Introduction to Boolean Satisfiability (SAT)

資料結構與程式設計

Data Structure and Programming

12/09/2015

Introduction to Boolean Satisfiability (SAT)

A fundamental problem in computer science

- Given a Boolean network F: $B^n \rightarrow B$,
  where B = { 0, 1 }, and
  n is the number of inputs I = { $x_1, x_2, \ldots, x_n$ }
- Boolean Satisfiability
  ➔ Finding an input assignment

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

  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.

$$\underline{(a+b+c)}\,\underline{(a'+b'+c)}\,\underline{(a'+b+c')}\,\underline{(a+b'+c')}$$

$\boxed{\text{Variables}}$ $\boxed{\text{Literals}}$ $\boxed{\text{Clauses}}$

   - To be satisfied, all the clauses should be ‘1’

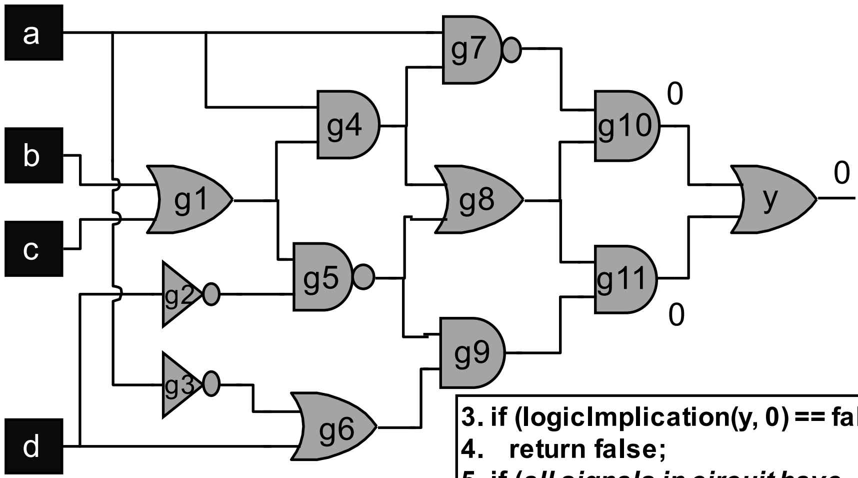
2. Circuit-Based
   - Boolean function is represented as a circuit netlist
   - SAT algorithm is directly operated on the netlist

A Very Basic (Circuit-Based) SAT Algorithm

```
bool sat(Gate g, Value v)
{
   if (logicImplication(g, v) == false)
      return false;
   if (all signals in circuit have been implied)
      return true;
   pick an unassigned signal s
   if (sat(s, V0) == true)
      return true;
   backtrack(s);
   if (sat(s, ~V0) == true)
      return true;
   backtrack(s);
   return false;
}
```

Proving always(y == 1)

sat(y, 0)



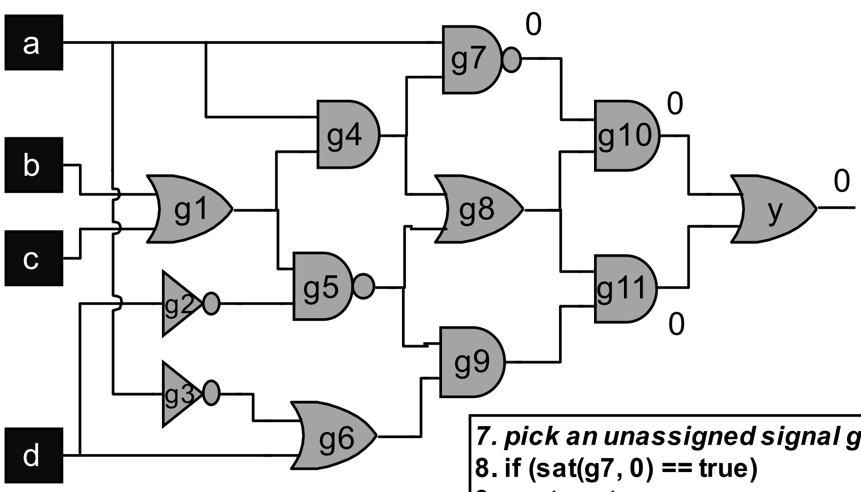

```
3. if (logicImplication(y, 0) == false)
4.    return false;
5. if (all signals in circuit have
6.    been implied)   return true;
```


A Very Basic SAT Algorithm

```
1. bool sat(Gate g, Value v)
2. {
3.    if (logicImplication(g, v) == false)
4.       return false;
5.    if (all signals in circuit have been implied)
6.       return true;
7.    pick an unassigned signal s
8.    if (sat(s, V0) == true)
9.       return true;
10.   backtrack(s);
11.   if (sat(s, ~V0) == true)
12.      return true;
13.   backtrack(s);
14.   return false;
15. }
```

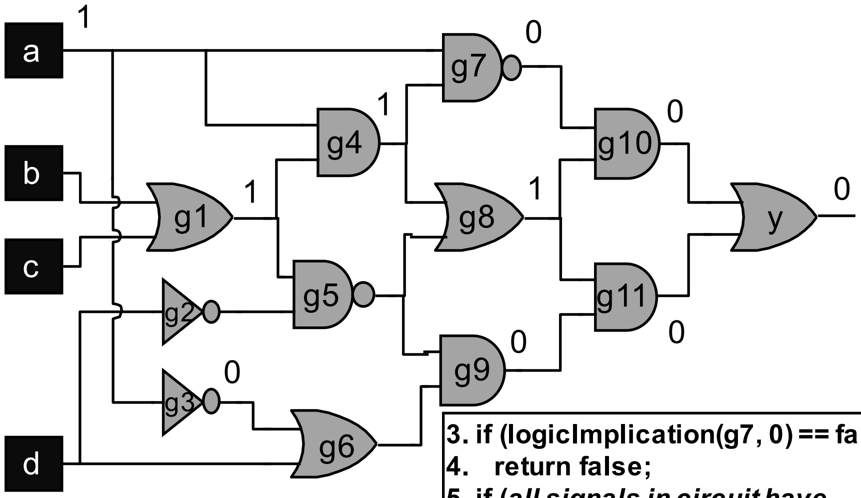
sat (y, 0)

sat (g7, 0)



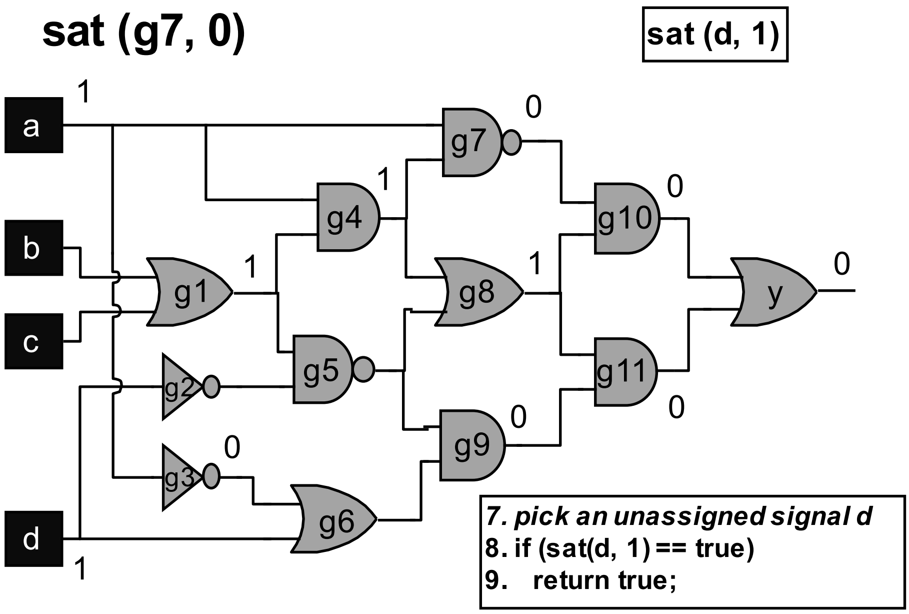
7. pick an unassigned signal g7
8. if (sat(g7, 0) == true)
9.   return true;

sat (g7, 0)



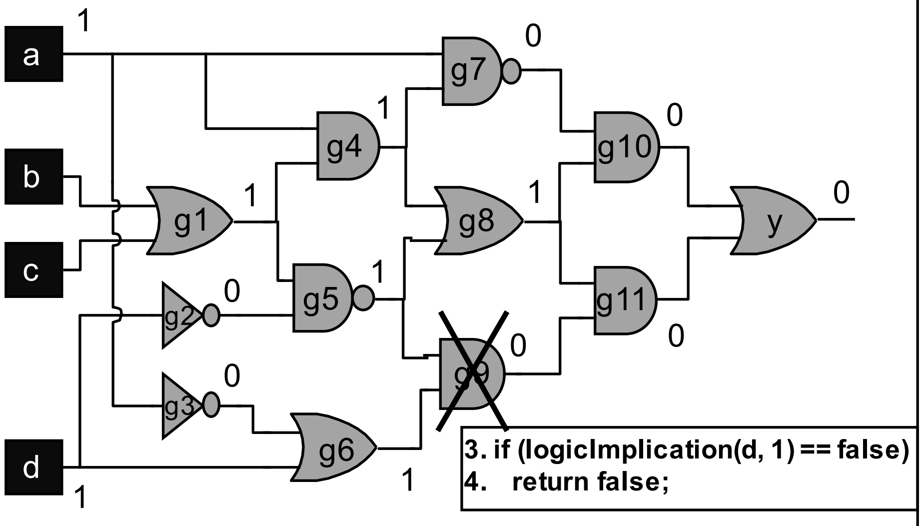
3. if (logicImplication(g7, 0) == false)
4.   return false;
5. if (*all signals in circuit have*
6.   *been implied*) return true;

sat (g7, 0)



7. pick an unassigned signal d
8. if (sat(d, 1) == true)
9. return true;

sat (d, 1)

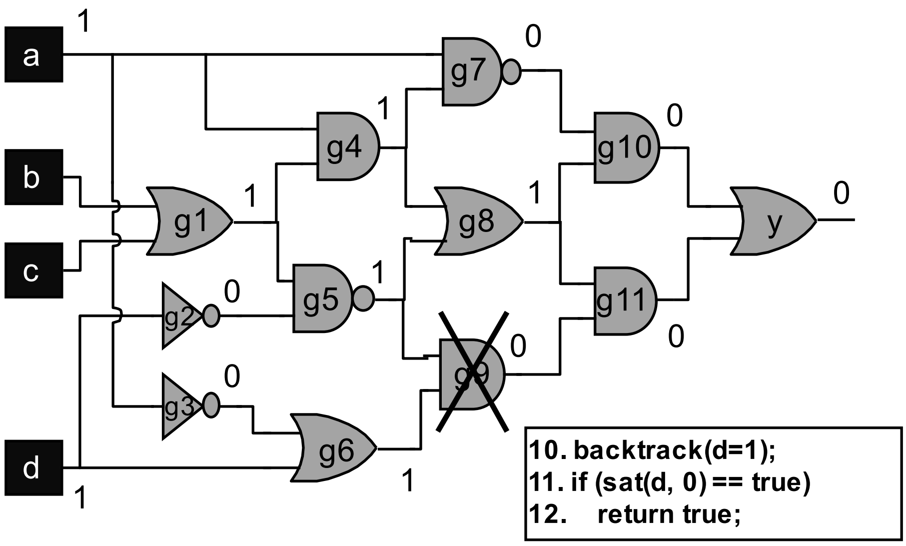


3. if (logicImplication(d, 1) == false)
4. return false;

A Very Basic SAT Algorithm

```
1. bool sat(Gate g, Value v)
2. {
3.    if (logicImplication(g, v) == false)
4.       return false;
5.    if (all signals in circuit have been implied)
6.       return true;
7.    pick an unassigned signal s
8.    if (sat(s, V0) == true)
9.       return true;
10.   backtrack(s);
11.   if (sat(s, ~V0) == true)
12.      return true;
13.   backtrack(s);
14.   return false;
15. }
```

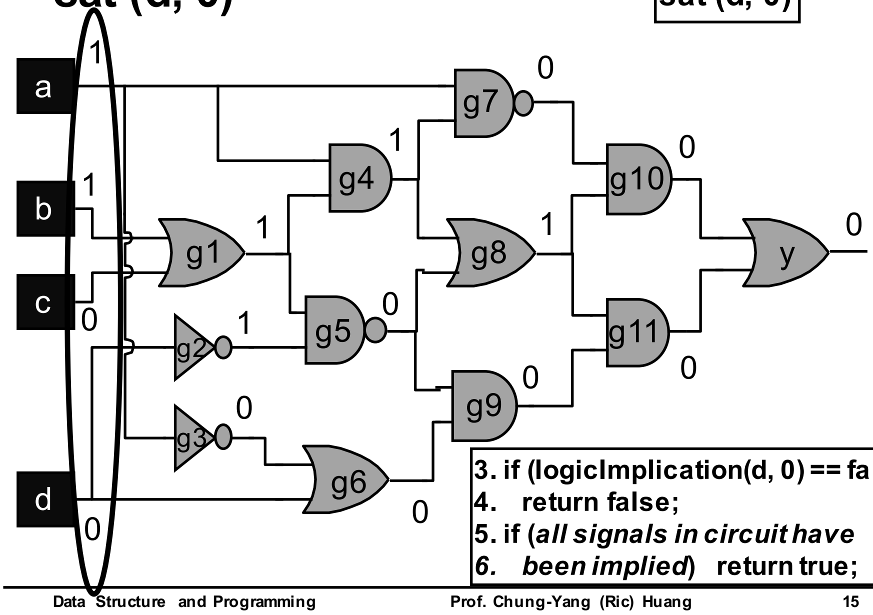

sat (d, 1)



10. backtrack(d=1);
11. if (sat(d, 0) == true)
12. return true;

sat (d, 0)

sat (d, 0)




```
3. if (logicImplication(d, 0) == false)
4.    return false;
5. if (all signals in circuit have
6.    been implied)   return true;
```


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

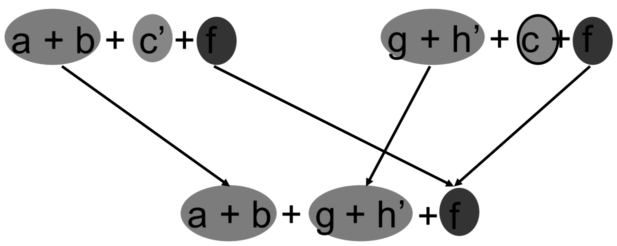
➔ In the following slides, we will focus on the easier-to-implement solver, CNF SAT, only.

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. Stålmarcks 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", *J. of ACM*, Vol. 7, pp. 201-214, 1960 (360 citations in citeseer)

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

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

(a + c + e)(c' + e + f)

(a + e + f)

SAT

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

(a + b) (a + b') (a' + c) (a' + c')

(a) (a' + c) (a' + c')

(c) (c')

( )

UNSAT

Potential memory explosion problem!

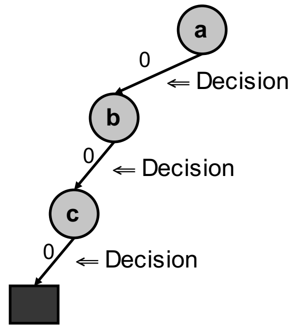
Souce: Prof. Sharad Malik, "The Quest for Efficient Boolean Satisfiability Solvers"

Boolean Satisfiability (SAT) Algorithm

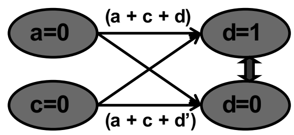
1. Davis, Putnam, 1960
   - Explicit resolution based
   - May explode in memory
2. Davis, (Putnam), Logemann, Loveland, (D(P)LL) 1962
   - Search based.
   - Most successful, basis for almost all modern SAT solvers
   - Learning and non-chronological backtracking, 1996
3. Stålmarcks algorithm, 1980s
   - 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)



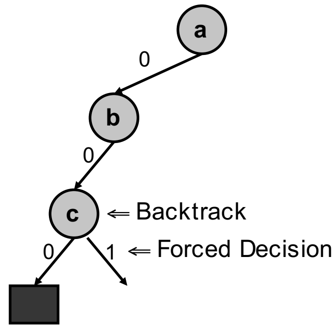
Implication Graph



Conflict!

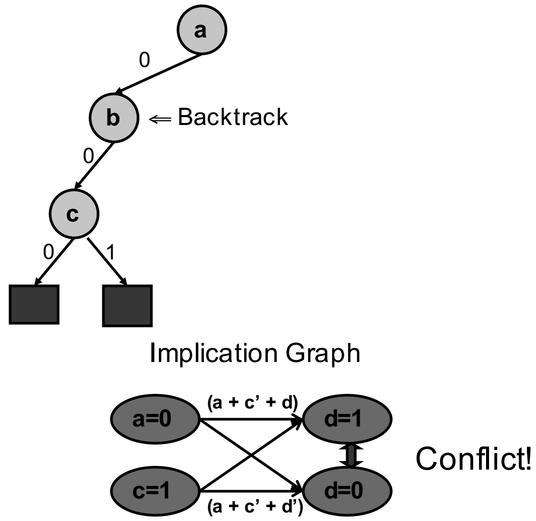
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



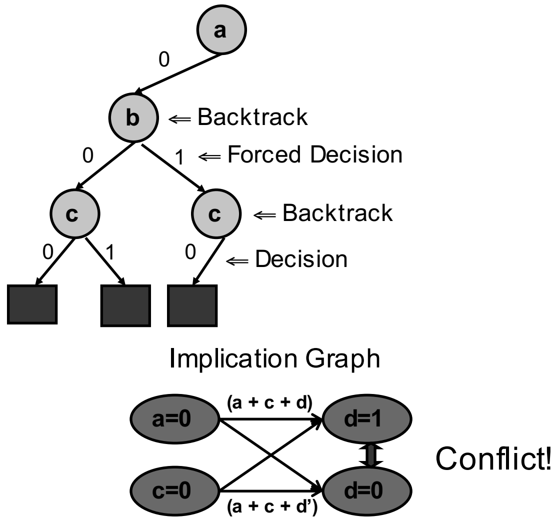
Basic DLL Procedure - DFS

- (a’ + b + c)
- (a + c + d)
- (a + c + d’)
- (a + c’ + d)
- (a + c’ + d’)
- (b’ + c’+ d)
- (a’ + b + c’)
- (a’ + b’ + c)



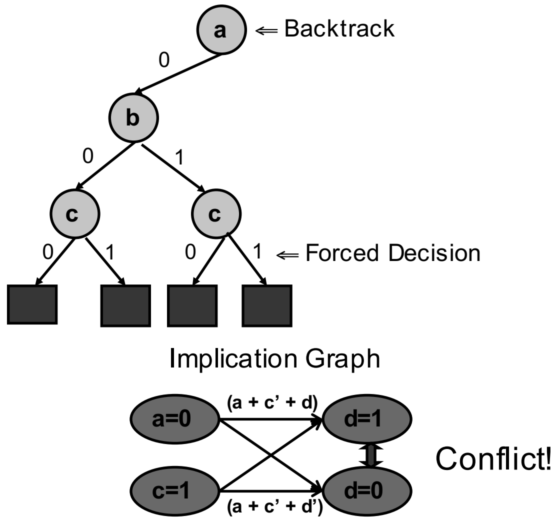
Basic DLL Procedure - DFS

- (a’ + b + c)
- (a + c + d)
- (a + c + d’)
- (a + c’ + d)
- (a + c’ + d’)
- (b’ + c’+ d)
- (a’ + b + c’)
- (a’ + b’ + c)



Basic DLL Procedure - DFS

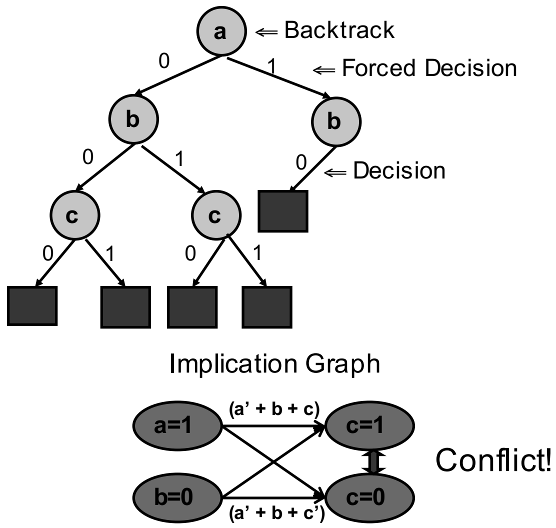
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



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)



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')

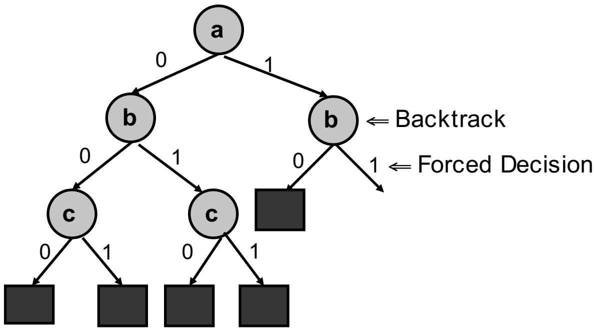
(a + c' + d)

(a + c' + d')

(b' + c' + d)

(a' + b + c')

(a' + b' + c)



Basic DLL Procedure - DFS

(a' + b + c)

(a + c + d)

(a + c + d')

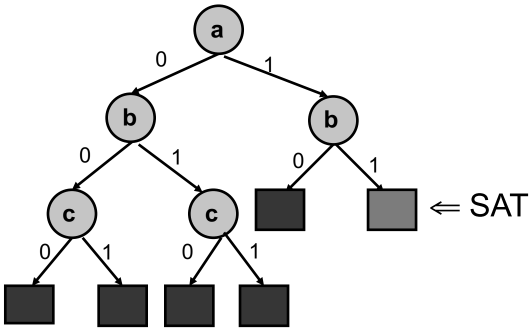
(a + c' + d)

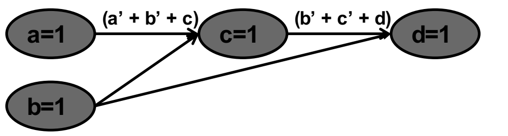
(a + c' + d')

(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)

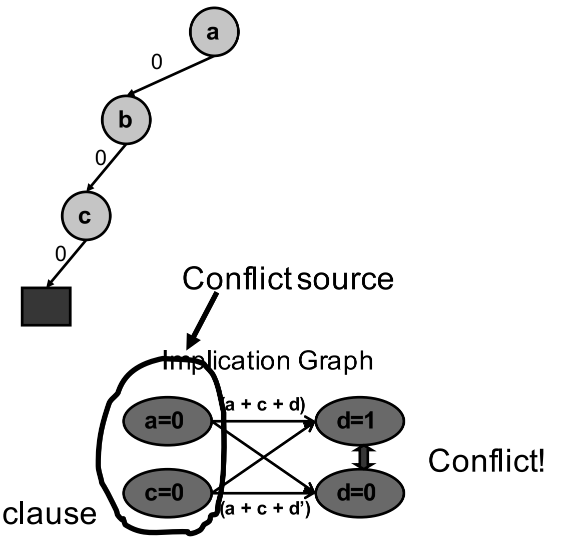
(a + c' + d')

(b' + c' + d)

(a' + b + c')

(a' + b' + c)

(a + c) Learned clause



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

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)

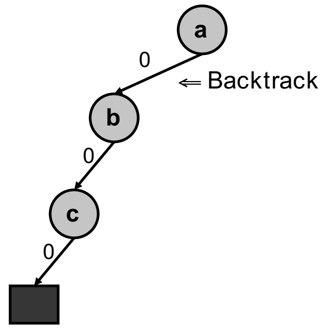
(a + c' + d')

(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

Deduced Implication from Learned Clause

(a' + b + c)

(a + c + d)

(a + c + d')

(a + c' + d)

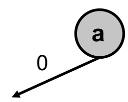
(a + c' + d')

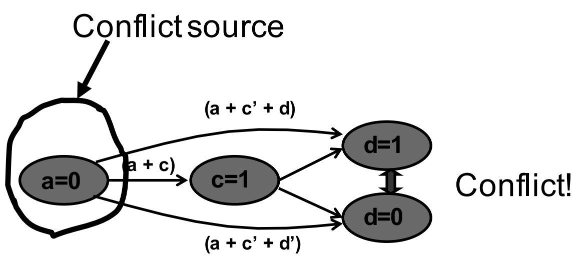
(b' + c' + d)

(a' + b + c')

(a' + b' + c)

(a + c)





Deduced Implication from Learned Clause

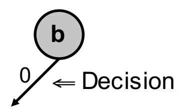
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)
(a + c)
(a) Learned clause

- Since there is only one variable in the learned clause

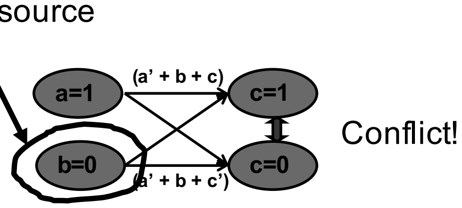
  → No one is the next-to-the-last variable
- Backtrack all decisions

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)



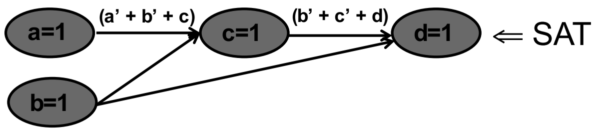
Conflict source



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

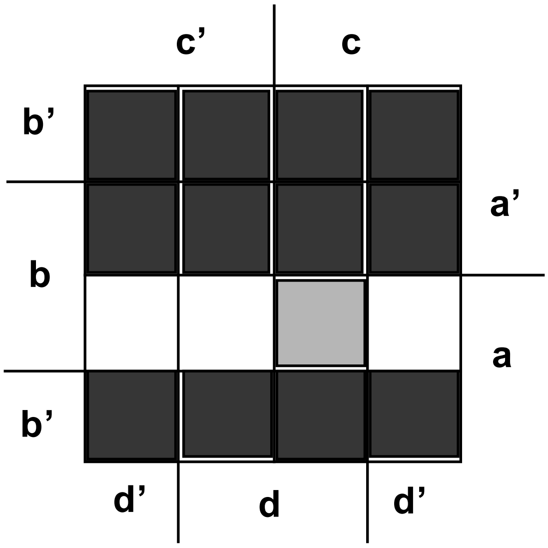
Deduced Implication from Learned Clause

(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)
(a) (b) Learned clause



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

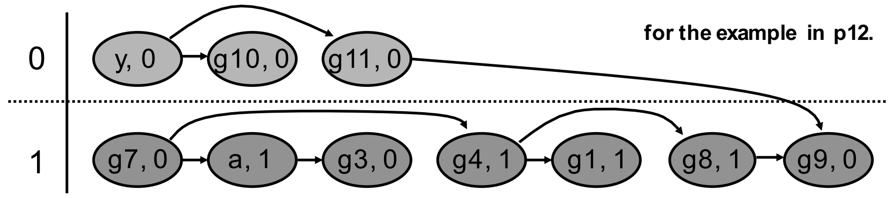
What does conflict learning tell us?



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!!

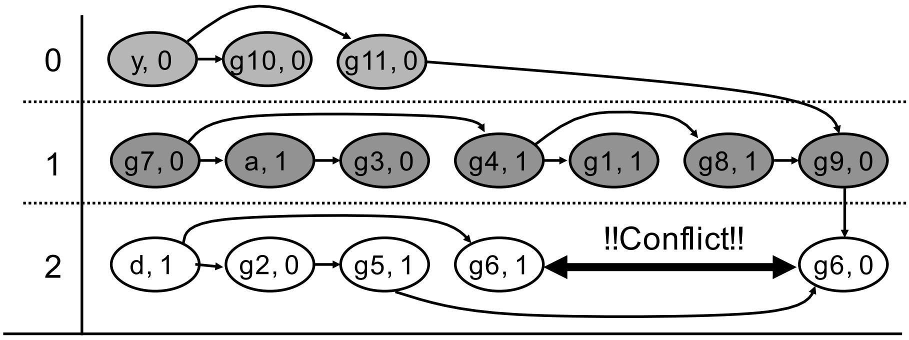
A Closer Look at the Implication Graph (a conceptual implementation)

- 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)

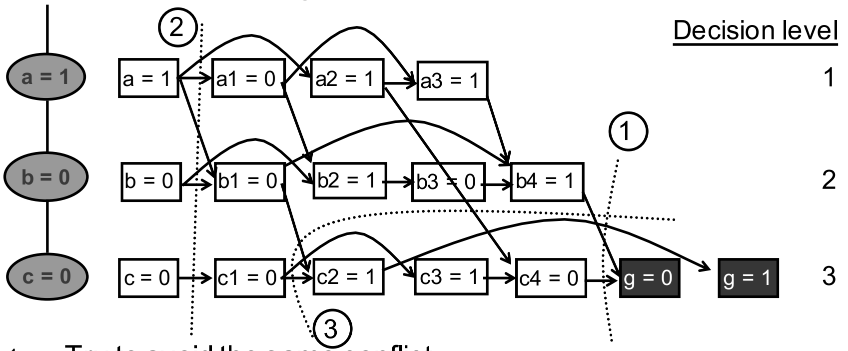


Conflict Analysis

- 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

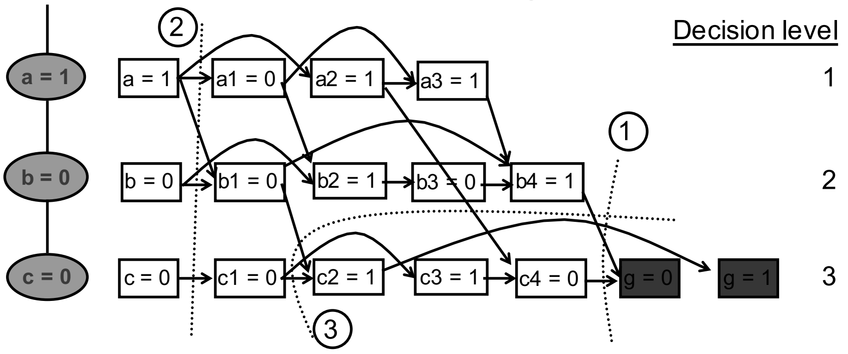


Conflict Analysis



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?

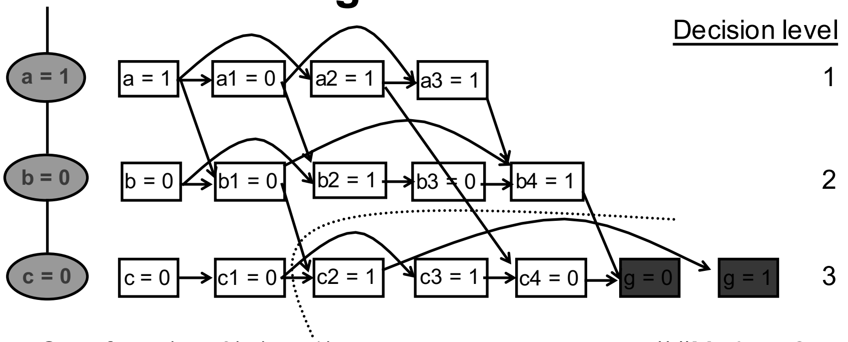
- [Zhang, *et al*, ICCAD 2001] Experiment shows that "first-UIP" (1st-UIP) is the best
  - UIP: Unique Implication Point
    - In a cut that there is only one node in the last (where conflict happens) decision level (why UIP cut?)
    - Starting from the conflict gate, the first encountered UIP is namely 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;
}
```

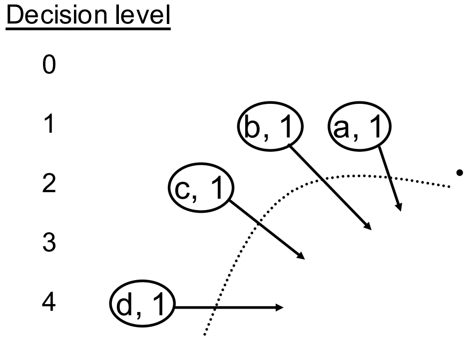
Linear-Time Algorithm to Find First UIP



- 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

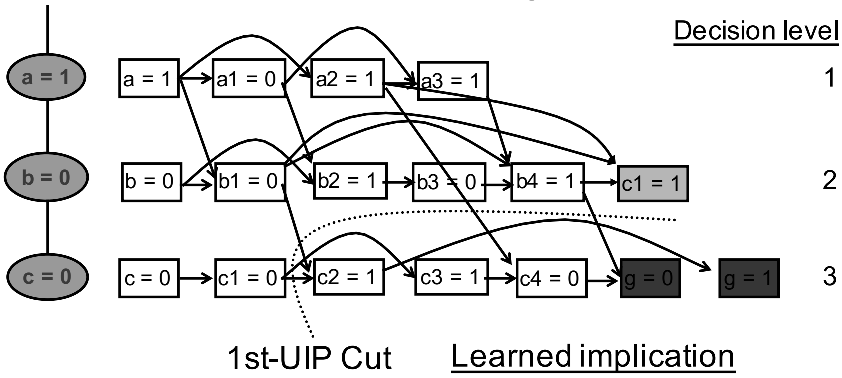
⇩

(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

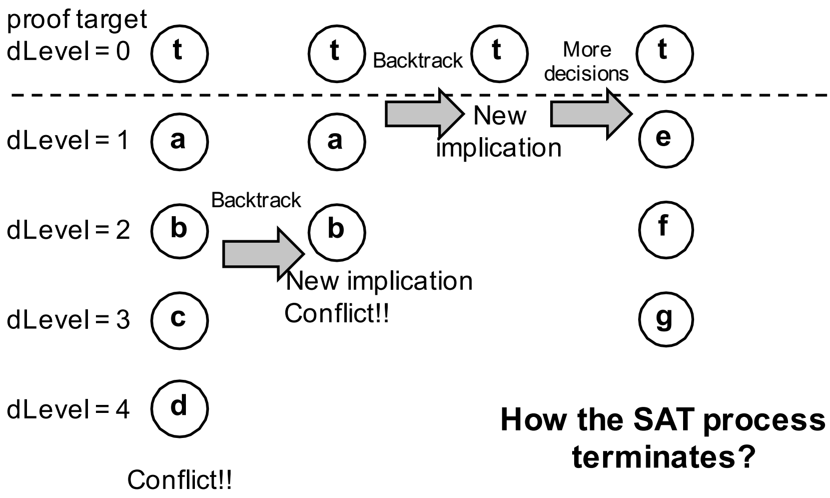
Conflict-Driven Learning



Learned implication

b1' && a2 && b4 ➔ c1

Conflict-Driven Non-Chronological Backtracking --- Algorithm



How the SAT process terminates?

Conflict-Driven Non-Chronological Backtracking --- Algorithm

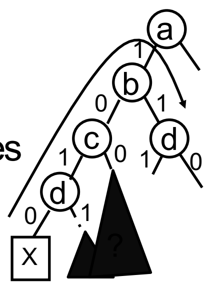
1. When conflict occurs, check if the conflict level == 0 (implication level for the SAT target)
   a) If yes, return *unsatisfiability* (Why?)
   b) Else, continue to 2
2. Find the 1st-UIP cut as the conflict causes
3. Backtrack to the max decision level of the nodes other than UIP in the cut
4. The 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?

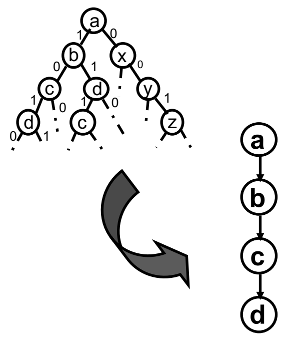
- May lead to SAT solution ealier
- But some portion of the decision tree may not be covered
  - Not a complete search anymore
  - May also miss some bugs

➔ 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



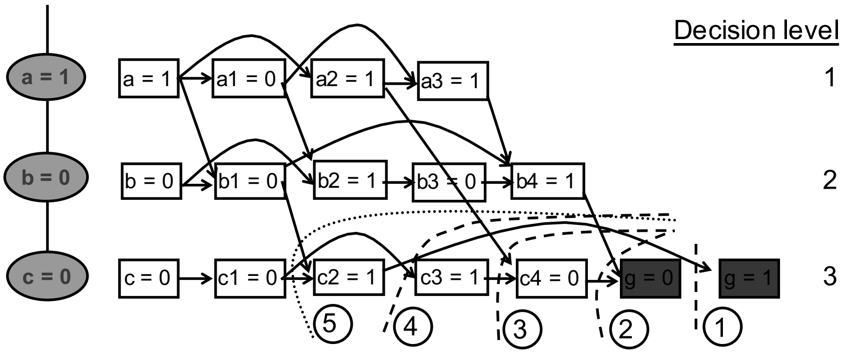
Conflict-Driven Non-Chronological Backtracking --- Completeness

- 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



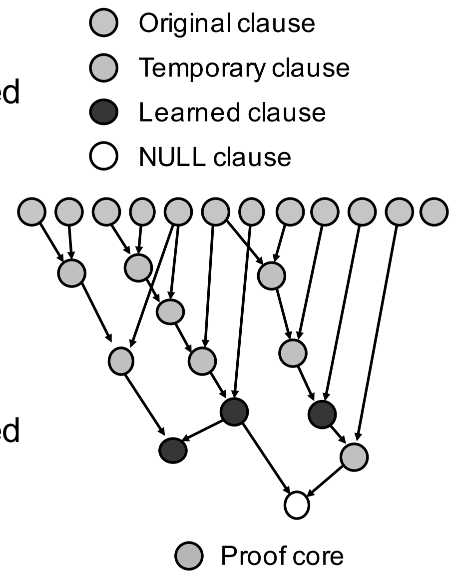
(1): (c2' + g)
(2): (b4' + c4 + g') → (b4' + c2' + c4)
(3): (a2' + c3' + c4') → (a2' + b4' + c2' + c3')
(4): (c1 + c3) → (a2' + b4' + c1 + c2')
(5): (b1 + c1 + c2) → (a2' + b1 + b4' + c1)

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?

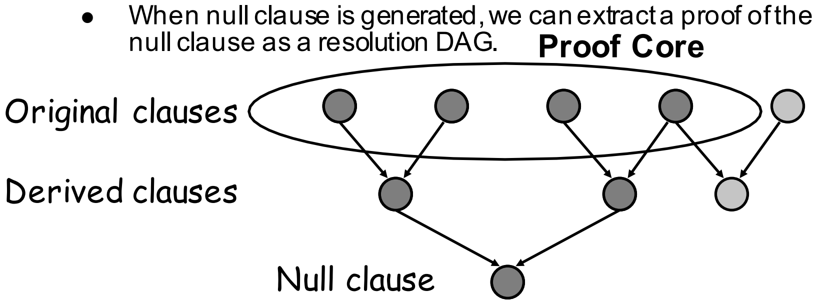
Resolution Graph

- 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.



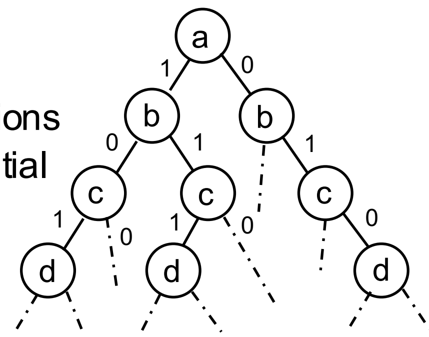
What affect the SAT efficiency?

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

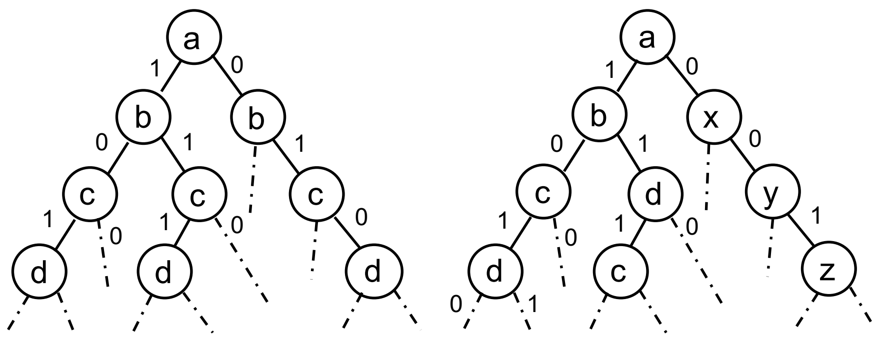
More decision heuristics...

- 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’

L1 + L2 + …+ L50 + L98 + L99 + L100

x → 0   x

seach for ‘x’

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 !!

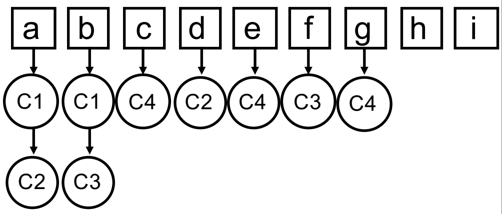
- a + b + **c** + d + **e** // b = 0; NO action
- a + b + c + **d** + **e** // c = 0; UPDATE watches !!
- a + b + c + **d** + **e** // Backtrack, NO action !!
- a + b + c + **d** + **e** // c = 0; NO action !!
- a + b + c + **d** + **e** // d = 1; NO action !!

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 ← 0

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

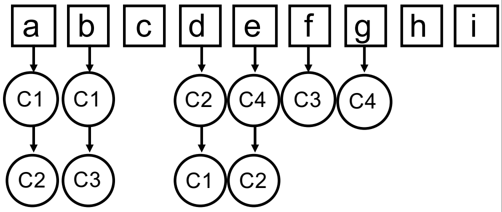
[Note] Don’t need to check ‘C1’

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)



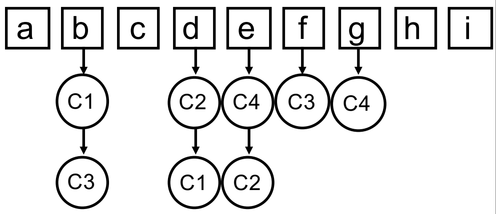
a ← 0
- Update watched literal pointer for C1 (only choice, to ‘d’)
- Update watched literal pointer for C2 (for example, to ‘e’)
- Erase a’s watching-clause list
- Add ‘C1’ to d’s and ‘C2’ to e’s watching-clause lists

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 ← 0
- No more unknown literal for C1 : d = 1
- No more unknown literal for C3 : f = 1

[Note] No change on watched literals

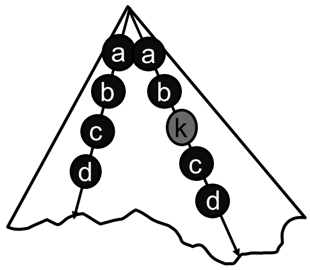
Caching Effect: Reducing from O(n) to almost O(C)

- 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.

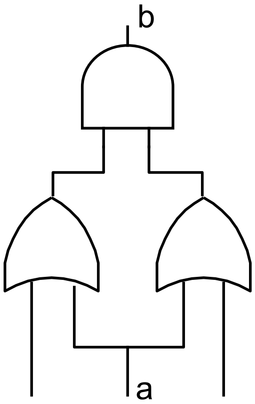
Various Learning Techniques

- 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
  ($a \rightarrow b \;\; \equiv \;\; !b \rightarrow !a$)
- e.g.



a = 1 → b = 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

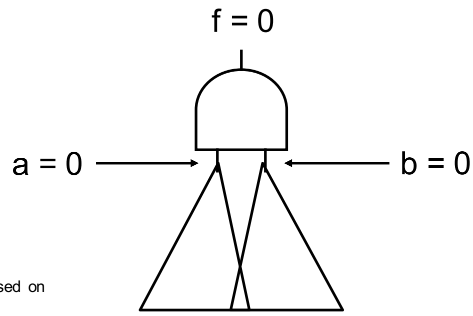
Learned by Signal Correlations

- 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

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
- Too many details, we will skip them here...

Appendix: Interpolation

Interpolation

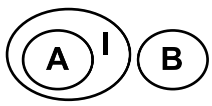
(Craig,57)

- If $A \wedge B$ = false, there exists an *interpolant* I for (A,B) such that:

$$A \Rightarrow I$$

$$I \wedge B = \text{false}$$

I refers only to common variables of A,B

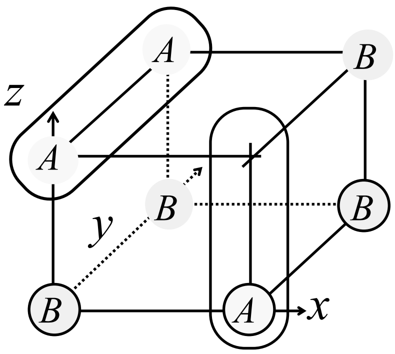


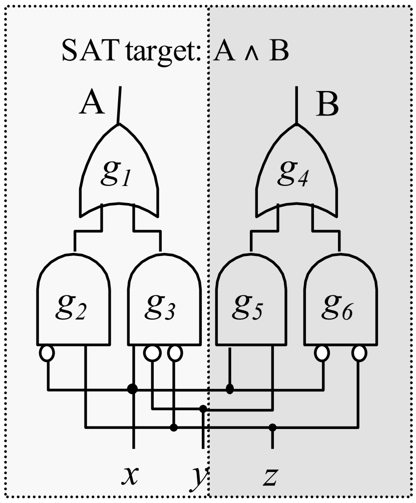
- Example:
  - $A = p \wedge q$, $B = \neg q \wedge r$, $A' = q$
- New result
  - given a resolution refutation of $A \wedge B$,
    A' can be derived in linear time.

(Pudlak,Krajicek,97)

Another Example of Interpolation

A: $(\neg x \wedge z) \vee (x \wedge \neg y \wedge \neg z)$
B: $(x \wedge y) \vee (\neg x \wedge \neg z)$

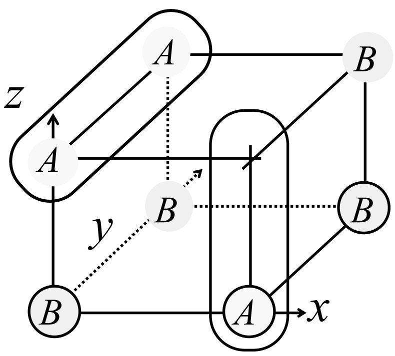


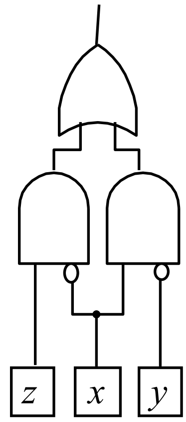


Another Example of Interpolation

A: $(\neg x \wedge z) \vee (x \wedge \neg y \wedge \neg z)$
B: $(x \wedge y) \vee (\neg x \wedge \neg z)$

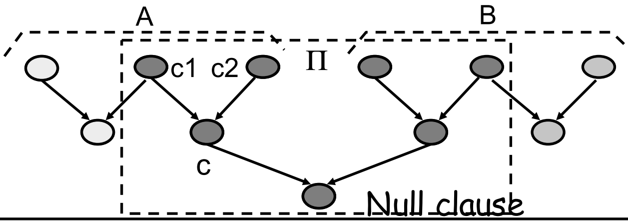
I = $(\neg x \wedge z) \vee (x \wedge \neg y)$





Some Definitions for Unsatisfiability Proof

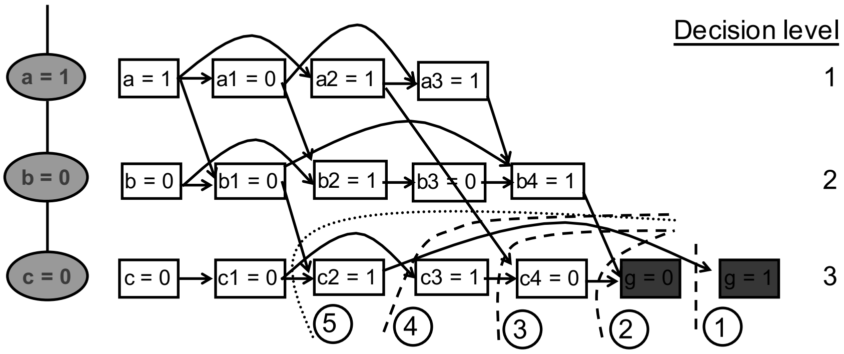
- Let (A,B) be a pair of clause sets and let Π be a proof of unsatisfiability of $A \cup B$
  - Π is a DAG (V_Π, E_Π)
  - Each vertex $c \in \Pi$ in the graph corresponds to a clause and has exactly 2 predecessors, say c1, c2
    - c is called the "resolvent" of c1 and c2
    - The resolved variable v is called the "pivot" variable
  - Π has exactly 1 leaf vertex which is a False (null clause)
  - The roots are original clauses in $A \cup B$



Some Definitions for Unsatisfiability Proof

- Let (A,B) be a pair of clause sets and let Π be a proof of unsatisfiability of $A \cup B$
  - Π is a DAG (V_Π, E_Π)
  - Each vertex $c \in \Pi$ in the graph corresponds to a clause and has exactly 2 predecessors, say c1, c2
    - c is called the "resolvent" of c1 and c2
    - The resolved variable v is called the "pivot" variable
  - Π has exactly 1 leaf vertex which is a False (null clause)
  - The roots are original clauses in $A \cup B$
- Global/Local variable/literal
  - With respect to (A,B), a variable/literal is global if it appears in both A and B
  - It is called local to A if it appears only in A

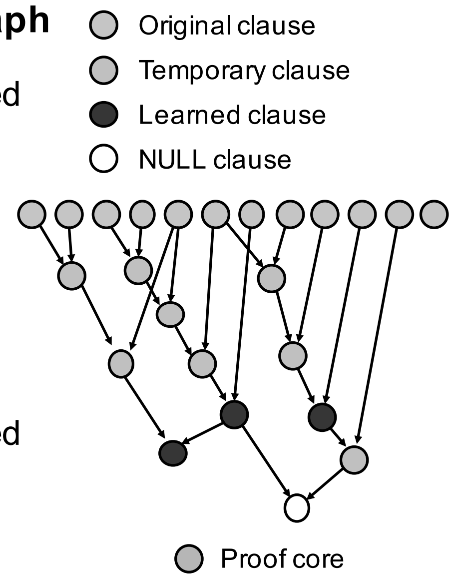
Again, Implication graph, resolution, and learning



(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)

Again, 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



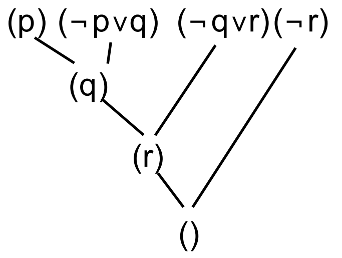
Interpolants from Proofs

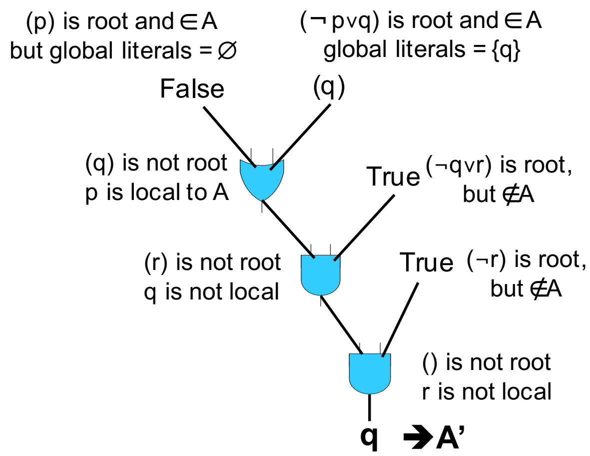
- Deriving interpolant from Π
  ➔ Calling itp(leaf vertex)
-

```
itp(c) { // c ∈ V_Π let p(c) be a
  if c is a root, then
    if c ∈ A then
      itp(c) = the disjunction of the
               global literals in c
    else itp(c) = constant True
  else, let c1, c2 be the predecessors of c
        and let v be their pivot variable
    if v is local to A
      then itp(c) = itp(c1) ∨ itp(c2)
    else itp(c) = itp(c1) ∧ itp(c2)
}
```


Interpolants from Proofs

A = (p)(¬p∨q) B = (¬q∨r)(¬ r)






```
itp(c) { // c ∈ VΠ let p(c) be a
  if c is a root, then
    if c ∈ A then
      itp(c) = the disjunction of the global literals in c
    else itp(c) = constant True
  else, let c1, c2 be the predecessors of c
       and let v be their pivot variable
    if v is local to A
      then itp(c) = itp(c1) ∨ itp(c2)
    else p(c) = p(c1) ∧ p(c2)
}
```