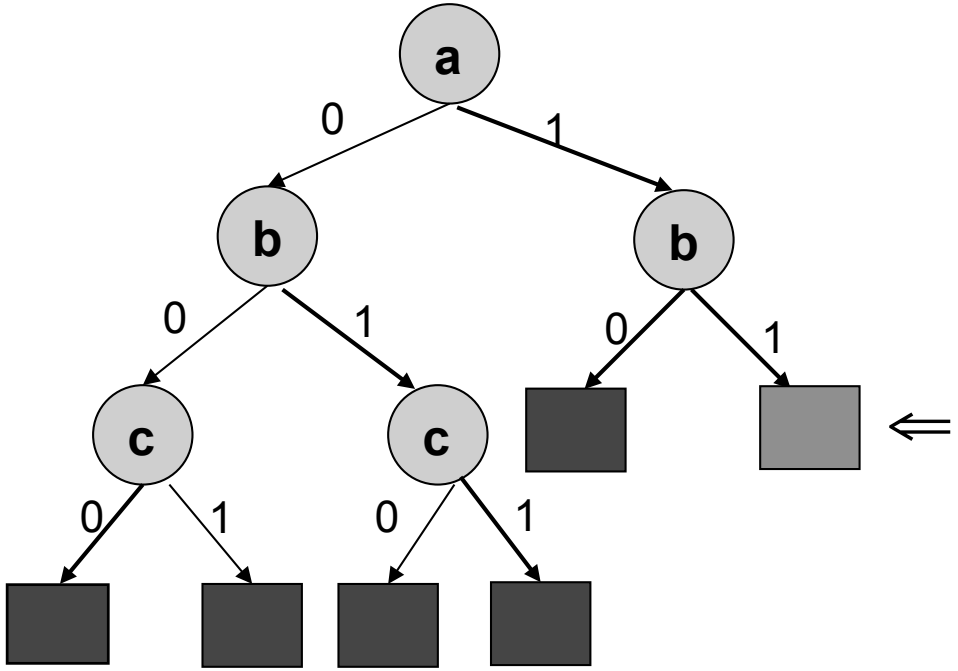
(a + c’ + d)

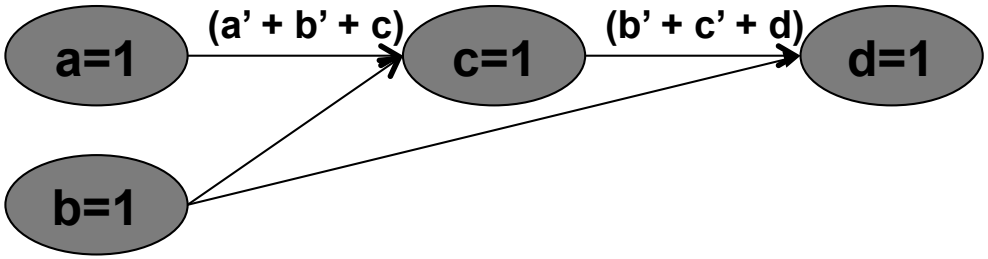
(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)





Potentially exponential complexity!!

Did you see any unnecessary work?

SAT Improvements

1. Conflict-driven learning
   - Once we encounter a conflict

     ➔ Figure out the cause(s) of this conflict and prevent to see this conflict again!!

Conflict-Driven Learning

(a' + b + c)

(a + c + d)

(a + c + d')

(a + c' + d)

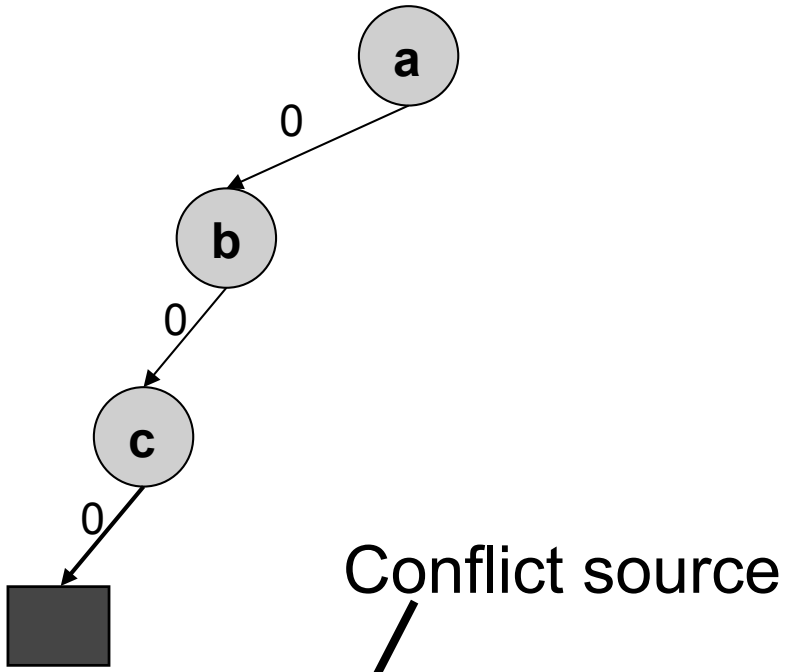
(a + c' + d')

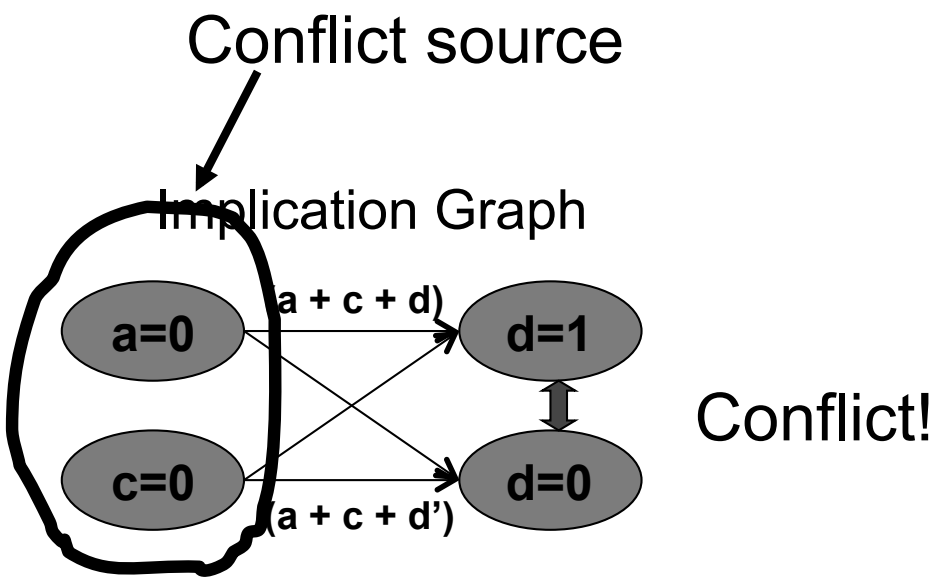
(b' + c' + d)

(a' + b + c')

(a' + b' + c)

(a + c) Learned clause





SAT Improvements

2. Non-chronological backtracking
   - Since we get a learned clause from the conflict analysis…

     ➔ Instead of backtracking 1 decision at a time, backtrack to the "next-to-the-last" variable in the learned clause

Non-Chronological Backtracking

(a' + b + c)

(a + c + d)

(a + c + d')

(a + c' + d)

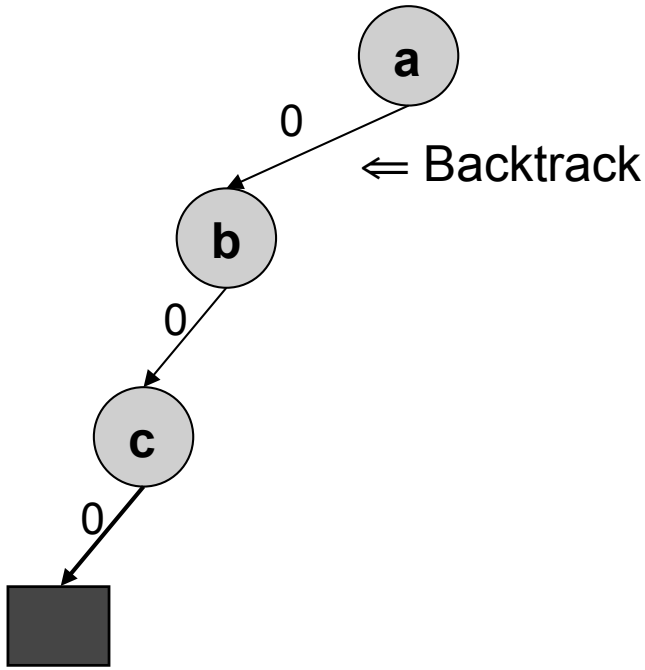
(a + c' + d')

(b' + c' + d)

(a' + b + c')

(a' + b' + c)

(a + c) Learned clause



- 'a' is the next-to-the-last variable in the learned clause
- Backtrack c = 0 && b = 0

Modified from: Prof. Sharad Malik, "The Quest for Efficient Boolean Satisfiability Solvers"

Deduced Implication from Learned Clause

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

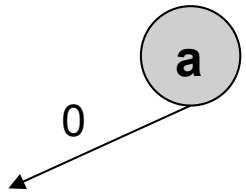
(a + c’ + d’)

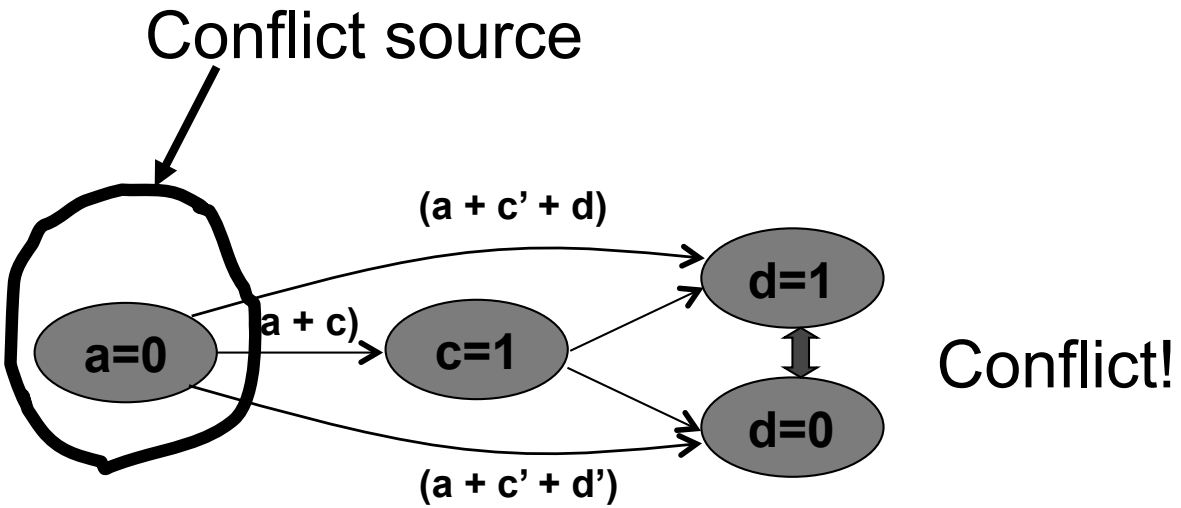
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c)





Deduced Implication from Learned Clause

(a' + b + c)

(a + c + d)

(a + c + d')

(a + c' + d)

(a + c' + d')

(b' + c' + d)

(a' + b + c')

(a' + b' + c)

(a + c)

(a) Learned clause

- Since there is only one variable in the learned clause

  → No one last variable

- Backtrack all decisions

Deduced Implication from Learned Clause

(a’ + b + c)

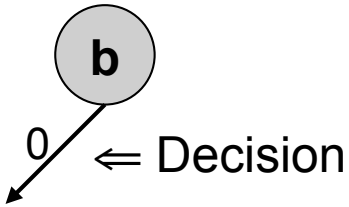
(a + c + d)

(a + c + d’)

(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)



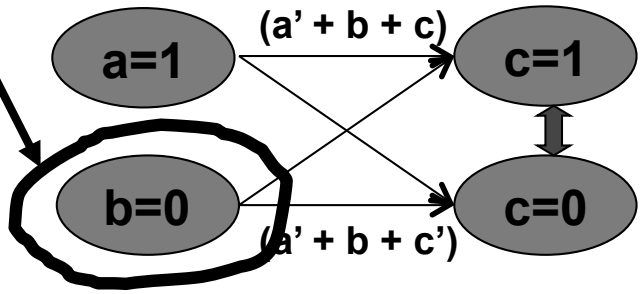
(a’ + b + c’)

(a’ + b’ + c)

(a + c)

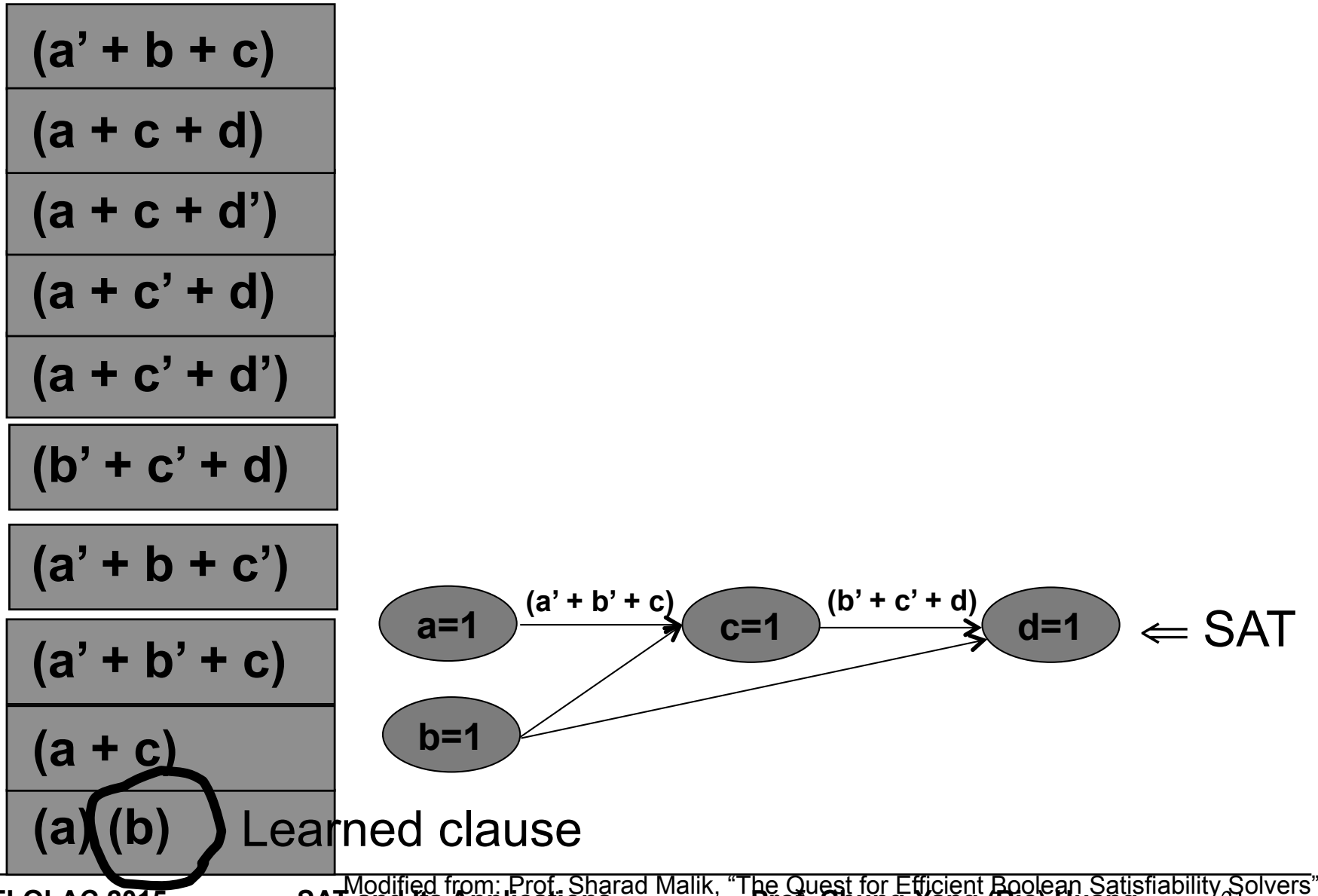
(a)

Conflict source

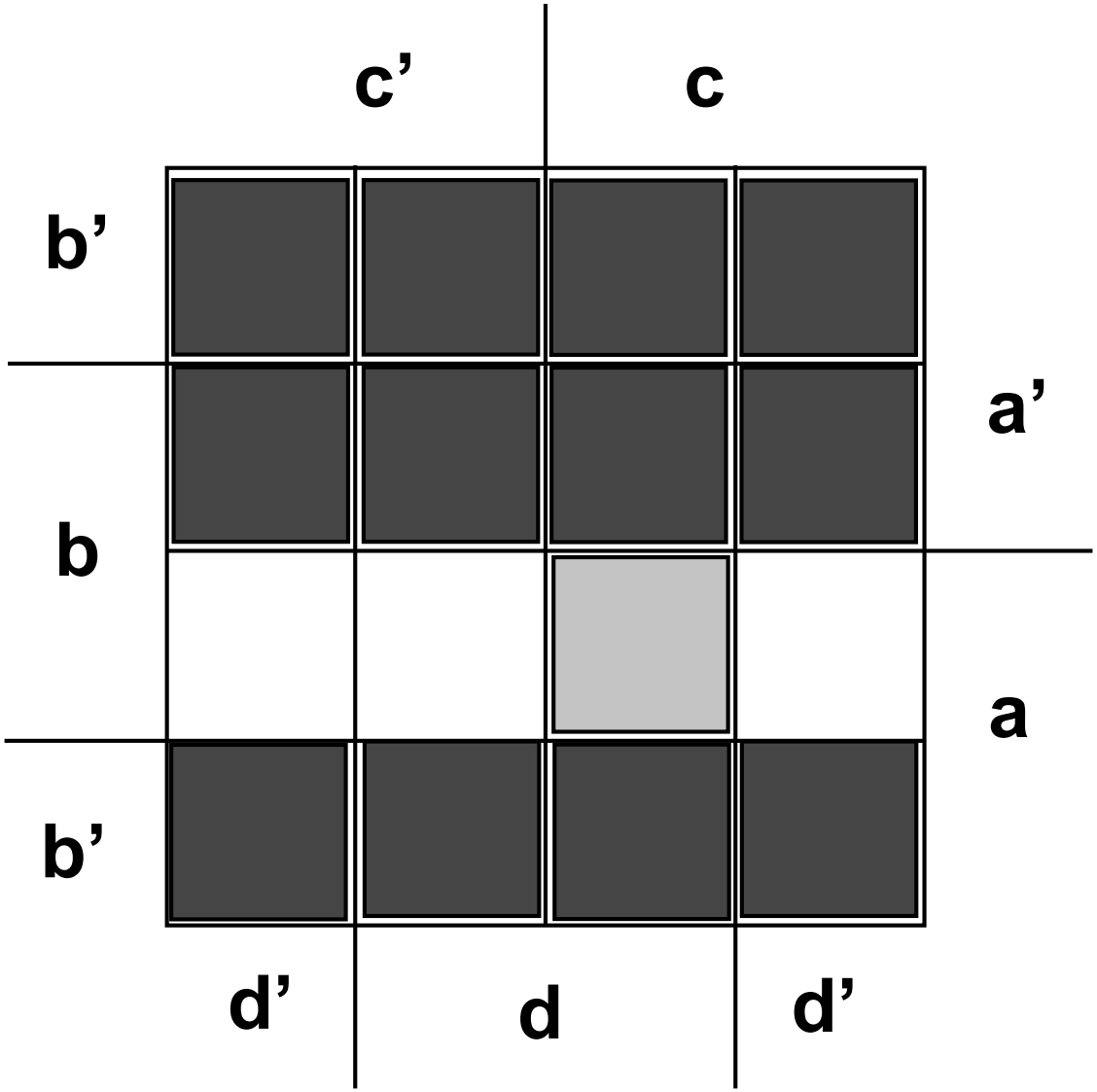


Conflict!

Deduced Implication from Learned Clause



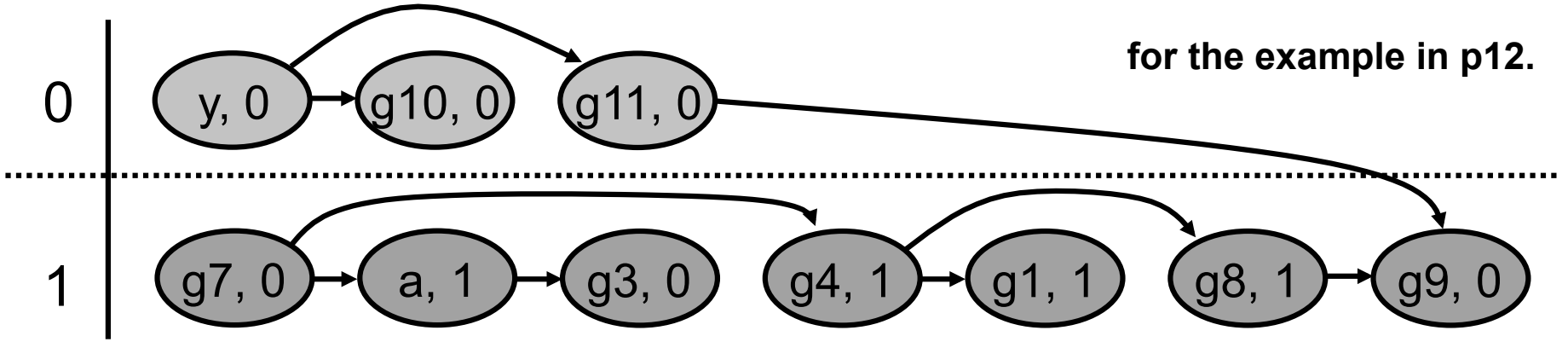
What does conflict learning tell us?



Decision: a = 0
Decision: b = 0
Decision: c = 0
conflict!!
Learned: (a + c)
Backtrack: c = 0, b = 0
Implied: c = 1
Decision: b = 0
conflict!!
Learned: (a)
Implied: a = 1
Decision: b = 0
conflict!!
Learned: (b)
Implied: b = 1
Implied: c = 1, d = 1
SAT!!

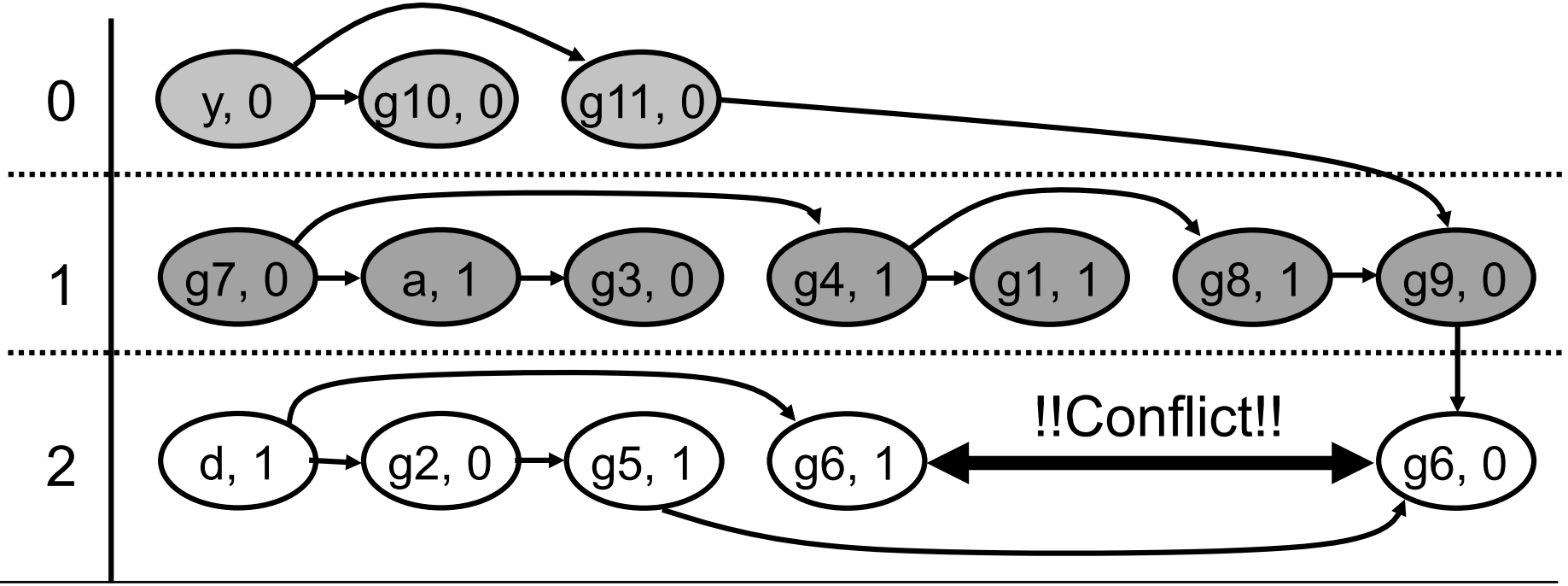
A Closer Look at the Implication Graph (a conceptual implementation)

- Implications are grouped into different decision levels
  - Level 0: target imp; constants
  - Level 1+: decisions
- Node (gate, value): implications
- Incoming edge(s) of a node: implication sources (reasons)
  - The nodes with no incoming edges are called “root implication nodes”
  - There should only be ONE root implication node for each decieion level >= 1 (which is the decision in that level)

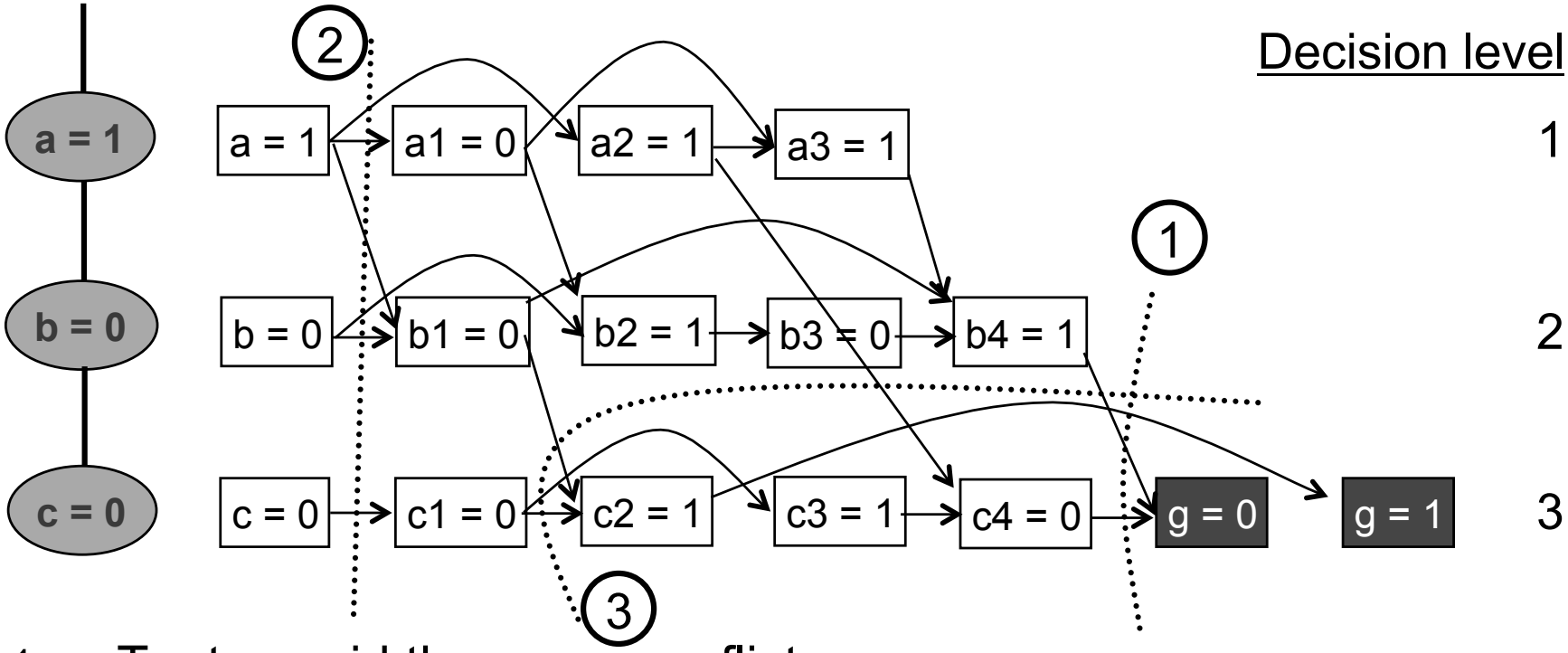


Conflict Analysis

- When we encounter a decision conflict, we want to figure out the causes so that ---
  1. Try to avoid the same conflict
  2. Backtrack as many decisions as possible

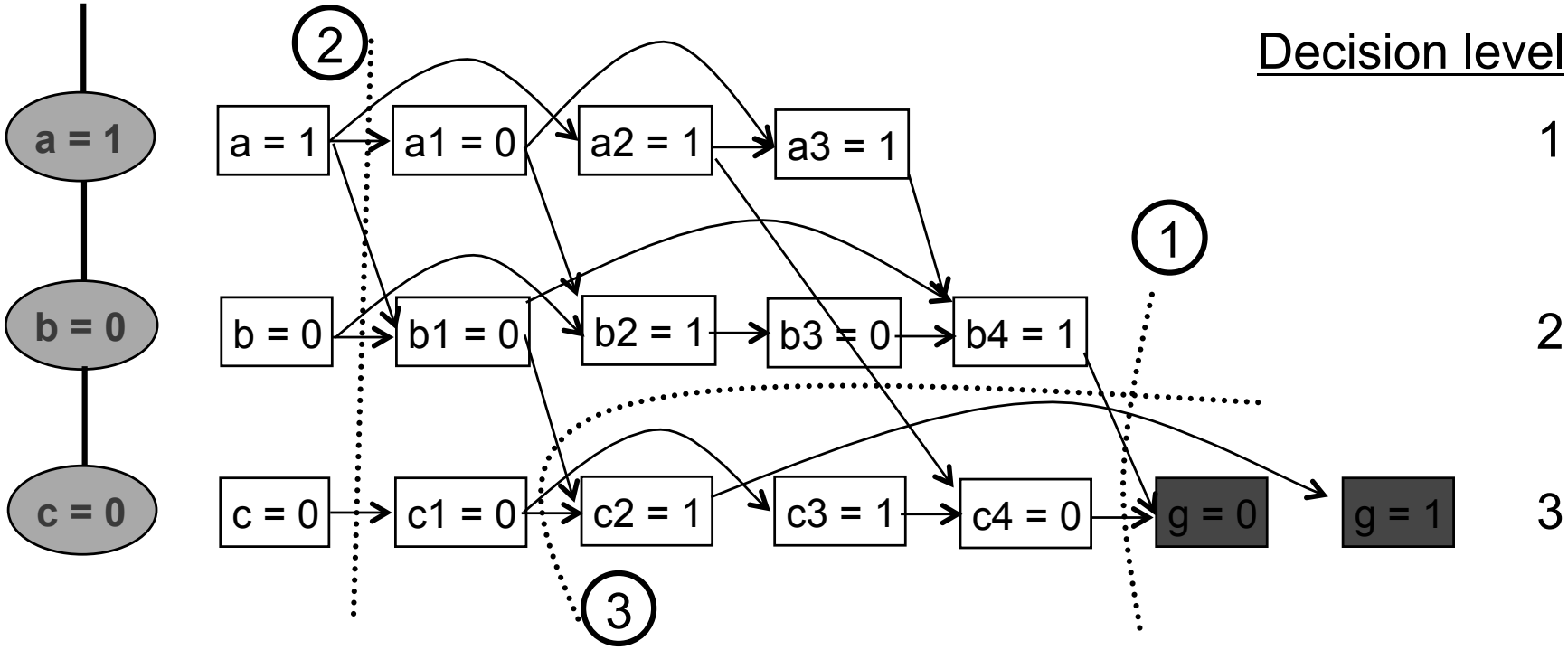


Conflict Analysis



1. Try to avoid the same conflict
   - Starting from the conflict implications (g = 0) & (g = 1), backward trace their implication sources
   - (An informal explanation) Any cut in the implication graph defines a set of conflict causes
   - Add a constraint for the conflict causes to prevent the conflict from happening again

Conflict-Driven Learning



- Add a constraint to prevent the same conflict
  1. b4 && c2 && c4' = 0; ➔ (b4' + c2' + c4)
  2. a && b' && c' = 0; ➔ (a' + b + c)
  3. b4 && a2 && b1' && c1' = 0; ➔ (b4' + a2' + b1 + c1)

Which constraint is the best to add?

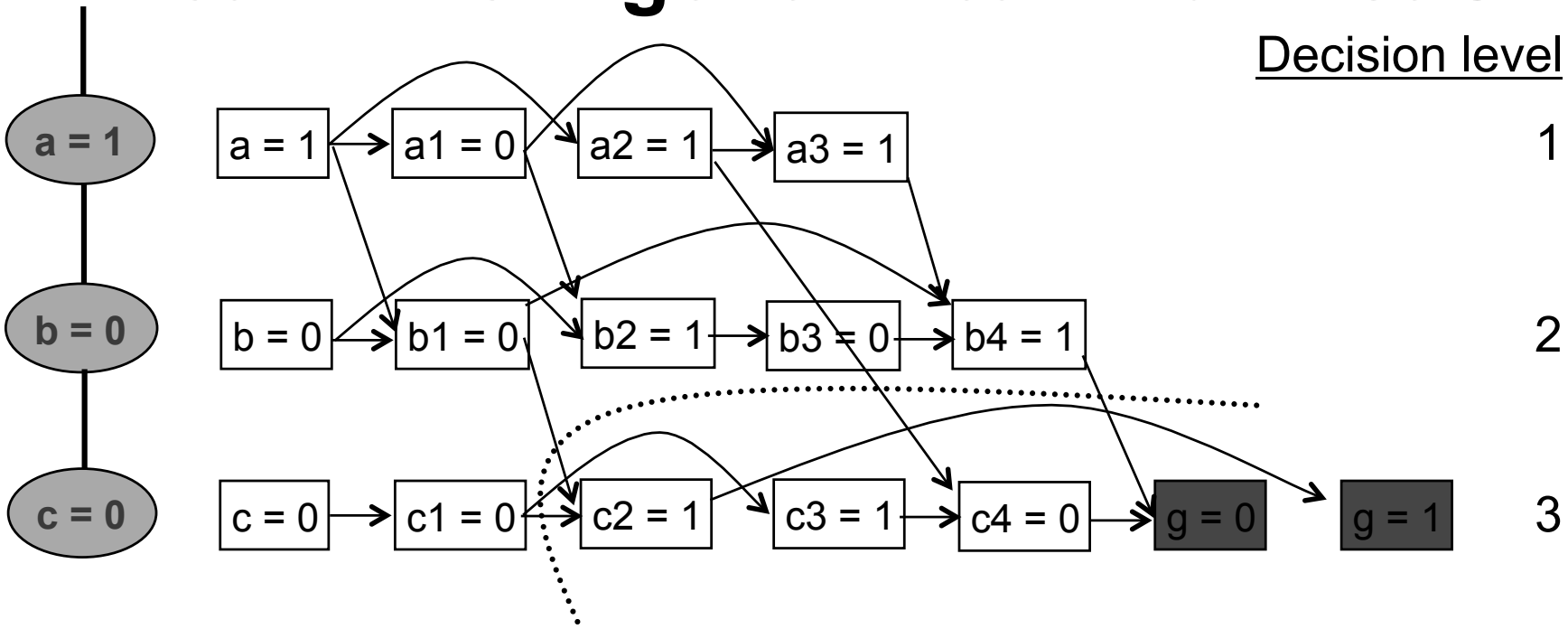
- [Zhang, , ICCAD 2001] Experiment shows that “first-UIP” (1st-UIP) is the best
  - UIP: Unique Implication Point
    - In a cut that there is only one node in the last (where conflict happens) decision level (why UIP cut?)
    - Starting from the conflict gate, the first encountered UIP is namely first UIP
    - The cut with only decision nodes is called the last-UIP
    - In the previous example, (2) is the last UIP, and (3) is the first UIP

Complexity to find the first UIP?

```
conflictAnalysis(imp0Src, imp1Src) {
  int nMarked = 0;
  for_each_imp(imp, imp0Src)
    checkImp(imp, nMarked,conflictSrc);
  for_each_imp(imp, imp1Src)
    checkImp(imp, nMarked,
     conflictSrc);
  for_each_imp_rev(imp, lastDLevel) {
    if (!imp.isMarked()) continue;
    if (--numMarked == 0) {// UIP found!!
      conflictSrc.push_back(imp);
      break;  // ready to return
    }
    imp.unsetMark();
    for_each_imp_src(imp_src, imp) {
      checkImp(imp_src, nMarked,
                    conflictSrc);
    }
  }
  for_each_imp(imp, conflictSrc)
    imp.unsetMark();
  return conflictSrc;
}

checkImp(imp, nMarked, conflictSrc) {
  if (imp.isMarked()) return;
  imp.setMark();
  if (!imp.isLastDecisionLevel())
    conflictSrc.push_back(imp);
  else ++numMarked;
}
```

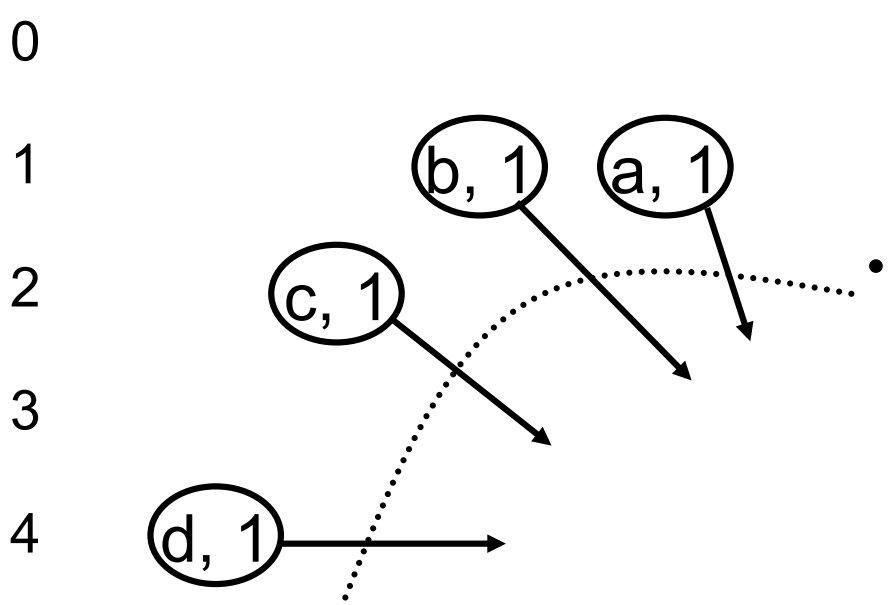
Linear-Time Algorithm to Find First UIP



- Start from (g = 0), (g = 1) // #Marks = 2
- Unmark (g = 1), mark (c2 = 1) // #Marks = 2
- Unmark (g = 0), mark (c4 = 0), add (b4 = 1) // #Marks = 2
- Unmark (c4 = 0), mark (c3 = 1), add (a2 = 1) // #Marks = 2
- Unmark (c3 = 1), mark (c1 = 0) // #Marks = 2
- Unmark (c2 = 1), add (b1 = 0) // #Marks = 1
- Find first UIP: (c1=0), conflict sources: { (c1=0), (b1=0), (a1=0), (b4=1) }

UIP for Non-chronological Backtracking

- Since in UIP cut there is only one node with the last decision level…
- And we add a constraint for the UIP cut



Constraint

(a && b && c && d) = 0

⇩

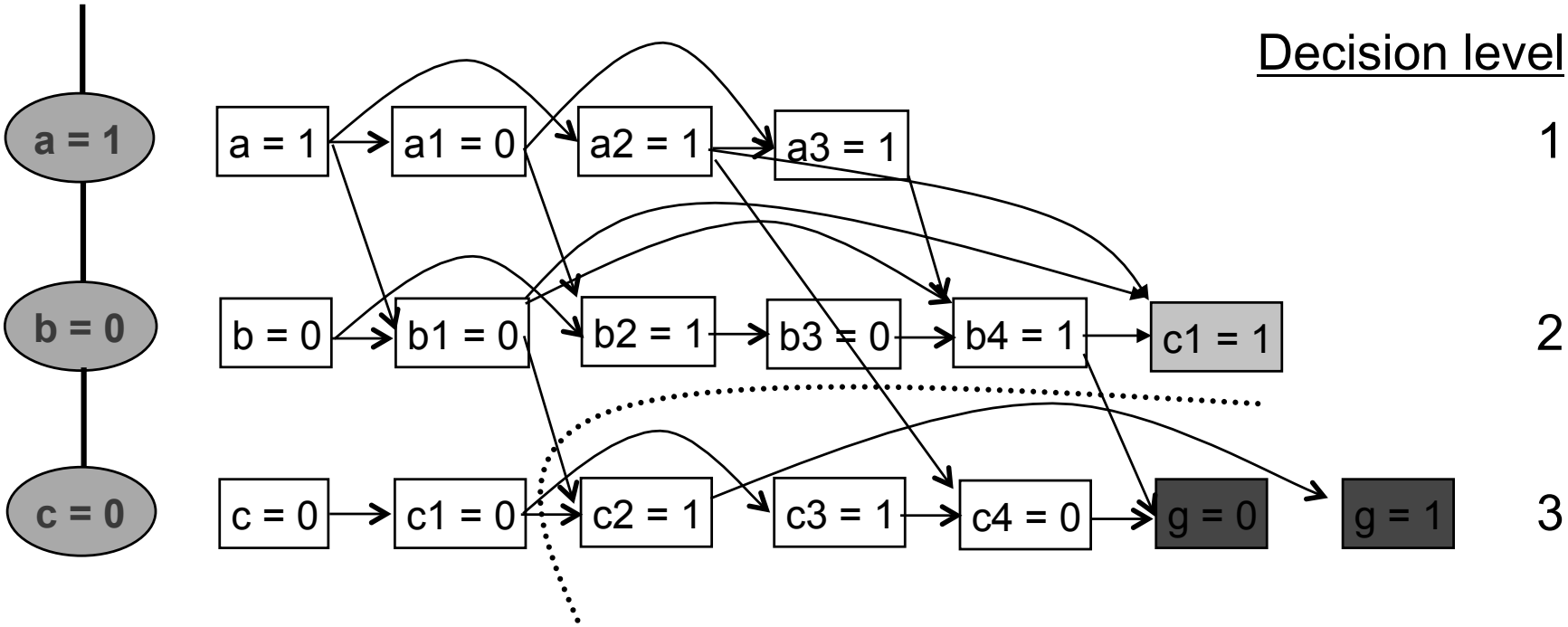
(a && b && c) ➔ d'

⇩

If we backtrack to the max decision level of { a, b, c }

1. { a, b, c } still have the original implications
2. d can be implied with the opposite value at the max level above

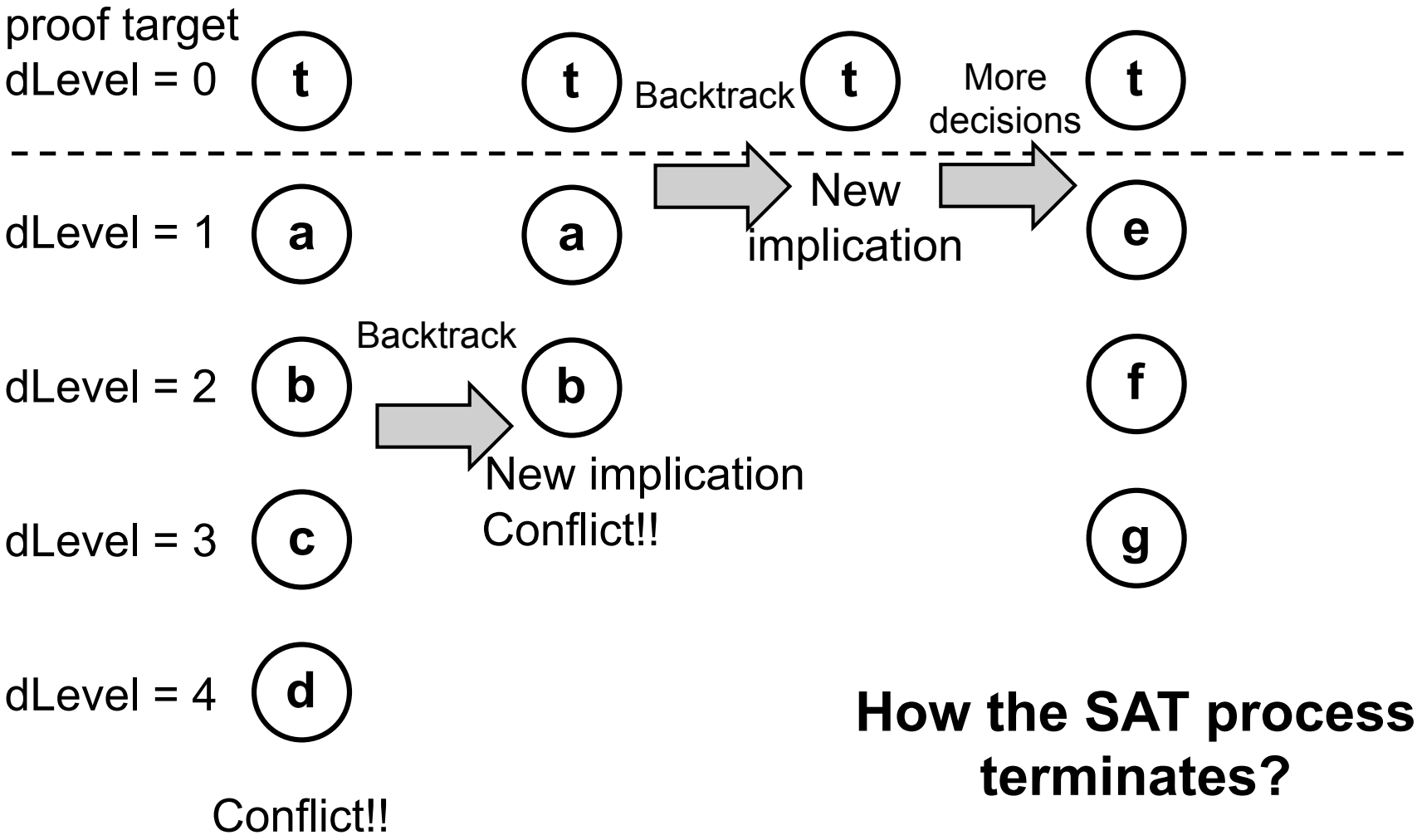
Conflict-Driven Learning



Learned implication

b1' && a2 && b4 → c1

Conflict-Driven Non-Chronological Backtracking --- Algorithm



How the SAT process terminates?

Conflict-Driven Non-Chronological Backtracking --- Algorithm

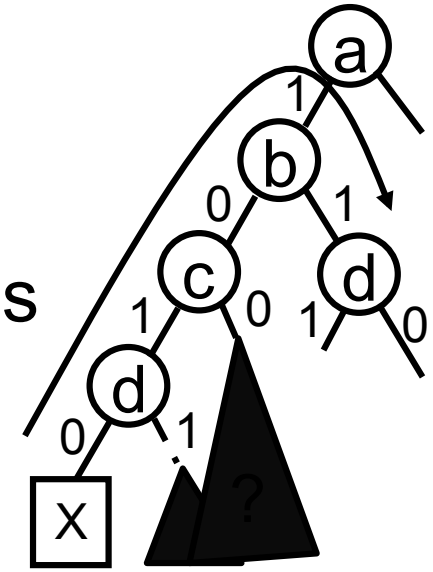
1. When conflict occurs, check if the conflict level == 0 (implication level for the SAT target)
   a) If yes, return *unsatisfiability* (Why?)
   b) Else, continue to 2
2. Find the 1st-UIP cut as the conflict causes
3. Backtrack to the max decision level of the nodes other than UIP in the cut
4. The UIP gate will be implied with the opposite value
5. Perform the new implication
6. If conflict, go to 1, else continue for the next decision

A closer look at binary decision tree

In general, is non-chronological backtracking safe?

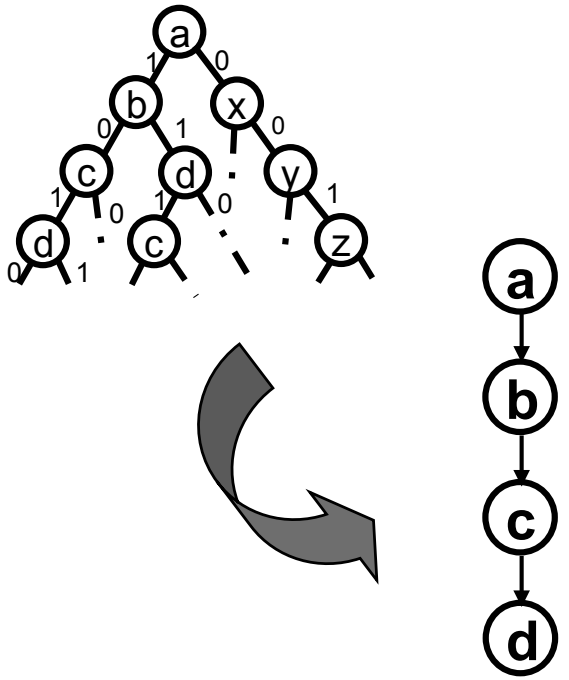
- May lead to SAT solution ealier
- But some portion of the decision tree may not be covered
  - Not a complete search anymore
  - May also miss some bugs

➔ Difficult to record which branches haven't been searched



Conflict-Driven Non-Chronological Backtracking --- Completeness

- But with conflict-driven learning, SAT search is still guaranteed to be complete
- SAT search is not a binary decision tree anymore…
  - Becomes a decision stack
  - Conflict
    - → Learned clause (gate)
    - → Indicate where to backtrack
    - → Learned implication



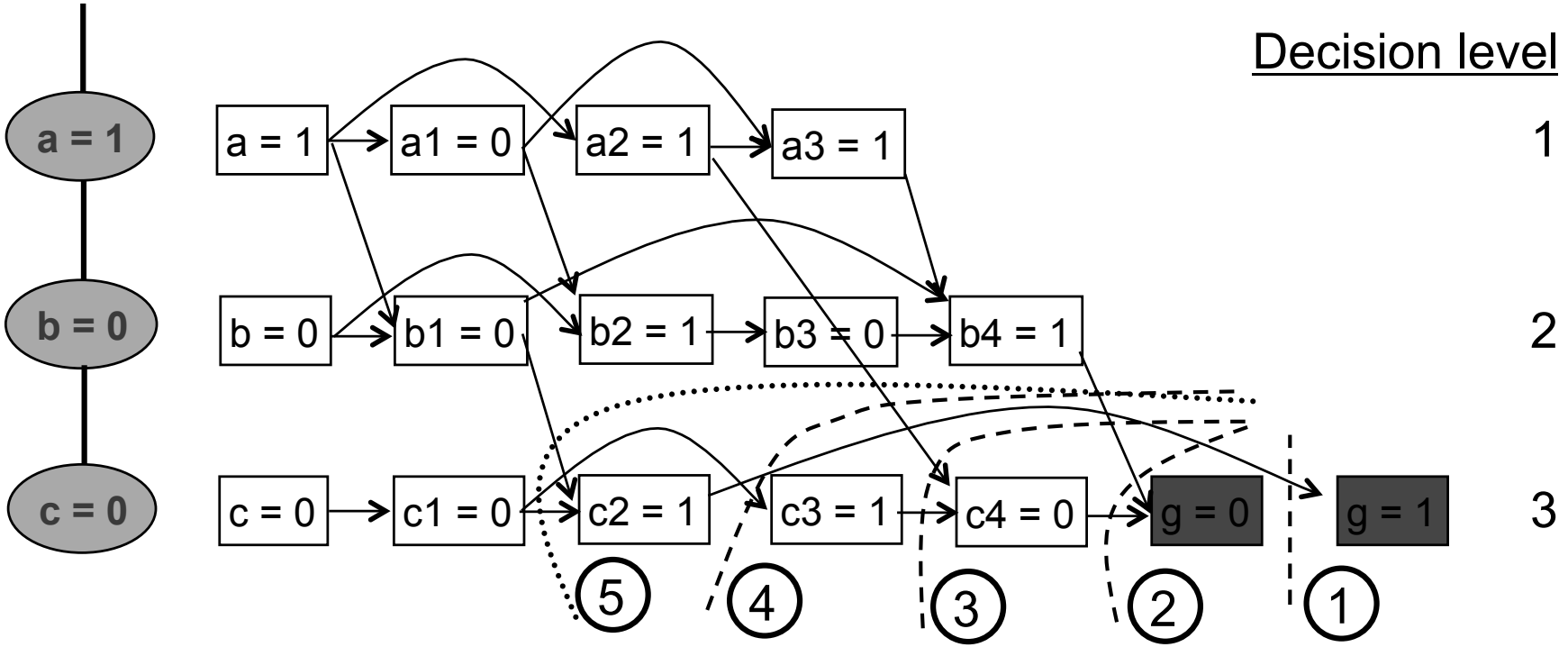
Conflict-Driven Non-Chronological Backtracking --- Completeness

- Branch-and-bound algorithm for Constraint Satisfaction Problem (CSP) becomes a "constraint refinement process"

➔ Search region is gradually narrowed down

➔ At the end, either becomes empty, or finds the solution !!

Implication graph, resolution, and learning



(1): (c2' + g)

(2): (b4' + c4 + g') → (b4' + c2' + c4)

(3): (a2' + c3' + c4') → (a2' + b4' + c2' + c3')

(4): (c1 + c3) → (a2' + b4' + c1 + c2')

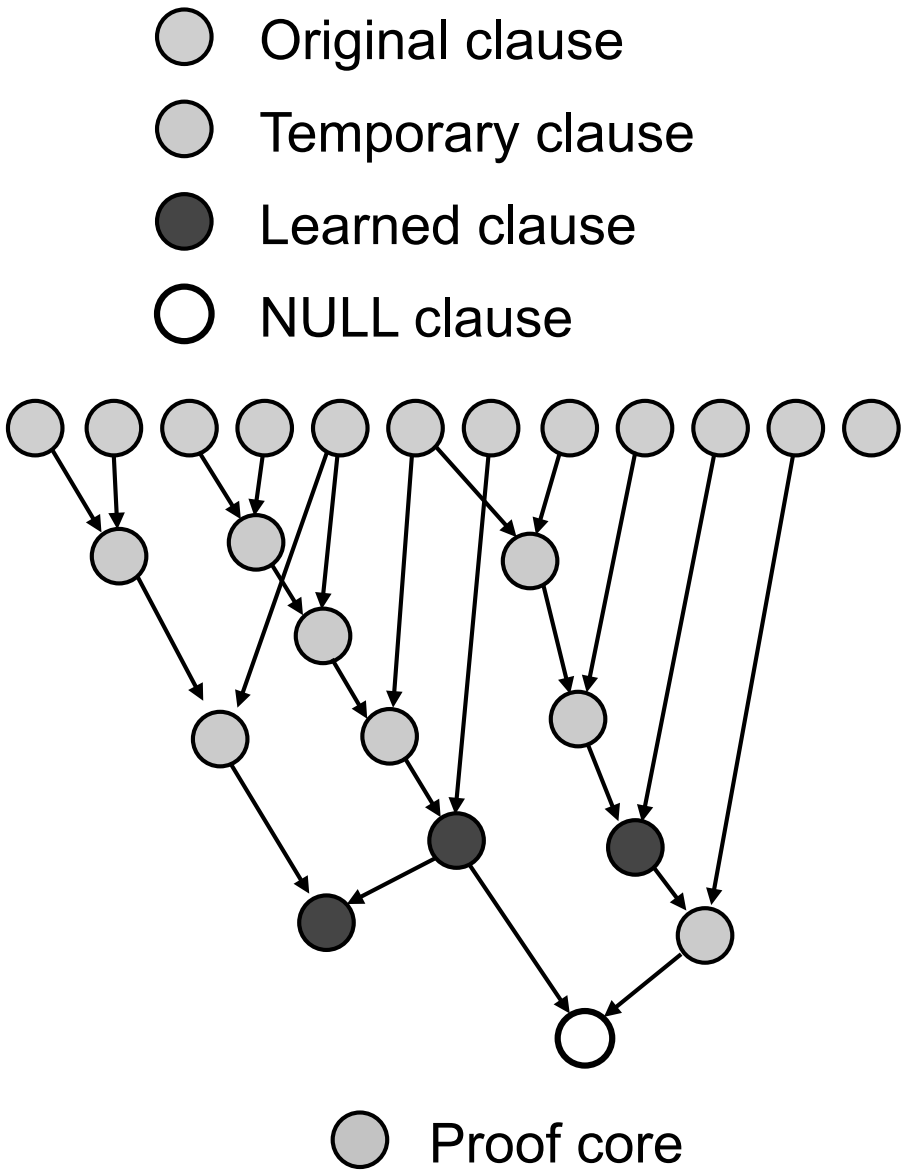
(5): (b1 + c1 + c2) → (a2' + b1 + b4' + c1)

The validity of learned information and incremental SAT

- Note that, learned clause is a resolution of clauses that are involved in the implication process.
  - As long as these clauses are still in the proof database, the learned information is always valid.
- Incremental SAT
  - (For example) Proving two properties in a circuit --- the learned information obtained in proving one property can be reused in proving another.
  - (Challenge) What if some of the clauses or variables are deleted?

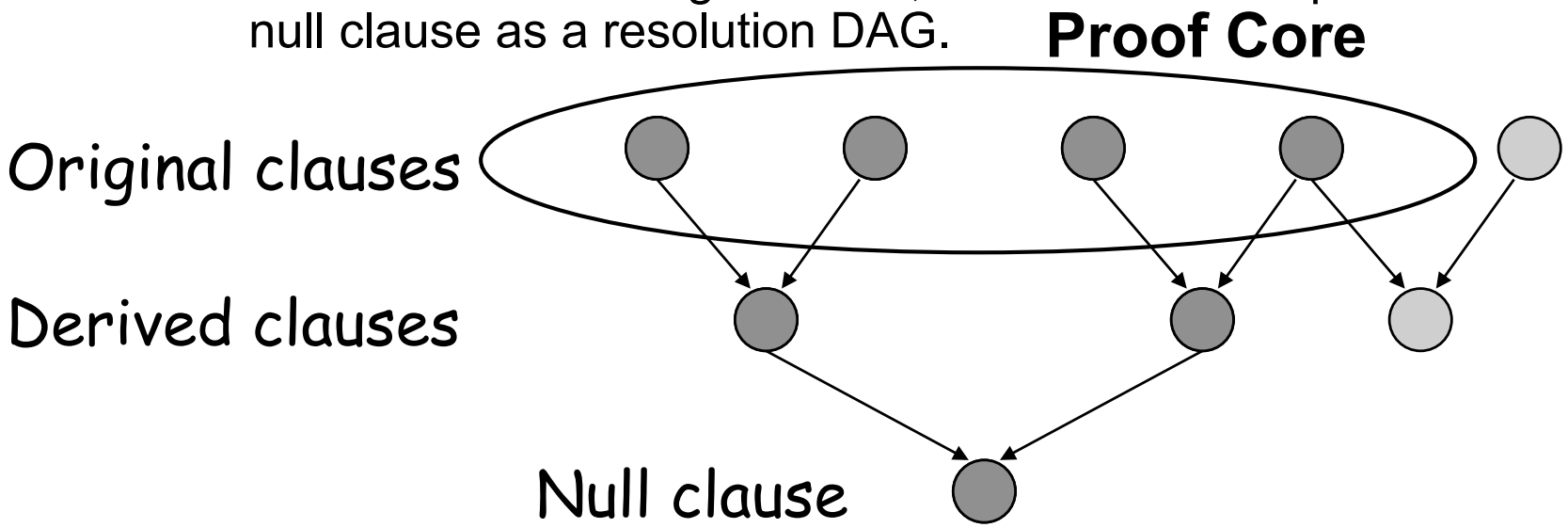
Resolution Graph

- A conflict is encountered
  - A learned clause is generated
- More conflicts are resolved...
- A conflict is encountered in decision level 0
  - Problem is proven UNSAT



Refutation / Proof Core of a SAT Problem

- Remember: Resolution-based SAT?
  - A problem is proven UNSAT if the resolution steps end up in a NULL clause
- Refutation = a proof of the null clause
  - Also called "proof core" or "UNSAT core"
  - Record a DAG containing all resolution steps performed during conflict clause generation.
  - When null clause is generated, we can extract a proof of the null clause as a resolution DAG.



What can/should be covered in this topic?

- Fundamentals of Boolean Satisfiability (SAT)
- Techniques to improve SAT solving
- ~~Circuit-based SAT algorithms~~
- SAT-based (hardware) verification
  - Bounded model checking (BMC)
  - Inductive proof
  - ~~SAT-based abstraction and refinement~~
  - ~~Interpolation-based method~~
  - ~~Property directed reachability~~

What affect the SAT efficiency?

1. Decision order

2. Logic implication (Boolean Constraint Propagation, BCP)

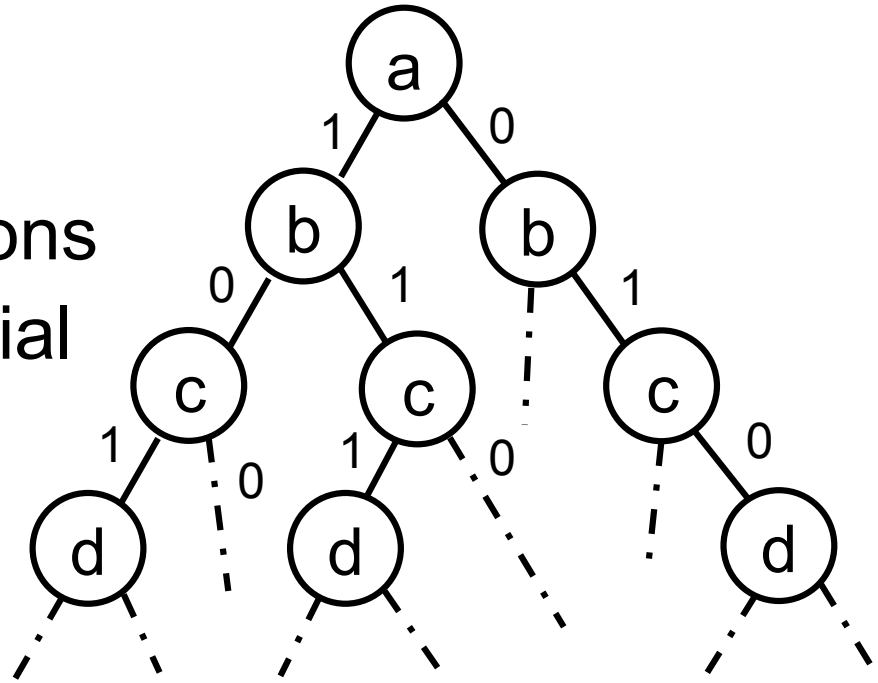
3. Various learning techniques

4. Database simplification

Impact of Decision Ordering

- Decision ordering: the order of gates that the corresponding decisions are made
  1. Order of gates
  2. Decision values

  ➔ Good and bad decisions can lead to exponential difference (e.g. 2^{10} vs. 2^{50})



- (Think) Does the decision value matter? (i.e. should we decide on '1' or '0' first?)

Static Decision Ordering

- Decision order and values are pre-computed in the beginning and remain unchanged

1. Topological
   - Depth-first
   - Breadth-first
   - Guided by gate types
2. Probability-based
   - Controllability / Observability
   - Signal probability
   - (Weighted) Random
3. Influence-based
   - Literal count
   - #fanins / #fanouts
   - Influence of implications

Dynamic Decision Ordering

- Decision order and values are dynamically determined based on current implication values, justification frontier, etc.
  - Use similar criteria as static method
  - But can mix different rules dynamically
- Pros
  - May lead to better decisions
  - Avoid useless decisions
- Cons
  - Overhead in computing dynamic ordering may be high
  - Effectiveness sometimes is hard to predict

➔ However, experiences show that the best is:

1. Has a good initial decision ordering
2. Adaptively adjust the decision order after a certain amount of backtracks

zChaff's Variable State Independent Decaying Sum (VSIDS) Decision Heuristic

(1) Each variable in each polarity has a counter, initialized to 0.

(2) When a clause is added to the database, the counter associated with each literal in the clause is incremented.

(3) The (unassigned) variable and polarity with the highest counter is chosen at each decision.

(4) Ties are broken randomly by default, although this is configurable

(5) Periodically, all the counters are divided by a constant.

Zhang, et al, DAC 2001

Berkmin – Decision Making Heuristics

E. Goldberg, and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver", *Proc. DATE* 2002, pp. 142-149.

- Identify the most recently learned clause which is unsatisfied
- Pick most active variable in this clause to branch on
- Variable activities
  - updated during conflict analysis
  - decay periodically
- If all learnt conflict clauses are satisfied, choose variable using a global heuristic
- Increased emphasis on "locality" of decisions

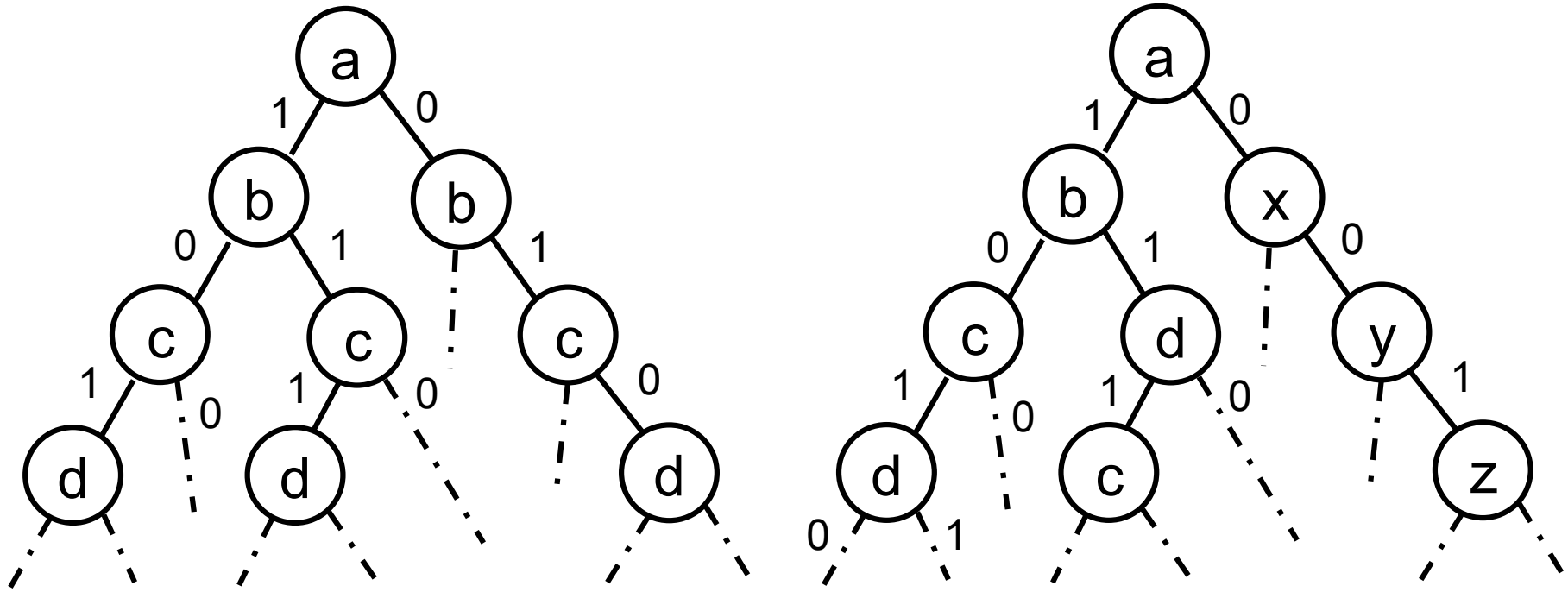
More decision heuristics...

- Variable Move-To-Front (VMTF)
- Clause Based Heuristic (CBH)
- Resolution Based Scoring (RBS)
- ...

- In general, there is no single decision heuristic that works for every case.
  - ➔ How to adaptively move to a good decision heuristic may be the winner...

A closer look at binary decision tree

Should the decision orderings on all branches be the same?



Remember when we talked about

conflict-driven learning,

we mentioned that

by adding a learned clause

we can do non-chronological backtracking,

while still achieve complete proof

How??

The Constraint Refinement Process

- Search region is gradually narrowed down by the learned constraints
- Learned information is universally true
  - Independent of the target implication, only related to the circuit function
  - The proof efforts between different properties can be shared
    - ➔ Incremental SAT
- Decision process can “restart” any time any where!!
  - Can use different decision ordering to explore different area in the decision tree
    - Previous efforts will not be wasted

What affect the SAT efficiency?

1. Decision order

2. Logic implication (Boolean Constraint Propagation, BCP)

3. Various learning techniques

4. Database simplification

BCP Checking for CNF-Based SAT

- If a literal in a clause gets an implication ‘1’
  - ➔ The clause is satisfied
- If a literal in a clause gets an implication ‘0’
  - ➔ Check: how many literals in the clause have unknown value?
    - >= 2 : no operation
    - = 1 : the remaining literal will be implied ‘1’
    - = 0 : the clause is evaluated to ‘0’ ➔ a conflict !!

Complexity for BCP

- Initially all literals are ‘x’
- A decision is made
  - Which clauses are affected?
  - Which of the above should produce new implications? Which of the above may lead to conflict?
  - Which clauses are affected due to new implications?
  - What happens if backtrack is needed?

A naïve/brute-force BCP approach

- a + b + c + d + e // all literals are ‘x’
- a + b + c + d + e // a = 0; any new imp?
- a + b + c + d + e // b = 0; any new imp?
- a + b + c + d + e // c = 0; any new imp?
- a + b + c + d + e // If conflict on other clause, and b, c are undone
- a + b + c + d + e // c = 0; any new imp?
- a + b + c + d + e // d = 0; any new imp?

How to improve the naïve/brute-force BCP approach?

- a + b + c + d + e // all literals are ‘x’
- a + b + c + d + e // a = 0; any new imp?
- a + b + c + d + e // b = 0; any new imp?

➔ Do we really need to check this, if we know there are more than two literals are ‘x’?

➔ How do we know there are at least two literals with value ‘x’?

➔ Do we need to check it, if we know there is a literal with value ‘1’?

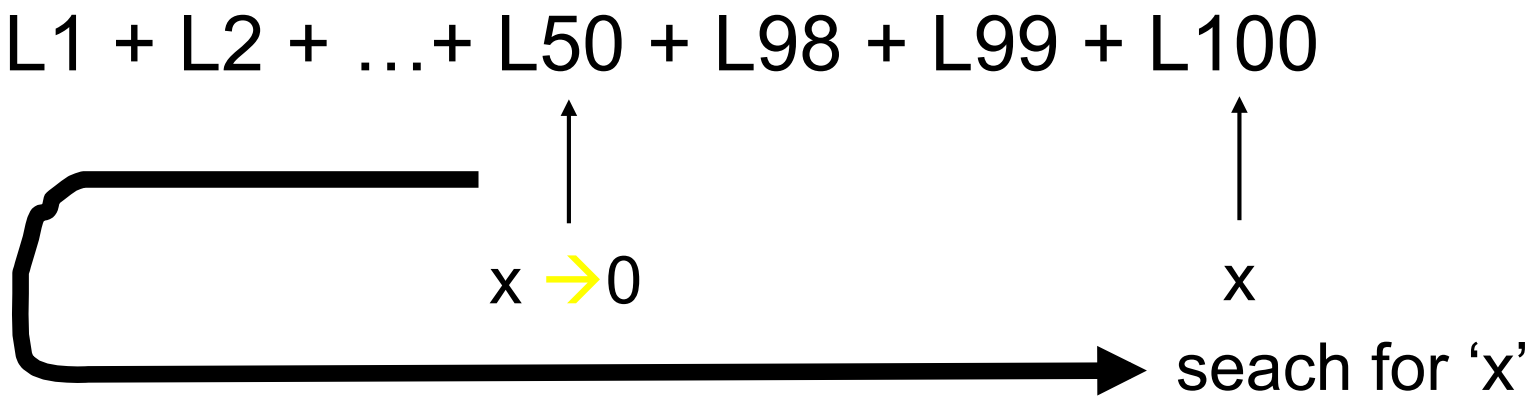
➔ How do we know there is a literal with value ‘1’?

2-Watched-Literal Algorithm

H. Zhang, SATO, CADE 97; M. Moskewicz *et al*, Chaff, DAC 2001

- For each clause, keep 2 pointers on 2 literals that have "non-0" values
  - If any watched literal gets implication '0'
    - Scan in the clause for another literal with "non-0" value
    - If found, update the watched literal pointer

      Else, imply the other watched literal with value '1'



In the previous example…

- a + b + **c** + d + **e** // Let ‘c’ and ‘e’ are watched
- a + b + **c** + d + **e** // a = 0; NO action

➔ How do we know ‘a’ is NOT watched?

➔ Keep a “watching list” for each literal !!

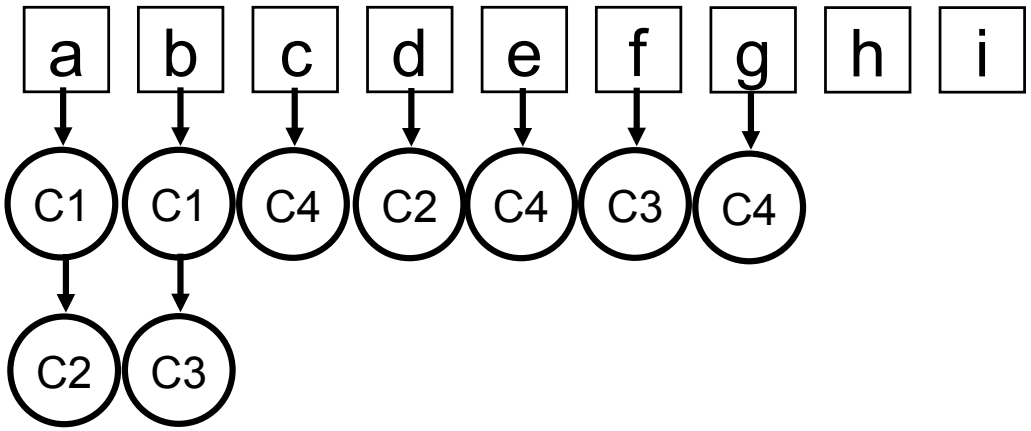
- a + b + **c** + d + **e** // b = 0; NO action
- a + b + c + **d** + **e** // c = 0; UPDATE watches !!
- a + b + c + **d** + **e** // Backtrack, NO action !!
- a + b + c + **d** + **e** // c = 0; NO action !!
- a + b + c + **d** + **e** // d = 1; NO action !!

2-Watched-Literal Algorithm Example

Each clause stores:
2 watched literal pointers

Each literal stores:
A list of watching clauses

C1: (a + b + c + d)
C2: (a + d + e + f + g)
C3: (b + f)
C4: (c + e + g + h + i)



c
- Update watched literal pointer for C4 (for example, to ‘g’)
- Erase c’s watching-clause list
- Add ‘C4’ to g’s watching-clause list

[Note] Don’t need to check ‘C1’

2-Watched-Literal Algorithm Example

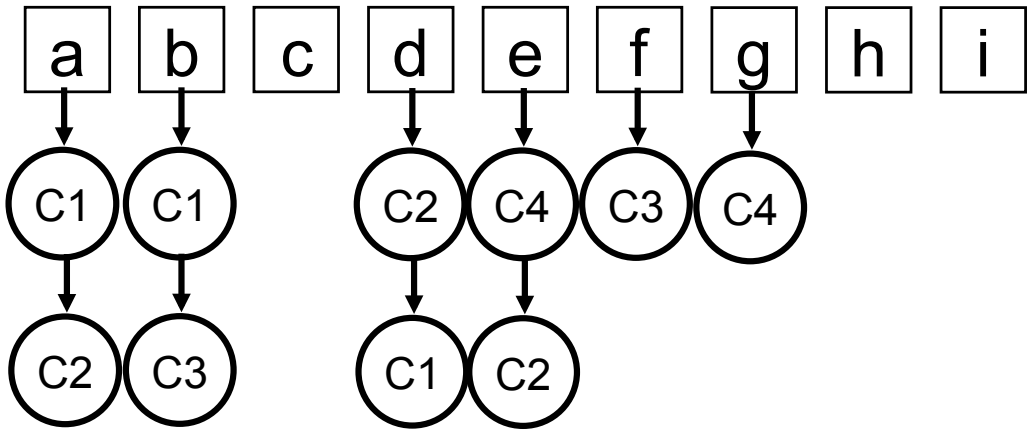
Each clause:

2 watched literal pointers

Each literal:

A list of watching clauses

C1: (a + b + c + d)
C2: (a + d + e + f + g)
C3: (b + f)
C4: (c + e + g + h + i)



a

- Update watched literal pointer for C1 (only choice, to ‘d’)
- Update watched literal pointer for C2 (for example, to ‘e’)
- Erase a’s watching-clause list
- Add ‘C1’ to d’s and ‘C2’ to e’s watching-clause lists

2-Watched-Literal Algorithm Example

Each clause:

2 watched literal pointers

Each literal:

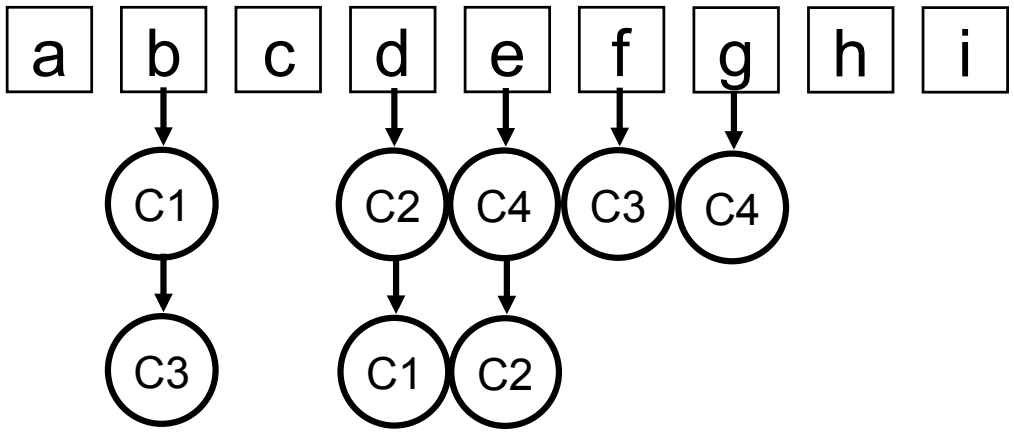
A list of watching clauses

C1: (a + b + c + d)

C2: (a + d + e + f + g)

C3: (b + f)

C4: (c + e + g + h + i)



b

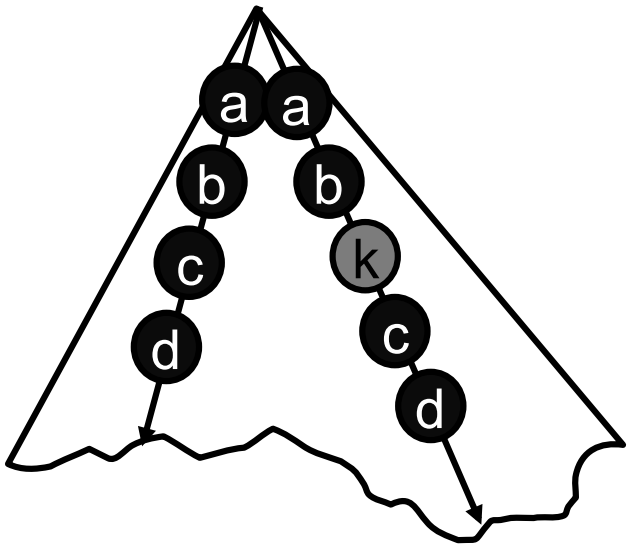
- No more unknown literal for C1 : d = 1
- No more unknown literal for C3 : f = 1

[Note] No change on watched literals

Caching Effect: Reducing from O(n) to almost O(C)

- The fact
  - Most of the time, the decision orderings at different parts of the decision tree are quite similar during a proof (or even from proof to proof)

  ➔ Literals in a clause get the implications almost by the same order every time



- Watched literal

  → point to the last implied literal

  ➔ Don't update watched literals after backtrack. After backtracks, no evaluations from the other unwatched literals.

(L1 + L2 + L3 + L4 + L5 + L6)

Logic implication can be very efficient for CNF-based SAT by using “watch” scheme.

Can this idea be applied to circuit-based SAT?

Generic Watch Scheme

- It can be shown that the watch scheme can be applied to primitive gates (e.g. AND/OR) in a circuit SAT solver, and can be further extended to complex gates such as MUXes, Pseudo Boolean gates, etc.
- For more details, please refer to:
  - "QuteSAT: A Robust Circuit-based SAT Solver for Complex Circuit Structure", DATE 2007.

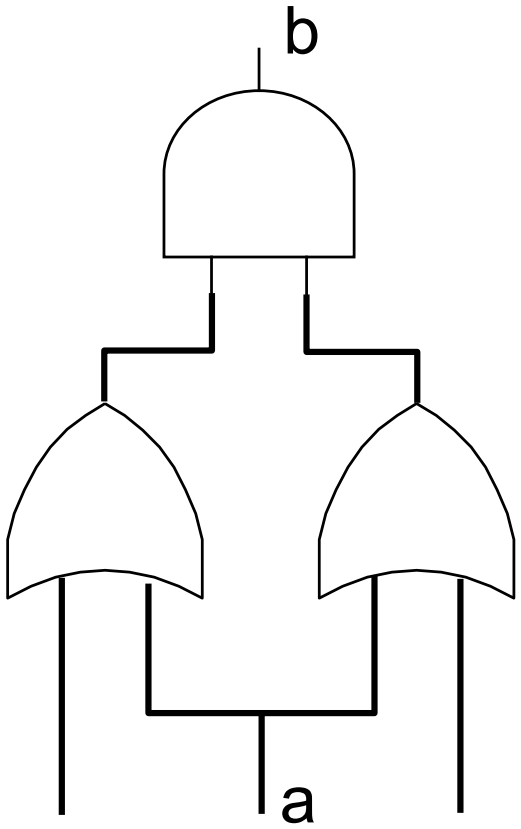
Various Learning Techniques

- Other than conflict-driven learning, there are many other learning techniques that can help
  - Derive more implications
    → may help find the conflict earlier
  - Provide information for decision ordering

1. Static learning
2. By signal correlations
3. Recursive learning
4. Success-driven learning

Static Learning

- Learn by contrapositive
  (a → b ≡ !b → !a)
- e.g.



a = 1
Learned b = 0 → a = 0

The question is:
which gate to learn??

Ref: “SOCRATES: A Highly Efficient Automatic Test Pattern Generation System”, Schulz *et.al*, TCAD 1988

Learned by Signal Correlations

- A proof-based approach
  - Since learned information is universally true, we can create some internal interesting properties, and use these properties to derive some interesting learning (by conflict analysis)

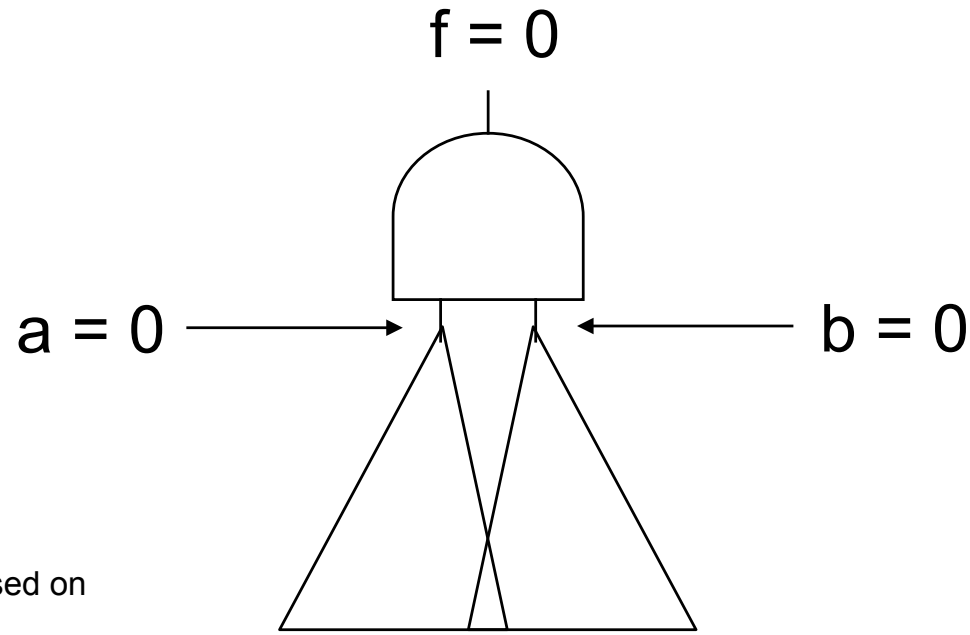
- e.g. By simulation, if we find a gate ‘g’ is very likely to stuck at some value ‘v’
  ➔ Witness “g = ¬v” (should produce many conflicts)
- e.g. By simulation, if two signals respond almost the same
  ➔ Witness “p != q”

- No matter the proof is finished or not
  - We can always learn something

Ref: Feng Lu, *et. al*, “A Circuit SAT Solver with Signal Correlation Guided Learning”, DATE 2003

Recursive Learning

- To justify f = 0
  - (a = 0) or (b = 0)
  - Let S_a and S_b be the set of implications from (a = 0) and (b = 0), respectively
  - Let $S = S_a \cap S_b$
    ➔ (f = 0) implies S
- A recursive process
- Deep recursion could be very expensive
- How to record the learned implicaiton?



Ref: “HANNIBAL: an efficient tool for logic verification based on recursive learning”, Wolfgang Kunz, ICCAD 1993

Conflict vs. Success-Driven Learning

Motivation: Traditional SAT approach finds only 1 solution, can we find more (or all) the solutions?

- How to record the solutions?
  - Hash table? (too expensive)
- Success-driven learning
  - Similar to conflict learning
  - When we find one solution, say (v1, v2, …, vn), add a blocking gate "v1 && v2 && … vn = 0" so that
    - This solution won't be repeated
    - May lead to new implication
    - Can continue the justification process for the next solution
  - At the end, all the solutions are recorded as set of blocking gates (or clauses)

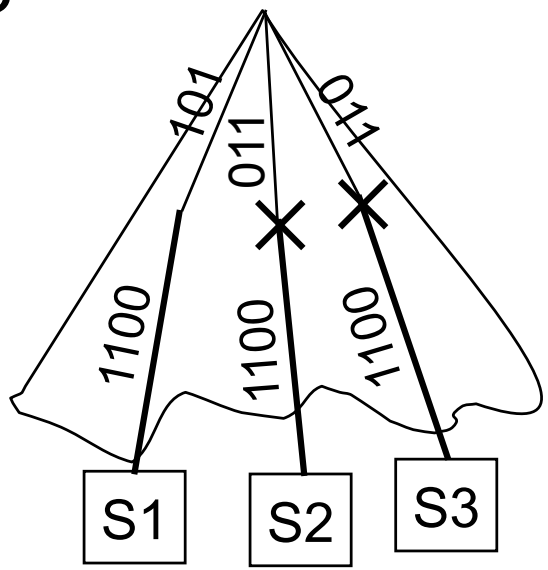
Conflict vs. Success-Driven Learning

- However, the number of solutions in a SAT problem can be very huge!!
  - ➔ Some solutions may look alike ---

    e.g. 1010011, 1100011, 0110011...

    s1 s2 s3

Can we predict that the sub -solutions under the sub-search tree are already covered?

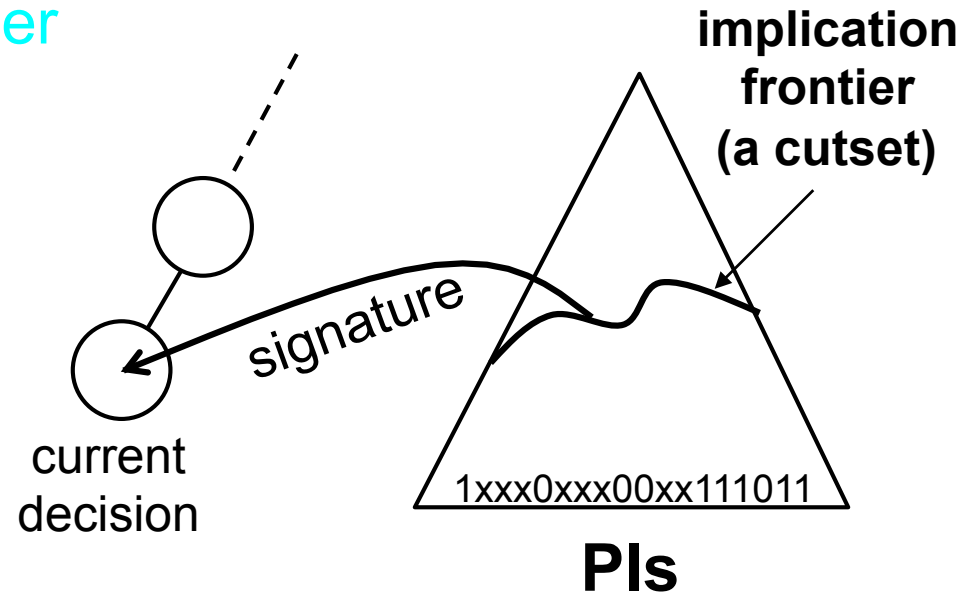


Success-Driven Learning

- Ref: Shuo, et al. DATE 2003
- Assume
  - ATPG-based technique (work on circuit)
  - Decisions on PIs only → forward implications
- Search State Equivalence
  - If two decisions have the same signature

  ➔ The "sub-solutions" under the sub-search space are the same!!

  ➔ No need to search
- Note: they also store the solutions in a "free BDD"



Although “learning” in general can lead to more implications and possibly lead to conflicts earlier (i.e. bound earlier) ---

1. It may slow down the implication process
2. It may affect the decision ordering, which may not necessarily reduce the #decisions

What can we do to make the learning useful?

1. Use learning to find better decision ordering
   - zChaff uses learned information to refine the decision ordering
   - BerkMin uses learned information to increase emphasis on “locality” of decisions
2. With conflict analysis, decision can restart any time
   - Change to different decision ordering heuristic to explore different areas in the input space
3. Modify the learned information
   - Remove least-used learned information
   - Simplify or synthesize the learned information
   - Any other idea?

What affect the SAT efficiency?

1. Decision order

2. Logic implication (Boolean Constraint Propagation, BCP)

3. Various learning techniques

4. Database simplification

Simplify SAT Database, why bother?

1. CNF proof instances generated from real-life problems (e.g. assertions in a circuit) are usually quite redundant
   - Better clausifier?
2. During SAT proof, the number of added learnt clauses will become much larger than the number of original clauses
   - A few thousands vs. millions
3. Slimmer clause database usually implies better proof efficiency

Many SAT proof database simplification techniques...

- Especially for CNF...
  1. Variable Elimination by Clause Distribution
  2. Clause Subsumption
  3. Self-Subsuming Resolution
  4. Simplification by Definition of a Gate
  5. Blocked Clause Elimination
  6. Equivalent Literal, Pure Literal Elimination, etc
- Also, many techniques to generate “better” CNF instances (from circuit problems)
  1. Tseitin Transformation
  2. Plaisted-Greenbaum Encoding
  3. Utilization of Logic Synthesis Techniques
- Note: in the following slides, (‘) for a literal/variable means negation; for clause/problem means another one.

Satisfiability Equivalent Problem

- A SAT proof instance P1 is a "satisfiability equivalent (SAT-EQ) problem" of another proof instance P2 iff:
  - P1 is SAT implies P2 is SAT, and
  - P1 is UNSAT implies P2 is UNSAT

  ➔ Note that P1 is NOT necessarily logically equivalent to P2
- "Resolution" preserves the SAT-EQ
  - Let $\otimes$ be the resolution operator, clauses $c_1 = (x + a_1 + ... + a_n)$, $c_2 = (x' + b_1 + ... + b_m)$ and $c = c_1 \otimes c_2$ is the resolvent of c_1 and c_2

  ➔ $c = (a_1 + ... + a_n + b_1 + ... + b_m)$

  ➔ $c_1 \wedge c_2$ implies c

Variable Elimination by Clause Distribution

- In a CNF proof instance S, let S_x and $S_{x'}$ be the sets of clauses in which x and x' occurs, respectively.
  ➔ $S = S_x \cup S_{x'}$
- Resolution operation can be lifted to sets of clauses as:
  - $S' = S1 \otimes S2 = \{ C1 \otimes C2 \mid \forall C1 \in S1, \forall C2 \in S2 \}$

  ➔ x will be eliminated from S

  ➔ S' is SAT-EQ to S

  ➔ Is S' always simpler than S?

  ➔ S' may contain several trivial clauses. (A clause is called trivial if it contains a variable and its negation)

  ➔ S' also contains many subsumed clauses

Clause Subsumption

- A clause C1 is said to (syntactically) subsume another clause C2 if C1 $\subseteq$ C2
  - e.g. (a + b) subsumes (a + b + c)
  - A subsumed clause is redundant in a SAT problem and can be removed from the proof
- [Remember] Variable eliminations by clause distribution usually lead to many subsumed clauses
- How to identify the subsumed clauses in a CNF?
  - [Ref: Een SAT2005]
  1. For each clause, a 64-bit signature is stored.
  2. Each literal is hashed to 0...63.
  3. The signature = bitwise_OR of the hashed literal indices
  4. Occur_list: literal → clauses
  5. Check subsumptions with the aid of the clause signatures

Self-Subsuming Resolution

- It's often that one clause can "almost subsume" the other. For example:
  - C1: (x' + a), C2: (x + a + b)

  ➔ C1 does not subsume C2
  - But if we do C2' = C2 ⊗ C1 = (a + b)

  ➔ C2' will subsume C2

  ➔ We say "C2 is strengthened by self-subsumption using C1"

  (i.e. Problem becomes: (x' + a) (a + b))
- Self-subsumption is a powerful technique in simplifying CNF database

Simplification by Definition of a Gate

- There are usually many functionally dependent variables in a CNF. For example:
  - ... (x + a' + b')(x' + a)(x' + b) ...

  ➔ x is actually equal to "a∧b"

  ➔ We call the equation "x = a∧b" the definition of x

  ➔ Can we remove the variable x?
- [Fact] If x has a definition and is eliminated by clause distribution, many redundant resolvents are generated.
  - [e.g.] 1 2 3 4 5 6

    (x + c)(x + d')(x + a' + b') (x' + a)(x' + b)(x' + e' + f)

    The resolvents are:
    - 1⊗4 1⊗5 2⊗4 2⊗5 3⊗6

      (c + a) (c + b) (d' + a) (d' + b) (a' + b' + e' + f) ➔ A
    - 3⊗4 3⊗5

      (a' + b' + a) (a' + b' + b) ➔ B
    - 1⊗6 2⊗6

      (c + e' + f) (d' + e' + f) ➔ C

  ➔ We will show in the next slide that A implies B ∪ C

Simplification by Definition of a Gate

- Let a CNF S contains a definition of a variable x, that is, $x = a \wedge b$ ---
  - ... $(x + a' + b')(x' + a)(x' + b)$ ...
- Let $S = G \cup R$,

  $G = (x + a' + b')(x' + a)(x' + b)$

  $R = S \setminus G$
- Let Gx and Gx' (Rx and Rx') be the set of clauses of G (R) in which x and x' occurs, respectively.
  - $S = (Gx \cup Rx) \cup (Gx' \cup Rx')$
  - $S' = (Gx \cup Rx) \otimes (Gx' \cup Rx') = S'' \cup G' \cup R'$

    where ---

    $S'' = (Rx \otimes Gx') \cup (Gx \otimes Rx')$

    $G' = Gx \otimes Gx'$

    $R' = Rx \otimes Rx'$

  ➔ S'' implies $G' \cup R'$

  That is, G' and R' are redundant and thus can be removed

Simplification by Definition of a Gate

- In the previous example ---
  - Example:
    $(x + c)(x + d')(x + a' + b')$ $(x' + a)(x' + b)(x' + e' + f)$

    Becomes...

    $((x + a' + b') \otimes (x' + e' + f))$
    $(\{(x + c), (x + d')\} \otimes \{ (x' + a), (x' + b) \})$

    ➔ $(a' + b' + e' + f)\ (c + a)\ (c + b)\ (d' + a)\ (d' + b)$

Blocked Clause Elimination (ref: Järvisalo TACAS 10)

- Blocking literal
  - A literal l in a clause C of a CNF F blocks C (w.r.t. F) if for every clause C' $\in$ F with l' $\in$ C', the resolvent of (C $\otimes$ C') on l is a tautology.
- Blocked clause
  - A clause is blocked if it has a literal that blocks it

  ➔ Removal of a blocked clause reserves satisfiability
- Example: (a' + b) (a + b' + c') (a' + c)
  - c blocks (a' + c) because (a + b' + c') $\otimes_c$ (a' + c) = 1

  ➔ Problem becomes (a' + b) (a + b' + c')
  - b' blocks (a + b' + c') ➔ Problem becomes (a' + b)

  ➔ Problem is satisfiable!!

Equivalent Literal, Pure Literal Eliminations

- Equivalent Literal
  - If both (a + b') and (a' + b) exist in CNF, a and b are equivalent ➔ Replace b with a
  - If (a + b'), (b + c') and (c + a') exist in CNF, a, b and c are equivalent ➔ Pick one representative literal

- Pure Literal Eliminations
  - If some variable exists only in one phase in all the clauses it appears (i.e. pure literal) ---

  ➔ Assigning these literals to '1' preserves satisfiability

  ➔ Removal of these clauses preserves satisfiability

Transforming Circuit Problems for CNF SAT

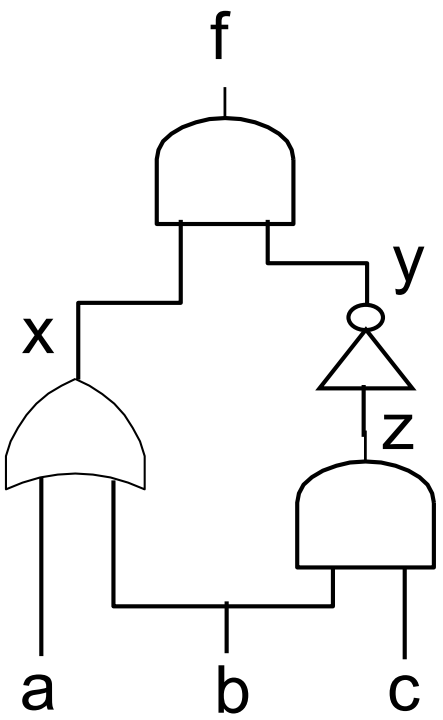
- Although the CNF and circuit-based SAT can be equally efficient, however, there are much more existing CNF solvers than circuit SAT.
  - It's often a need to transform a circuit problem to CNF

1. Tseitin Transformation
2. Plaisted-Greenbaum Encoding
3. Utilization of Logic Synthesis Techniques

Tseitin Transformation

1. Assign each gate with a variable
2. For each gate, generate CNF clauses for its input and output variables

- Example:



F: f (f ⇔ x ∧ y) (x ⇔ a ∨ b) (y ⇔ z') (z ⇔ b ∧ c)

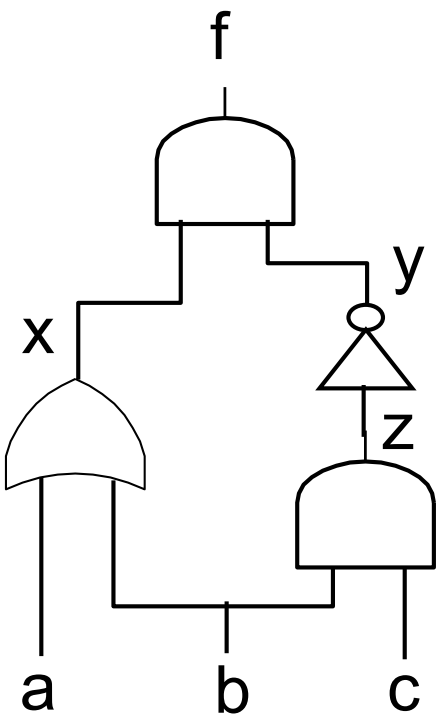
≡ f (f → x) (f → y) (f' → x' ∨ y')
(x' → a') (x' → b') (x → a ∨ b)
(y → z') (y' → z) ...

≡ f (f' + x) (f' + y) (f + x' + y')
(x + a') (x + b') (x' + a + b)
(y' + z') (y + z) ...

However, many redundant variables /clauses are generated...

Plaisted-Greenbaum Encoding

- Polarity-cared transformation
  - A is satisfiable iff $L_A \wedge A^+$
  - $\neg$A is satisfiable iff $L_{\neg A} \wedge A^-$



F: f ...

$\equiv$ f (f $\rightarrow$ x $\wedge$ y)...

$\equiv$ f (f' + x) (f' + y) (x $\rightarrow$ a v b)...

$\equiv$ (f' + x) (f' + y) (x' + a + b) (y $\rightarrow$ z')...

$\equiv$ f (f' + x) (f' + y) (x' + a + b) (y $\rightarrow$ b' v c')

$\equiv$ f (f' + x) (f' + y) (x' + a + b) (y' + b' + c')

Utilization of Logic Synthesis Techniques

- How many are there n-input Boolean functions?
  - e.g. 65536 for 4-input functions
  - Are they all distinct?

$$2^{2^n}$$

- NPN-equivalent functions
  - Two functions are called NPN-equivalent iff they are equivalent by negating parts of the inputs and outputs, and by permuting inputs
  - e.g. (a ∧ b) and (b ∨ a') are NPN-EQ
- How many are there distinct NPN-EQ n-input Boolean functions?
  - Well, no general formula...
  - 1-input: 2; 2-input: 4; 3-input: 14; 4-input: 222; 5-input: ??

Utilization of Logic Synthesis Techniques

- How to utilize this NPN-EQ concept in generating CNF formula from circuit?
  - For each NPN-EQ class, derive the “best” CNF representation
  - Partition the circuit into clusters of n-input “macro cells”. That is, each macro cell has exactly n inputs and 1 output.
    - e.g. 4-input
  - Generate CNF by table lookup

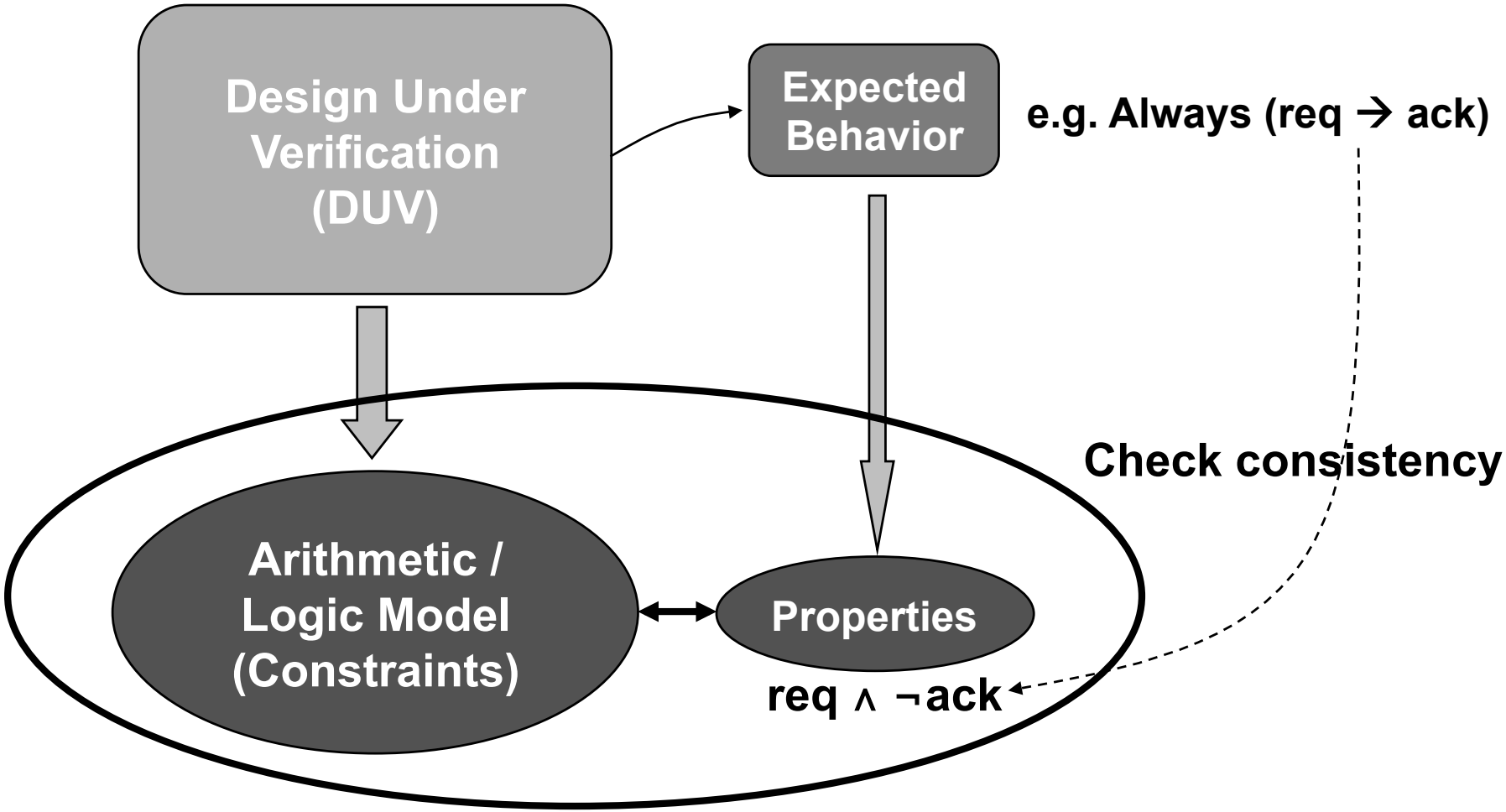
What we have learned...

- What is the Boolean Satisfiability (SAT) problem?
- Circuit SAT vs. CNF SAT
- Key factors for SAT efficiency
  - Boolean constraint propagation (BCP)
  - Decision ordering
  - Various learning
  - Database simplification

What can/should be covered in this topic?

- Fundamentals of Boolean Satisfiability (SAT)
- Techniques to improve SAT solving
- ~~Circuit-based SAT algorithms~~
- SAT-based (hardware) verification
  - Bounded model checking (BMC)
  - Inductive proof
  - ~~SAT-based abstraction and refinement~~
  - ~~Interpolation-based method~~
  - ~~Property directed reachability~~

Formal Verification Technologies



What is formal verification?

"The expression 'formal verification', as it appears in the literature, refers to a variety of (often quite different) methods used to prove that a model of a system has certain specified attributes. What distinguishes 'formal' verification from other undertakings also called 'verification' is that 'formal' verification conveys a promise of mathematical certainty. The certainty is that if a model is formally verified to have a given attribute, then no behavior or execution of the model ever can be found to contradict this"

Robert Kurshan, "Computer-Aided Verification of Coordinating Processes"

What is formal verification?

- In general, formal verification can be applied to various disciplines ---
  - Hardware design validation
  - Software verification
  - Protocol checking
  - and more...

- The “models” checked by formal verification can be ---
  - Continuous / discrete time
  - Finite / infinite states
  - Hardware / software
  - Deterministic / non-deterministic,... etc

In this class, we will focus on

"**Hardware Verification**",

where the design is usually modeled as a

"**Finite State Concurrent System**".

"**Model Checking**"

is the most widely studied and used technique.

Model Checking Problem

- Let M be the state transition graph obtained from the concurrent system.
- Let f be the specification expressed in temporal logic.

- Model Checking
  - Find all states s of M such that $M, s \models f$

The Process of Model Checking

1. Modeling
   - Convert a design into a formalism accepted by a model checking tool
   - Parsing, compilation, abstraction, reduction, etc
2. Specification
   - What are the properties the design must satisfy?
   - e.g. Temporal logic
3. Verification
   - Try to prove that the model is compliant with the specification
   - If not, manual debugging is usually required

"Model Checking", E.M. Clarke, et al.

Remember ---

- Model checking is an automatic technique for verifying “finite state concurrent systems”.

What can be the basic formalism for “finite state concurrent system” model?

(HDL, circuit, FSM, ??)

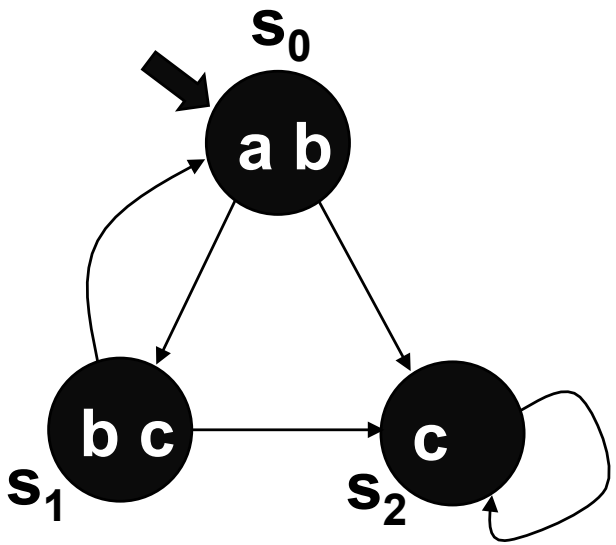
(FYI) Kripke Structure --- a type of state transition graph

- Kripke structure M over a set of atomic propositions AP is a four tuple $M = (S, S_0, R, L)$, where
  1. S is a finite set of states
  2. $S_0 \subseteq S$ is the set of initial states
  3. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$
  4. $L: S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.
- Think ---
  - What is the "state" for a concurrent system (e.g. hardware design)?
  - How many states does a circuit have?

What?

- Kripke structure M over a set of atomic propositions AP is a four tuple M = (S, S_0, R, L), where ---
  1. S is a finite set of states
     - e.g. A hardware circuit has finite set of states
  2. $S_0 \subseteq S$ is the set of initial states
     - Note: This item can be omitted if we don't care about initial states
  3. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that R(s, s')
     - i.e. The next state always exists → The system can keep on running
  4. $L: S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.
     - i.e. For a given state, some atomic propositions are true in this state, while others are false
     - On the other hand, for a given atomic proposition, what are the states on which this proposition is true?

Example of Kripke Structure

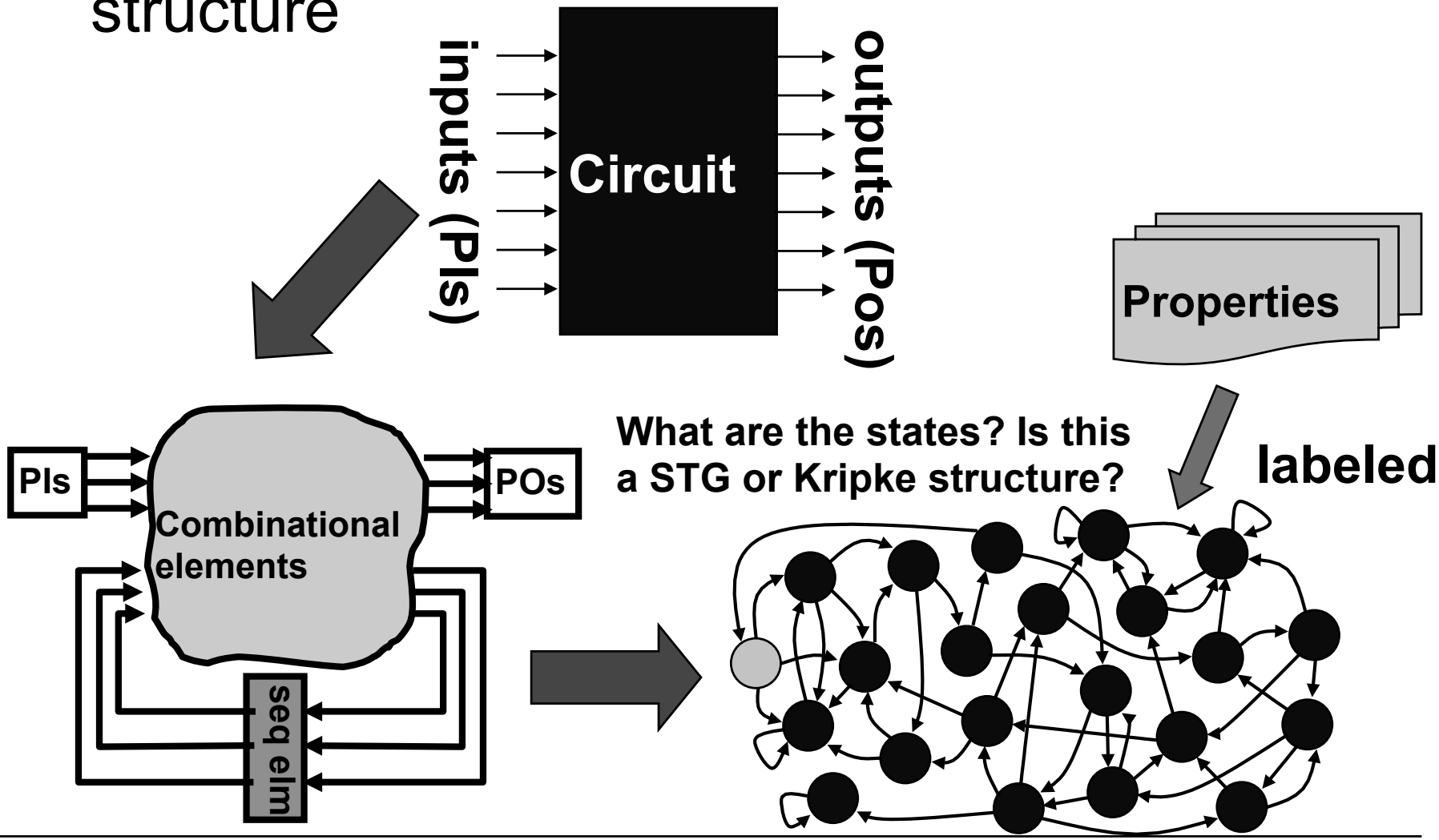


$S: \{s_0, s_1, s_2\}$

$S_0: \{s_0\}$

AP: {a, b, c}

- It can be easily shown that a synchronous digital circuit can be converted to a kripke structure



The Process of Model Checking

1. Modeling
   - Convert a design into a formalism accepted by a model checking tool
   - Parsing, compilation, abstraction, reduction, etc
2. Specification
   - What are the properties the design must satisfy?
   - e.g. Temporal logic
3. Verification
   - Try to prove that the model is compliant with the specification
   - If not, manual debugging is usually required

"Model Checking", E.M. Clarke, et al.

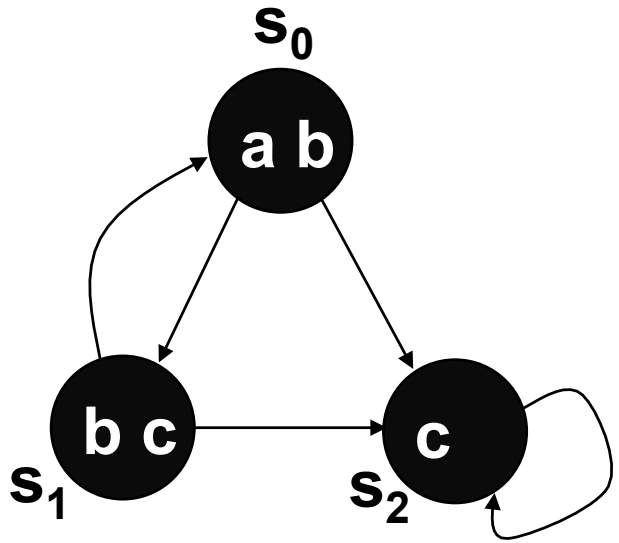
Property Specification

- Remember, the definition of Kripke Structure
  4. L: S $\rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.
     - i.e. For a given state, some atomic propositions are true in this state, while others are false
     - On the other hand, for a given atomic proposition, what are the states that this proposition is true?

➔ What are the formula for the atomic proposition?

➔ In terms of what? States, state variables, or?

Property Specification



What are the formulae to describe propositions a, b, and c?

➔ **'a' is true iff state s = s_0**
(this type of description does not work if number of states is large)

➔ **'a' will never be true if 'c' is true for two consecutive states**

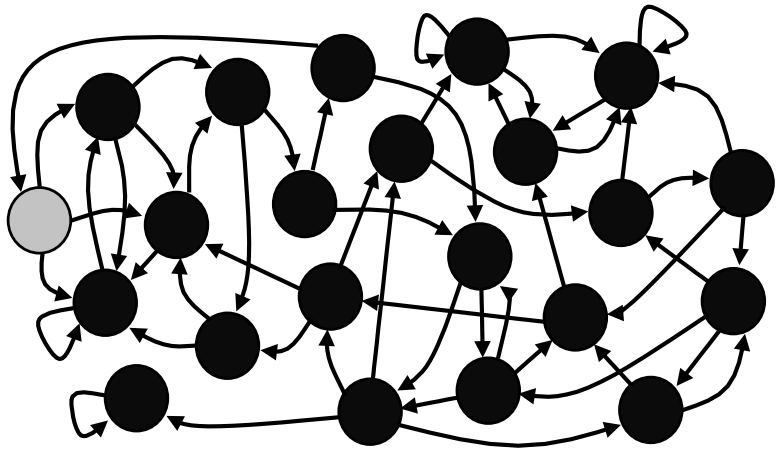
➔ **There is an execution trace that 'a' can be true infinitely often**

Property Specification

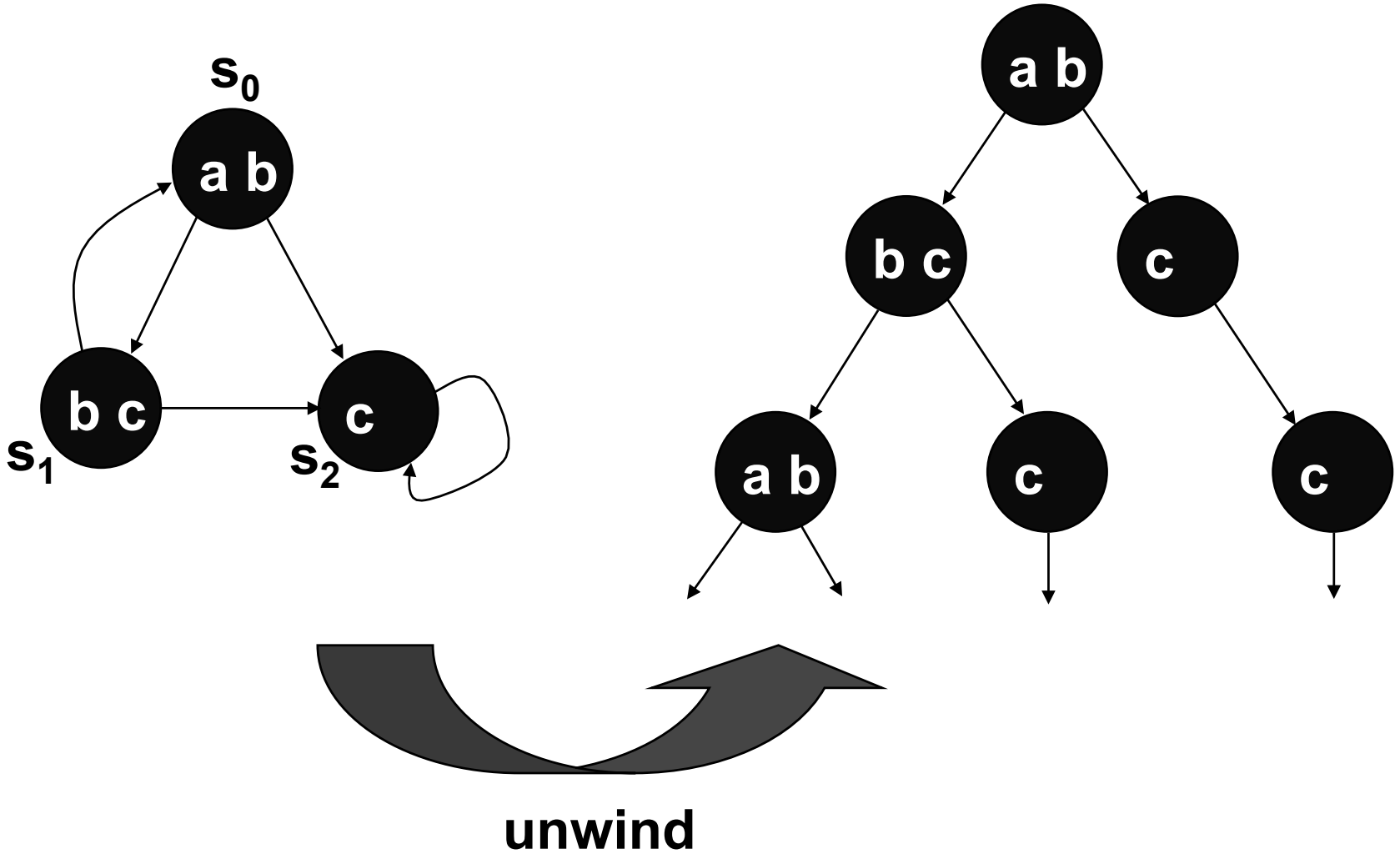
- To describe the formula for a proposition, in addition to the states or state variables, we also need the "*temporal expression*".

- We will first introduce "temporal logic" for the description of the proposition over the span of execution time

  - Never?
  - Two consecutive?
  - Execution trace?
  - infinitely often?



Kripke Structure vs. (Infinite) Computation Tree

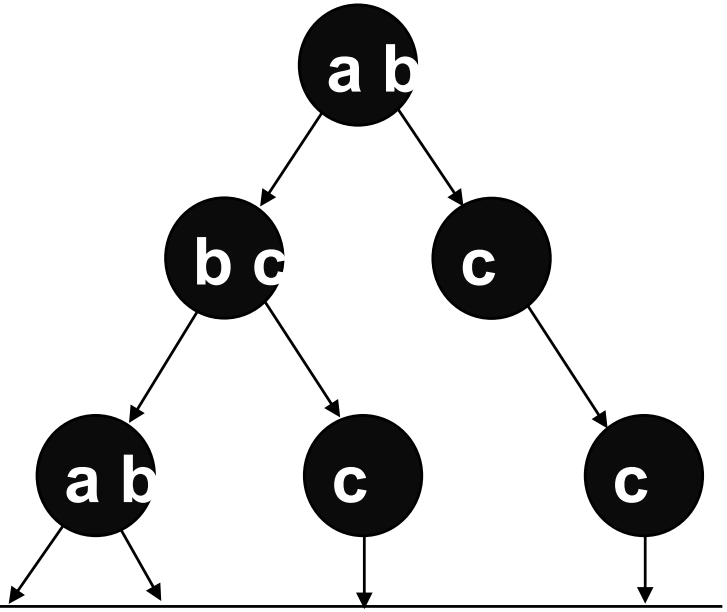


Describe Properties on Computation Tree

- There exists an execution path such that ‘b’ always holds

- For every execution path, c will eventually holds

➔ “**Path**” and “**Temporal**” operators/quantifiers

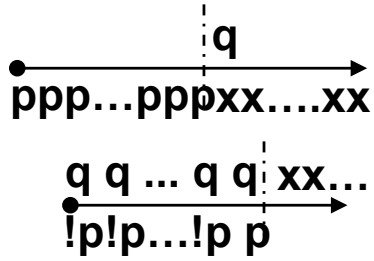


Computation Tree Logic* (CTL*)

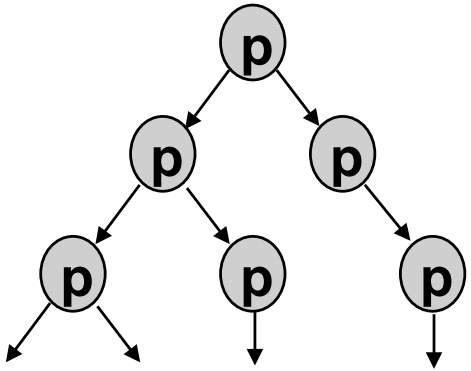
- Describe the properties for the propositions on the computation tree

1. Path quantifier
   - A --- “for every path”
   - E --- “there exists a path”

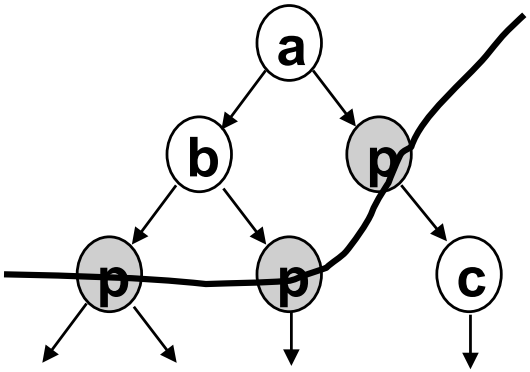
2. Temporal operator (State quantifier)
   - Xp --- p holds next time
   - Fp --- p holds sometime in the future
   - Gp --- p holds globally in the future
   - pUq --- p holds until q holds (exclusive)
   - pRq --- p release q (inclusive)
     q holds up to (and including) p holds



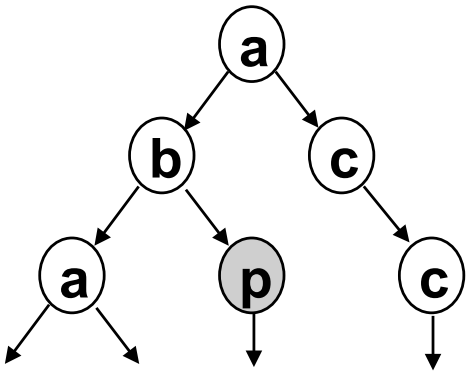
For example…



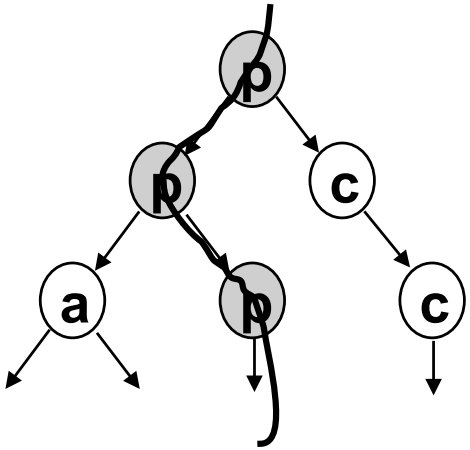
AGp



AFp



EFp



EGp

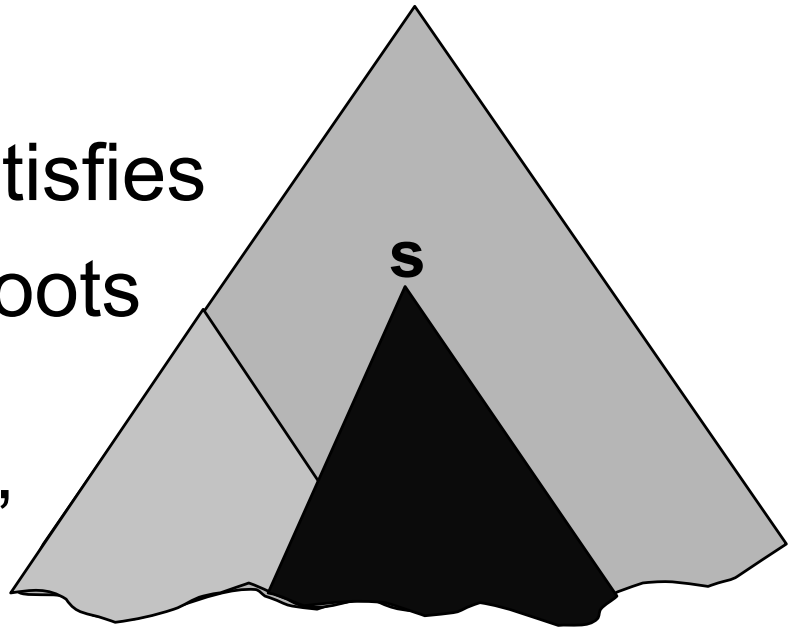
Recursive Definition

- In an infinite computation tree, any sub-tree is also an infinite computation tree

- Let Φ1, Φ2 be temporal formulae

  ➔ "Φ1 (Φ2)" means ---

  "For any state s that satisfies Φ1, the sub-tree that roots at this state s should satisfy the formula Φ2"



More examples…

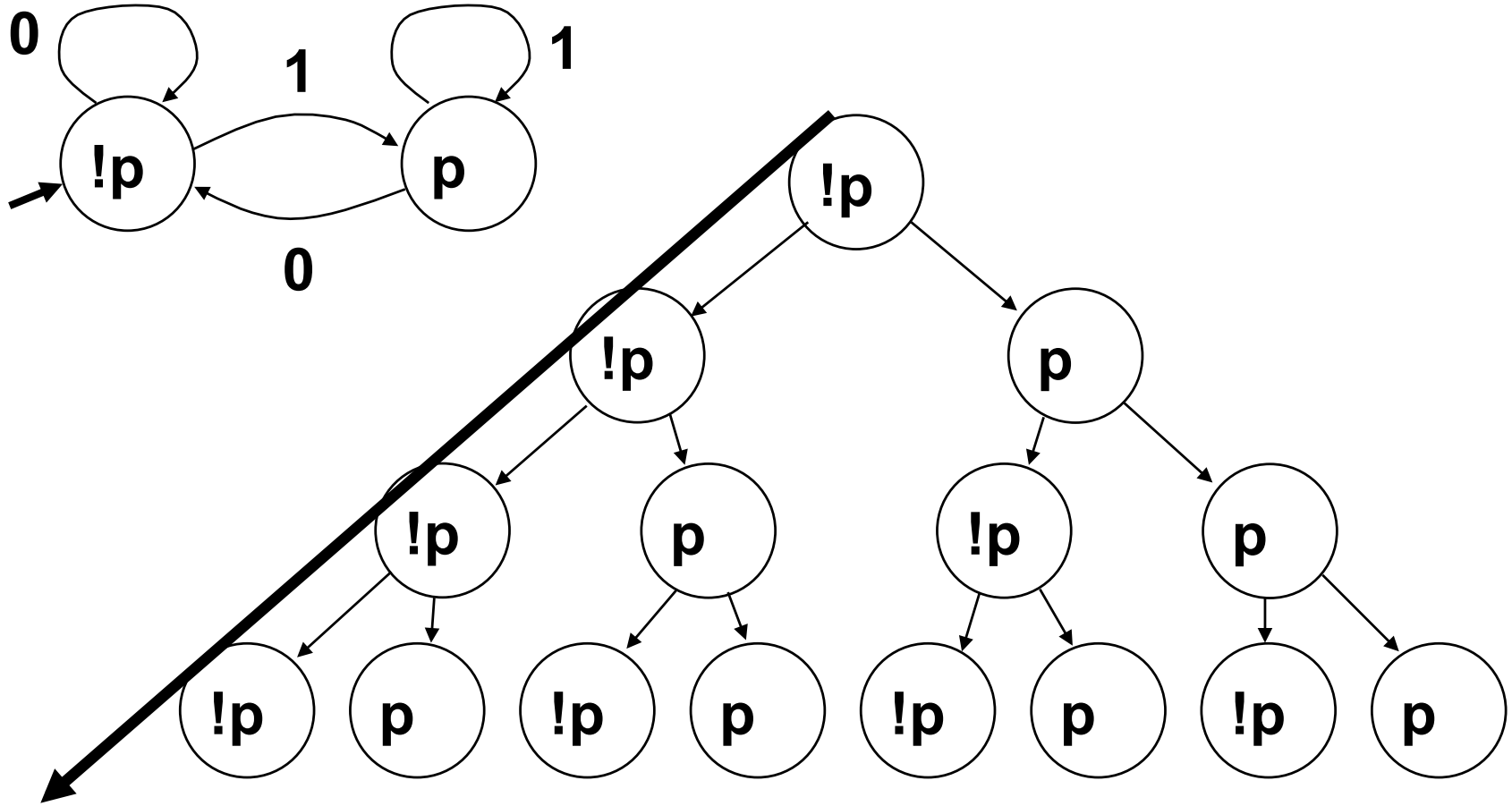
- AG(EF p)
  - For any state in the computation tree,
    its sub-tree should at least contain a state that satisfies p
  - e.g. AG(EF Restart) ≡ ¬deadlock
    - From any state it is possible to get to the Restart state

- AG(AF p)
  - For any state in the computation tree,
    its sub-tree should have a ”cut” that satisfies p
  - e.g. AG(AF DeviceEnabled)
    - From any state, any of its future computation path must see a DeviceEnabled
    - DeviceEnabled holds infinitely often on every computation path

Is AG(EF p) the same as AG(AF p)??

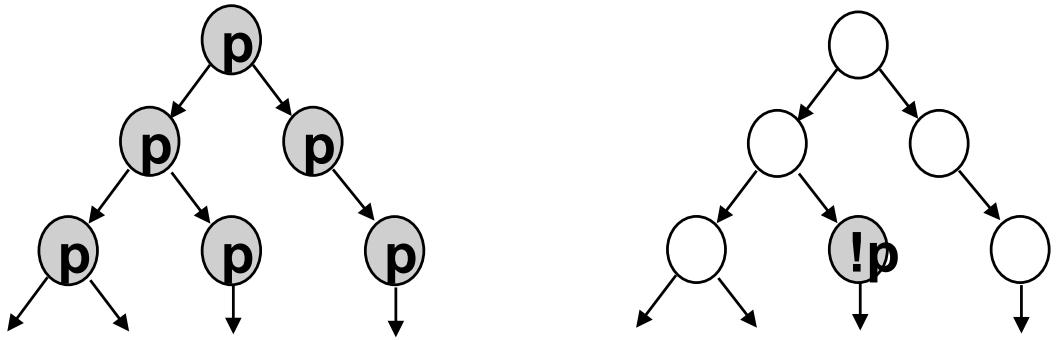
No, a counter-example is…

- Satisfies AG(EF p), but not AG(AF p)

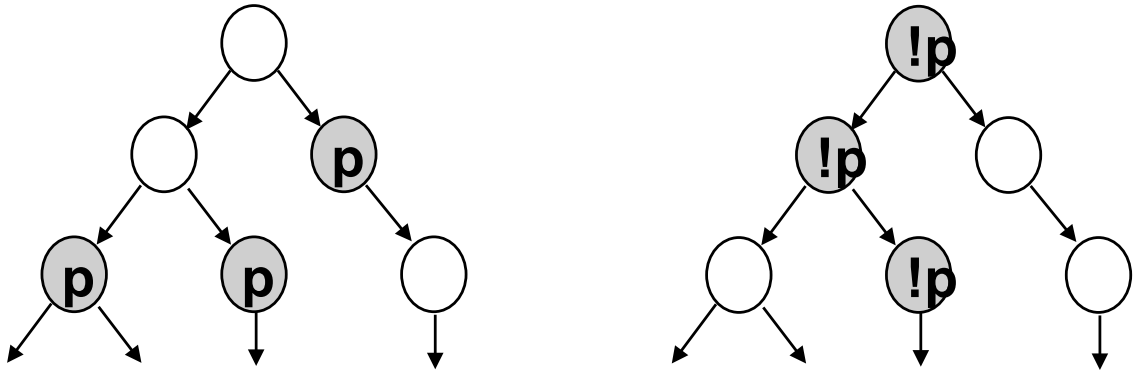


Equivalent Formulae

- AG(p) ¬ ¬

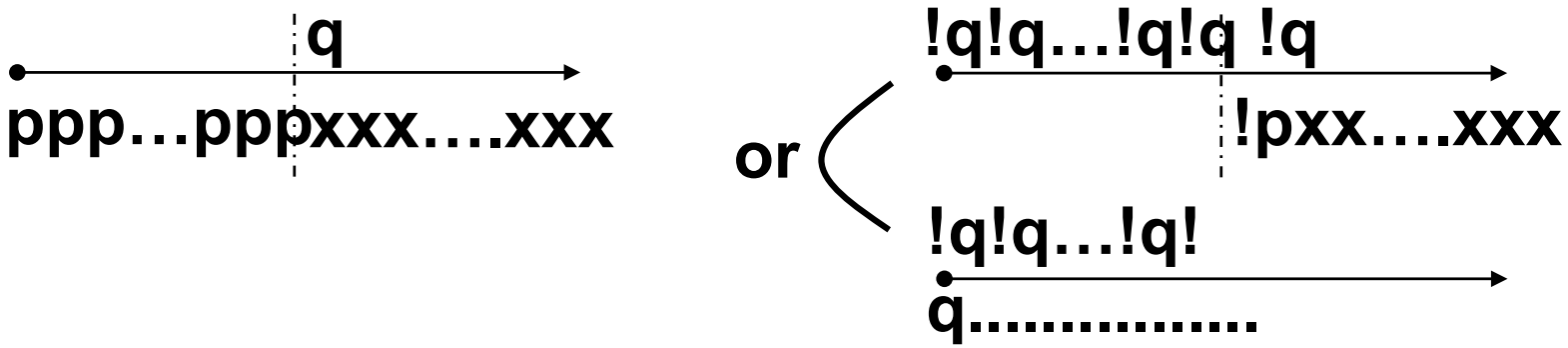


- AF(p) ¬ EG(¬p)



More Equivalent Formulae

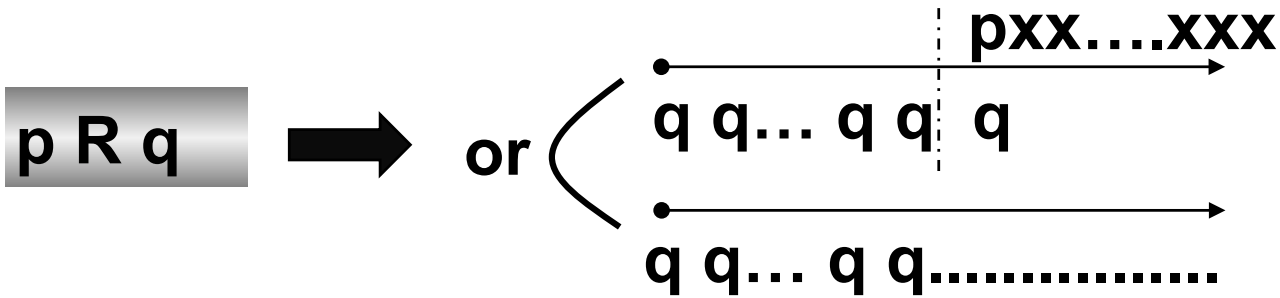
- AX p ¬ EX(¬ p)
- EF p E(true U p)
- A(p U q) ¬(E(¬q U (¬p ∧ ¬q)) ∨ EG ¬q)



- A(p R q) ¬ E(¬p U ¬q)
- E(p R q) ¬ A(¬p U ¬ q)

Atomic Operators for CTL*

- It can be shown that all CTL* formulae can be expressed by , , X, U, and E
  - $p \wedge q \equiv \neg(\neg p \vee \neg q)$
  - $p \, R \, q \equiv \neg(\neg p \, U \, \neg q)$
  - $F \, p \equiv True \, U \, p$
  - $G \, p \equiv \neg \, F \, \neg p$
  - $A \, p \equiv \neg \, E \, \neg p$



CTL (Computation Tree Logic)

- A restricted subset of CTL* that permits only branching-time operators. Each of the state quantifiers G, F, X and U must be immediately preceded by a path quantifier A or E

- 10 basic operators (path + state quantifiers)
  - AX, AF, AG, AU, AR
  - EX, EF, EG, EU, ER

- Formula

  [(CTL formula)] := <Path_quantifier> <state quantifier> [(CTL formula)]

  e.g. AG(p → EF q) ….. OK

  e.g. AGF p ….. Not OK, no A/E between GF

  e.g. AG(p → E q) ….. Not OK, missing state quantifier

LTL (Linear Temporal Logic)

- A restricted subset of CTL* that consists of the form “A f” where f is a path formula. The quantifiers in f must be state quantifiers G, F, X and U, followed by atomic proposition
  - ➔ f is the path formula that holds for ALL the paths in the computation tree

- Formula

  [(LTL formula)] := A <state quantifiers>... <atomic proposition>

  e.g. AGF(p) .... OK (p occurs infinitely often)

  e.g. AFG(p) .... OK

  e.g. AGAF(p) .... Not OK; CTL, not LTL

  e.g. EGF(p) .... Not OK; not start with A

CTL*, CTL, LTL, ... ^*^#(*^(*#^$%!^*#!@#

Many early model checking tools adopted these languages for property specification.

But many people thought that they were not easy to learn.

Therefore, in late 90's, several companies were extending HDLs or programming languages (e.g. C++) and making them into different "easier-to-learn" or say "programmable" *property specification languages*.

Standardization of Property Specification Language (PSL)

- Background
  - Different companies were using different property specification languages (Intel, IBM, Motorola,...)
  - Making verification tool support very difficult
- Accellera (http://www.accellera.org/home)
  - Formed in 2000, to drive development and use of standards required by systems, semiconductor and design tools companies
  - 4 major contenders for PSL
    - IBM sugar; Intel ForSpec; Verisity “e”; Motorola CBV
  - After long debates and voting, Accellera chose IBM sugar as the standard (http://www.eetimes.com/story/OEG20020425S0018)
  - However, Intel ForSpec was later combined with Synopsys Vera and then called OpenVera. It lastly became a part of the SystemVerilog standard

Learning New PSL

- Sugar PSL 1.1 LRM
  - 131 pages
  - 8 chapters

- SystemVerilog 3.1 LRM
  - 586 pages
  - 31 chapters

➔ Temporal language....
➔ Formal semantics.....
➔ Still a big burden for most of the (design) engineers!!

Think:
What's the goal of hardware verification?

- The goal: to fix *as many* bugs as possible
  - Bottom line: a do-or-die game (bug → recall)
  - Limitation: impossible to "know" how many bugs to fix
  - The fact: simulation is still the mainstream

- Formal method: to *prove* ONE property at a time
  - Bigger problem 1: *"Have I written a correct property?"*
  - Bigger problem 2: "*Have I written enough properties?"*
  - Bigger problem 3: *"What if proof aborts?"*
  - Dilemma:

    ➔ Complex property (but may be wrongly written) or
    Simple property (yet enough to detect bugs)?

Do we write a property to express the completeness of the specification,

or

we write a property to guide the model checker for bug hunting?

The Fact

- More than 90% of properties written for hardware verification are simply "*safety (invariance)* " properties
  - e.g. assert_never(readEn && writeEn);
  - e.g. assert_next(req, ack);

  ➔ Easier to write

  ➔ Higher proof completion percentage

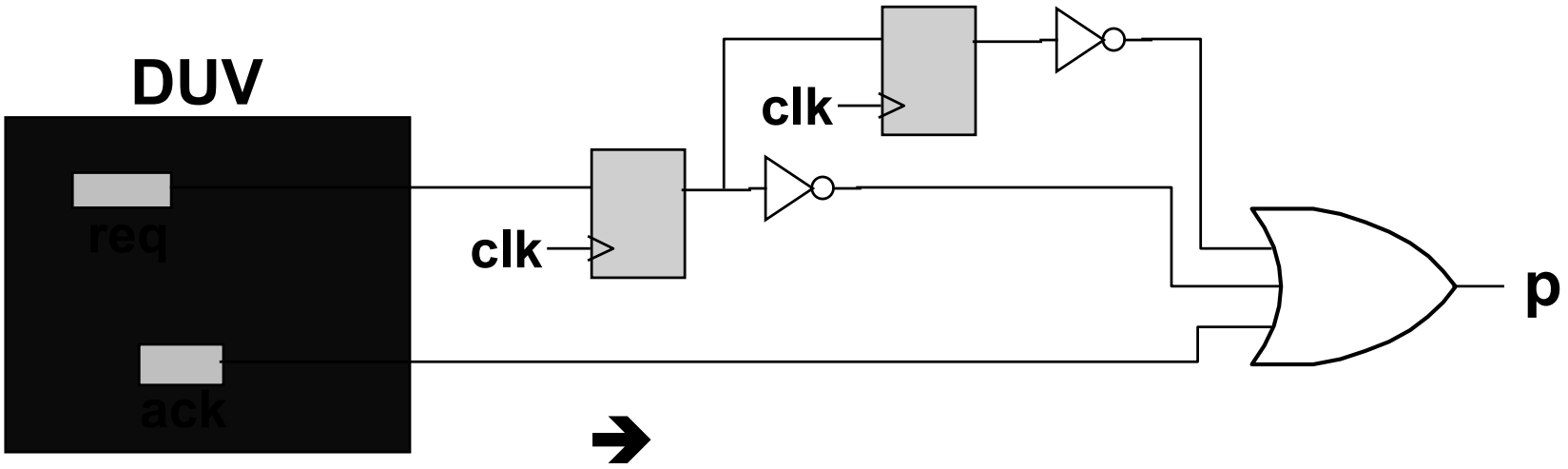
  ➔ Enough to detect bugs

  How to quickly prove all of them is the key issue

Safety Property (Invariance)

- Something GOOD should always hold;
  Something BAD should never happen
- Without lost of generality, an assertion property on a circuit can be transformed into an “assert_always (atomic_proposition)” property with some extra gates
  - e.g. assert_never(p) ≡ assert_always(¬p);
  - e.g. assert property

    (@(posedge clk) req |-> ##[1:2] ack)



Proving “assert_always(p)”

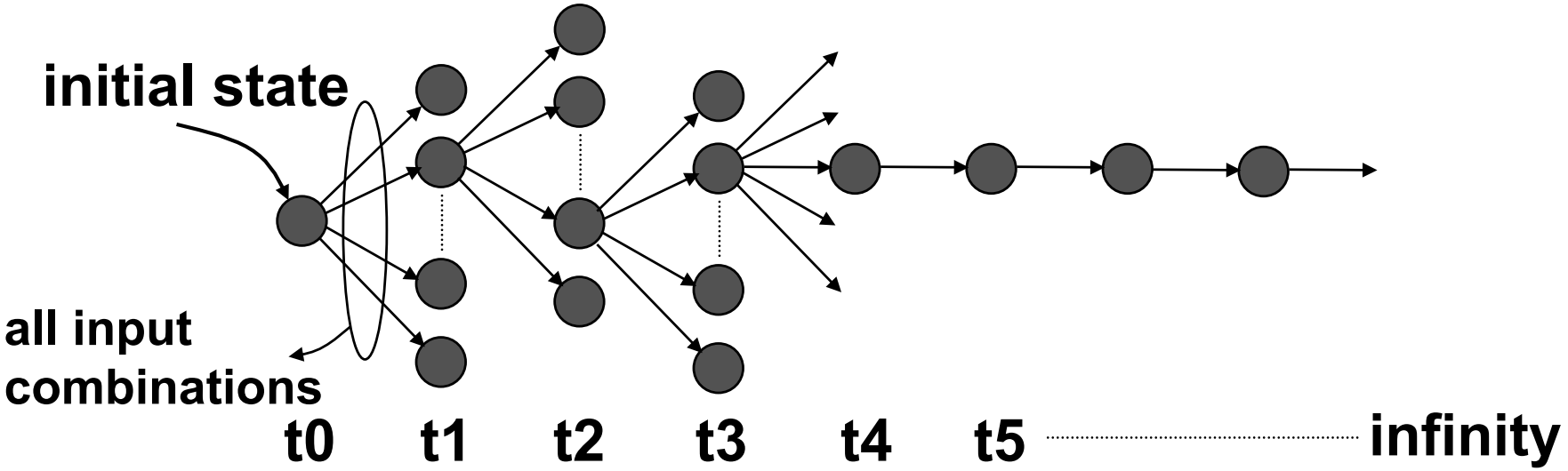
- In later slides, we will mostly focus on how to prove the property “assert_always(p)”, instead of proving a complex temporal logic formula

- assert_always(p) ≡ AG (p) ≡ ¬ EF (¬p)

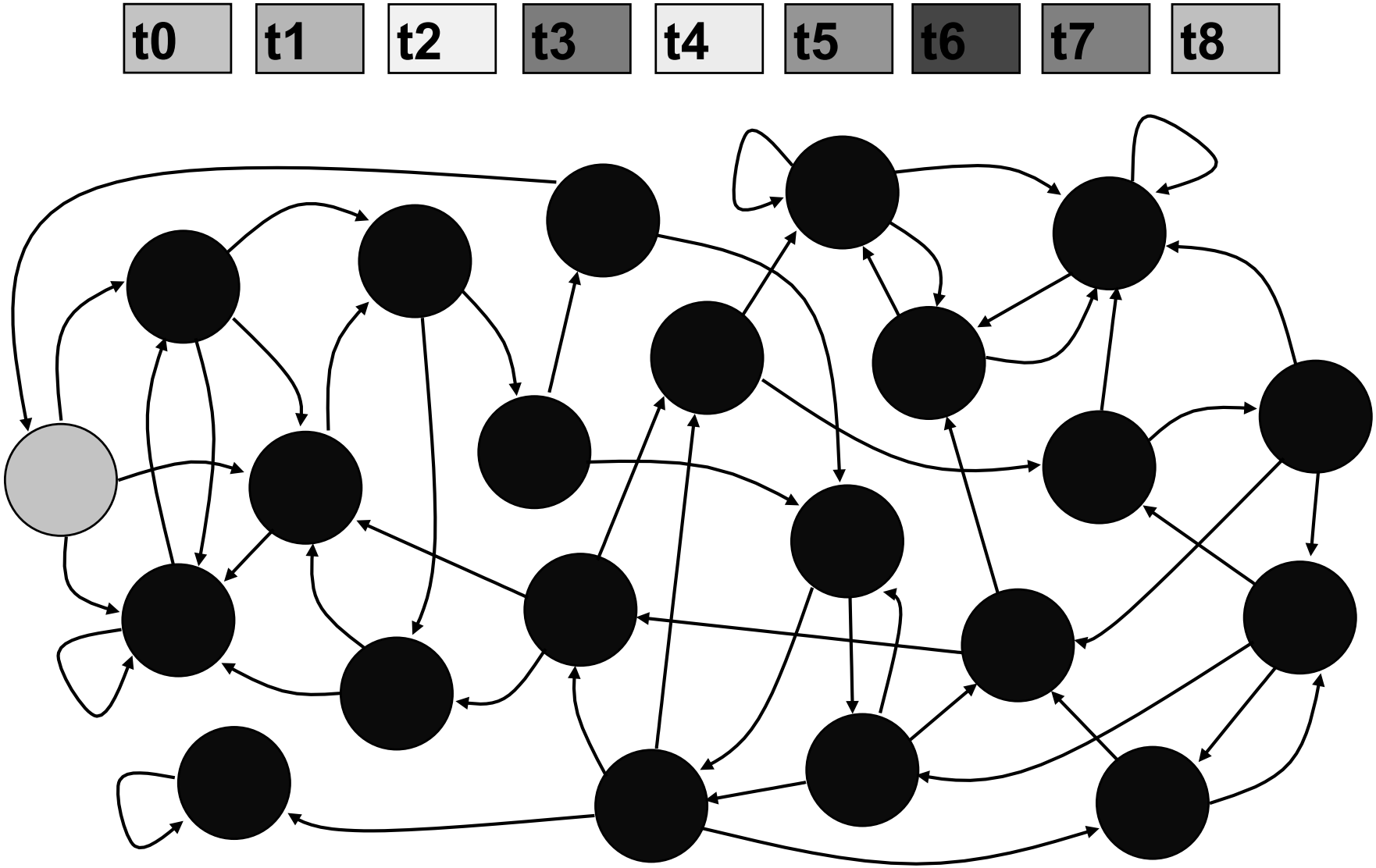
- Either
  “proving p is true for all states on the state transition graph” or
  “finding a trace that can disprove p”

“Mathematical Certainty” in Formal Verification

- Space exhaustiveness
  - Verify all input combinations of the system
- Time exhaustiveness
  - Verify system behavior from initial state to time infinity

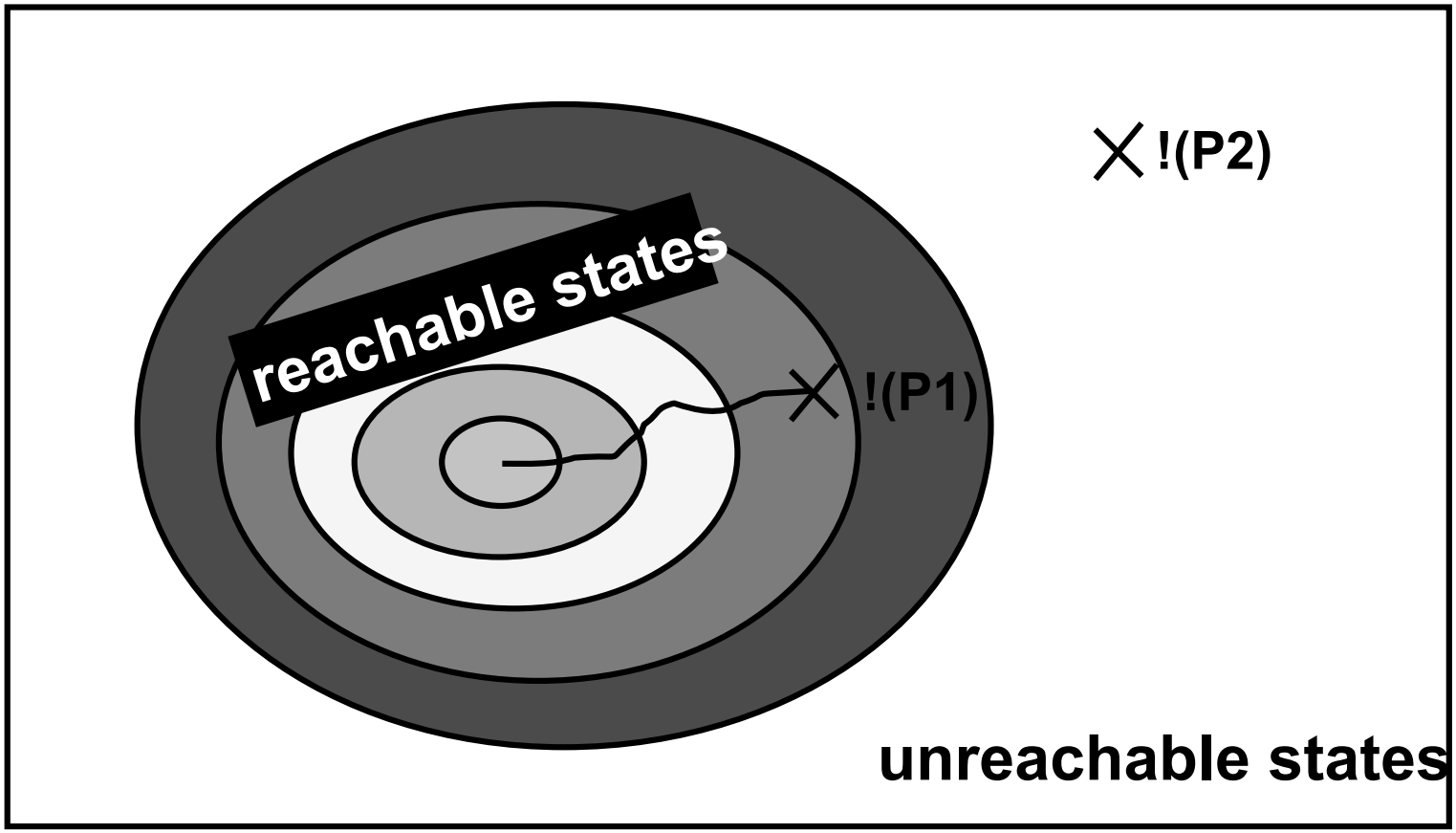


Set of Reachable States



Boolean State Space

Boolean space of n state variables (2^n)



Formal Verification Engines for AG

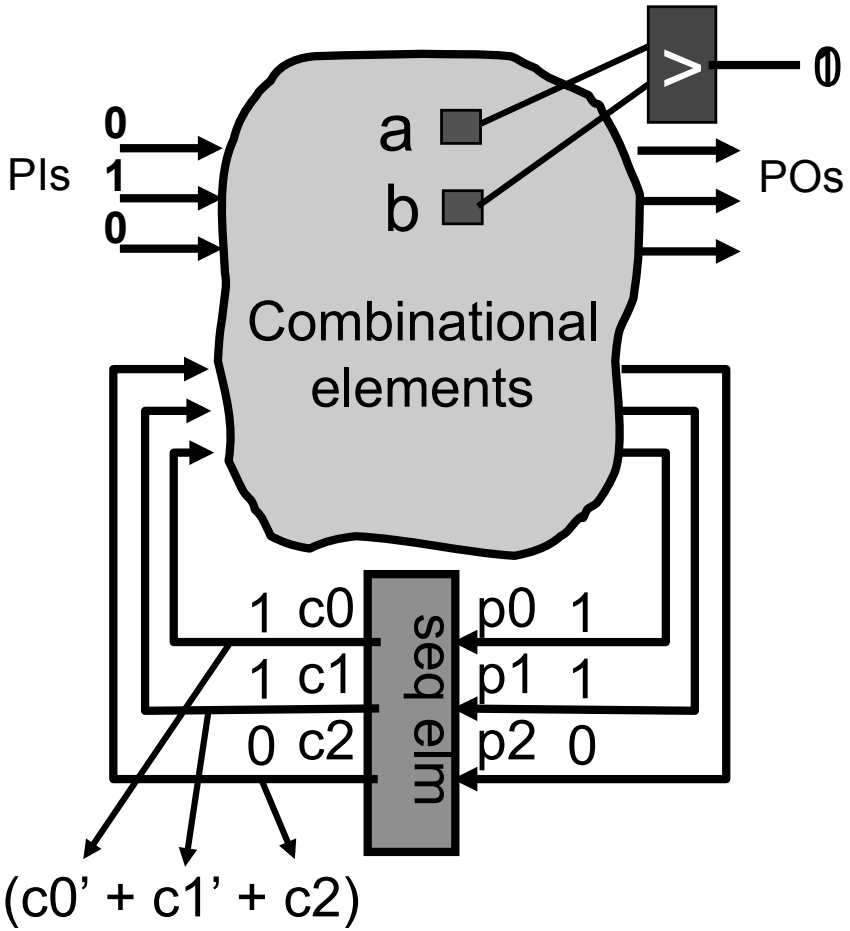
- Our later discussion of formal verification engines on AG property will either be
  1. How to compute the set of reachable states
  2. How to generate a trace to (¬p)
  3. How to prove that there is no trace to (¬p)

Think.... BDD vs. SAT

- (FYI) For Binary Decision Diagram (BDD), we compute the set of reachable states by iteratively applying transition relationship (TR) on current set of states (recorded as BDDs)

- However, SAT is a propositional constraint solver. How to “record” the set of states?

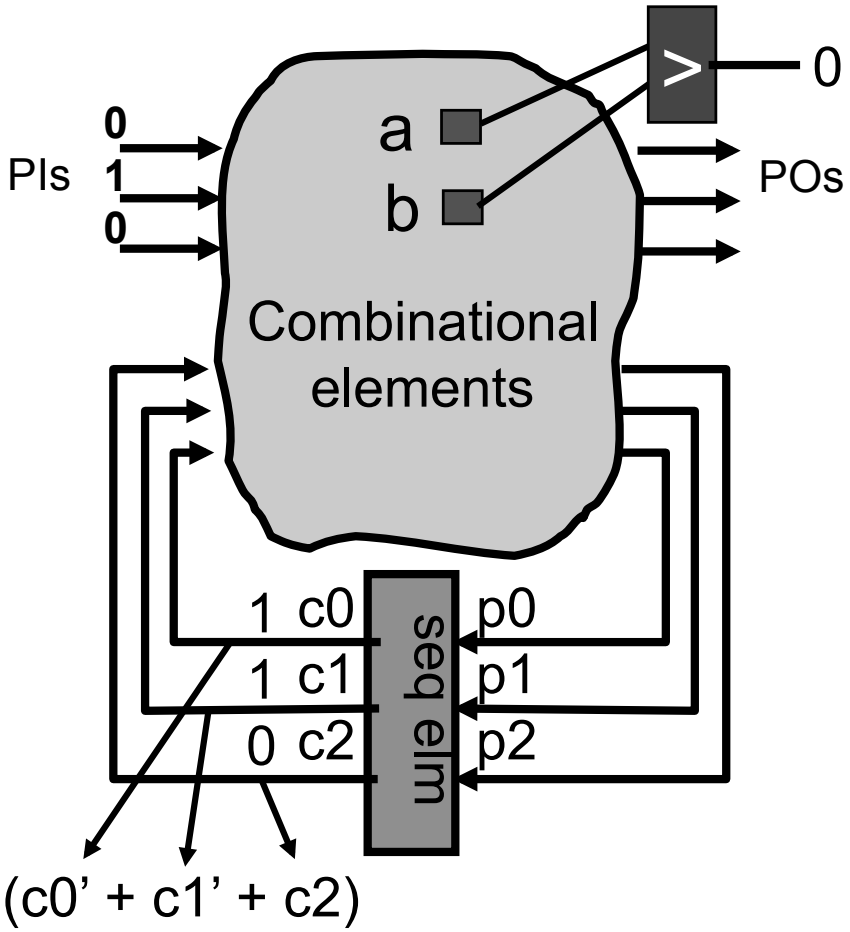
Using Blocking Clauses for Sequential SAT



Suppose we are solving the property "a > b"

1. registers (for !p)
   e.g. (c0, c1, c2) = (1,1, 0)
2. Add a "blocking clause"
   (c0' + c1' + c2) to the original CNF
   ➔ Won't get the same state again!!
3. Repeat 2 for another solution in the same timeframe..., or
4. Apply the solution "(c0, c1, c2) = (1, 1, 0)" to the previous state as "(p2, p1, p0) = (1, 1, 0)" and continue to the search in the previous timeframe (for p)

A closer look on the above algorithm...



-
  - Call SAT(p == 0)
  - Put the current state value (e.g. 110) to the previous state variables and call SAT(p0p1p2 = 110)
-

  among different SAT calls be shared?
  - Yes, by "assumpProve()"
  - Also an "incremental SAT" approach

Using Blocking Clauses for Sequential SAT

- The above process needs to continue until ---
  1. The initial state is reached
  2. No new state can be found (i.e. all in blocking clauses)

  ➔ BFS or DFS (in terms of timeframe traversal)?

- However, in the above approach, we are solving one state (cube) at a time.

  Comparing to BDD, which finds all the reachable state in one timeframe at once.

  SAT seems inefficient...

The bottomline is---

- SAT is a satisfiabilty solver (i.e. to answer "satisfiability"; to find ONE solution)

➔ It is NOT natural for it to enumerate ALL the solutions

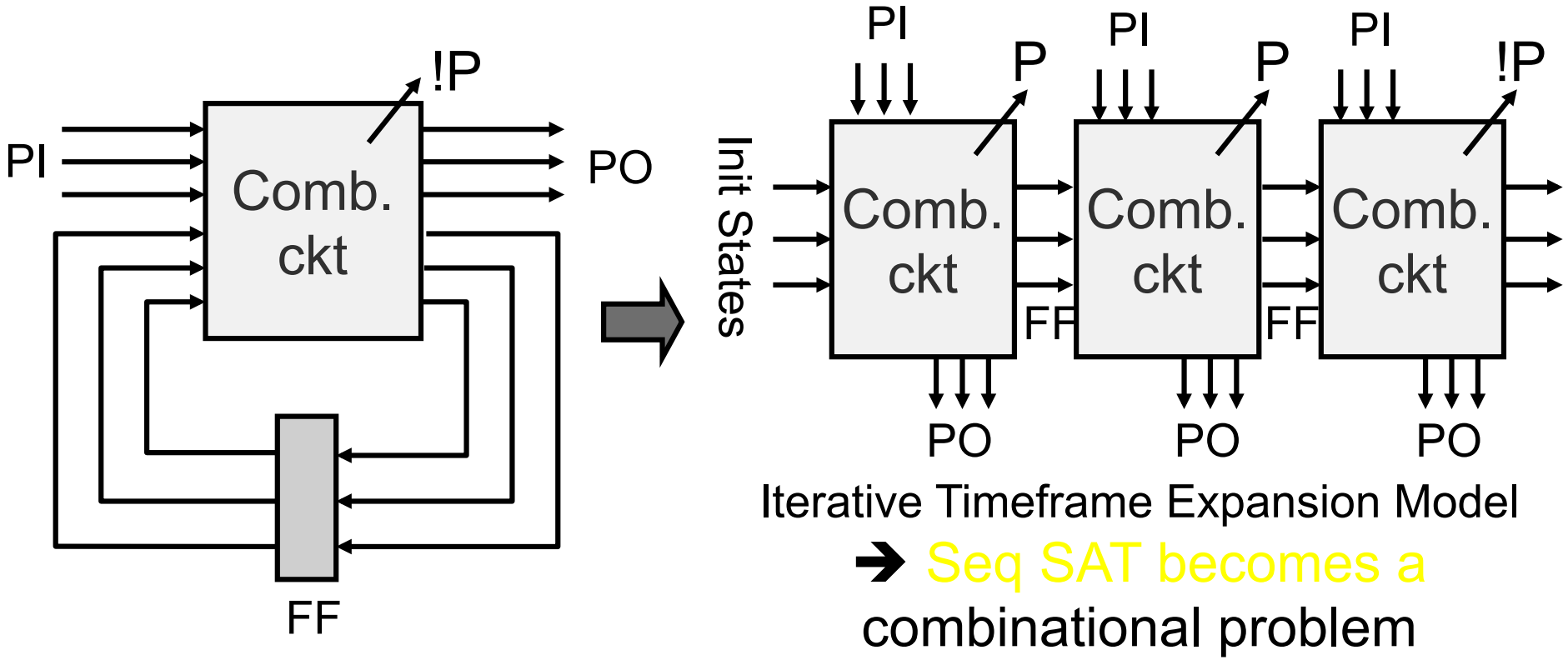
➔ It is NOT a structure for data storage (e.g. hash, BDD)

- SAT solves only propositional constraints

➔ No temporal logic

Iterative Timeframe Expansion Model

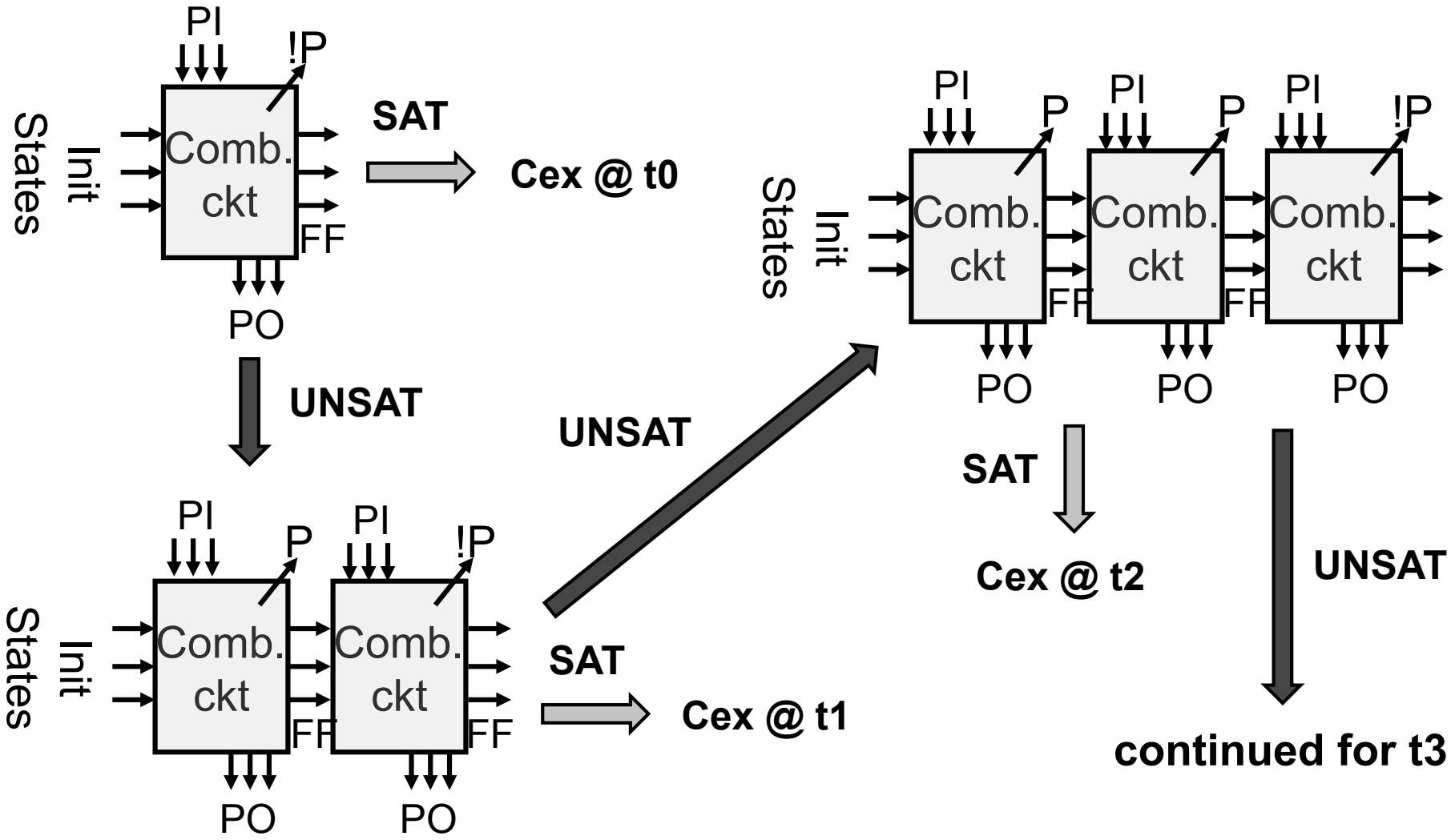
- There's another way of using SAT for sequential property checking



Iterative Timeframe Expansion Model

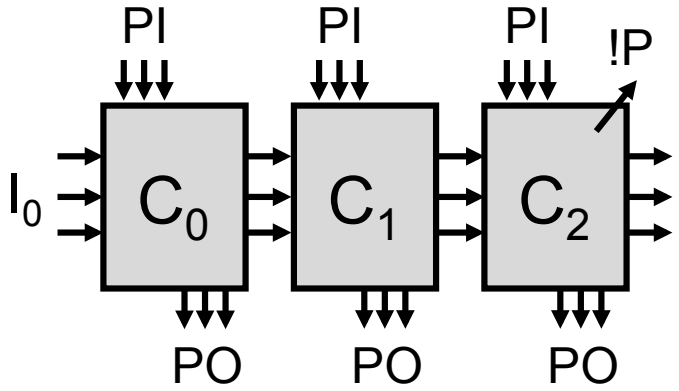
➔ Seq SAT becomes a combinational problem

In other words...



Bounded Model Checking (BMC) Algorithm

- Let ‘C’ be the set of constraints on the combinational circuit
  ➔ For an iterative model that unfolds the circuit for n times, let ‘C_i’ correspond to the i-th iteration of the circuit constraint ($0 <= i <= n - 1$)
- Let ‘I_0’ be the initial state value
- Let ‘P’ be the property to prove




```
BMC(P) {
    let k = 1;
 loop:
    if (SAT(I0∧C0∧...∧Ck-1∧!Pk-1))
       return "Find a counter-example @ (K-1)";
    k = k + 1;
    goto loop;
}
```

How far should we go?

- What's the limit of K?

  (How many iterations do we need before concluding the property is always true?)

  ➔ Impossible to know in the above BMC algorithm

  ➔ A loose upper bound is 2^N (N is the number of registers)

Application of BMC

- BMC is particularly useful when BDD encounters the memory explosion problem

- If the property is false, BMC can find a counter-example with the shortest length
- However, BMC cannot conclude that a property is true...

➔ It can only conclude that the property holds up to certain number of timeframes

➔ NOTE: BMC timeframe is different from the number of cycles in a simulation trace!!!

(BMC is best used in “bug-finding”)

Extension of BMC for Unbounded Proof

- BMC, combined with various techniques, can be extended to unbounded model checking

1. K-step Induction
2. Simple-path constraint
3. Counter-example-based abstraction
4. Proof-based abstraction
5. Image computation by SAT
6. Over-approximated image computation using interpolation

etc...

K-induction

SSS2000

- Induction:

$$\frac{P(s_0) \quad \forall i: P(s_i) \Rightarrow P(s_{i+1})}{\forall i: P(s_i)}$$

- k-step induction:

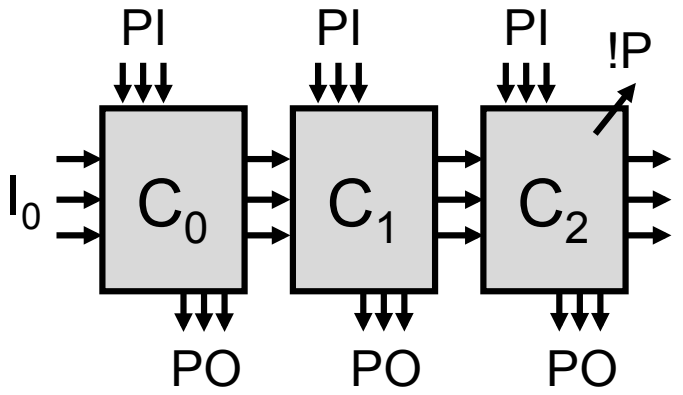
$$\frac{P(s_{0..k-1}) \quad \forall i: P(s_{i..i+k-1}) \Rightarrow P(s_{i+k})}{\forall i: P(s_i)}$$

* Some of the following slides in this lecture note are adopted and modified from Dr. Ken McMillan's CAV03 tutorial

K-induction with a SAT solver

- Let:

  $U_k = C_0 \wedge C_1 \wedge \ldots \wedge C_k$



- Two formulas to check:
  - Base case:

    $$I_0 \wedge U_{k-1} \Rightarrow P_0 \ldots P_{k-1}$$

  - Induction step:

    $$U_k \wedge P_0 \ldots P_{k-1} \Rightarrow P_k$$

- If both are valid, then P always holds.
- If not, increase k and try again.

Induction SAT

```
for (k = 0 to infinity)
    S = U_k ∧ F_k  // F_k = P_0 ∧... ∧ P_{k-1} ∧ !P_k
    T = I_0  ∧ S
    // induciton step
    if (SAT(S) == false)
       return NO_SOLUTION;  // i.e. P is true
    // normal proof: base case for next k
    if (SAT(T) == true)
       return HAS_SOLUTION; // i.e. CEX is found
    if (effort exceeds limit)
       return ABORT;
endfor
```

Induction SAT

- In other words, let

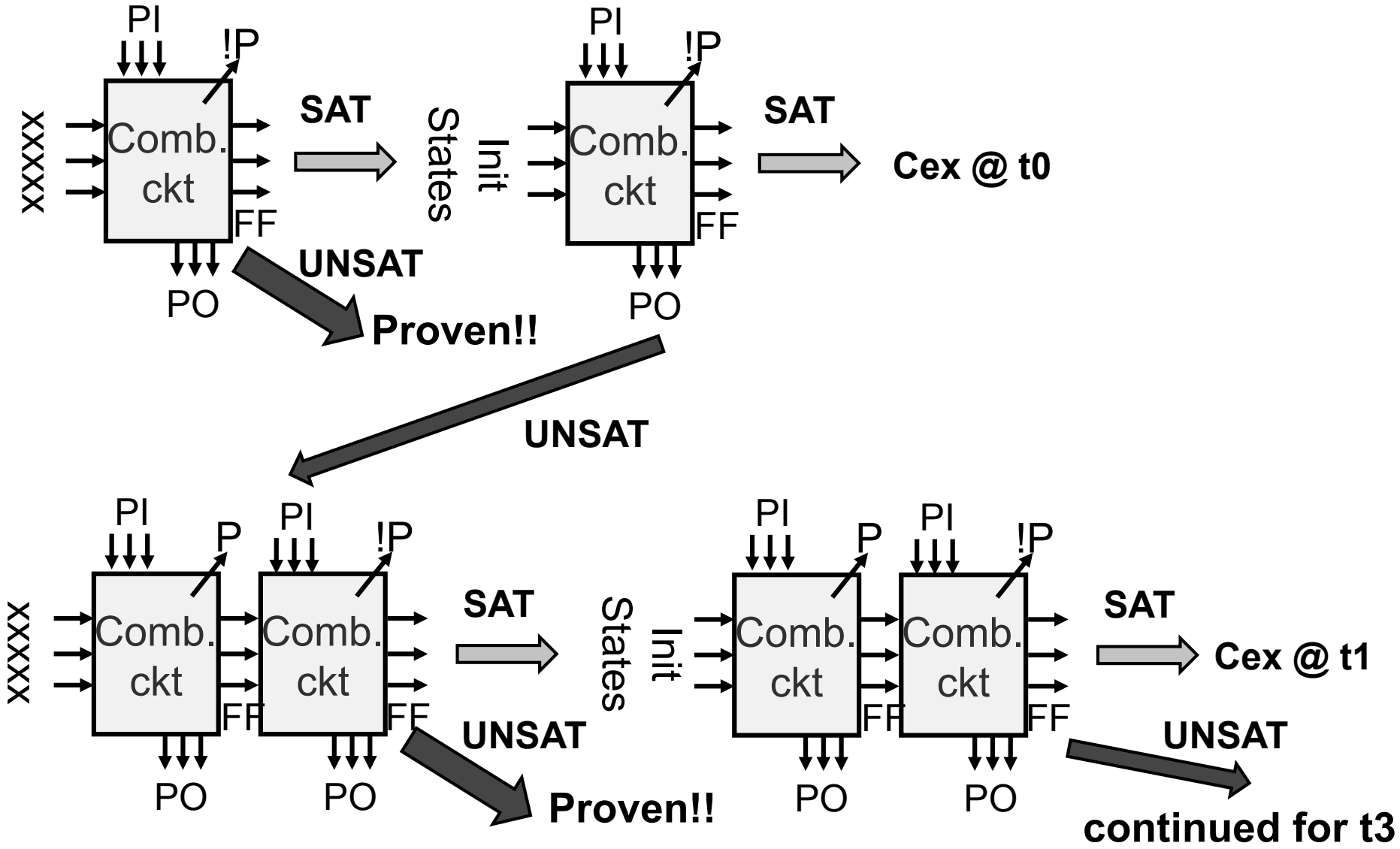
$U_k \wedge F_k$ // induction step

$I_0 \wedge S$ // BMC step

- Induction SAT...

```
if (S(0) == UNSAT)   return UNSAT;
if (T(0) == SAT)     return SAT;
if (S(1) == UNSAT)   return UNSAT;
if (T(1) == SAT)     return SAT;
...
```

In other words...



Does “Induction SAT” guarantee convergence?

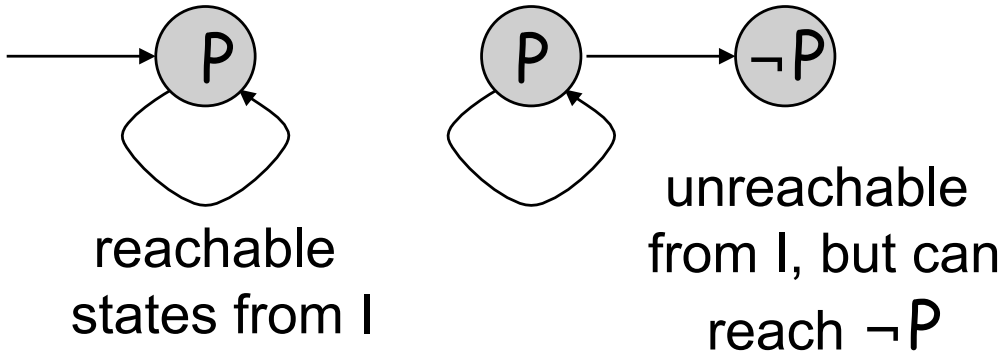
i.e. Will we either (given enough time/memory)

1. conclude no solution in induction step

or 2. find a counter-example in normal proof with a finite number k ???

Simple path assumption

- Unfortunately, k-induction is not complete.
  - Some properties are not k-inductive for any k.



- Simple path restriction:
  - There is a path to ¬P iff there is a *simple* path to ¬P (path with no repeated states).

Induction over simple paths

- Let simple($s_{0..k}$) be defined as:
  - $\forall i,j \text{ in } 0..k : (i \neq j) \Rightarrow s_i \neq s_j$
- k-induction over simple paths:

$$\frac{P(s_{0..k-1}) \qquad \forall i: \text{simple}(s_{0..k}) \wedge P(s_{i..i+k-1}) \Rightarrow P(s_{i+k})}{\forall i: P(s_i)}$$

Must hold for k large enough, since a simple path cannot be unboundedly long. Length of longest simple path is called *recurrence diameter*.

...with a SAT solver

- For simple path restriction, let:

  $S_k = \forall t=0..k, t'=t+1..k: \ \neg\ (\forall v \text{ in } V : v_t = v_{t'})$

  (where V is the set of state variables).
- Two formulas to check:
  - Base case:

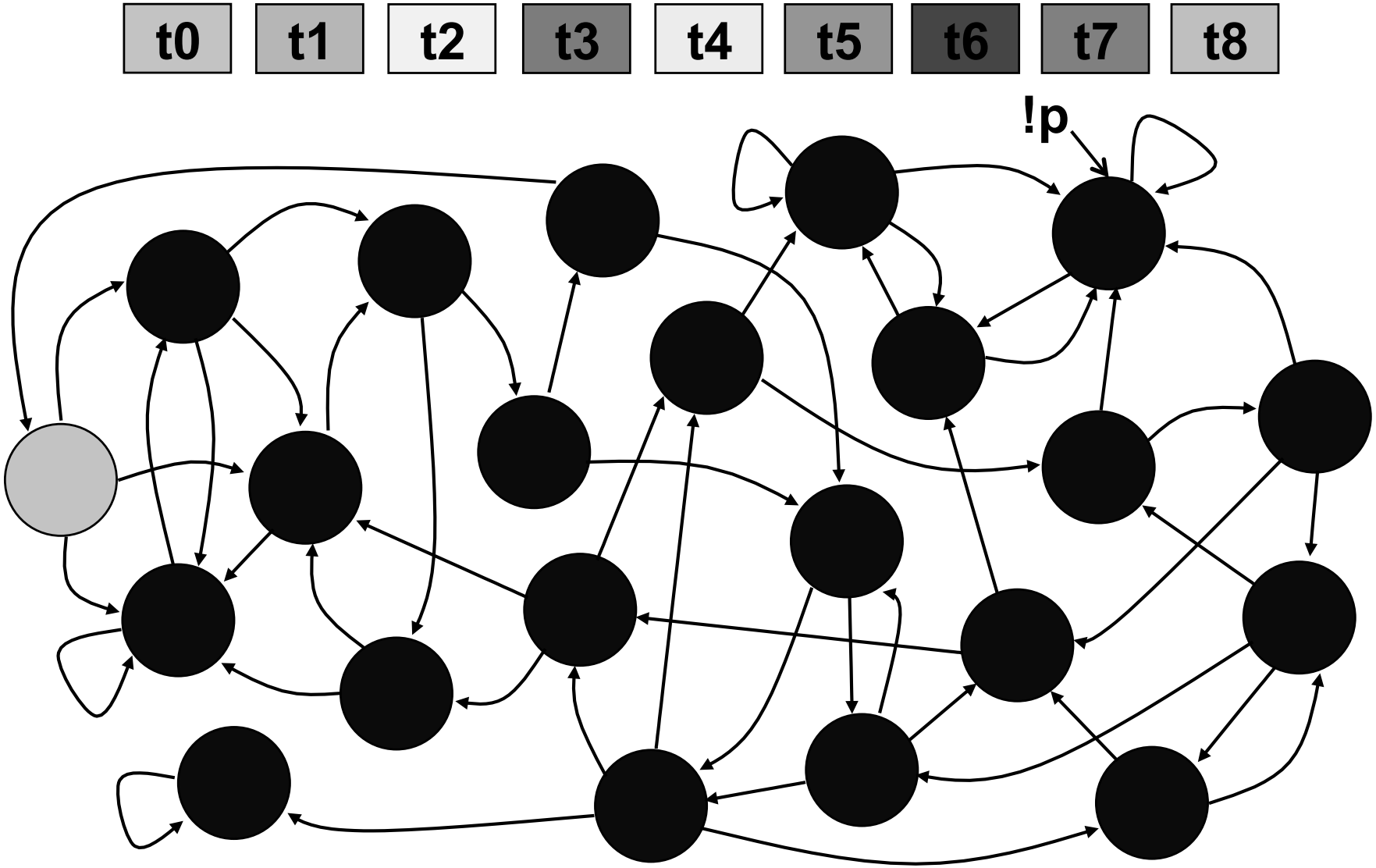
    $$I_0 \ \wedge\ U_{k-1} \ \Rightarrow\ P_0 ... P_{k-1}$$

  - Induction step:

    $$S_k \ \wedge\ U_k \ \wedge\ P_0 ... P_{k-1} \ \Rightarrow\ P_k$$
- If both are valid, then P always holds.
- If not, increase k and try again.

Is the recurrence diameter the same as the diameter (the distance from initial state to any state, i.e. depth of fixed point)??

Recurrence Diameter vs. Diameter



Termination

- Termination condition:

  k is the length of the longest simple path of the form

  $$P^* \neg P$$

- This can be exponentially longer than the diameter.
  - example:
    - loadable mod 2^N counter where P is (count $\neq 2^N-1$)
    - diameter = 1
    - longest simple path = 2^N
- Nice special cases:
  - P is a tautology (k=0)
  - P is inductive invariant (k=1)

Limitations of simple path constraint

- Although simple path constraint can make the induction-based SAT a complete algorithm for sequential proof, it has the limitation in reality that the circuitry for the simple path constraint can grow too big ($O(n^2)$)
  - Not really applicable in real cases
- What if we limit the simple path constraint to “no repeat states within k timeframes”, where k is a small enough number?
  - Is the algorithm still complete?

What can/should be covered in this topic?

- SAT-based logic synthesis
  - Redundancy addition and removal
  - ~~Functional dependency~~
  - ~~SAT-based re-synthesis techniques~~
  - ~~Engineering Change Order (ECO)~~
- From SAT to optimization problems
  - Pseudo Boolean satisfiability/optimization problems
- ~~General SAT-based model checking algorithms~~
- ~~Quantified Boolean Formula (QBF)~~
- ~~Bit-vector/Arithmetic solver~~
- ~~Satisfiability Modulo Theories (SMT)~~

Applications of Logic Implication

- We have learned that logic implication can be very efficient for both CNF and circuit-based SAT solvers
- Logic implication is actually also a powerful approach in exploring signal correlations in the circuit
- Any application?

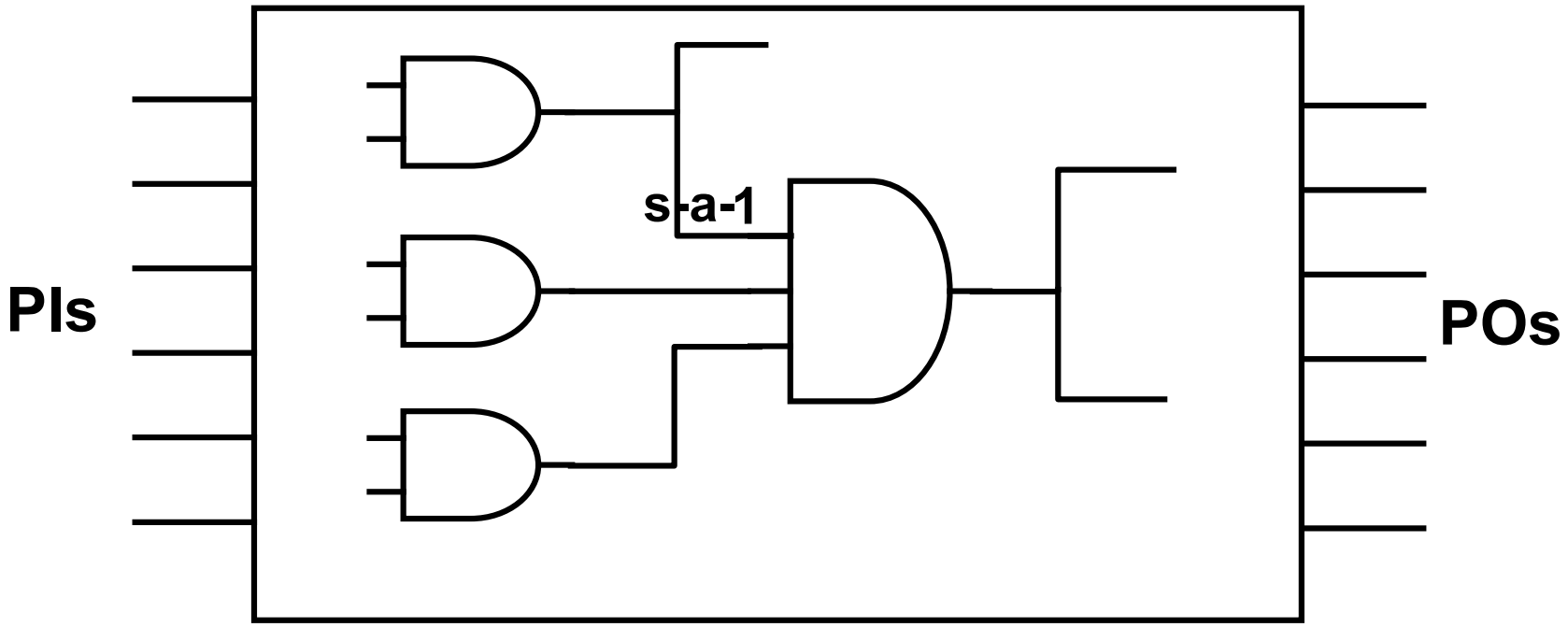
  Redundancy addition and removal

Redundancy Addition and Removal (RAR)

- Redundancy to a circuit
  - When removing or adding some signal/gate to a circuit, the circuit functionality remains unchanged
- Motivations
  - Removing redundancy in a circuit can gradually lead to smaller area, timing, power, etc
  - When (*deliberately*) adding some redundancy to a circuit, we may cause ***other*** part of the circuit become redundant
    - Incremental circuit restructuring (rewiring)
    - Can be used for incremental optimization (e.g. timing, area, etc)

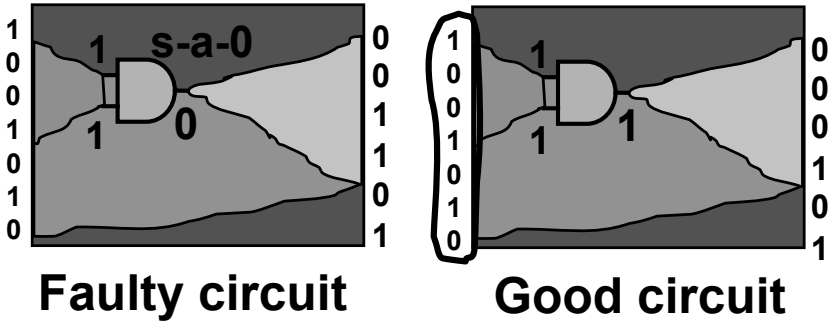
Redundancy in a Combinational Circuit

- Redundancy in a combinational circuit
  = Single stuck-at fault untestable



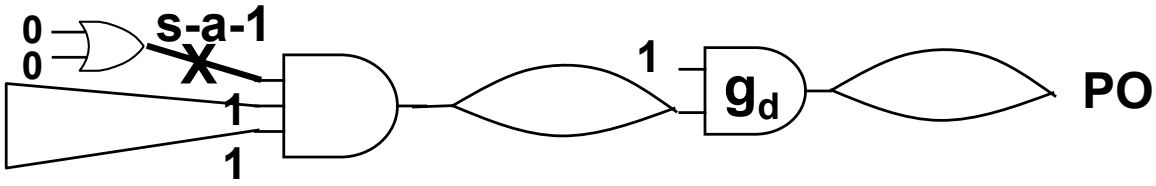
Background: Single Stuck-at Fault Untestable

- Sufficient untestable condition
  - ➔ The mandatory assignment (MA) of the stuck-at fault has conflict



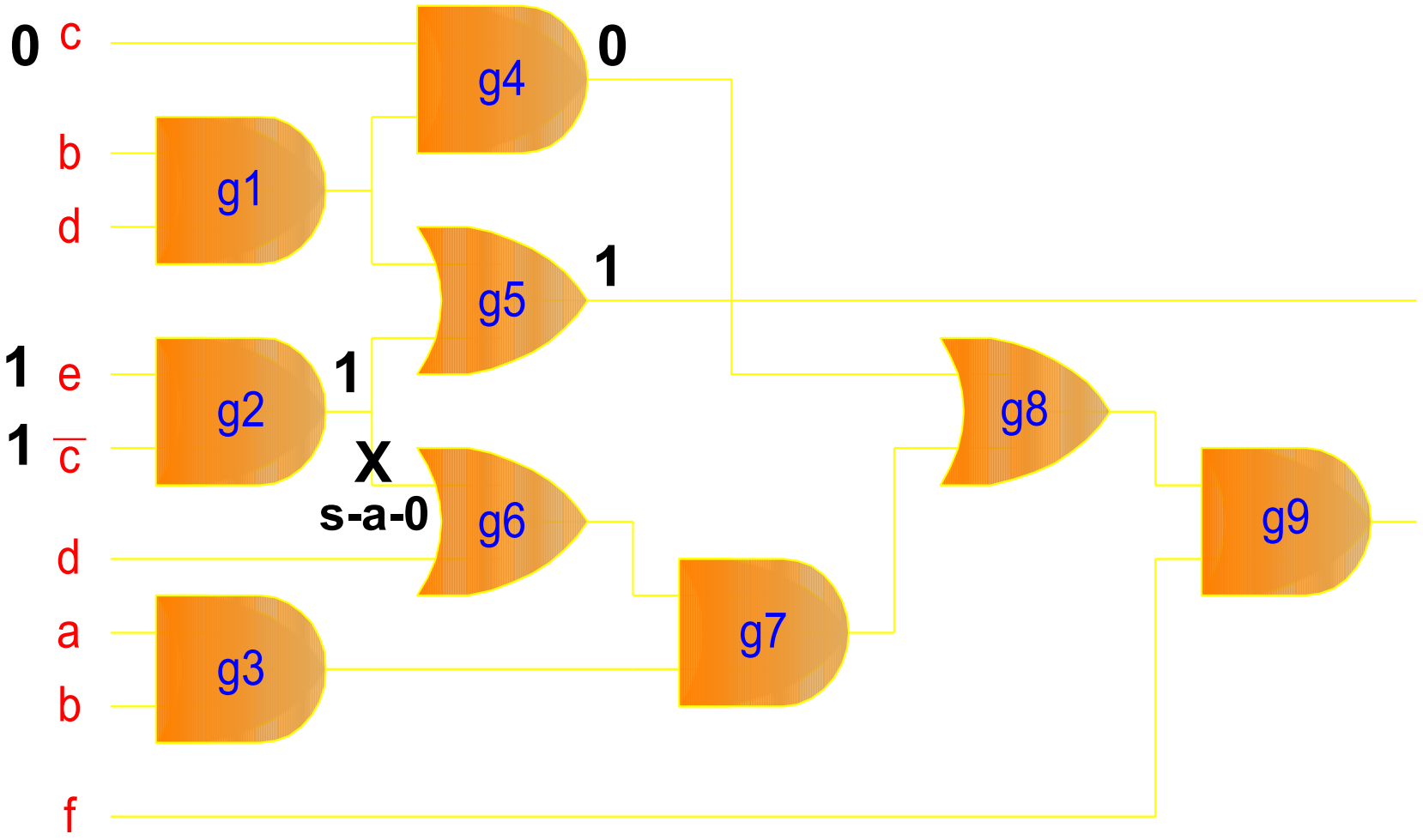
- Mandatory assignment of a fault
  - Denoted as MA(w) or MA(g), where ‘w’ or ‘g’ is the fault location (wire or gate)
  - Implications of
    1. Fault sensitization @ fault site
    2. Fault propagation @ the side inputs of the dominators

- Dominators of a fault
  - The gates where all the paths from the fault site to the POs must intersect



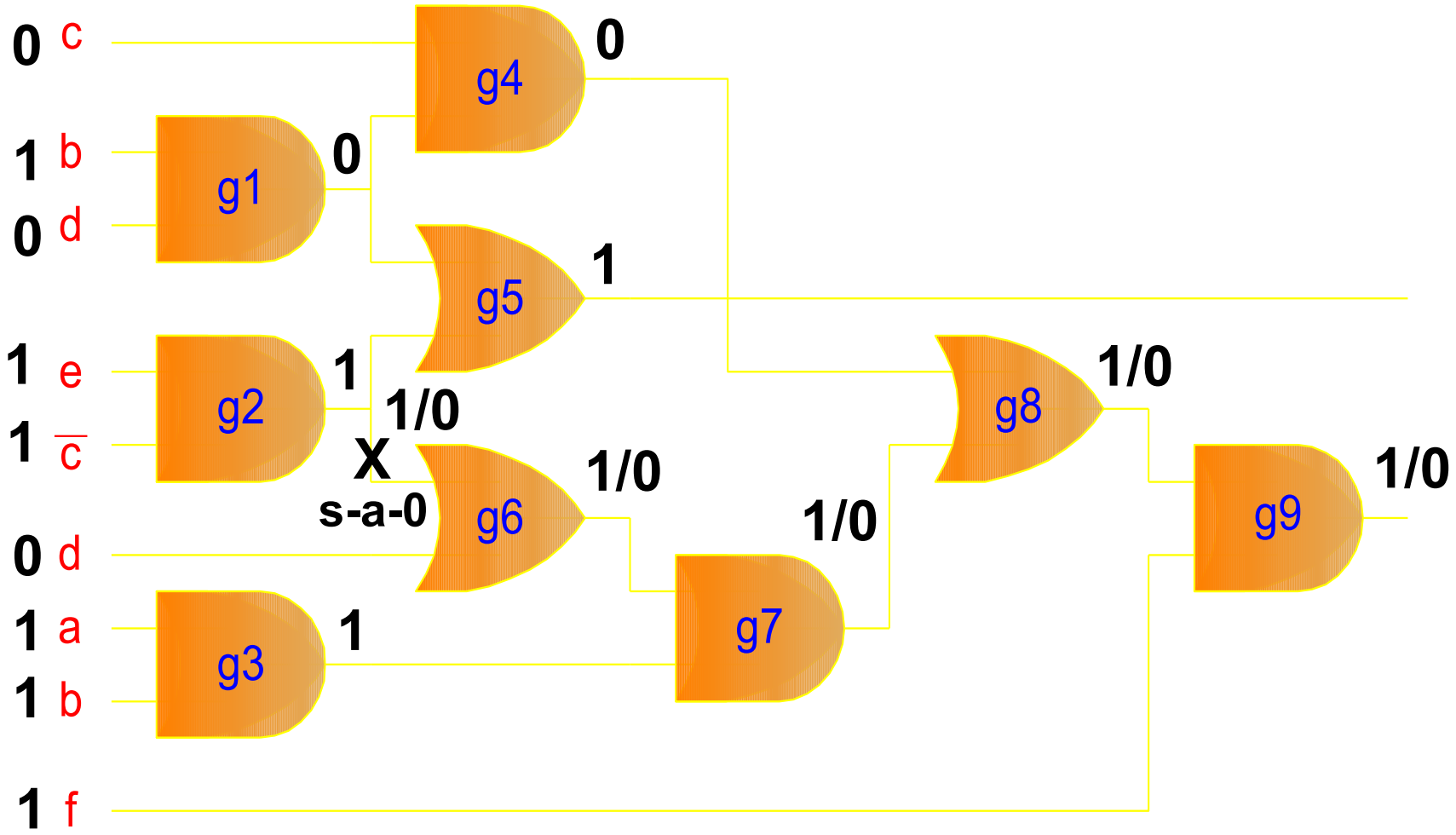
Mandatory Assignment Example

(1) Fault sensitization: g2 = 1



Mandatory Assignment Example

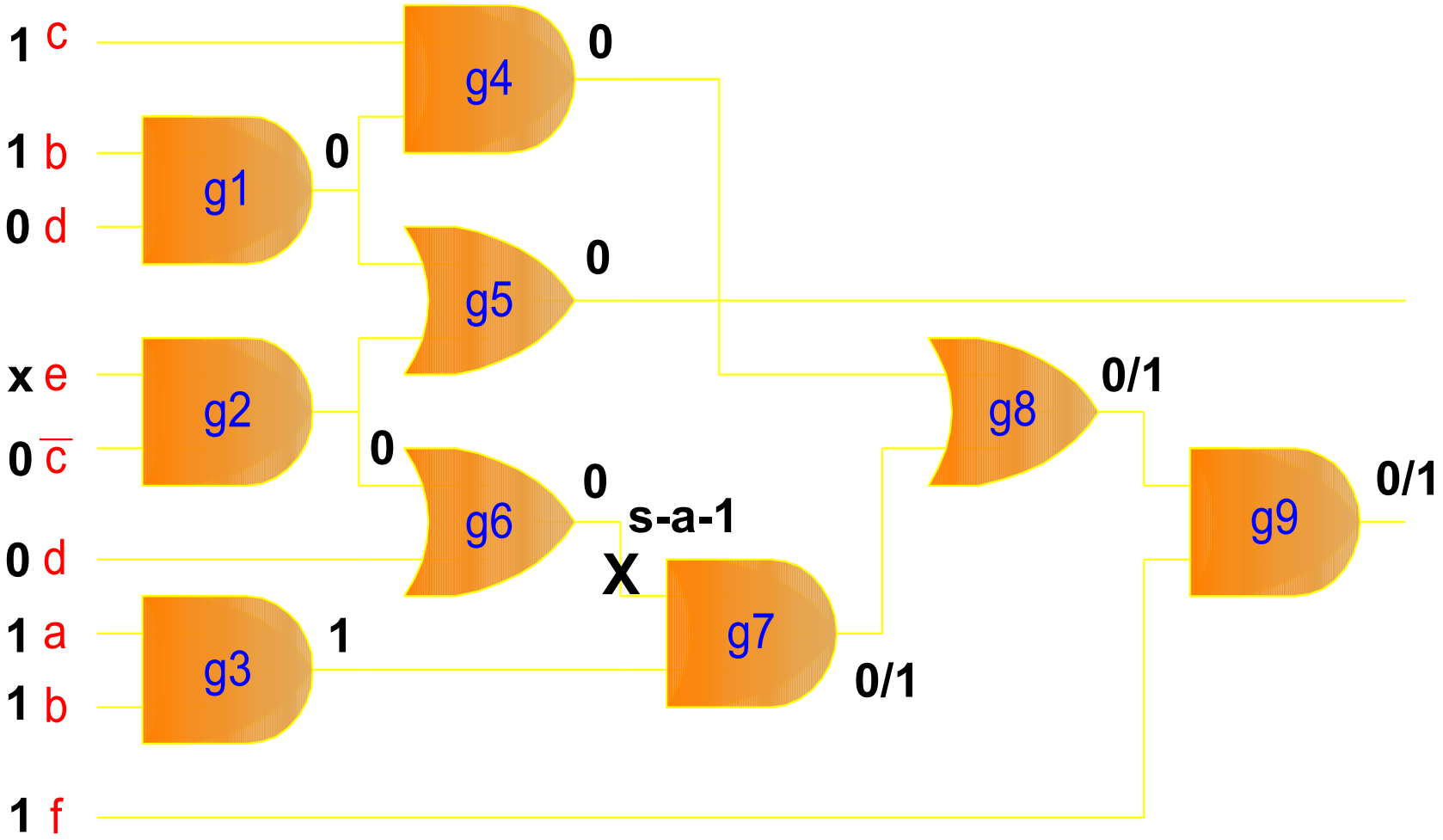
(2) Fault propagation: d = 0, g3 = 1, g4 = 0, f = 1



1. How do we know a wire in a combinational circuit is redundant?
   - ➔ Its corresponding stuck-at fault is untestable
     (s-a-1 for AND inputs; s-a-0 for OR inputs), or
   - ➔ MA of the fault has conflict

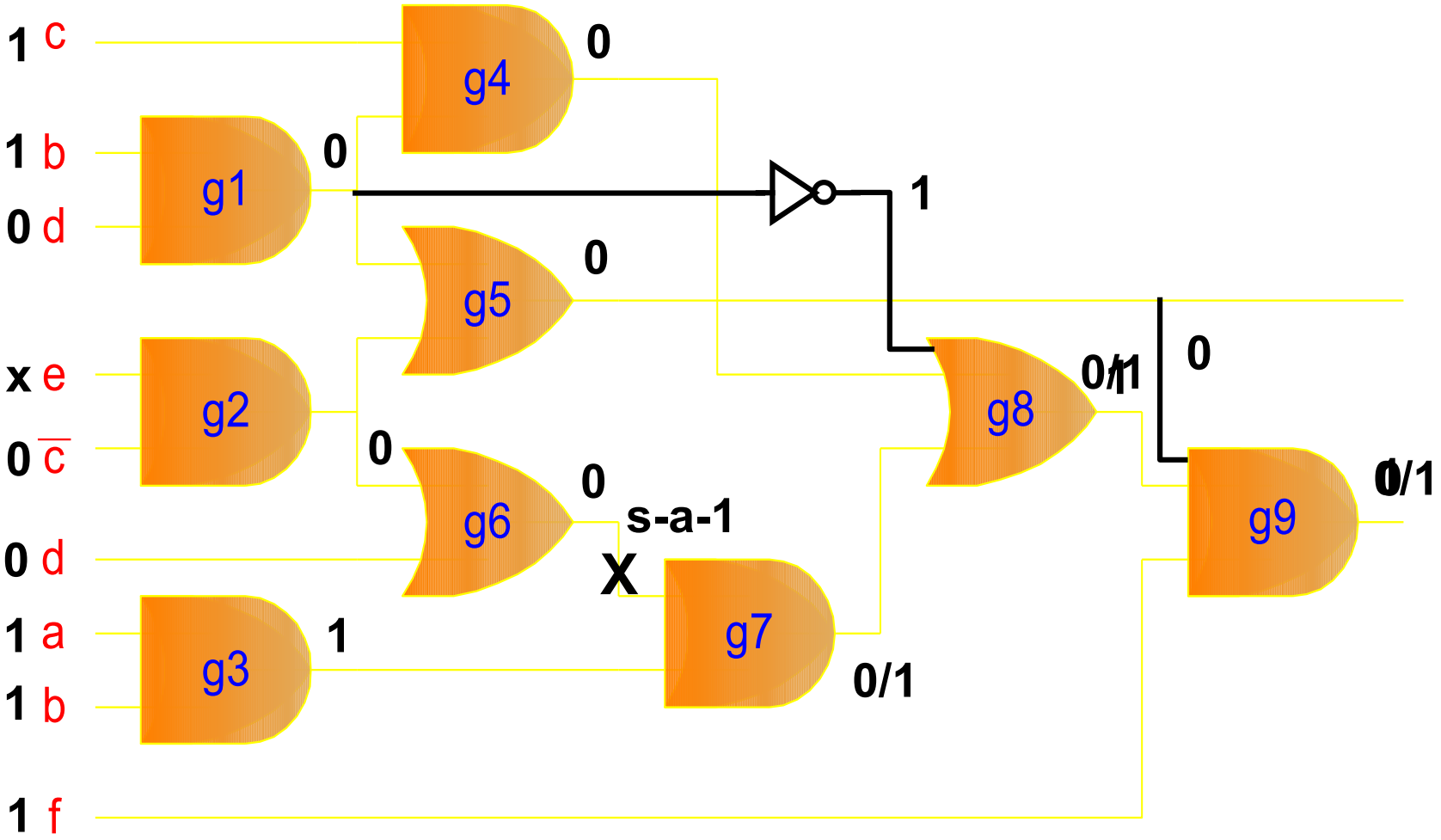
2. If a wire is NOT redundant, can we add an extra wire to make this wire redundant?
   - ➔ Yes, but the extra wire itself must be redundant
   - ➔ Add a redundant wire to make the originally irredundant wire become redundant

Target: remove g6



g6 is testable and thus NOT redundant

How to add an extra wire to make the s-a-1 fault @ g6 untestable?



Adding a wire (or with inverter) from any implied gate to a dominator

But remember,

the added wire must be redundant!!

RAMBO: Redundancy Addition and removal for Multi-level Boolean Optimization

[Cheng et.al. TCAD 1995]

1. Given a target wire, perform its mandatory assignments (MA) for its corresponding s-a fault

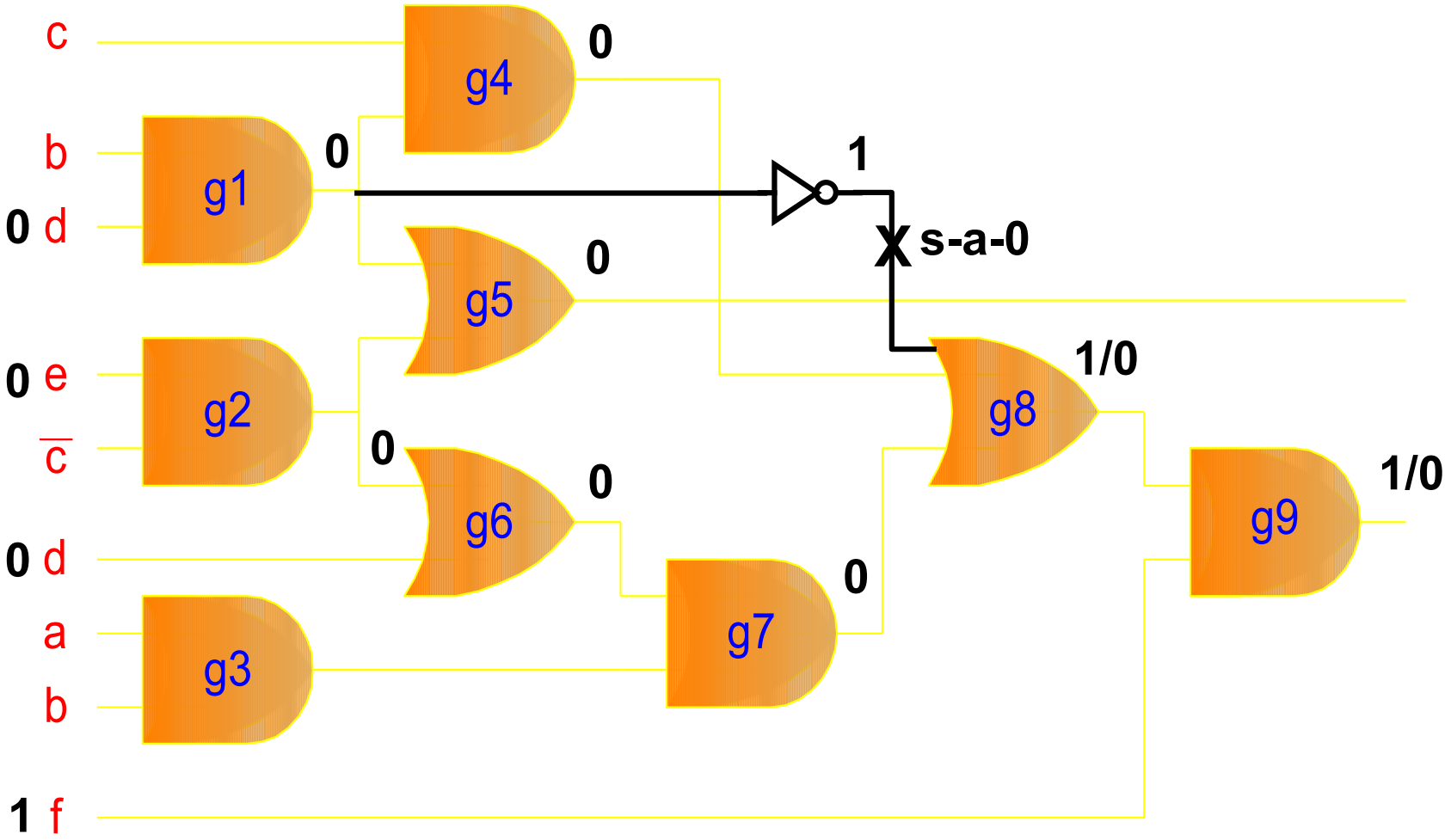
2. For each gate g_m in the set of MA,

   For each dominator g_d, test the fault on the added wire (g_m → g_d)

   a. If value(g_m) = 0 and g_d is an AND → direct connection
   b. If value(g_m) = 1 and g_d is an AND → add an inverter
   c. If value(g_m) = 0 and g_d is an OR → add an inverter
   d. If value(g_m) = 1 and g_d is an OR → direct connection

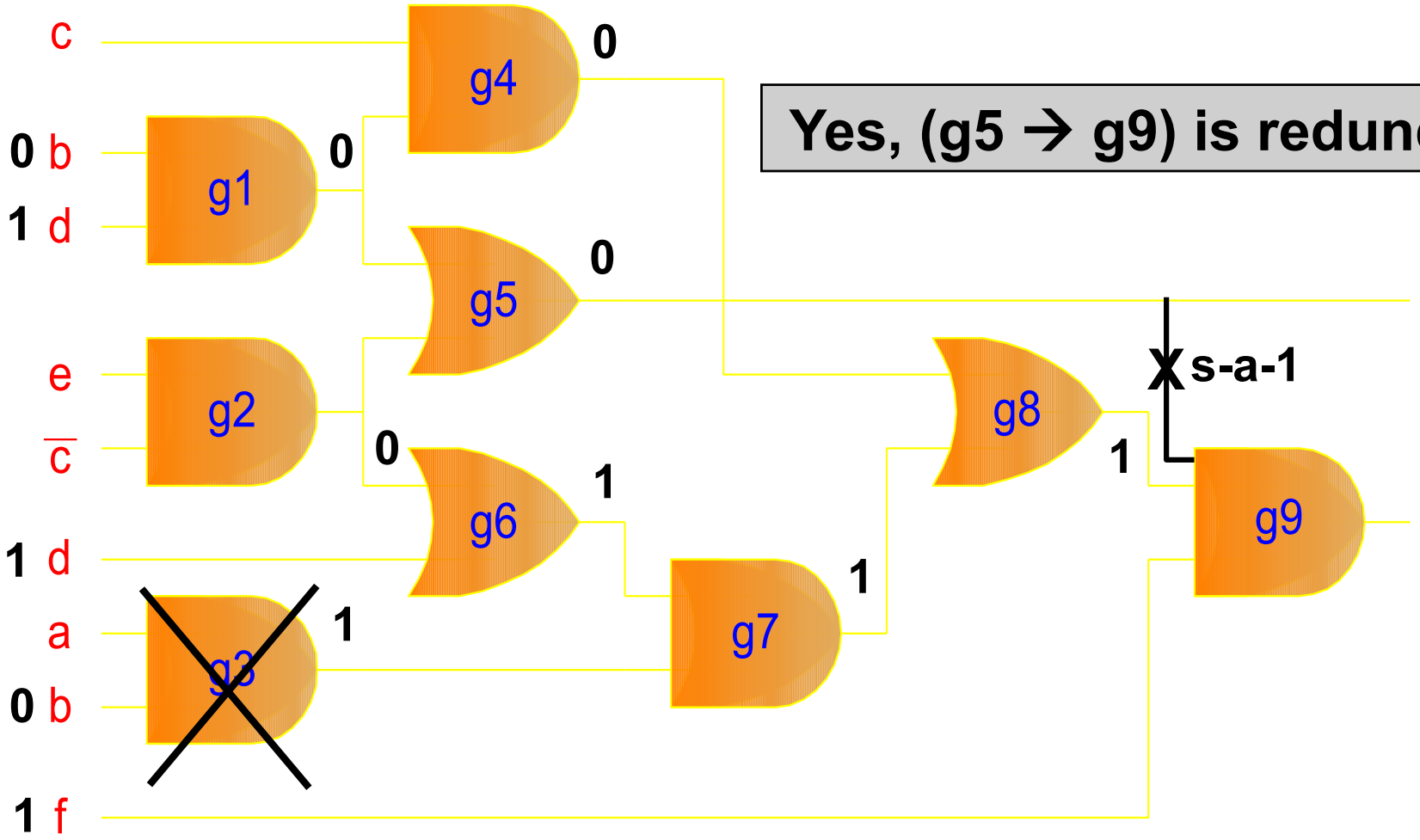
3. If the fault on the added wire in 2.a ~ 2.d is untestable,
   → the added wire is redundant and can be an alternative wire to remove the target wire

Is (!g1 → g8) redundant?



No, (!g1 → g8) is NOT redundant

Is (g5 → g9) redundant?



Yes, (g5 → g9) is redundant

We can remove g6 and then g7

RAMBO Algorithm Complexity

- Need to perform (M * D) redundancy tests
  - M: number of gates in MA
  - D: number of dominators
  - ➔ Could be a BIG number

- "Perturb and Simplify" (Chang, et. al. TCAD 1996)
  - Propose several rules to filter out impossible candidates

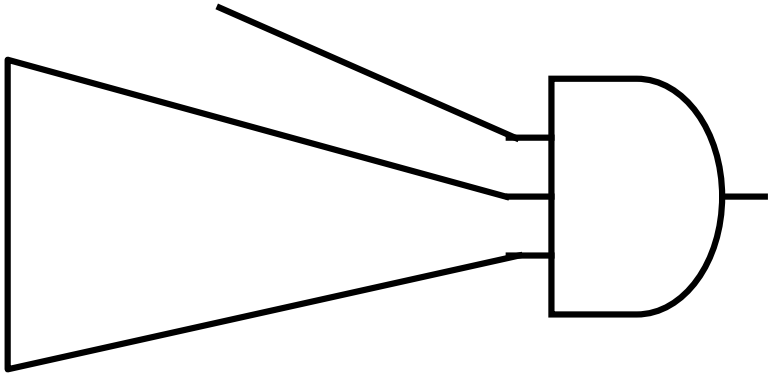
In short, given a target wire, it is easy to add a wire to its dominator to make this wire redundant.

The problem is, need to make sure the added wire is redundant. This may require a large number of fault tests.

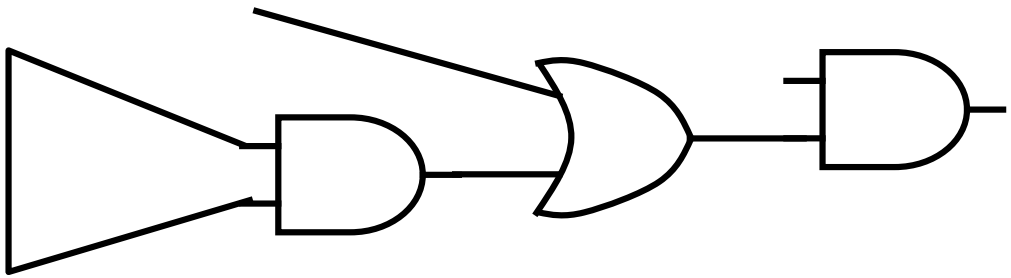
So, can we deliberately add something to a circuit, and guarantee that it is redundant?

How do we add “something” to a circuit and guarantee it is redundant?

- Add a wire

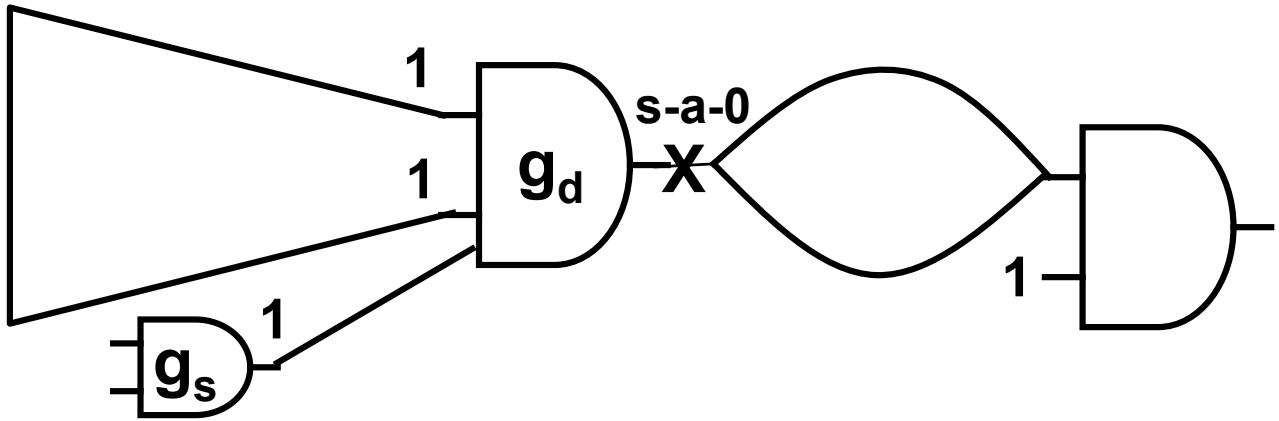


- Add a gate



Creating a Redundant Wire

- e.g. Add to the input of an AND gate g_d
  1. Test the output s-a-0 fault of this AND gate
  2. Perform MA of this fault

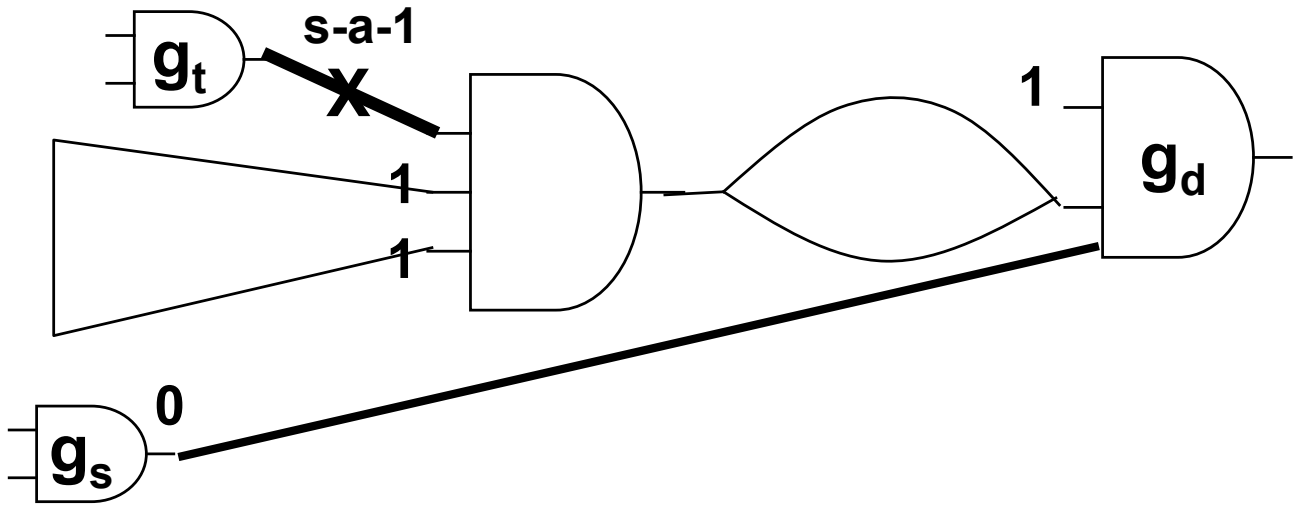


  3. For each gate g_s in the MA, there is a corresponding redundant wire (or with inverter) to g_d

Why??

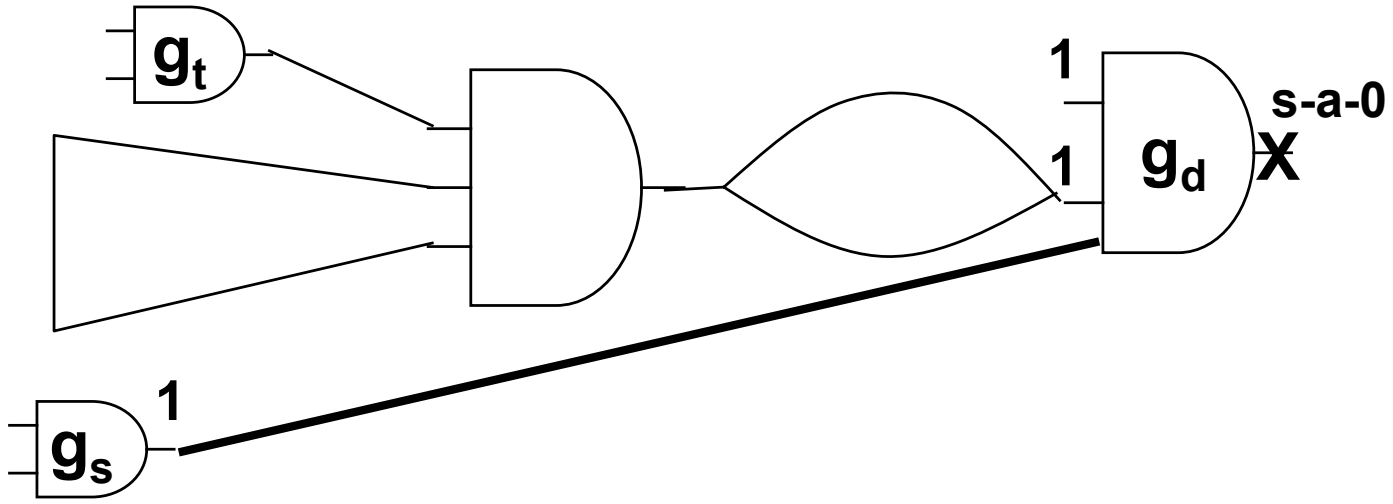
A Two-Way Redundancy Addition and Removal (2-Way RAR) Algorithm

1. Given a target wire on g_t, perform MA(g_t)
   - Adding a wire from a gate g_s in MA(g_t) to any of its dominator g_d can make this target wire redundant
   - e.g. value(g_s) = 0 → AND gate g_d

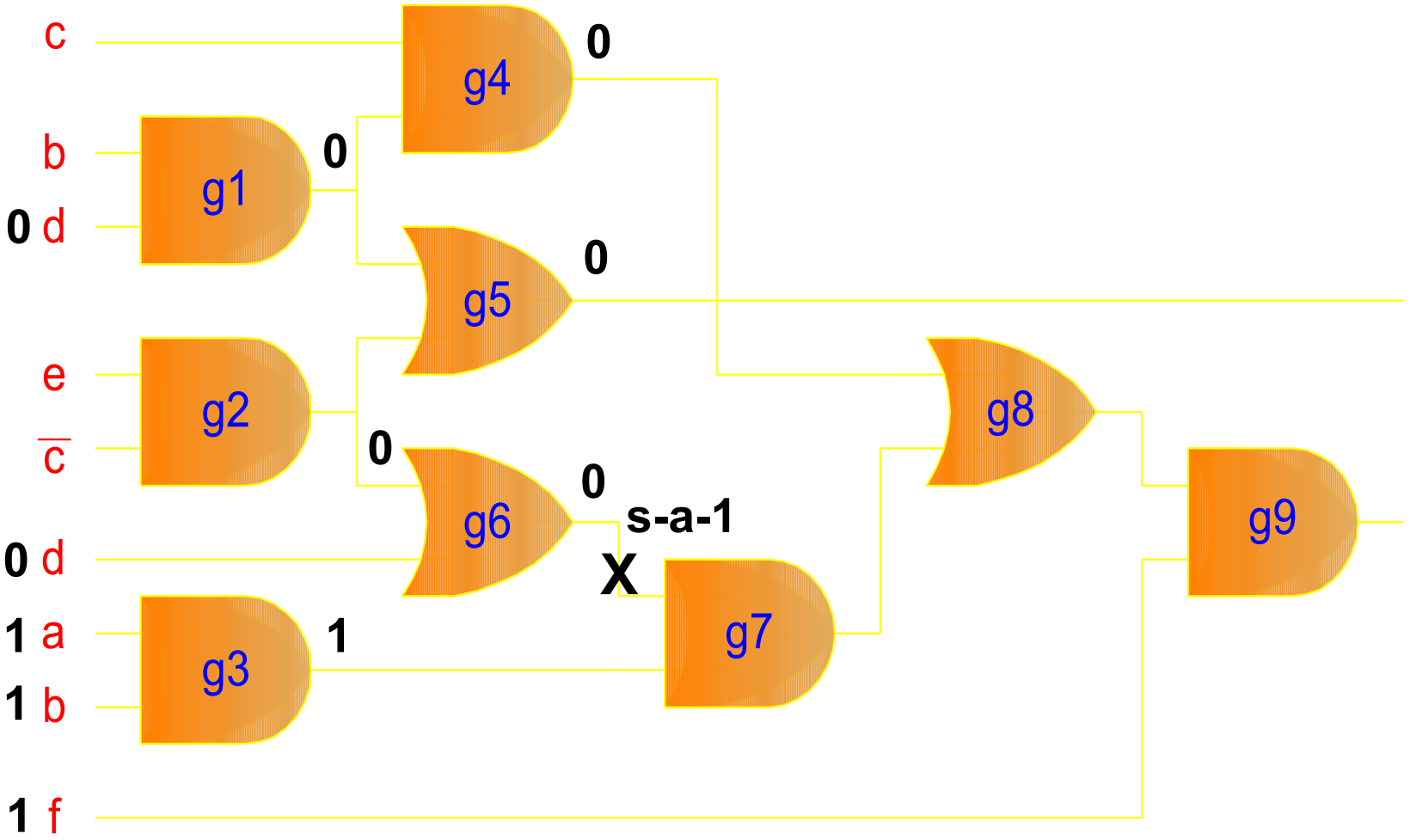


2-Way RAR Algorithm

1. Given a target wire on g_t, perform MA(g_t)
   - Adding a wire from a gate g_s in MA(g_t) to any of its dominator g_d can make this target wire redundant
   - e.g. value(g_s) = 0 → AND gate g_d
2. Given a destination gate g_d(dominator of the target wire g_t), perform MA(g_d)
   - Any wire from a gate g_s in MA(g_d) to this gate g_d can be redundant
   - e.g. value(g_s) = 1 → AND gate g_d

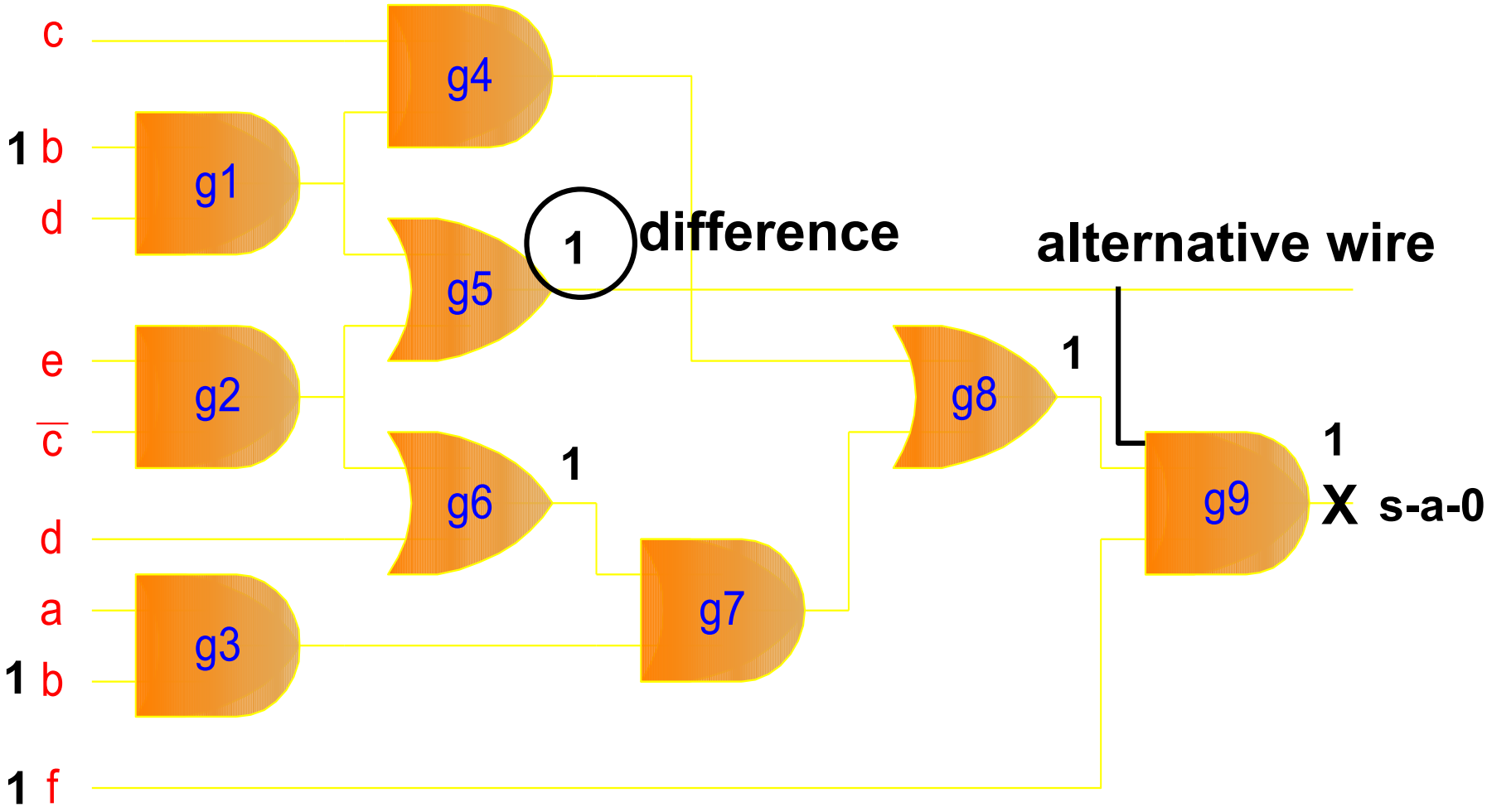


RAR Example (w_t: g6 → g7)



1. MA of w_t : g6 → g7 s-a-1

RAR Example (w_t: g6 → g7)



2. Try MA of g_d : g9 s-a-0

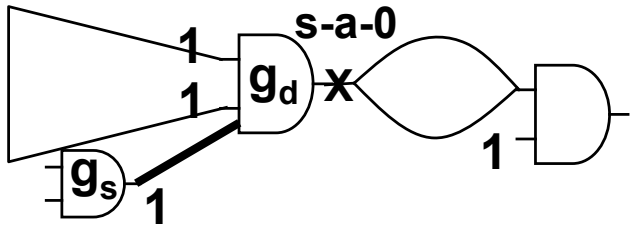
2-Way RAR Algorithm

1. Given a target wire on g_t, perform MA(g_t)
2. Given a destination gate g_d(dominator of the target wire g_t), perform MA(g_d)
3. Perform intersection on (1) & (2)
4. Any contradiction on a gate g_s, implies an alternative wire ($g_s \rightarrow g_d$) for the target wire on g_t
   - Can be generalized for adding a gate or adding a sub-circuit

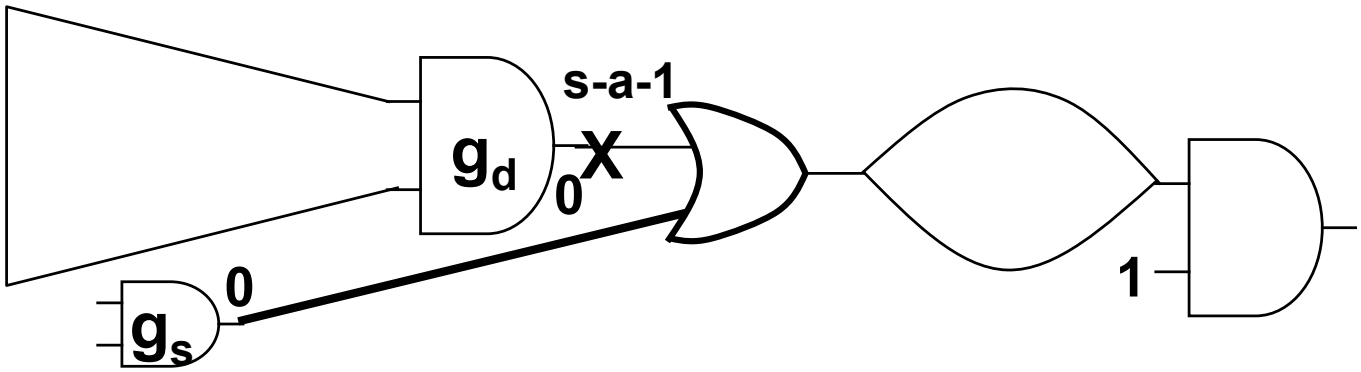
[ref: Huang ISPD 1998]

Creating a Redundant Gate (1)

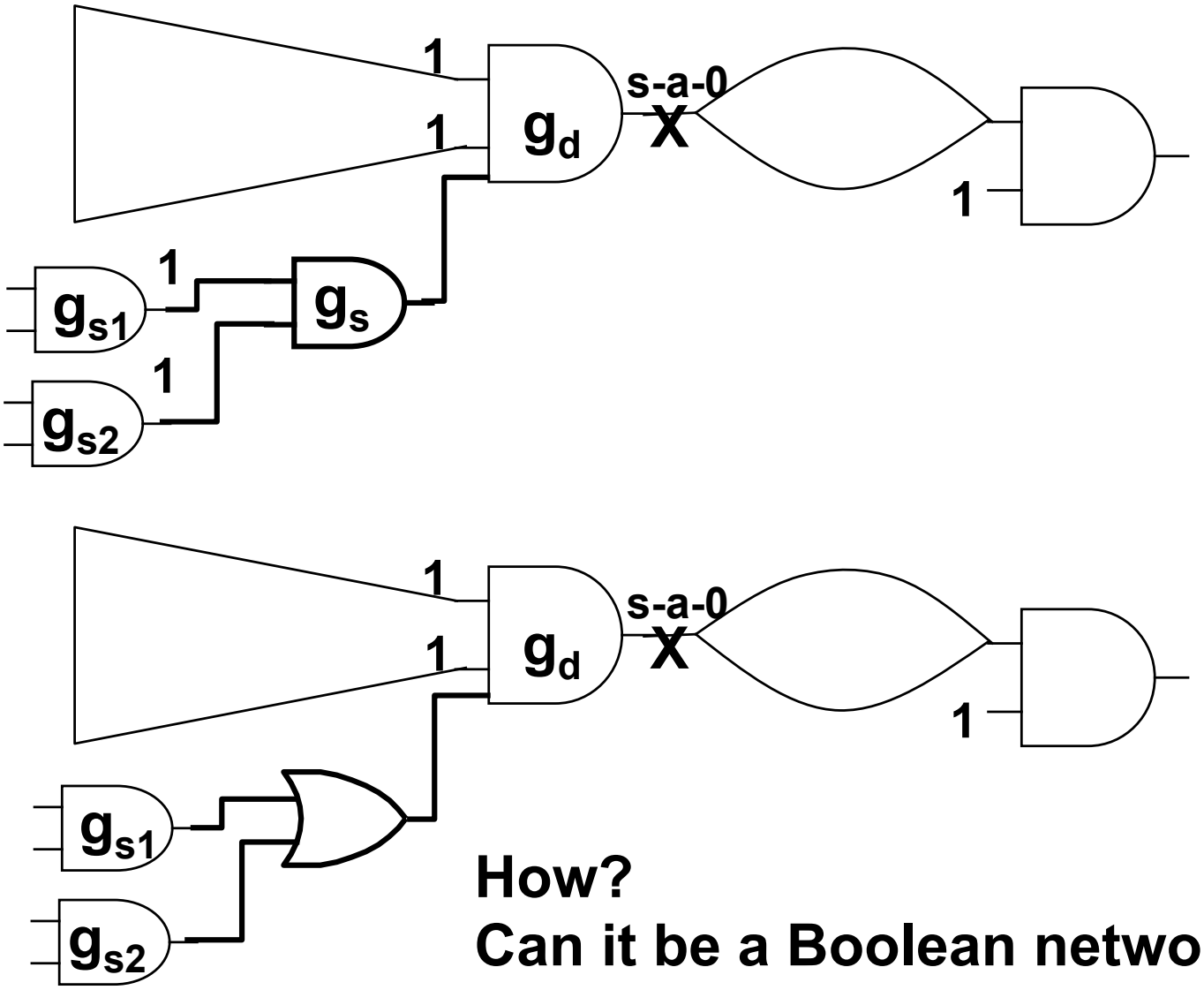
- Refresh “add a redundant wire”:
  e.g. Add to the input of an AND gate g_d
  1. Test the output s-a-0 fault of this AND gate
  2. Perform MA of this fault
  3. For each gate g_s in the MA, there is a corresponding redundant wire (or with inverter) to g_d



- How about adding a redundant gate?
  ➔ Test the output s-a-1 fault of an AND gate?



Creating a Redundant Gate (2)



How?
Can it be a Boolean network?

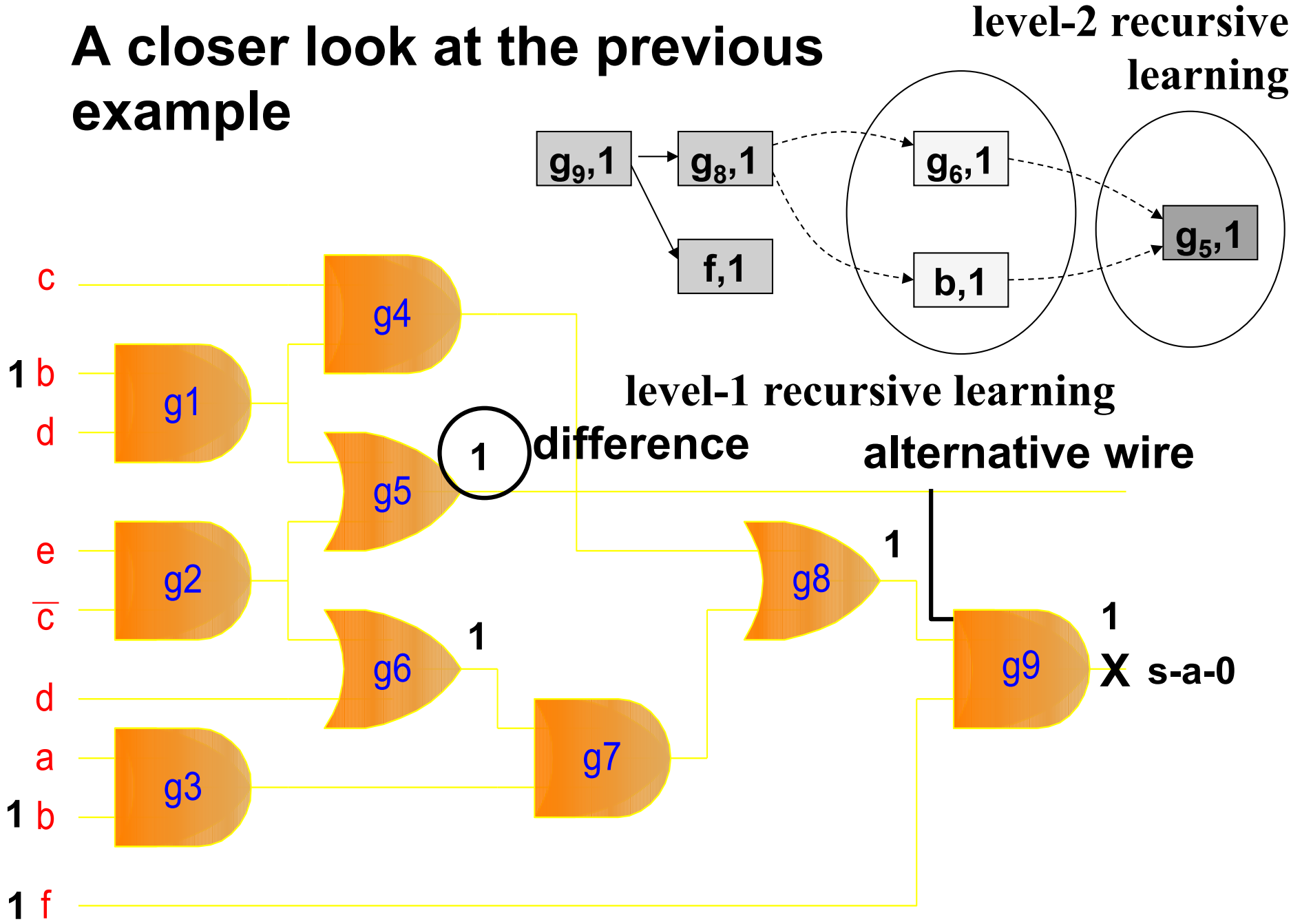
2-Way RAR Algorithm

- Pros
  - No need to perform (M*D) redundancy test as in RAMBO
  - Potential orders of speed-up
- Cons
  - Only connect to dominators?
    (Can we connect to fanins of dominators?)
  - Still need to try for each dominator
  - MA on target wire may NOT intersect with MA on dominators
    ➔ Or just find some trivial alternative wires (e.g. DeMorgan Law)

➔ Methods to deriving more MAs (e.g. Recursive learning) are often used (but could be expensive)

➔ How can we increase the number of MAs?

A closer look at the previous example



SAT-Controlled RAR (SatRAR) [Huang, ASPDAC 2009]

- Problems with previous RAR techniques
  - RAMBO: too many redundancy tests
  - 2-Way RAR: expensive implication technique needed
- SAT-controlled RAR
  - NOT just take the advantage of the advancements from the modern SAT solvers (covered later)
    - Efficient BCP, conflict-driven learning, etc
  - A seamless integration of SAT and RAR algorithms
  - Extensions for general RAR
    - Alternative wire, gate, sub-circuit identification
  - Options to “control” the RAR optimization quality

Single Wire Replacement Theorem in SatRAR

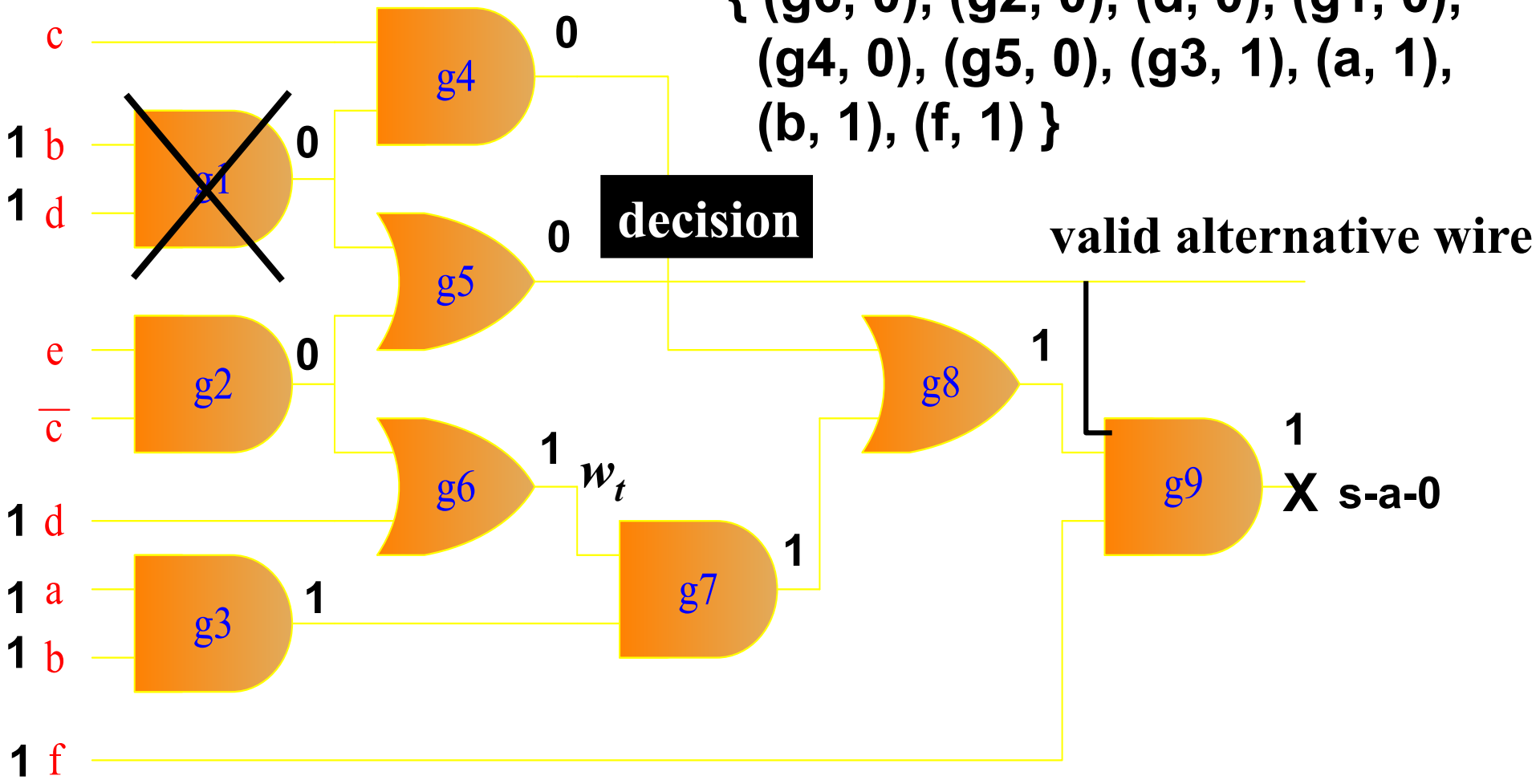
- *Let **MA(w_t)** and **MA(g_d)** be the mandatory assignments for the fault tests of the target wire w_t and its dominator g_d, respectively.*
- *Let **<g_s, v>** belong to MA(w_t) but not MA(g_d), and g_s be not in the fanout cone of g_d.*
- *If we make a decision <g_s, v> on top of MA(g_d) and encounter a conflict, then*

  (i) MA(g_d) $\Rightarrow$ <g_s, $\neg v$>

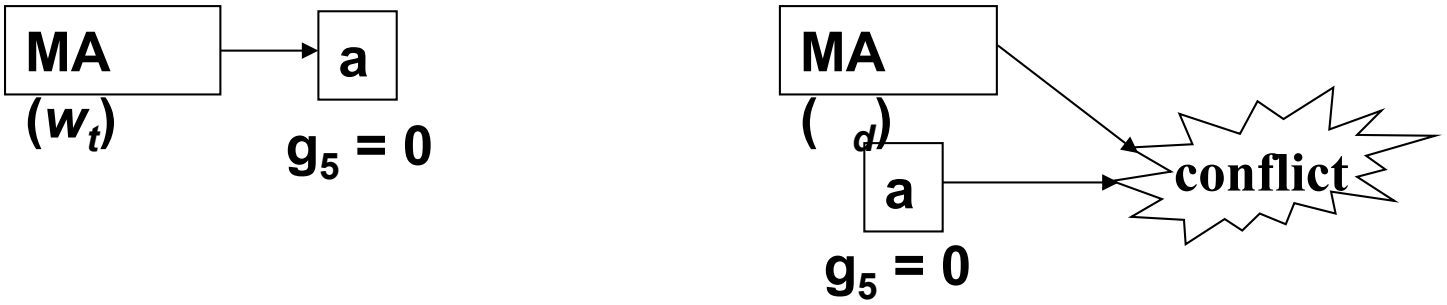
  (ii) ($g_s \rightarrow g_d$) or ($g_s \rightarrow\!\!\circ\ g_d$) must be a valid alternative wire for w_t

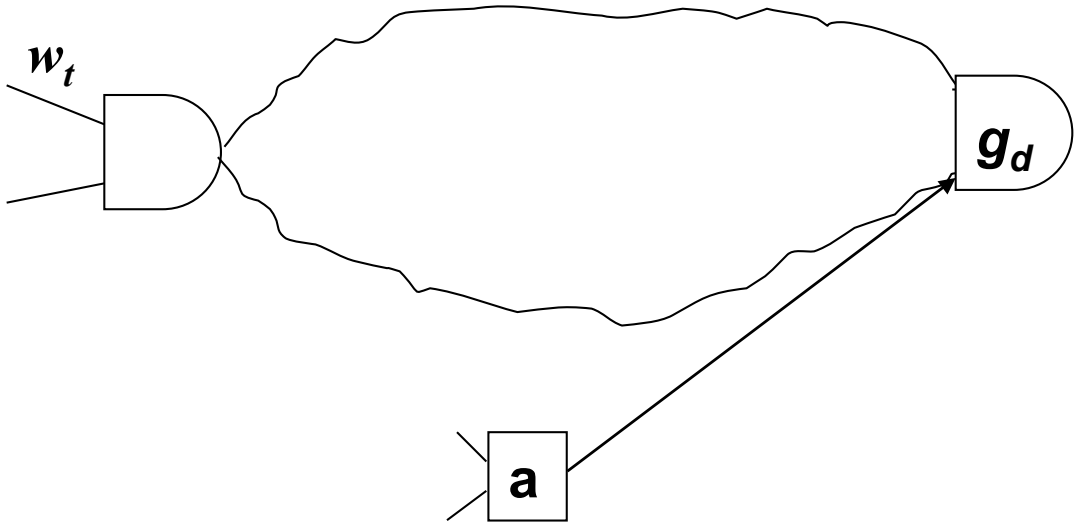
Single Wire Replacement in SatRAR

$MA(w_t = g_6) =$
$\{ (g6, 0), (g2, 0), (d, 0), (g1, 0),$
$(g4, 0), (g5, 0), (g3, 1), (a, 1),$
$(b, 1), (f, 1) \}$



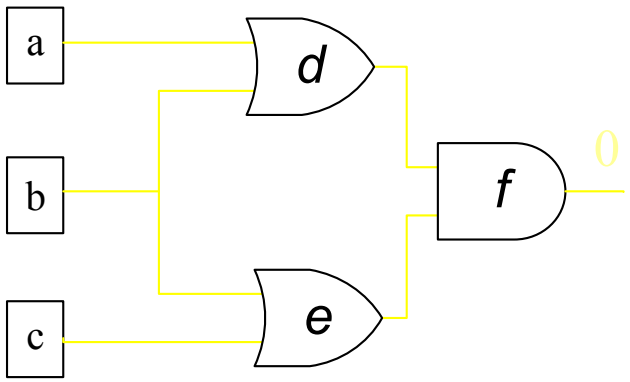
A closer look...





SRAR-Wire vs. 2-Way RAR

- Similarity
  - Based on the conflicting implications between MA(w_t) and MA(g_d)
- Difference
  - SAT decision (conflict-driven leanring) vs. Recursive learning



- **Recursive learning:**

f = 0 ⇒ d = 0 or e = 0
⇒ { a=0, b=0 } or { b=0, c=0 }
⇒ b = 0 (Cannot be recorded)

- **Conflict-driven learning:**

f = 0; decision b = 1 results in conflict
➔ f = 0 ⇒ b = 0 (Recorded!!)

SRAR-Wire vs. Original RAMBO (FYI)

- Similarity (looks like...)
  - For each assignment g_s in { MA(w_t) - MA(g_d) }... vs.
    For each assignment g_s in MA(w_t)...
- Difference
  - Incremental SAT vs. Independent redundancy tests
- Incremental SAT in SatRAR
  - MA(g_s) is performed on top of MA(g_d)
    - Sharing of different MA(g_{di})
  - Conflict-driven learning
    - Learning & RAR at the same time
  - Implication filter
    - Reduce #decisions
  - More importantly, can be extended for alternative gate/ sub-circuit replacements

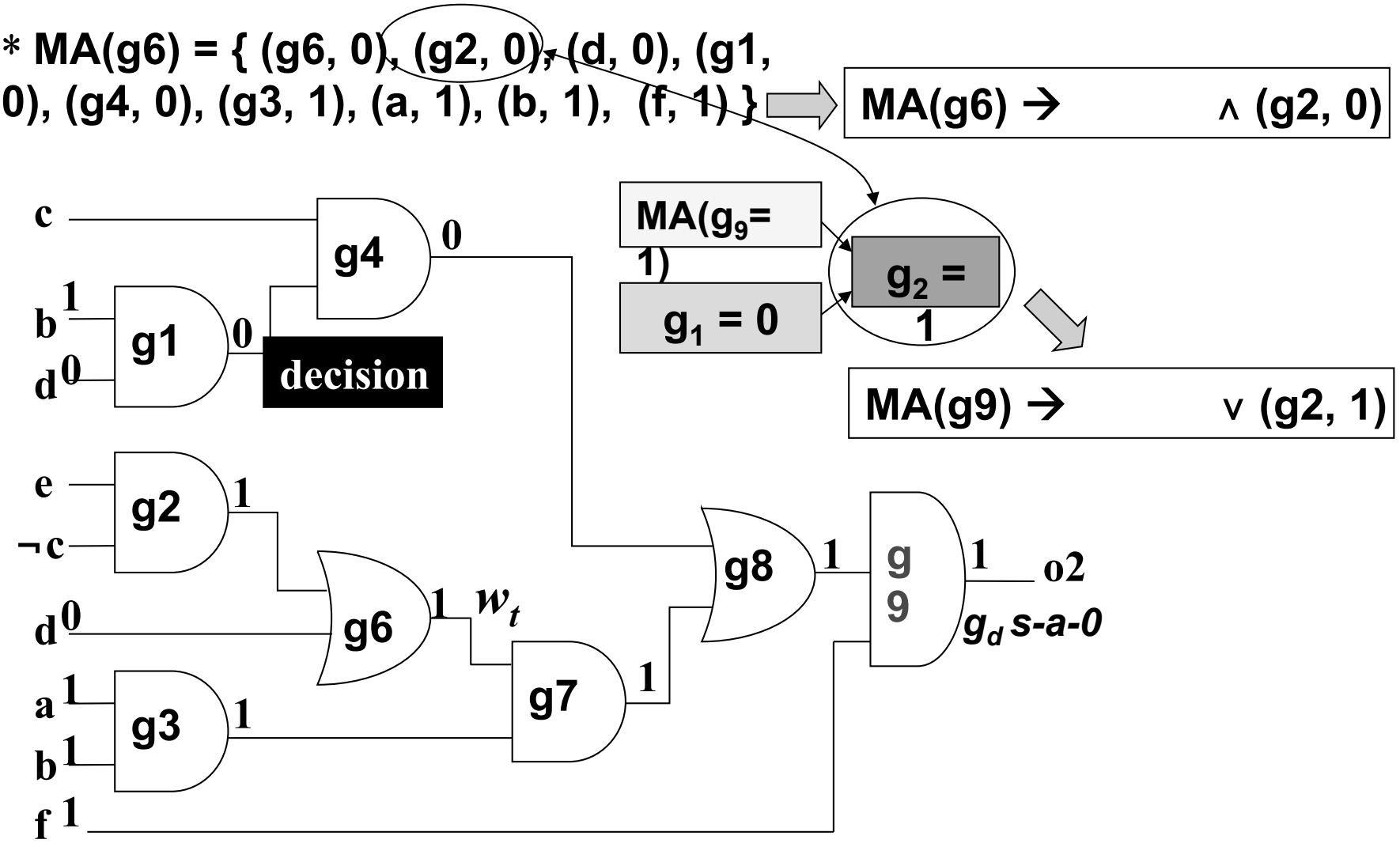
Single Gate Replacement Theorem in SatRAR

- *Let **MA(w_t)** and **MA(g_d)** be the mandatory assignments for the fault tests of the target wire wt and its dominator gd, respectively.*
- *Let both **<g_s, u>** and **<g_t, v>** belong to MA(w_t), and be not in the fanout cone of gd.*
- *Suppose we make the decision <g_s, u> after MA(g_d) and result in an implication <g_t, ¬v>.*
- *Let a gate g_n = AND(<g_s, u>, <g_t, v>). Then*

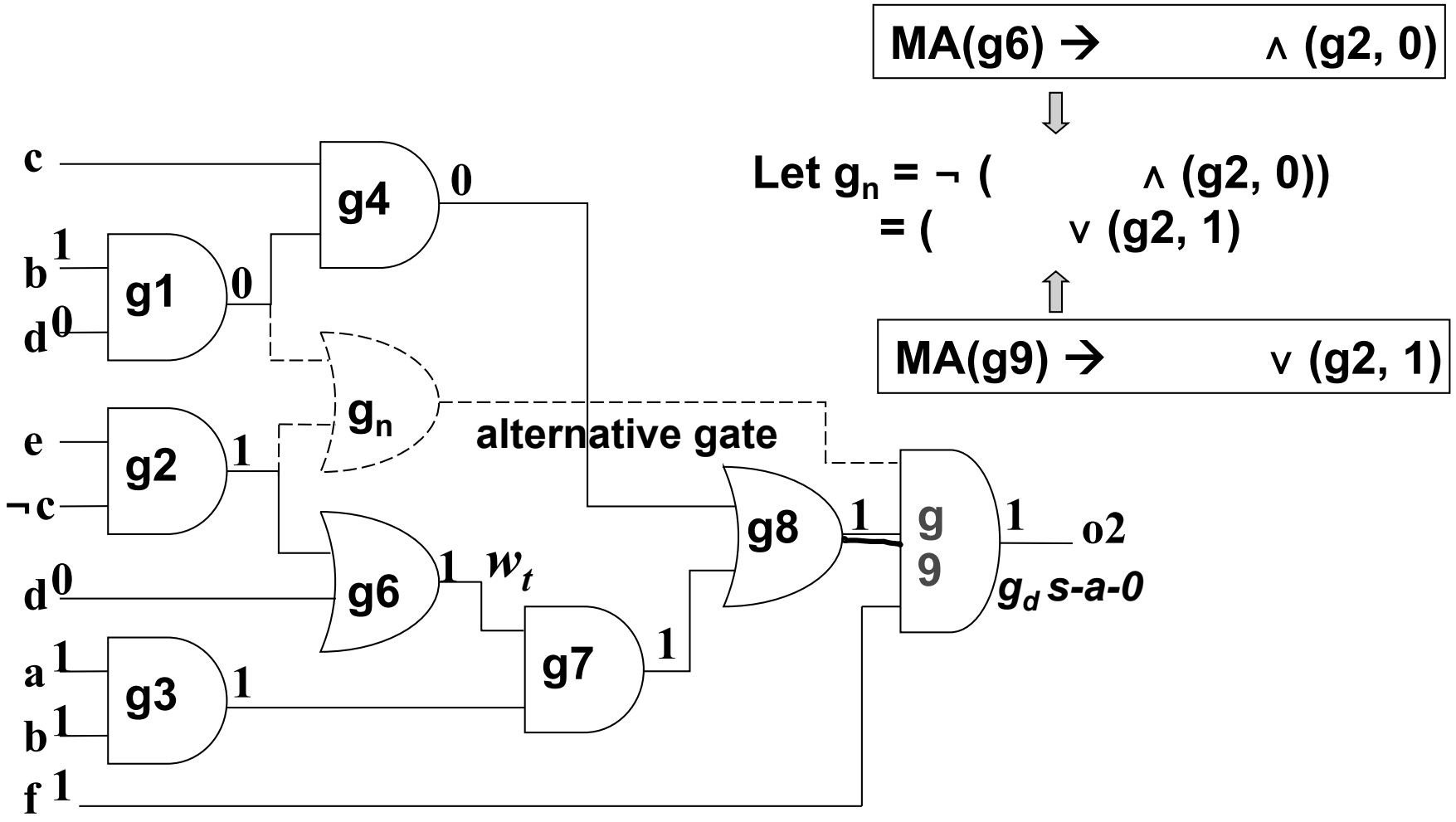
  (i) MA(g_d) ⇒ ¬ g_n,

  (ii) g_n or ¬ g_n, when connected to g_d, must be a valid alternative gate for w_t

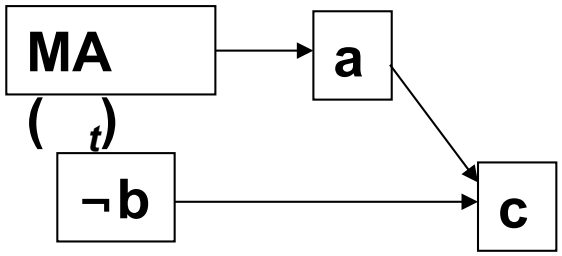
Single Gate Replacement in SatRAR

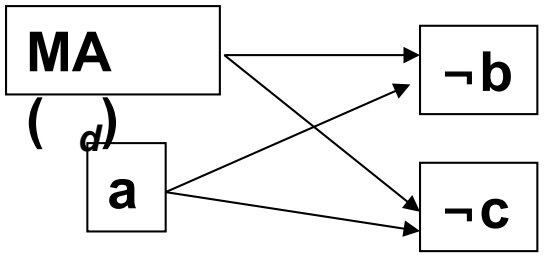


Single Gate Replacement in SatRAR



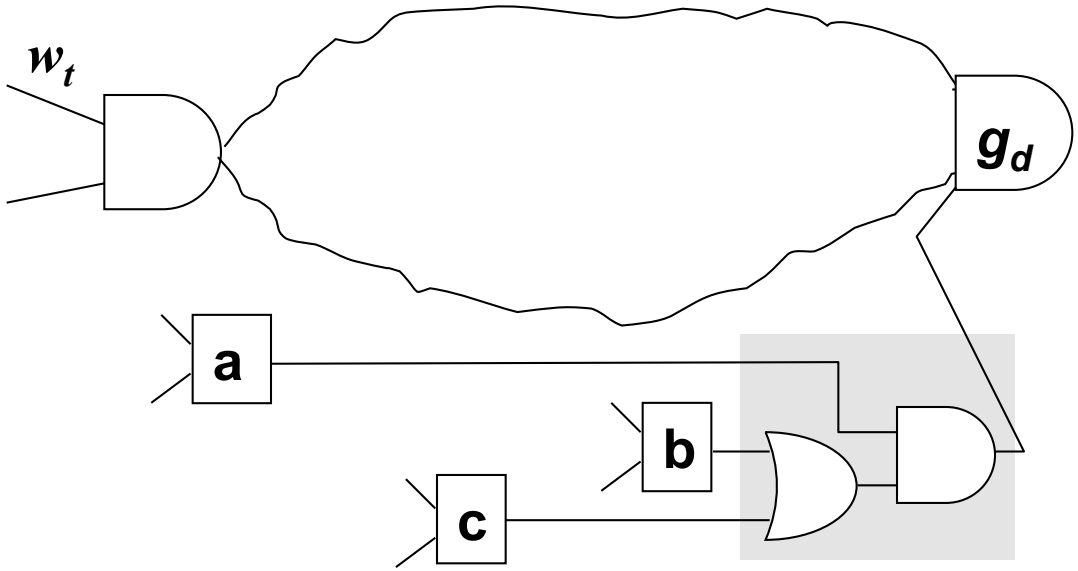
Alternative Sub-circuit by SatRAR





⇨ MA(w_t) → a ∧ (b ∨ c)

⇨ MA(g_d) → ¬ (a ∧ (b ∨ c))



What can/should be covered in this topic?

- SAT-based logic synthesis
  - Redundancy addition and removal
  - ~~Functional dependency~~
  - ~~SAT-based re-synthesis techniques~~
  - ~~Engineering Change Order (ECO)~~
- From SAT to optimization problems
  - Pseudo Boolean satisfiability/optimization problems
- ~~General SAT-based model checking algorithms~~
- ~~Quantified Boolean Formula (QBF)~~
- ~~Bit-vector/Arithmetic solver~~
- ~~Satisfiability Modulo Theories (SMT)~~

Optimization problems? Finding all solutions of a SAT instance?

- It seems that with learning techniques (e.g. blocking clauses, success-driven learning), we can find all the solutions of a SAT problem

- With a (target) cost function, can SAT be used for optimization problems?
  - minimize(or maximize) $f(x)$;
    subject to $X = \{ x \mid g_i(x) \geq b_i, \quad i = 1...m \}$;
    where
    - $x = (x_1,..., x_n)$ are optimization (or decision) variables,
    - $f(x)$ is the objective function, and
    - $g_i(x)$ and b_i form the constraints for the valid values of x.

  ➔ Find an assignment $A \in X$ such that $f(A)$ is min(Max)

A Brute Force Approach

```
SatOpt(C, F) {
  Let bestCost = INT_MAX;
  while (SAT(C) has a solution A) {
    if (F(A) < bestCost) {
      bestScore = F(A);
      bestAssign = A;
    }
    // adding blocking clause
    C ← C ∧ ¬A;
  }
}
```

➔ Number of solutions for SAT(C) may be huge!!

➔ Any better approach?

A Better Approach

```
SatOpt(C, F) {
  Let bestCost = INT_MAX;
  while (SAT(C) has a solution A) {
    if (F(A) < bestCost) {
      bestScore = F(A);
      bestAssign = A;
    }
    // adding new constraint
    C ← C ∧ (F < bestScore);
  }
}
```

➔ Excluding all "F ≥ bestScore" solutions at a time

➔ But how to transform the inequality "F < bestScore" to CNF ?

Using SAT to solve optimization problems

- In the following, we will learn ---

1. Pseudo Boolean (PB) optimization problems
2. How to transform a PB optimization into a SAT problem
3. SAT vs. PB learning

Using SAT as a Pseudo-Boolean Constraint Solver

- A Pseudo Boolean (PB) constraint is an inequality on a linear combination of Boolean variables
  - PB: $c_0x_0 + c_1x_1 + \ldots + c_{n-1}x_{n-1} \geq k$
    where c_i is an integer, $x_i \in \{0, 1\}$
  - A PB constraint is said to be *satisfied* if the LHS of the PB is greater or equal to k
  - Many problems are more naturally expressed in PB format!!
- PB SAT/OPT problem
  - Given a set of PB constraints
  - Given a target function of the form:
    $a_0x_0 + a_1x_1 + \ldots + a_{n-1}x_{n-1}$
  - ➔ Find an assignment that satisfies all the PB constraints and minimizes the target function

Normalization of PB Constraints

- Normal form for PB constraints?
  - Usually many syntactically different, yet semantically equivalent, constraints
    - e.g. $4x + 3y - 3z \geq -1$ and $y + \neg z + x \geq 1$
    - Difficult to prove their equivalence
  - No known good method to canonicalize the PB constraints
- But, try to apply some normalization steps ---
  - Simplifies the implementation by giving fewer cases to handle
  - May reduce some constraints and make the subsequent translation more efficient.

Normalization Steps in miniSat+

1. ≤ constraints are changed into ≥ constraints by negating all constants.
2. Negative coefficients are eliminated by changing x into ¬x and updating the RHS.
3. Multiple occurrences of the same variable are merged into one term x or ¬x :
4. The coefficients are sorted in ascending order: $a \leq a$ if $i < j$.
5. Trivially satisfied constraints, such as “x + y ≥ 0” are removed

Normalization Steps in miniSat+

6. Trivially unsatisfied constraints (“x + y ≥ 3”) will abort the parsing and report Unsatisfiable.
7. Coefficients greater than the RHS are trimmed to (replaced with) the RHS.
8. The coefficients of the LHS are divided by their greatest common divisor (“gcd”).
9. The RHS is replaced by “RHS/gcd”, rounded upwards

Normalization Example

- $4x + 3y - 3z \geq -1$

➔ $4x + 3y + 3\neg \geq 2$ (positive coefficients)

➔ $3y + 3\neg \geq 2$ (sorting)

➔ $2y + 2\neg \geq 2$ (trimming)

➔ $y + \neg \geq 1$ (gcd)

(note: $\neg z = 1 - z$)

More Preprocessing on PB Constraints

- Trivial constraint propagation
  - e.g.

    "$3x + y + z \geq 4$" ➔ x must be "TRUE" (why?)

- Constraint splitting
  - e.g.

    $4x_1 + 4x_2 + 4x_3 + 4x_4 + 2y_1 + y_2 + y_3 \geq 4$

    ➔ $x_1 + x_2 + x_3 + x_4 + \neg z \geq 1$ (clause part)

    $2y_1 + y_2 + y_3 + 4z \geq 4$ (PB part)

    ,where z is a new variable not present in any PB

    (what does z mean?)

Optimization Procedure

Given a set of PB constraints and a target function ---

1. First run the solver on the set of constraints (without considering the objective function) to get an initial solution $F(x_0) = k$.
   - How? DPLL? LP relaxation? (Covered later)
2. Then add the PB constraint $F(x) < k$ and run again
3. Run until no more solution in 2 is possible

➔ What's the difference between this and the SAT brute-force approach shown earlier?

➔ Solve as a PB problem, or as a SAT problem?

➔ How to use SAT to solve PB constraints?

Using SAT to solve optimization problems

- In the following, we will learn ---

1. Pseudo Boolean (PB) optimization problems
2. How to transform a PB optimization into a SAT problem
3. SAT vs. PB learning

Translating PB Constraints to CNF

- Converting PB constraints into circuit netlist first!!
  How? (e.g. miniSat+)

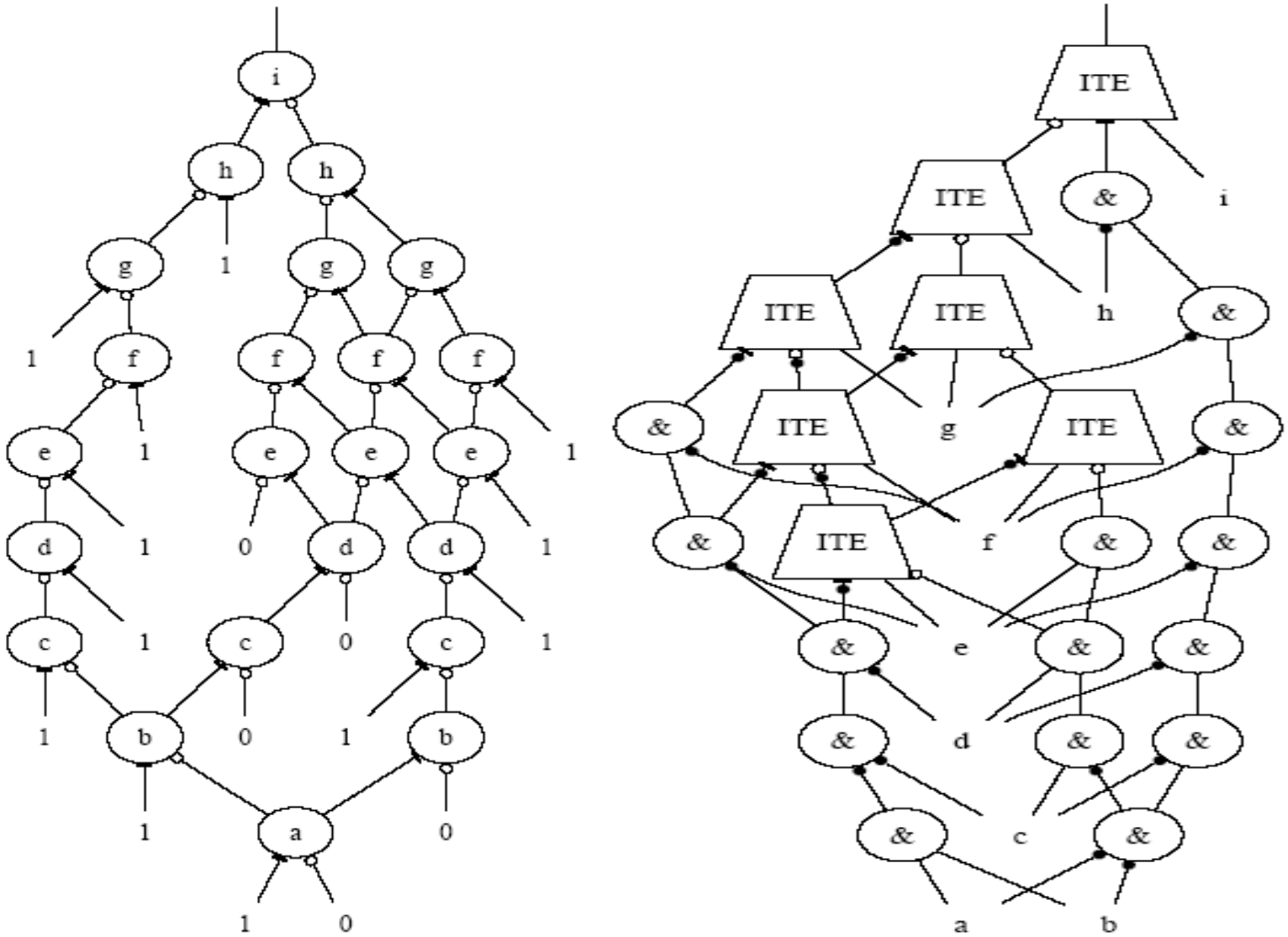
1. BDD
2. Adder network
3. Sorter network

- Basic step: the Tseitin transformation
  - ITE(s, t, f)
    - (~s+~t+ x)(~s+ t+~x) (s+~f+ x) (s+ f+~x) (~t+~f+ x) (t+ f+~x)
  - FA_sum(a, b, c): as XOR(a, b, c)
  - FA_carry(a, b, c): as $a + b + c \geq 2$
  - HA_sum: as XOR
  - HA_carry: as AND

Translation of PB-constraint (BDD)

- Variable order:
  - Largest coefficient to the smallest.
- Once the BDD is built, it can simply be treated as a circuit of ITEs and translated to clauses by the Tseitin transformation.

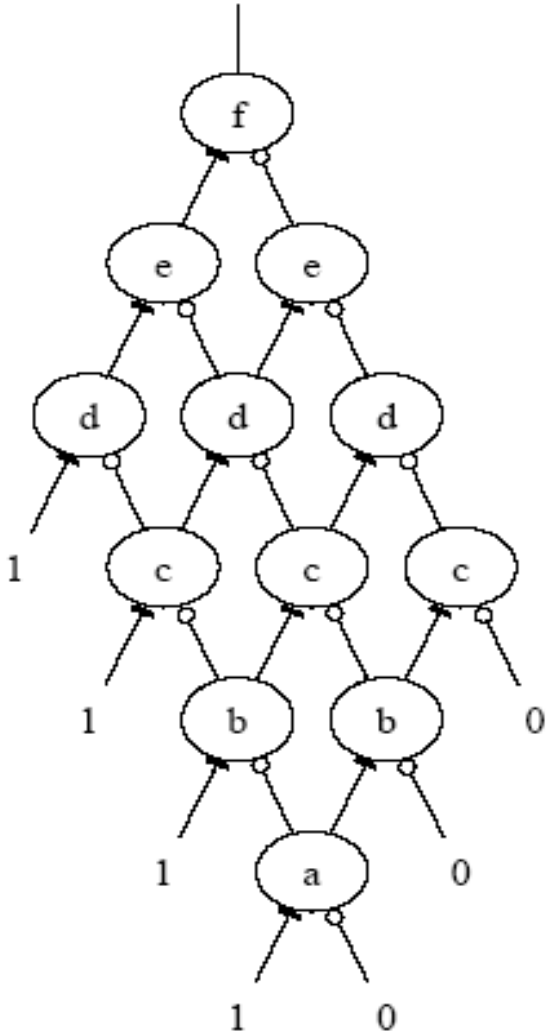
1. BDD Translation of
a + b + 2c + 2d + 3e + 3f + 3g + 3h + 7i ≥ 8



Translation of PB-constraint (BDD)

- Transform every BDD node by a ITE gate, with worst case exponential.
- But for cardinality linear PBCs, linear size BDDs is constructed, so it is very efficient.

  eg. BDD for a+b+c+d+e+f $\geqq$3



2. Adder Translation of
2a + 13b + 2c + 11d + 13e + 6f + 7g + 15h ≥ 12

```
  00a0
  bb0b
  00c0
  d0dd
  ee0e
  0ff0
  0ggg
+ hhhh
```

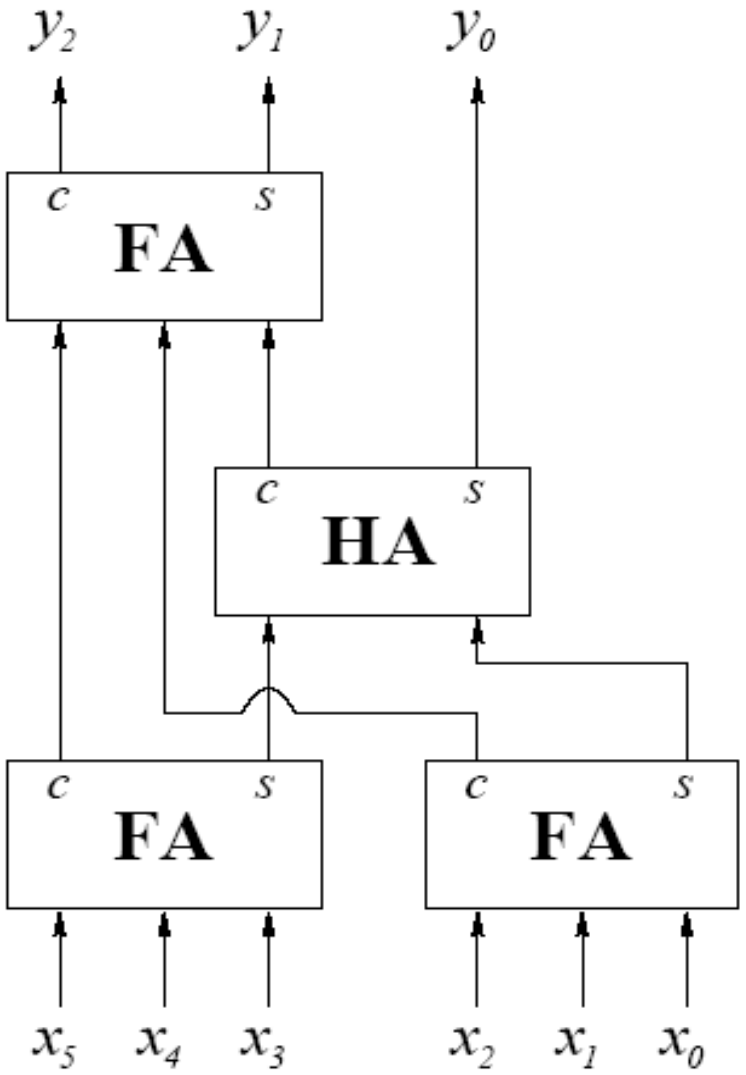
Bucket:	Content:
1-bits:	[b,d,e,g,h]
2-bits:	[a,c,d,f,g,h]
4-bits:	[b,e,f,g,h]
8-bits:	[b,d,e,h]

➔ **Generate a binary sum and then compare with the RHS (12)**

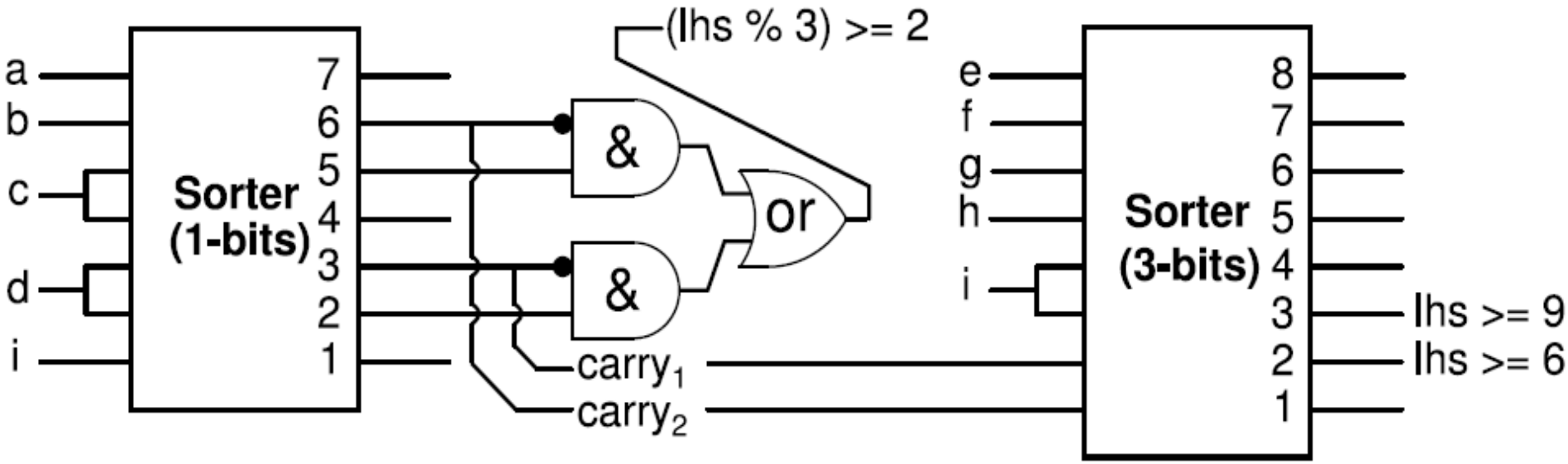
Translation of PB-constraint (Adder)

- For each bit (bucket), an adder network is established.

  eg.
  Adder circuit for $x_0+\ldots+x_5$



3. Sorter Translation of
a + b + 2c + 2d + 3e + 3f + 3g + 3h + 7i ≥ 8



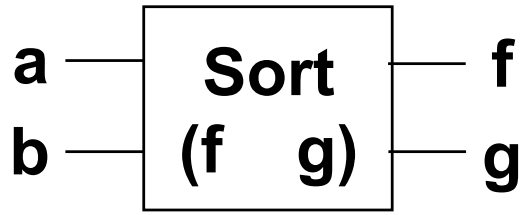
➔ Base of 3 representation (above example)

➔ A comparison network with RHS is then constructed

(LHS ≥ 9) ((LHS ≥ 6) (LHS % 3 ≥ 2))

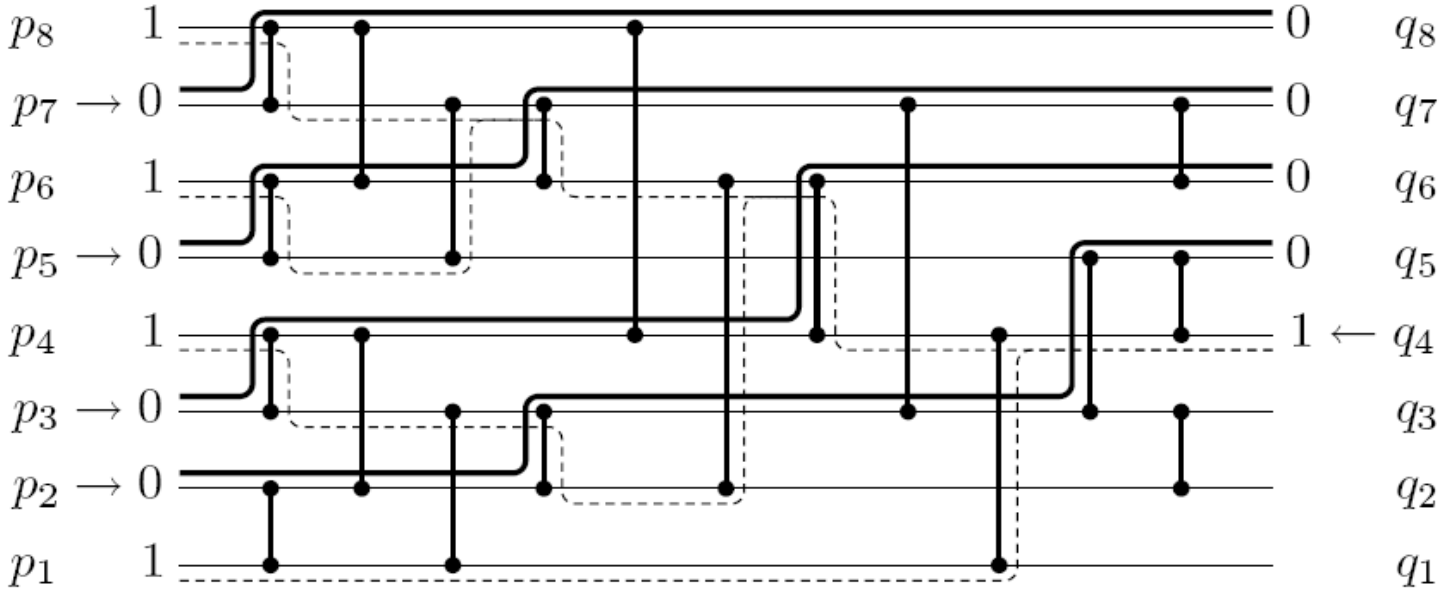
Translation of PB-constraint (Sorter)

- Basic operator



How?

- Odd-even merge sorter

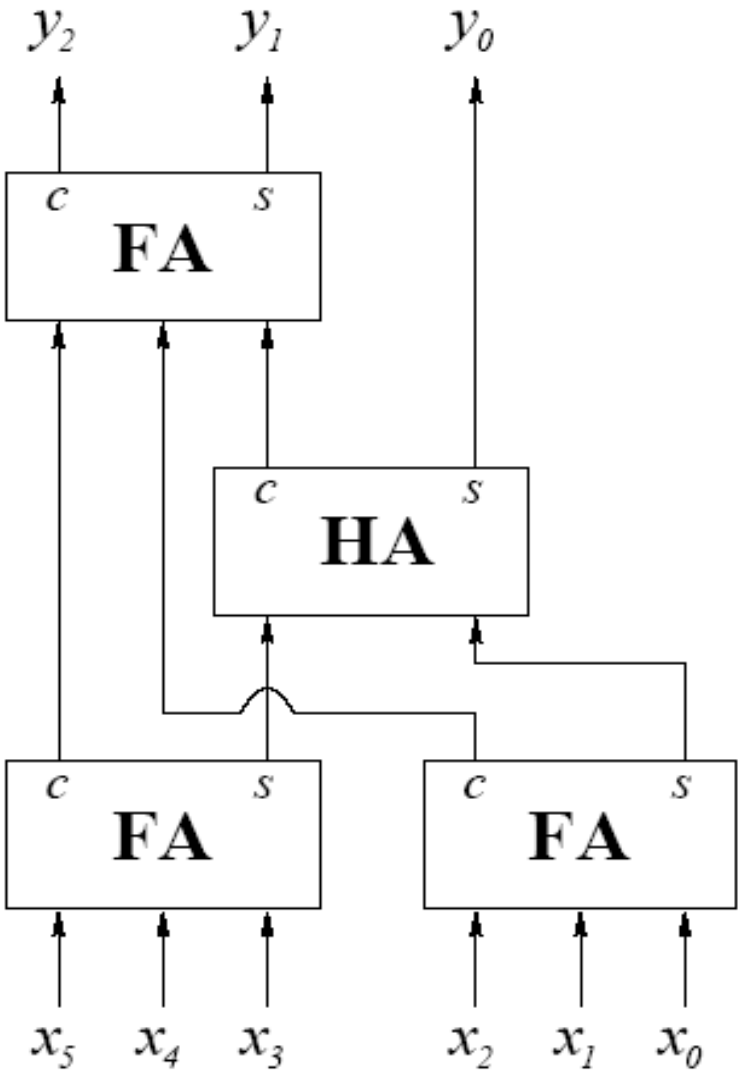


SAT-based approach issue: arc-consistency

- Whenever the original constraint C(X) get an implication, the translation Φ(X, T) will get the same implication.
  - X: original variables
  - T: introduced variables
- Then we say that the translation is arc-consistency

e.g. Adder network has no arc-consistency

- $x + x + x + x + x + x \geq 4$
- $x = 0$ and $x = 0$
  - ➔ All other variables = 1
  - ➔ Cannot be derived from the adder network



Arc Consistency and Complexity Analysis

- BDD
  - Arc-consistency
  - #clauses: worst case exponential, but linear when the PBC is cardinality.
- Adder
  - Not arc-consistency
  - Weaker implicativity
  - O(n) clause size complexity
- Sorter
  - Not arc-consistency
  - Stronger implicativity
  - $O(n \log^2 n)$ clause size complexity

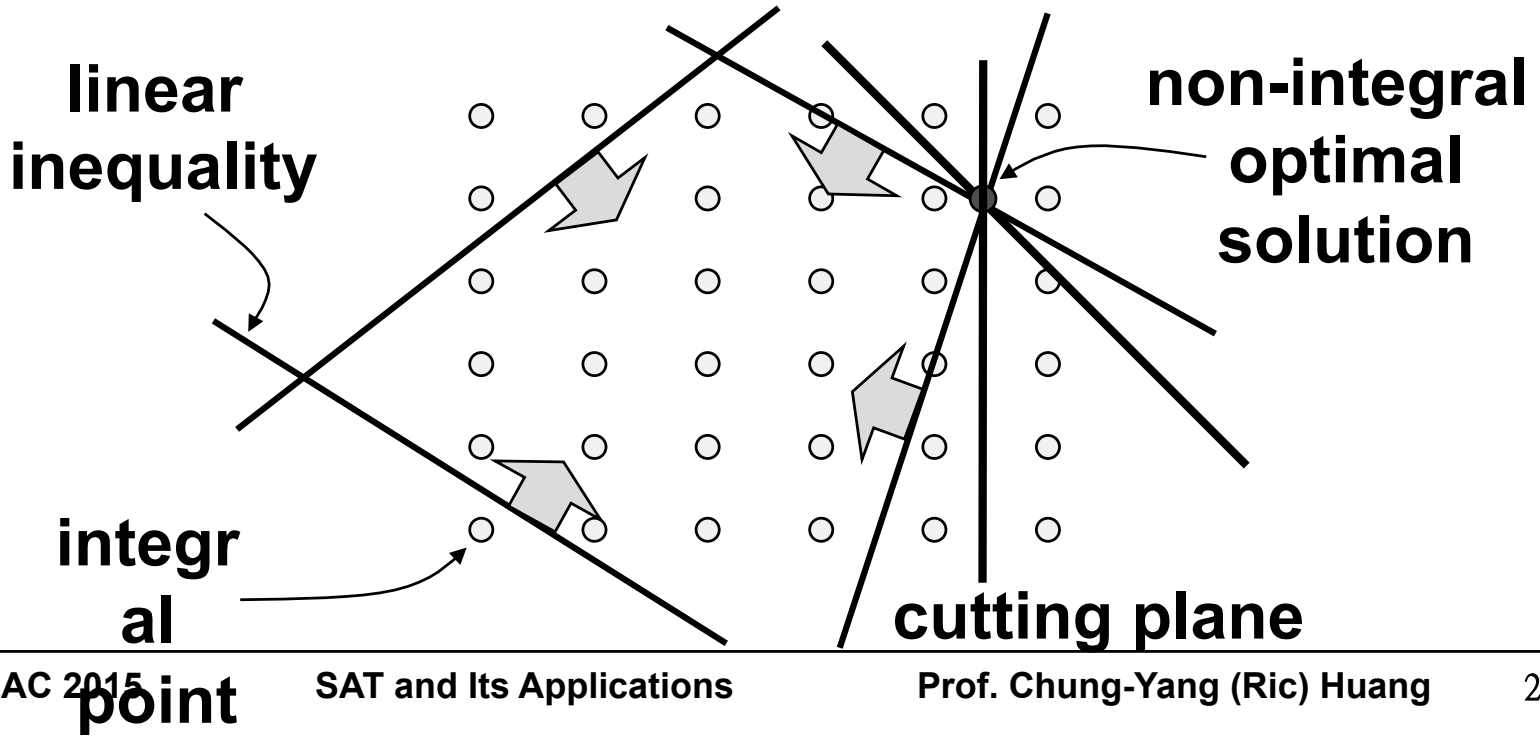
We have seen how to translate a PBC problem into a SAT one.

➔There may be some overhead and drawbacks...

How about solving PBC on PB data structure?

PBC solver

- Traditionally, PB (or in general ILP) constraint satisfaction/optimization problem can be solved by linear programming relaxation...
  - ➔ Common approach for ILP problems
  - ➔ Cutting plane / branch and cut / lift and cut... etc

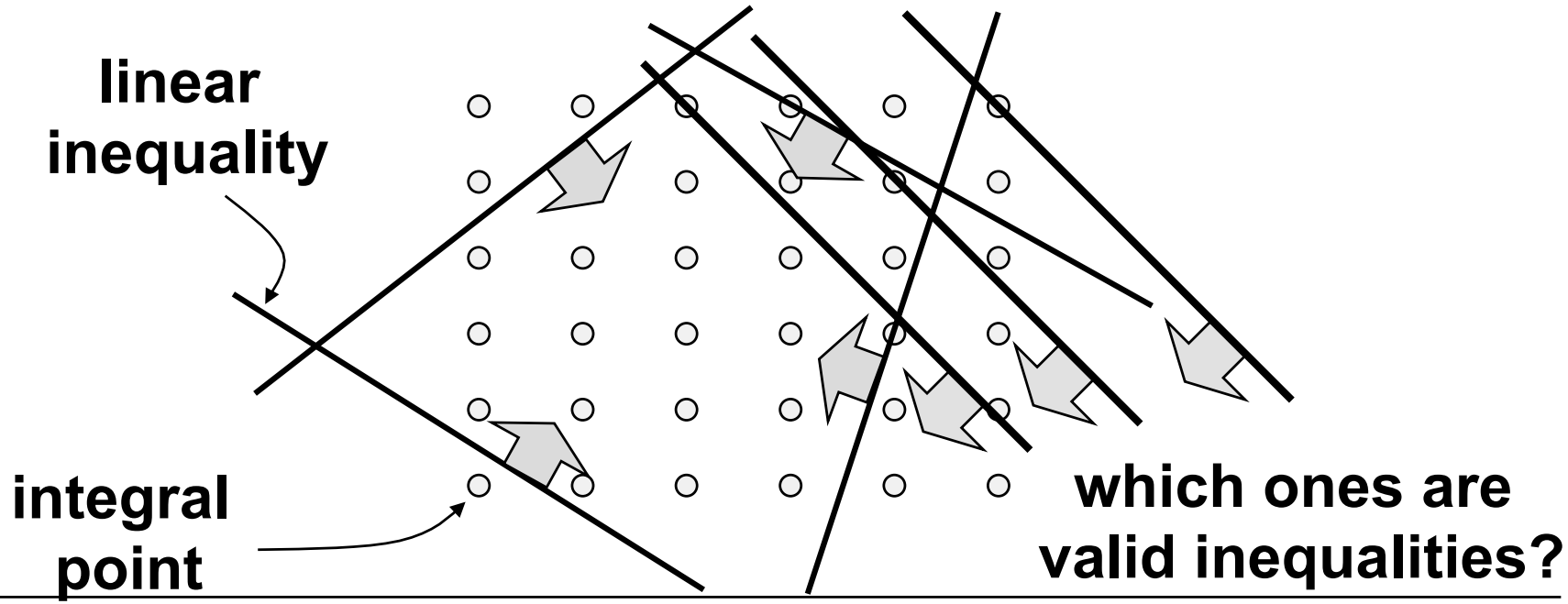


Valid Inequality

- Given a integer program:

  $\max\{ cx: x \in X \}$, where $X = \{ x: Ax \le b, x \in Z^+ \}$

  ➔ An inequality $\pi x \le \pi_0$ is called a "valid inequality" if $\pi x \le \pi_0$ for all $x \in X$.



Chvátal-Gomory procedure

- Chvátal-Gomory procedure to construct a valid inequality

  Given $X = \{ x: Ax \le b, x \in Z^+ \}$

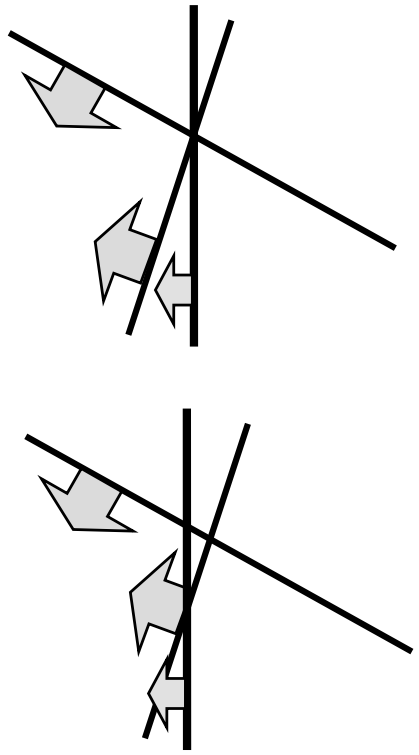
  1. Let u be a row vector with nonnegative coefficients

     ➔ $uAx \le ub$ is a valid inequlity

     (i.e. linear combination)

  2. The inequality $\lfloor uAx \rfloor \le \lfloor ub \rfloor$ is also valid

     (also called: lifting)



Chvátal-Gomory procedure example

- $Ax \leq b$ ---

  $7x_1 - 2x_2 \leq 14$

  $x_2 \leq 3$

  $2x_1 - 2x_2 \leq 3$

1. Multiply u = (2/7, 37/63, 0)

   ➔ $uAx = 2x_1 + 1/63\ x_2 \leq 121/21 = ub$

2. Applying “floor” function on the inequality

   ➔ $2x_1 + 0x_2 \leq 5$

   ➔ $x_1 \leq 5/2$

   ➔ $x_1 \leq 2$

Cutting Plane Algorithms

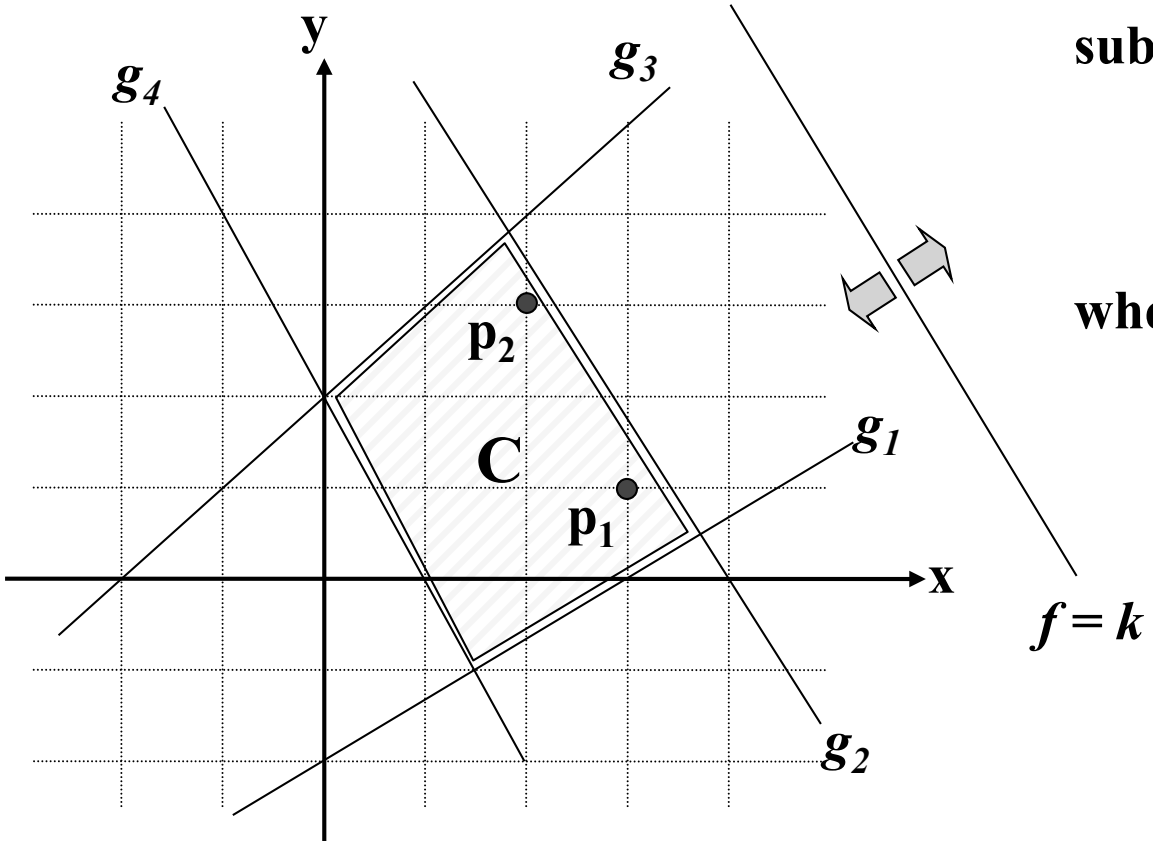
1. Using LP relaxation to find an optimal solution
   - e.g. Simplex
   - If the solution is integral, done.
2. Using cutting plane algorithms to find a (set of) valid inequality(ies)
3. Adding the valid inequalities to the constraints
   - Perhaps with some simplification (e.g. removing redundant inequlities)
4. Repeat 1

Cutting Plane Algorithms

- There are many other types of cuts
  - 0-1 Knapsack, odd hole, lift and project,...
  - ➔ More to be covered in “Discrete Optimization” class

- How efficient are these algorithms for PB problem? (i.e. 0-1 ILP problem)
  - How many iterations?
  - DPLL (Branch-and-bound) for PB problem?

FYI: Branch-and-Cut for ILP problems



maximize f: $12x + 7y$

subject to g_1: $2x - 3y \leq 6$

$2 \quad \leq$

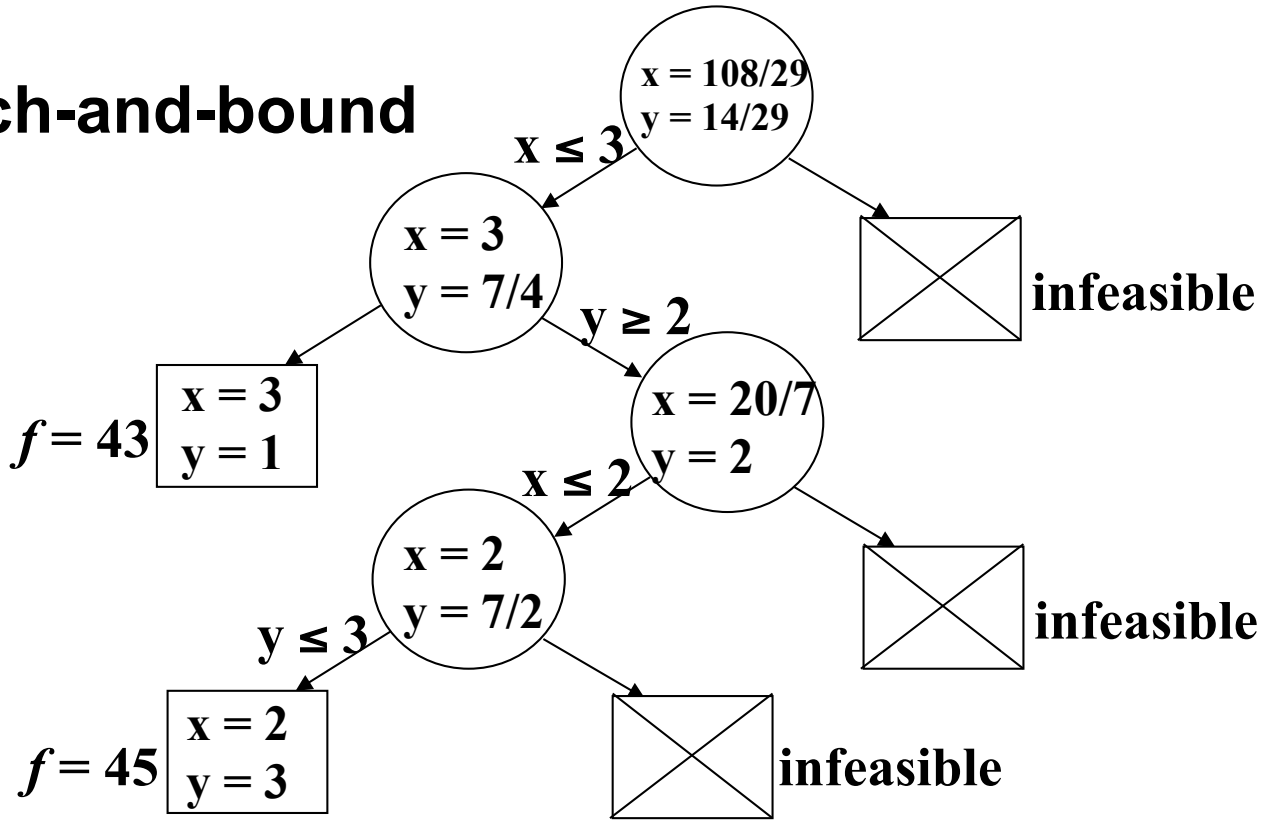
$3 \quad \leq$

$4 \quad \leq$

where $x, y \in Z$

FYI: Branch-and-Cut for ILP problems

Branch-and-bound



However, #decisions can be exponential...

Branch-and-cut = branch-and-bound + cutting plane

But PB is 0-1 ILP. Any better algorithm?

Compare: DPLL and new SAT algorithms

- What make the modern DPLL-based SAT solvers efficient ---
  1. Efficient BCP
  2. Conflict-driven learning with non-chronological backtracking
  3. Clause/Circuit reduction/simplification

Can PB solvers have the counterparts?

Let's summarize the types of PB solvers first...

1. Pure SAT-based method
   - Converting PB constraints to CNF
   - e.g. miniSat+
2. ILP-based
   - Cutting plane, branch and cut, etc.
   - e.g. CPLEX
3. Hybrid method (SAT + ILP)
   - e.g.
   1. Donald Chai and Andreas Kuehlmann, "A Fast Pseudo-Boolean Constraint Solver", TCAD 2005
   2. Hossein M. Sheini and Karem A. Sakallah, "Pueblo: A Hybrid Pseudo-Boolean SAT Solver", JSAT 2006

Hybrid PB Solver Algorithm

```
While (MakeDecision() != done) {
    while (PBCP() == conflict) {
        CNF_Learning();
        PB_Learning();
        if (learning.conflict())
            return UNSAT;
    }
}
return SAT;
```

Pseudo Boolean Constraint Propagation (PBCP)

- Let the PB constraint be normalized so that:
  - $a_1x_1 + a_2x_2 + \ldots + a_nx_n \geq k$, where $a_i, k \in Z^+, x_i \in B$
  - Let $\alpha = \Sigma_{(x_i \text{ != } 0)} a_i$ ..... coefficients of non-zero terms
- Let s denote the slack of a constraint that
  - $s = \alpha - k$
  - If $s < 0$ ➔ UNSAT
- To make sure no conflict
  - $\alpha \geq k$, i.e. $s \geq 0$
- An indirect implication is generated if ---
  - $s = (\alpha - k) < a_{max}$, where a_{max} = max unassigned literal

  ➔ $x_{i|a_{max}} = 1$

Pseudo Boolean Constraint Propagation (PBCP)

- Example:

  $6x_1 + 5x_2 + 5x_3 + 3x_4 + 2x_5 + 2x_6 + x_7 \geq 12$

  - $s = 24 - 12 > 0$
- Assign: $x_3 = 0$, $x_4 = 0$
  - $\alpha = 16 \geq 12$ ➔ no conflict
- However ---
  - $s = (16 - 12) < 6$, where $a_{max} = a_1 = 6$

  ➔ Indirect implication $x_1 = 1$
- Then ---
  - $s = (16 - 12) < 5$, where $a_{max} = a_2 = 5$

  ➔ Indirect implication $x_2 = 1$

➔ Multiple implications are inferred

Watch Scheme for PBCP

- Can we do “watch” for PBCP?
- Dynamic watch
  - Let L_w be a set of watch literals such that
  - Every literal in L_w is non-negative (1 or x)
  - $\Sigma_{(x_i \in L_W)} a_i = \alpha_w \geq k + a_{max}$

  ➔ However, since α_w will change along with implications, L_w needs to be updated and thus the number of watch literals will also change
- All watch
  - Watch all literals
- Static watch (ref: QuteSat)
  - A conservative but more efficient than all and dynamic watches

Conflict-Driven Learning

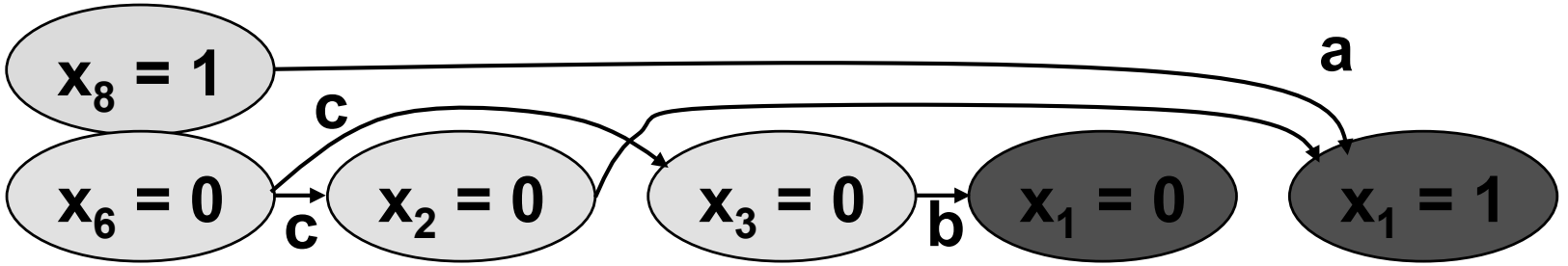
- How to perform conflict-driven learning for PB?
  - Learn a clause or an inequality?
- The bottom line is, any learned constraint must exhibit the following properties:

1. The learned constraint must remain in conflict under the current partial assignment.
   - This ensures that we backtrack from the conflict

2. After backtracking from the conflict, there must exist some decision level at which the constraint will generate one or more implications for the respective partial assignments

Problematic Conflict Driven Learning

- $3x_1 + 2x_2 + x_7 + 2\overline{x}_8 \geq 3$ (a)
  $3\overline{x}_1 + x_3 + x_5 + x_9 \geq 3$ (b)
  $\overline{x}_2 + \overline{x}_3 + x_6 \geq 2$ (c)
- Assume
  - $x_8 \leftarrow 1$ @ some previous decision level
  - $x_6 \leftarrow 0$ @ current decision level
- Implication graph



- Deriving learned constraint

(d) $2x_2 + x_7 + 2\overline{x}_8 + x_3 + x_5 + x_9 \geq 3$ (a)+(b) // remove x_1

(e) $x_2 + x_7 + 2\overline{x}_8 + x_5 + x_9 + x_6 \geq 3$ (d)+(c) // remove x_3

(f) $\overline{x}_3 + x_7 + 2\overline{x}_8 + x_5 + x_9 + 2x_6 \geq 4$ (e)+(c) // remove x_2

➔ **Does not conflict with ($x_8 = 1$, $x_6 = 0$)**

A closer look...

- Deriving learned constraint

(a) $3x_1 + 2x_2 + x_7 + 2\overline{x}_8 \geq 3$

(b) $3\overline{x}_1 + x_3 + x_5 + x_9 \geq 3$

slack (c) $\overline{x}_2 + \overline{x}_3 + x_6 \geq 2$

---

0 (d) $2x_2 + x_7 + 2\overline{x}_8 + x_3 + x_5 + x_9 \geq 3$ // (a)+(b)

0 (e) $x_2 + x_7 + 2\overline{x}_8 + x_5 + x_9 + x_6 \geq 3$ // (d)+(c)

---

0 (f) $\overline{x}_3 + x_7 + 2\overline{x}_8 + x_5 + x_9 + 2x_6 \geq 4$ // (e)+(c)

➔ Remember, if slack ≥ 0, no conflict!!

➔ Over satisfied!!

➔ Need to “weaken” the PB constraint!!

Weakening the PB constraints

(a) $3x_1 + 2x_2 + x_7 + 2\overline{x}_8 \geq 3$

(b) $3\overline{x}_1 + x_3 + x_5 + x_9 \geq 3$

➔ Removing non-0 literal // (b) + "$\overline{x}_9 \geq 0$"

➔ $3\overline{x}_1 + x_3 + x_5 \geq 2$

➔ Saturating x_1

(b') $2\overline{x}_1 + x_3 + x_5 \geq 2$

slack (c) $\overline{x}_2 + \overline{x}_3 + x_6 \geq 2$

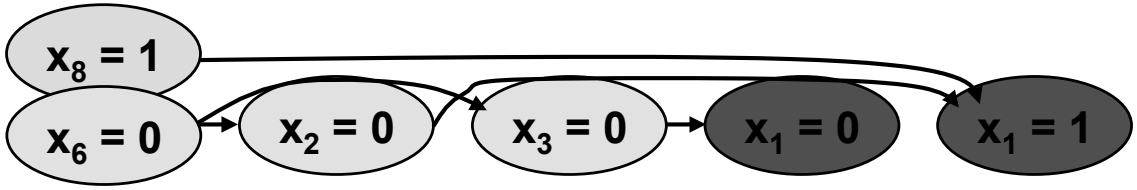
-1 (d) $4x_2 + 2x_7 + 4\overline{x}_8 + 3x_3 + 3x_5 \geq 6$ // 2(a)+3(b')

-1 (e) $x_2 + 2x_7 + 4\overline{x}_8 + 3x_5 + 3x_6 \geq 6$ // (d)+3(c)

-1 (f) $\overline{x}_3 + 2x_7 + 4\overline{x}_8 + 3x_5 + 4x_6 \geq 7$ // (e)+(c)

➔ Maintain conflict with ($x_8 = 1$, $x_6 = 0$)!!

Conflict-Driven Learning

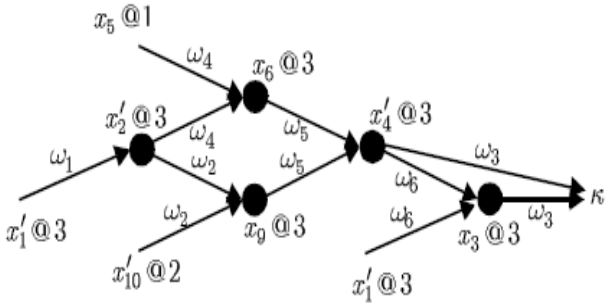


1. CNF learning
   - In the previous example, we can learn a CNF clause ($\overline{x}_8 + x_6$) by traversing on the implication graph

     ➔ Does not depend on the resolutions on the PB constraints
   - Similar to SAT by ---
     - Record the implication sources in PBCP
     - Backtrack to the UIP and learn a clause
2. PB learning
   - By applying resolutions and weakening on the PB constraints

Learning CNF and PB constraints simultaneously

$\omega_1 : 3x_1 + 2x'_2 + x_3 + x'_4 \geq 3$
$\omega_2 : 2x_2 + 2x_{10} + x_8 + x_9 \geq 2$
$\omega_3 : x'_3 + x_7 + x_4 \geq 2$
$\omega_4 : x_2 \vee x'_5 \vee x_6$
$\omega_5 : x'_6 \vee x'_4 \vee x'_9$
$\omega_6 : x_1 \vee x_3 \vee x_4$



Step	Implication Graph	var to Elim.	PB Learning	CNF Learning X_F
0		—	$x'_3 + x_7 + x_4 \geq 2$	$x'_3 \vee x_4$
1		x_3	$x'_3 + x_7 + x_4 \geq 2$ $x_1 \vee x_3 \vee x_4$ $x_1 + 2x_4 + x_7 \geq 2$	$x_1 \vee x_4$
2		x_4	$x_1 + 2x_4 + x_7 \geq 2$ $x'_6 \vee x'_4 \vee x'_9$ $x_1 + 2x'_6 + 2x'_9 + x_7 \geq 2$	$x_1 \vee x'_6 \vee x'_9$
3		x_6	$x_1 + 2x'_6 + 2x'_9 + x_7 \geq 2$ $x_2 \vee x'_5 \vee x_6$ $x_1 + 2x_2 + 2x'_5 + 2x'_9 + x_7 \geq 2$	$x_1 \vee x_2 \vee x'_5 \vee x'_9$
4		x_9	$x_1 + 2x_2 + 2x'_5 + 2x'_9 + x_7 \geq 2$ $2x_2 + 2x_{10} + x_8 + x_9 \geq 2$ $x_1 + 4x_2 + 4x_{10} + 2x'_5 + x_7 + 2x_8 \geq 4$	$x_1 \vee x_2 \vee x'_5 \vee x_{10}$
5		x_2	$x_1 + 4x_2 + 4x_{10} + 2x'_5 + x_7 + 2x_8 \geq 4$ $x'_1 \rightarrow x'_2$ $4x_1 + 4x_{10} + 2x'_5 + x_7 + 2x_8 \geq 4$	$x_1 \vee x'_5 \vee x_{10}$

□ implication @ conflict level (3)
△ assignment @ previous level
● decision @ conflict level

source: "Pueblo", JSAT 2006

Summary on PB Solver

- Many applications can be modeled as PB (0-1 ILP) problems
  - Logic optimization, verification, routing, operational research (OR), etc.
- Great attention on research these years
  - e.g. PB Evaluation:

    http://www.cril.univ-artois.fr/PB09/
- To be combined with word-level arithmetic, first-order logic, theorem proving,... techniques
  - SMT: Satisfiability Modulo Theory
  - e.g. SMT Competition:

    http://www.csl.sri.com/users/demoura/smt-comp/

Summary of SAT and Its Applications

- Remember: SAT is to answer the satisfiability problem of a proposition. Don't use it (directly) to compute all the solutions, nor to represent a Boolean formula (e.g. the set of reachability)
- The spirits of SAT solving are:
  - Local/greedy search
  - Conflict earlier, the better
  - Learn from the past
  - Lazy evaluation
  - Simplify from learned model
  - Profiling-based is the trend. Big data? (Haha)

To contact me…

- Any question? Please feel free to contact me…
  - Office: EE-II 444
  - E-mail: cyhuang@ntu.edu.tw
  - Tel: 02-3366-3644
  - Easiest ways to find me…

    ric2k1 on almost major media
    (PTT, P2, Skype FB, Line, WeChat…)