SAT-Solving: From Davis-Putnam to Zchaff and Beyond

Day 2: Efficient SAT Solving

Lintao Zhang

Microsoft Research

Davis Logemann Loveland Algorithm Framework

```
while(1) {
  if (decide_next_branch()) { //Branching
      while(deduce()==conflict) { //Deducing
            blevel = analyze_conflicts();
            if (blevel < 0)
                  return UNSAT;
            else back_track(blevel); //Backtracking
      }
  else //no branch means all variables got assigned.
      return SATISFIABLE;
  }
```

Chronological Backtracking

- Backtracking to the highest decision level that has not been tried with both values
- Originally proposed in the DLL paper in 1962
- OK for randomly generated instances, bad for instances generated in practical applications
- We can do better than that

Conflict Driven Learning and Non-Chronological Backtracking

- Marques-Silva and Sakallah [SS96,SS99]

  J. P. Marques-Silva and K. A. Sakallah, "GRASP -- A New Search Algorithm for Satisfiability,“ Proc. ICCAD 1996.

  J. P. Marques-Silva and Karem A. Sakallah, “GRASP: A Search Algorithm for Propositional Satisfiability”, *IEEE Trans. Computers*, C-48, 5:506-521, 1999.

- Bayardo and Schrag’s RelSAT also proposed conflict driven learning [BS97]

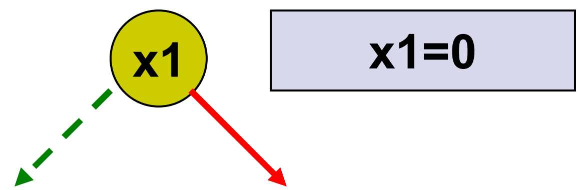
  R. J. Bayardo Jr. and R. C. Schrag “Using CSP look-back techniques to solve real world SAT instances.” *Proc. AAAI*, pp. 203-208, 1997

- Practical SAT instances can be solved in reasonable time

Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

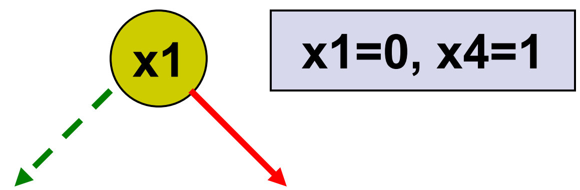
Conflict Driven Learning and Non-chronological Backtracking



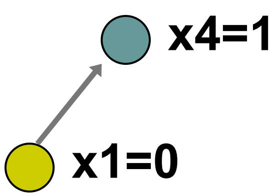
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

x1=0

Conflict Driven Learning and Non-chronological Backtracking

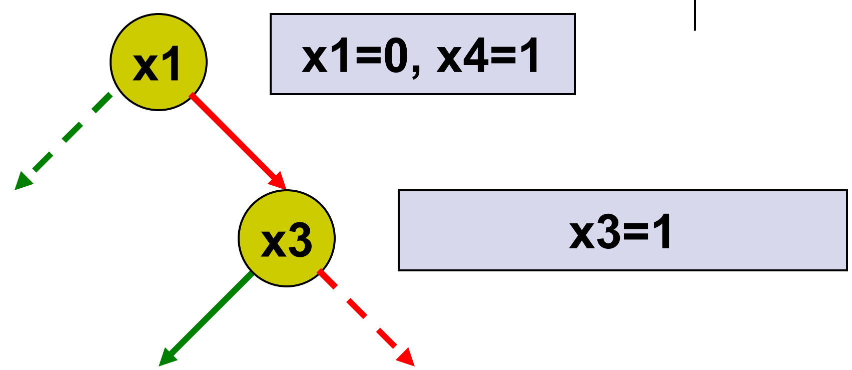


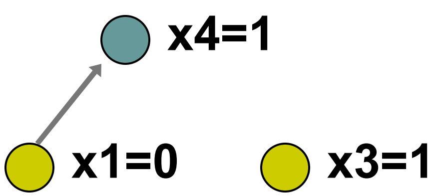
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’



Conflict Driven Learning and Non-chronological Backtracking

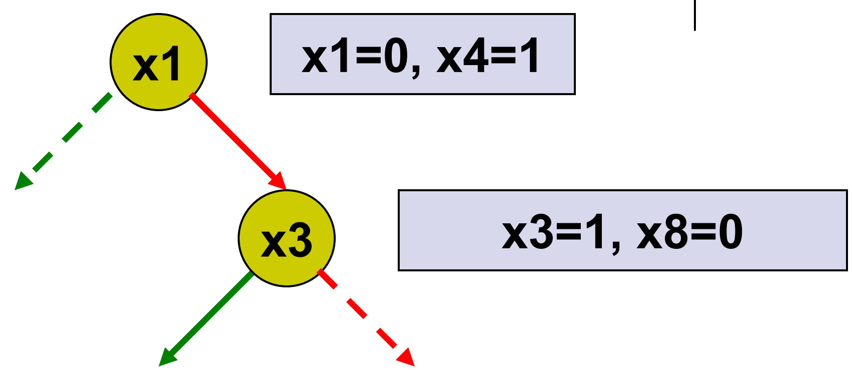
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

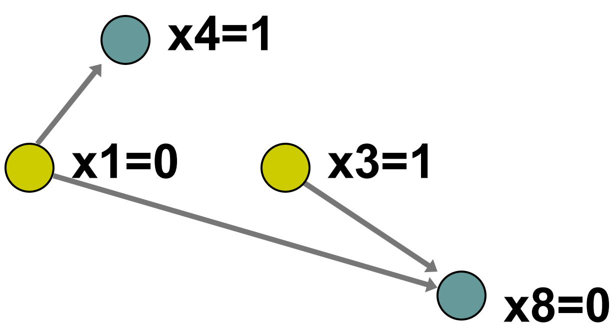




Conflict Driven Learning and Non-chronological Backtracking

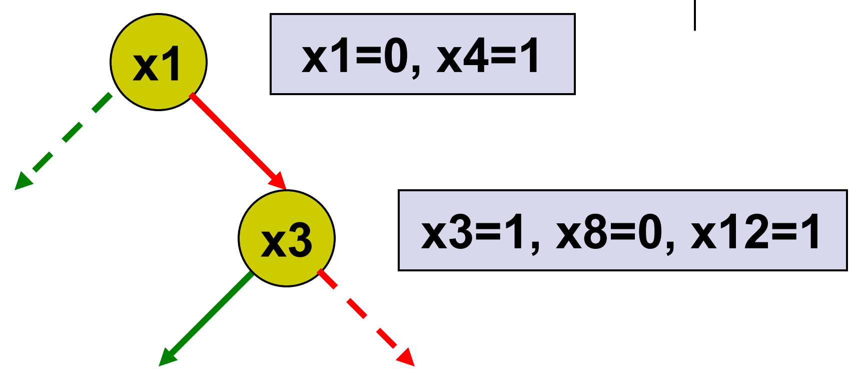
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

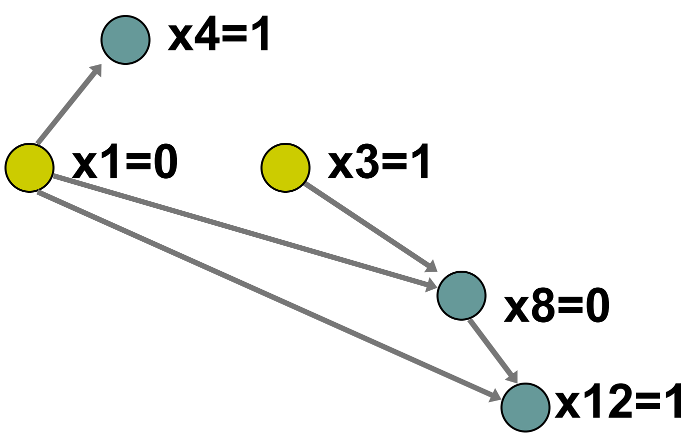




Conflict Driven Learning and Non-chronological Backtracking

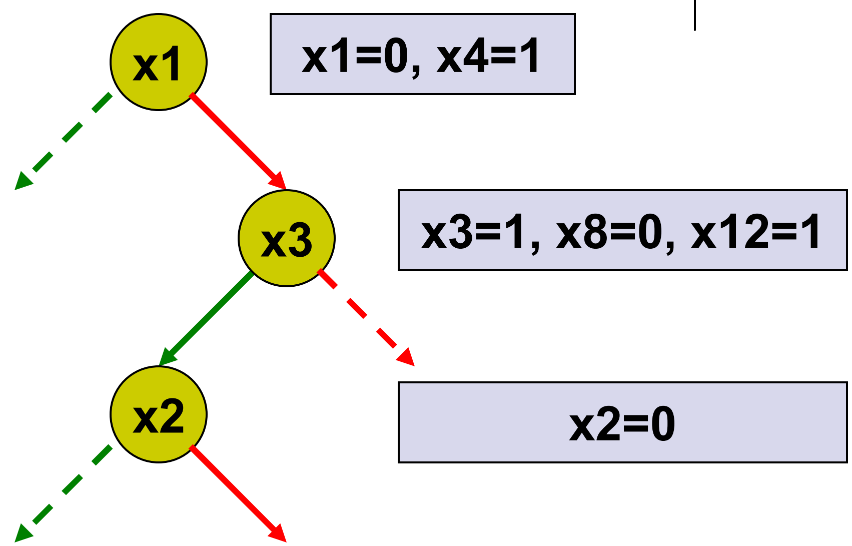
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

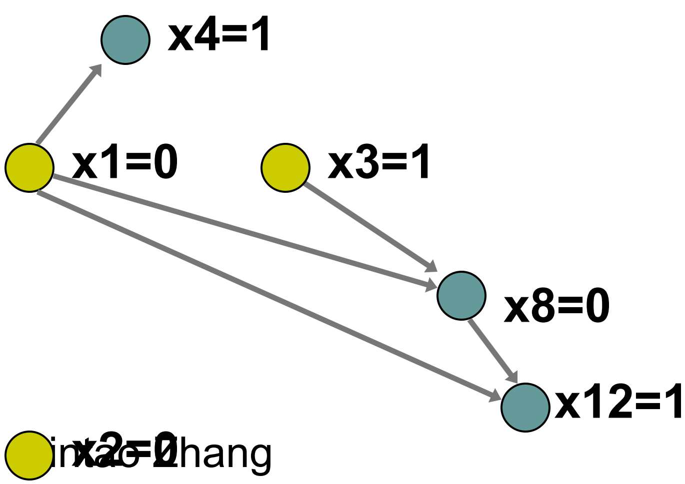




Conflict Driven Learning and Non-chronological Backtracking

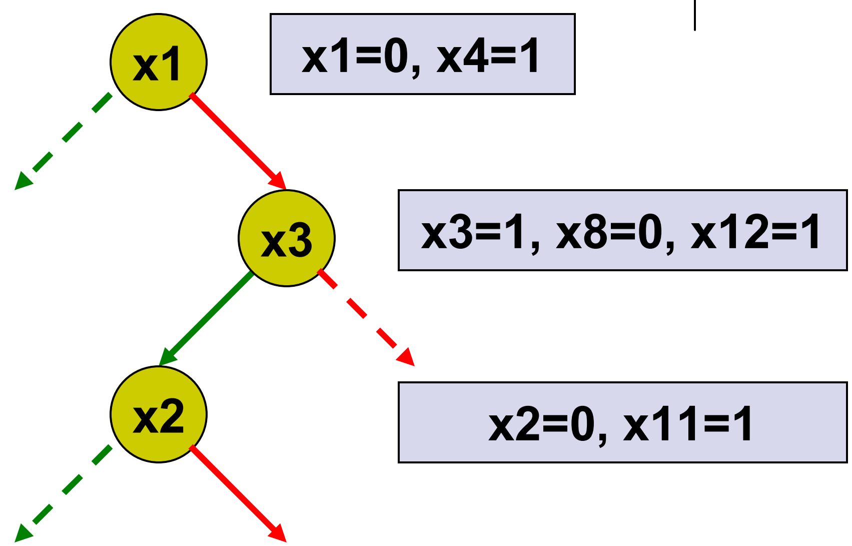
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

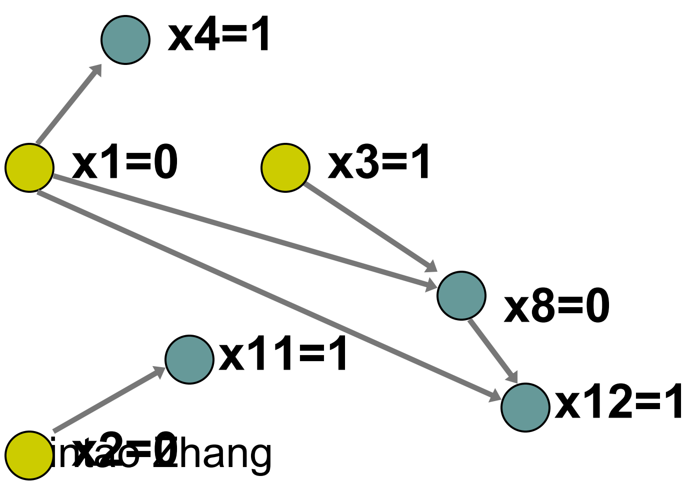




Conflict Driven Learning and Non-chronological Backtracking

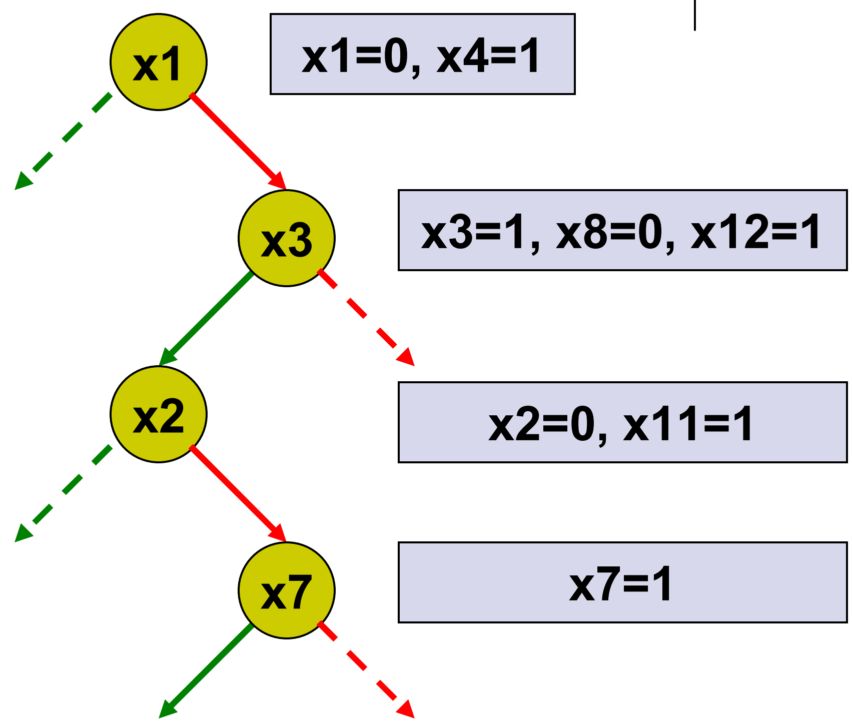
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

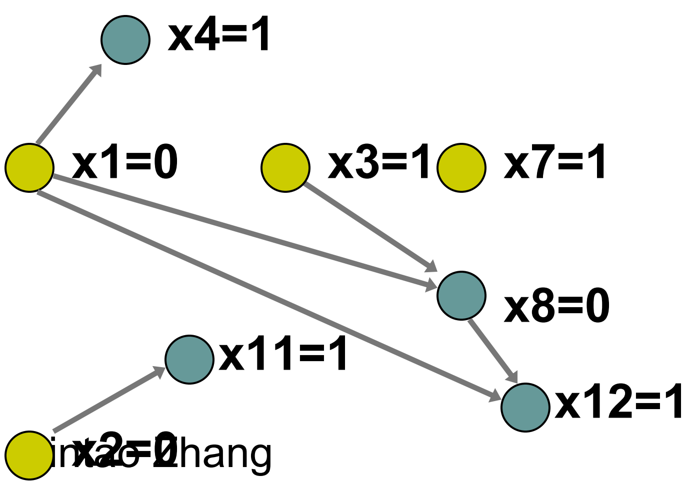




Conflict Driven Learning and Non-chronological Backtracking

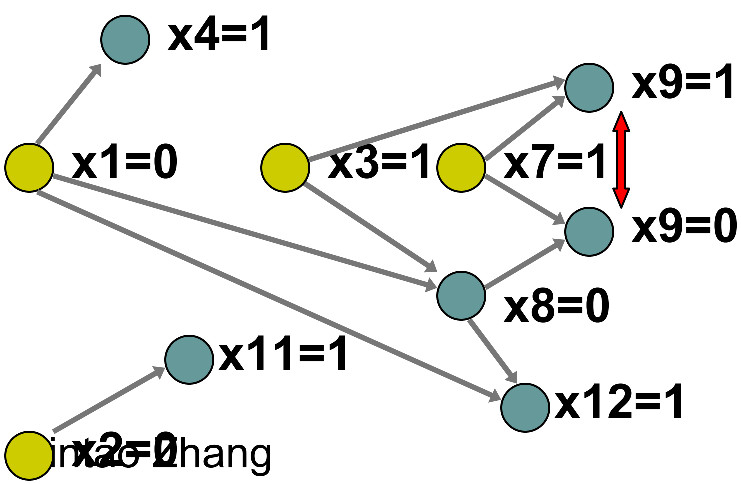
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

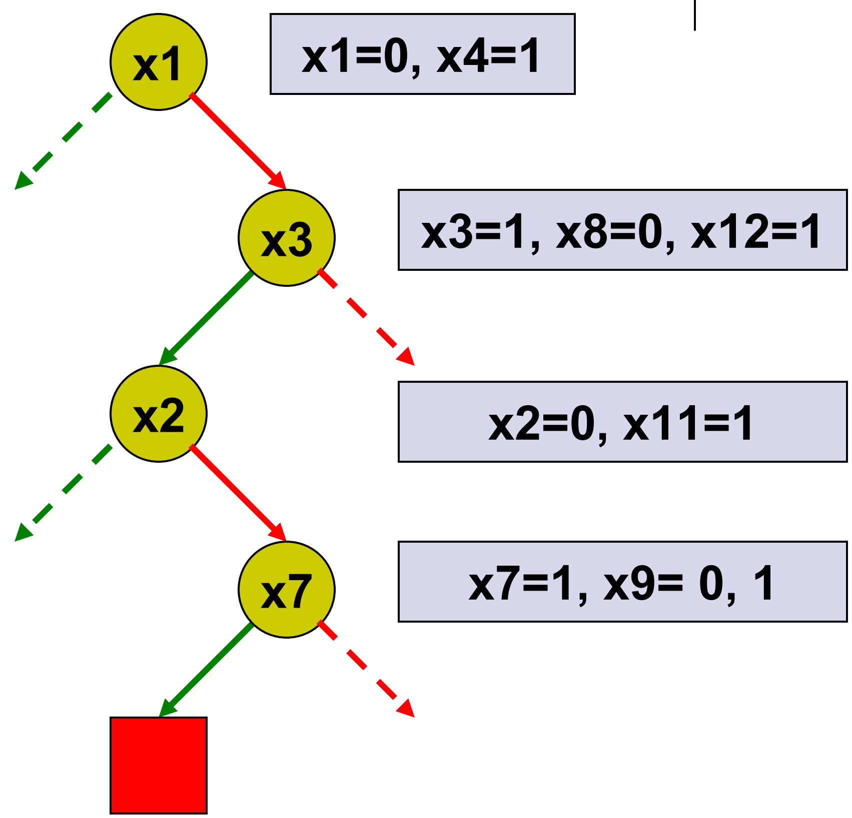




Conflict Driven Learning and Non-chronological Backtracking

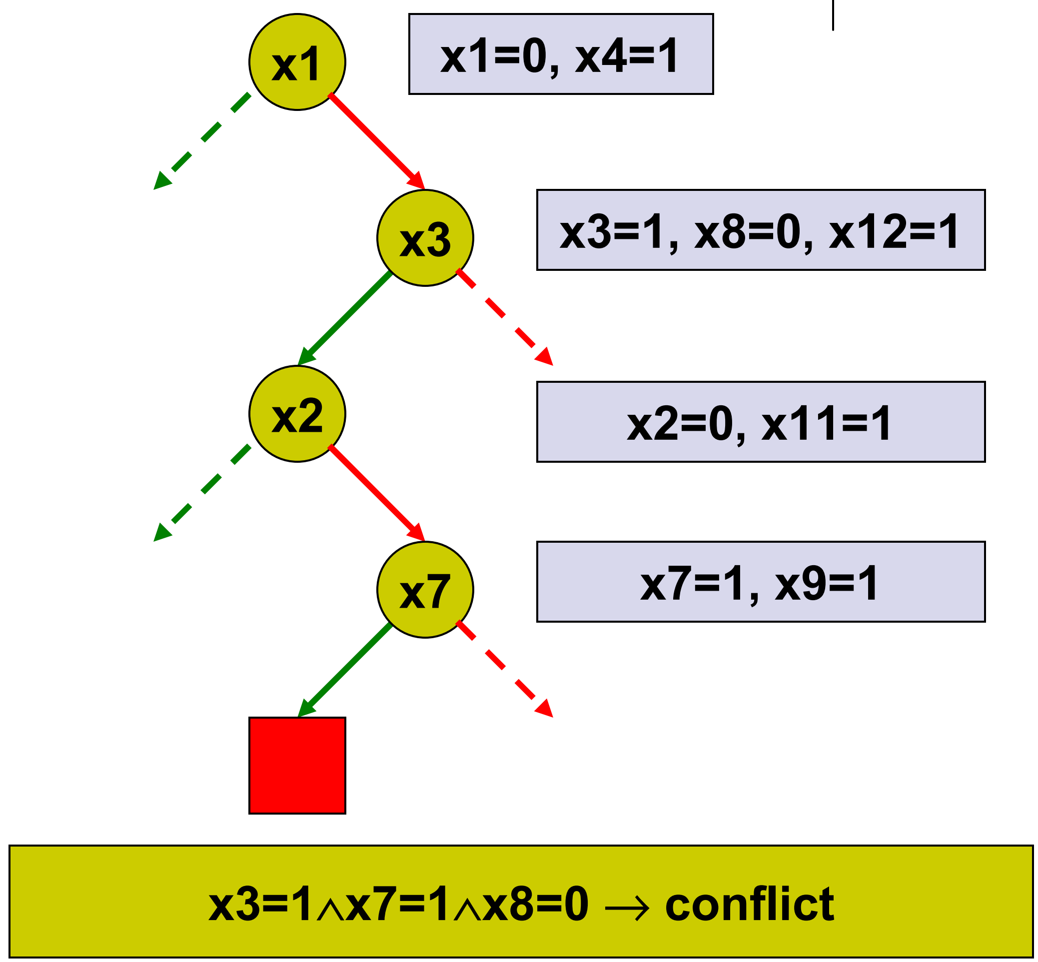
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

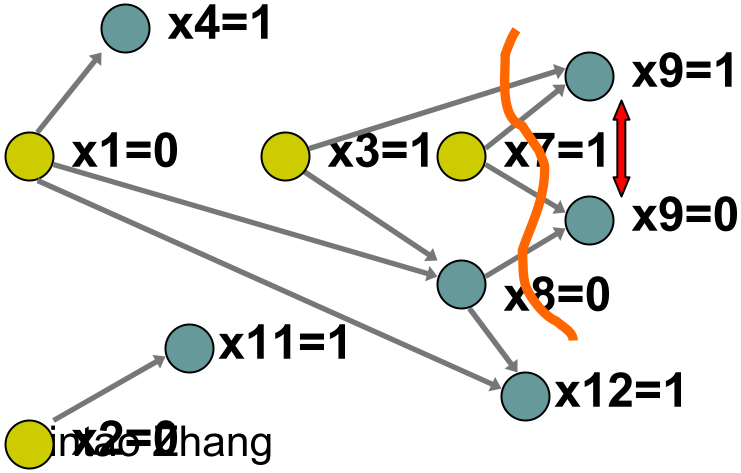




Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'





Contra-proposition:

- If a implies b, then b' implies a'

$$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow \text{conflict}$$

$$\text{Not conflict} \rightarrow (x3=1 \wedge x7=1 \wedge x8=0)'$$

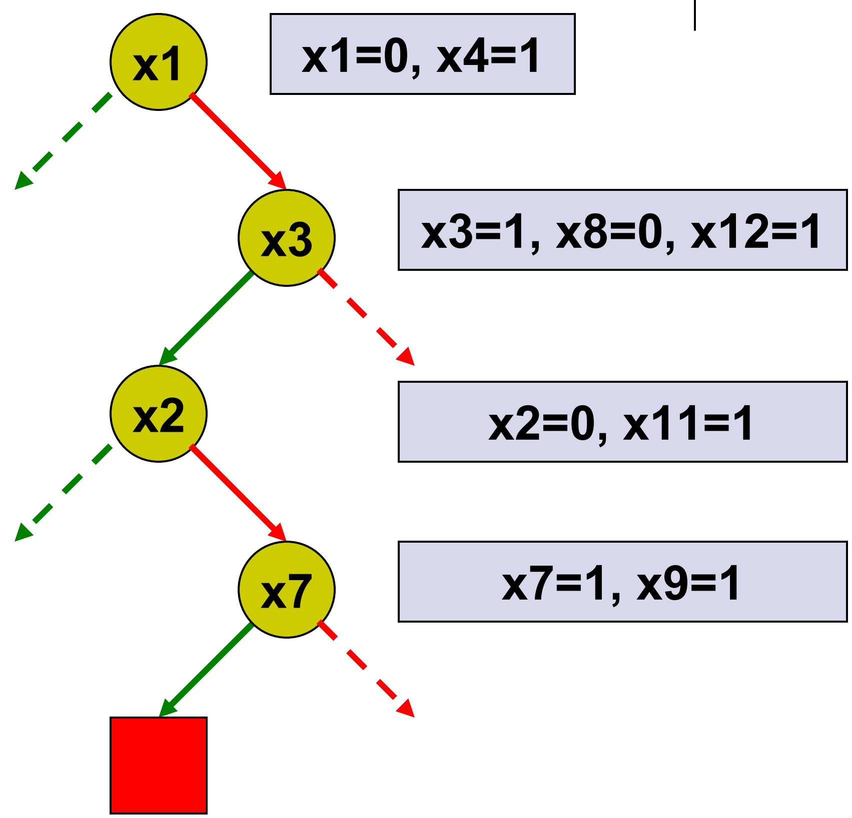
$$\text{true} \rightarrow (x3=1 \wedge x7=1 \wedge x8=0)'$$

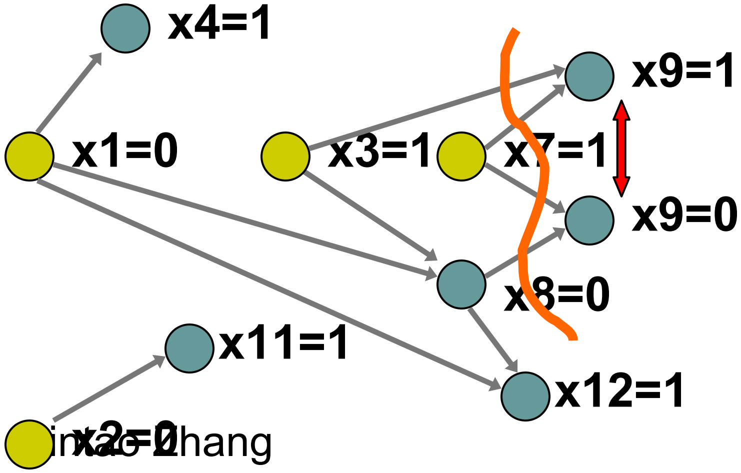
$$(x3=1 \wedge x7=1 \wedge x8=0)'$$

$$(x3' + x7' + x8)$$

Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’





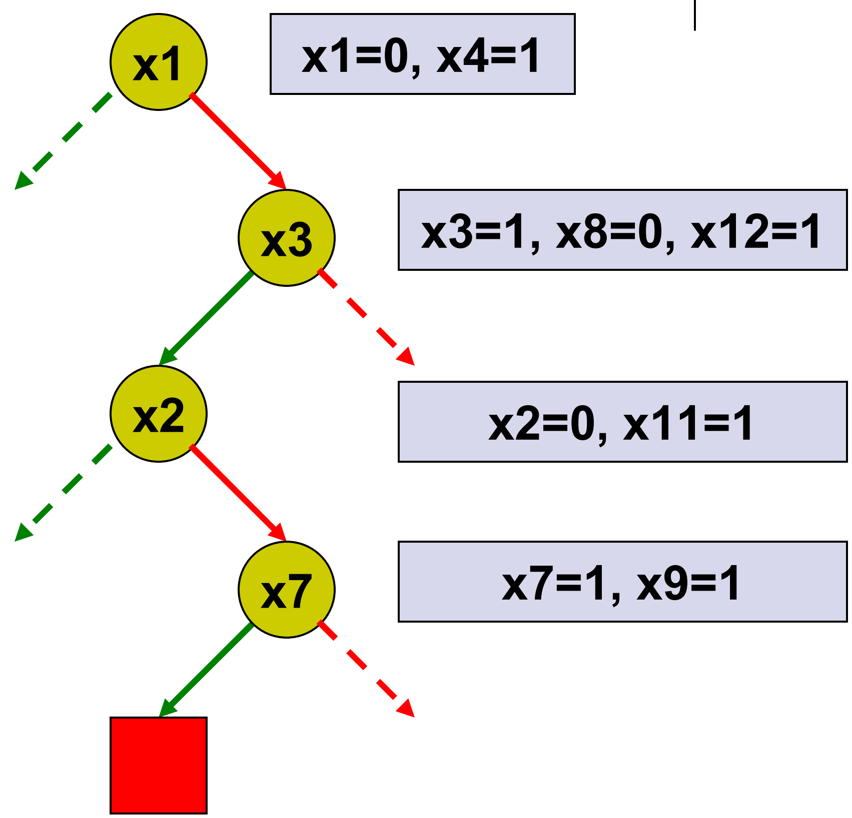
x3=1∧x7=1∧x8=0 → conflict

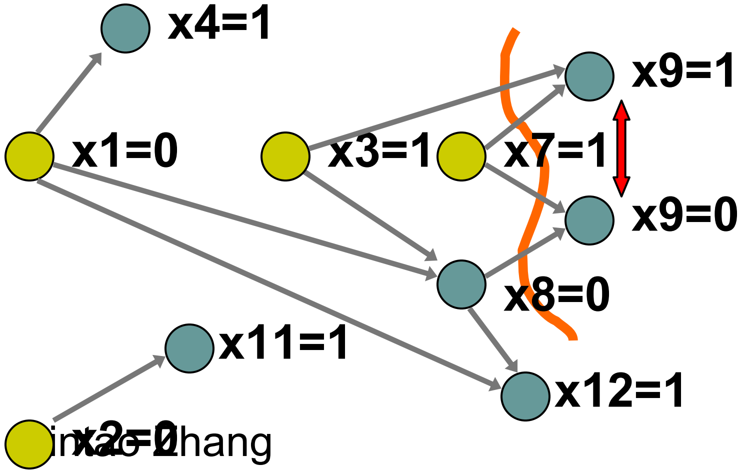
Add conflict clause: x3’+x7’+x8

Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

x3’+x7’+x8



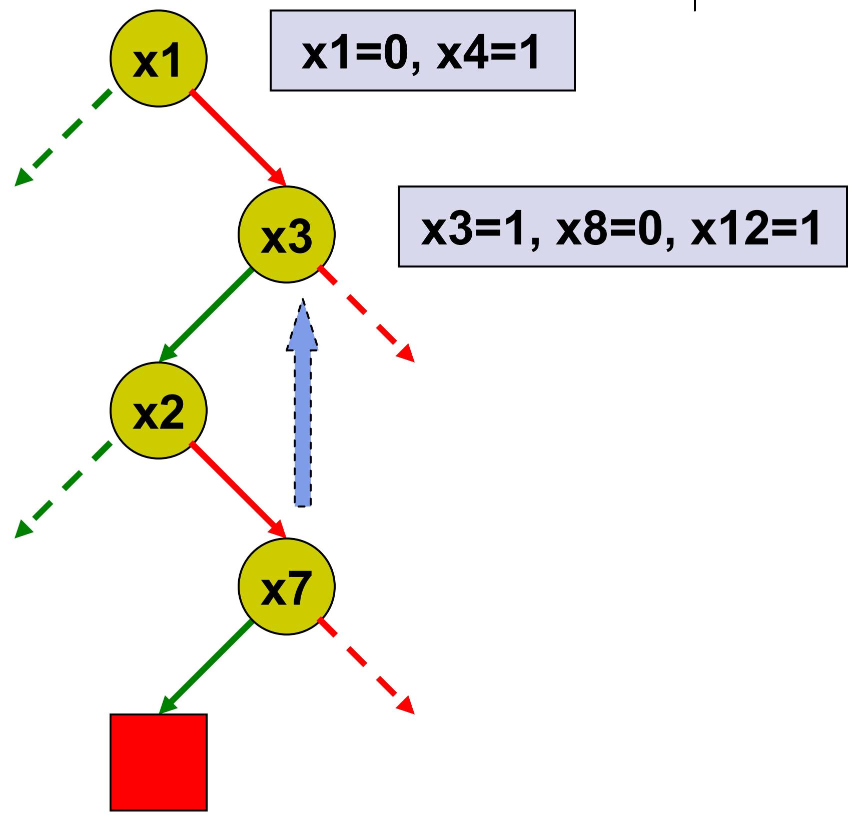


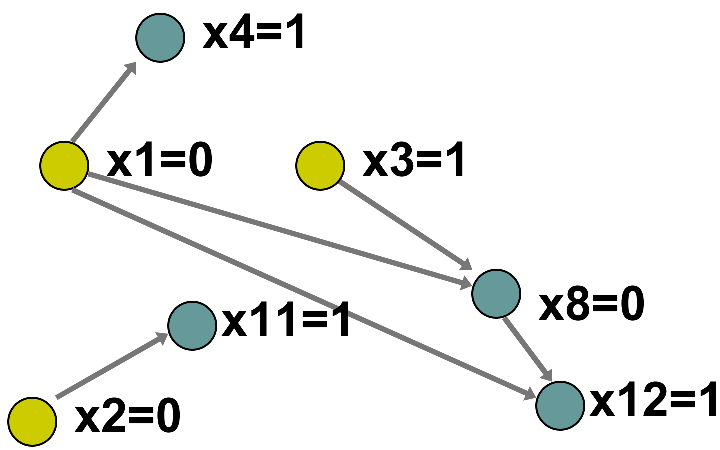
x3=1∧x7=1∧x8=0 → conflict

Add conflict clause: x3’+x7’+x8

DLL with Non-Chronological Backtracking and Learning

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'
x3' + x8 + x7'

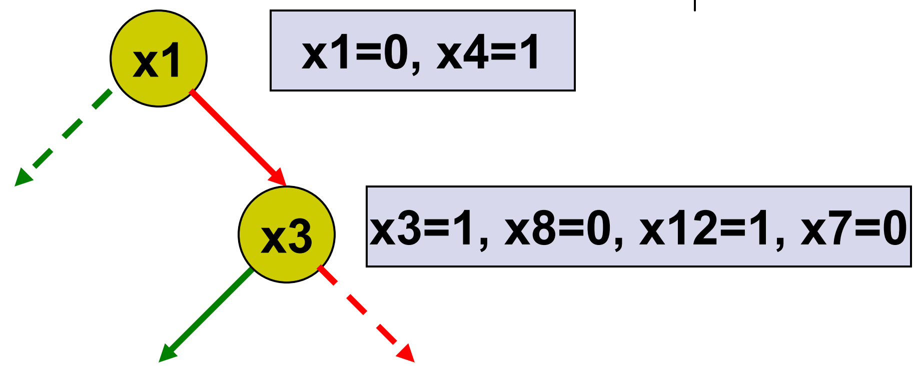


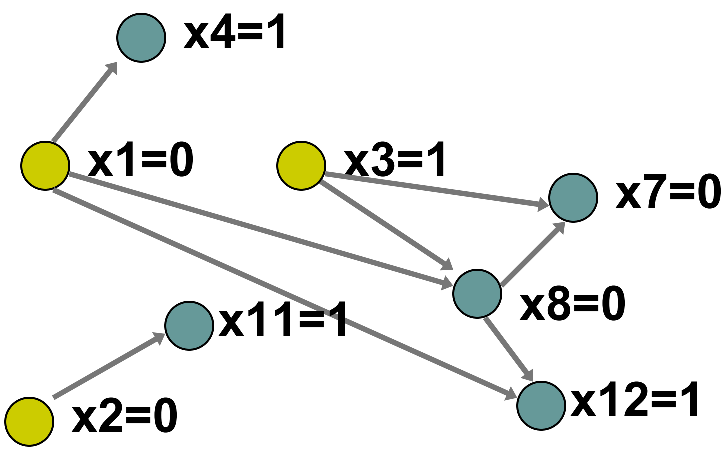


Backtrack to the decision level of x3=1:
x7 = 0

DLL with Non-Chronological Backtracking and Learning

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’
x3’ + x8 + x7’





Efficient Implementation of SAT Solvers

```
while(1) {
   if (decide_next_branch()) { //Branching
       while(deduce()==conflict) { //Deducing
             blevel = analyze_conflicts(); //Learning
             if (blevel < 0)
                    return UNSAT;
             else back_track(blevel); //Backtracking
       }
   else //no branch means all variables got assigned.
       return SATISFIABLE;
   }
```

Efficient Implementation of SAT Solvers

```
while(1) {
  if (decide_next_branch()) { //Branching
      while(deduce()==conflict) { //Deducing
            blevel = analyze_conflicts(); //Learning
            if (blevel < 0)
                  return UNSAT;
            else back_track(blevel); //Backtracking
      }
  else //no branch means all variables got assigned.
      return SATISFIABLE;
  }
```

Decision Heuristics

- If problem is SAT
  - Find satisfying assignment quickly
    - prune spaces where an assignment does not exist quickly
      - A: try and force a conflict (through implications) quickly
    - zoom in on the space where the solution exists
      - B: try and satisfy as many clauses as possible
- If problem is UNSAT
  - Prove unsatisfiability quickly
    - prune entire space quickly
      - A: try and force a conflict (through implications) quickly
- A, B above are the operational goals
- Cost benefit tradeoff
  - computation cost should not overweigh benefit of search space reduction

Simple Literal Counting

- RAND
  - pick a literal randomly (no counting!)
- Let:
  - CP(x) be the number of occurrences of x in unresolved clauses
  - CN(x) be the number of occurrences of x' in unresolved clauses
- DLCS (Dynamic Largest Combined Sum)
  - Pick variable with largest CP(x) + CN(x) value
  - if CP(x) ≥ CN (x), set x true, else set x false
- DLIS (Dynamic Largest Individual Sum)
  - Pick variable with largest value or all CP, CN
  - if CP(x) ≥ CN (x), set x true, else set x false
- Randomized DLIS (RDLIS), or RDLCS
  - select phase of the variable randomly

BOHM's Heuristic

- Select a variable with the maximum vector:

  $H_i(x) = \alpha \max(h_i(x), h_i(x')) + \beta \min(h_i(x), h_i(x'))$

  - $h_i(x)$: number of unresolved clauses of size i (remaining literals) with literal x in them
  - α, β selected by experimentation (suggested values 1, 2)
  - vectors compared in lexicographic order from left to right
- Intuition:
  - each selected literal gives preference to:
    - satisfying small clauses (when assigned true)
    - further reducing the size of small clauses when assigned false

MOM's Heuristic

- Maximum Occurrence's in Clauses of Minimum Size
- Select the literal that maximizes the function:

  $[f^*(x) + f^*(x')]*2^k + f^*(x) * f^*(x')$
  - f*(l): Number of occurences of l in the smallest non-satisfied clauses
  - k is a tuning parameter
- Intuition: Preference is given to clauses:
  - with a large number of occurences of either x or x' in them
  - and also variables that have a large number of clauses of both phases of x in them
  - focus on the currently smallest size clauses

Jeroslow-Wang Heuristics

- For a given literal l compute:

  $J(l) = \sum 2^{-|\omega|}$ Sum over all clause ω where l is in
- One sided Jeroslow-Wang (JW-OS)
  - select the literal with the highest value of J
- Two-sided Jeroslow-Wang (JW-TS)
  - select the variable with the highest value of (J(x) + J(x'))
  - if $J(x) \geq J(x')$ set x true, else set x false
- Intuition:
  - Weight occurrences in small clauses higher

Decision Heuristics – Conventional Wisdom

- DLIS is typical of common dynamic decision heuristics
  - Simple and intuitive
    - At each decision simply choose the assignment that satisfies the most unsatisfied clauses.
  - However, considerable work is required to maintain the statistics necessary for this heuristic – for one implementation:
    - Must touch every clause that contains a literal that has been set to true.
    - Maintain "sat" flag for each clause. When the flag transition 0→1, update rankings.
    - Need to reverse the process for unassignment.
  - The total effort required for this and similar decision heuristics is much more than that for BCP.
- Still based on static statistics in the sense that it does not take search history into consideration
  - The next decision will be determined by the current clause database and search tree, regardless of how you reach current state,

Static Statistics are not Enough

- We should differentiate learned clauses with the original clauses
- Why does the search process arrive in current state? There are some insight that we can leverage
- How to use dynamic information to guide search in the future?

Chaff Decision Heuristic - VSIDS

- Variable State Independent Decaying Sum (VSIDS)
  - Choose literal that has the highest score to branch
  - Initial score of a literal is its literal count in the initial clause database
  - Score is incremented (by 1) when a new clause containing that literal is added.
  - Periodically, divide all scores by a constant.

- VSIDS is semi-static because it does not change as variables get assigned/unassigned
  - Scores are much cheaper to maintain

- VSIDS is based on dynamic statistics because it take search history into consideration
  - Much more robust, highly effective in real world benchmarks

Decision Heuristic of BerkMin

- Literal score is incremented when the literal is involved in conflict clause generation
- Branch on free variables that are in the last unresolved learned clause with highest score
- It has similar property as VSIDS but seems to be more robust and more effective

Efficient Implementation of SAT Solvers

```
while(1) {
   if (decide_next_branch()) { //Branching
        while(deduce()==conflict) { //Deducing
               blevel = analyze_conflicts(); //Learning
               if (blevel < 0)
                      return UNSAT;
               else back_track(blevel); //Backtracking
        }
   else //no branch means all variables got assigned.
        return SATISFIABLE;
   }
```

Boolean Constraint Propagation (BCP)

- After setting a variable to a constant value, propagate the effect of the assignment
  - Find out all the unit clauses
  - Detect conflicts
- Backtrack: the reverse of BCP
  - when a variable is unassigned, how to unset a variable.
- BCP takes the major part of the run time of a DLL SAT solver
- Different implementation schemes for BCP may have significant effect on the efficiency of the solver

Literal Counting Scheme

- Each variable keeps a list of all its occurrence in the clauses, both in positive and negative form.
- Each clause maintains counters to indicate its status (number of 1/0/- assignments).
- When a variable is assigned, it will visit all the clauses that contain it and modify the status counters.
- When a variable is unassigned, it will also need to reverse the modification it did to the clauses.

Literal Counting as in GRASP

- Each clause maintains two counters:
  - Num_1_Lits
  - Num_0_Lits
  - Num_all_Lits (This is not a counter, just a constant)
- A Clause is unit if
  - (Num_0_Lits == Num_all_Lits – 1) && Num_1_Lits == 0
  - If this is true, solver needs to search through all the literals of the clause to find out the remaining free literal
- A Clause is a conflict clause if
  - Num_0_Lits == Num_all_Lits
- A SAT instance with n variables, m clauses, each clause has l literals on the average:
  - A variable assignment/unassignment takes *l m / n* operations on the average
- A Clause is SAT if
  - Num_1_Literals > 0
  - This is a constant time operation

A Better Literal Counting Scheme

- Each clause keeps one counter
  - Num_Non_Zero_Literals
- A Clause is Unit if
  - Num_Non_Zero_Literals == 1
  - And, the solver will search all the literals in the clause to find out the remaining literal with value other than 0, if it's unassigned, then it is implied. Otherwise, skip this clause.
- A Clause is Conflict if
  - Num_Non_Zero_Literals == 0
- A SAT instance with n variables, m clauses, each clause has l literals on the average:
  - A variable assignment/unassignment takes *l m / 2n* operations on the average
- A Clause is SAT if
  - Search all the literals in the clause to find if it has at least one of them with value 1.
  - This operation complexity is linear wrt the length of the clause

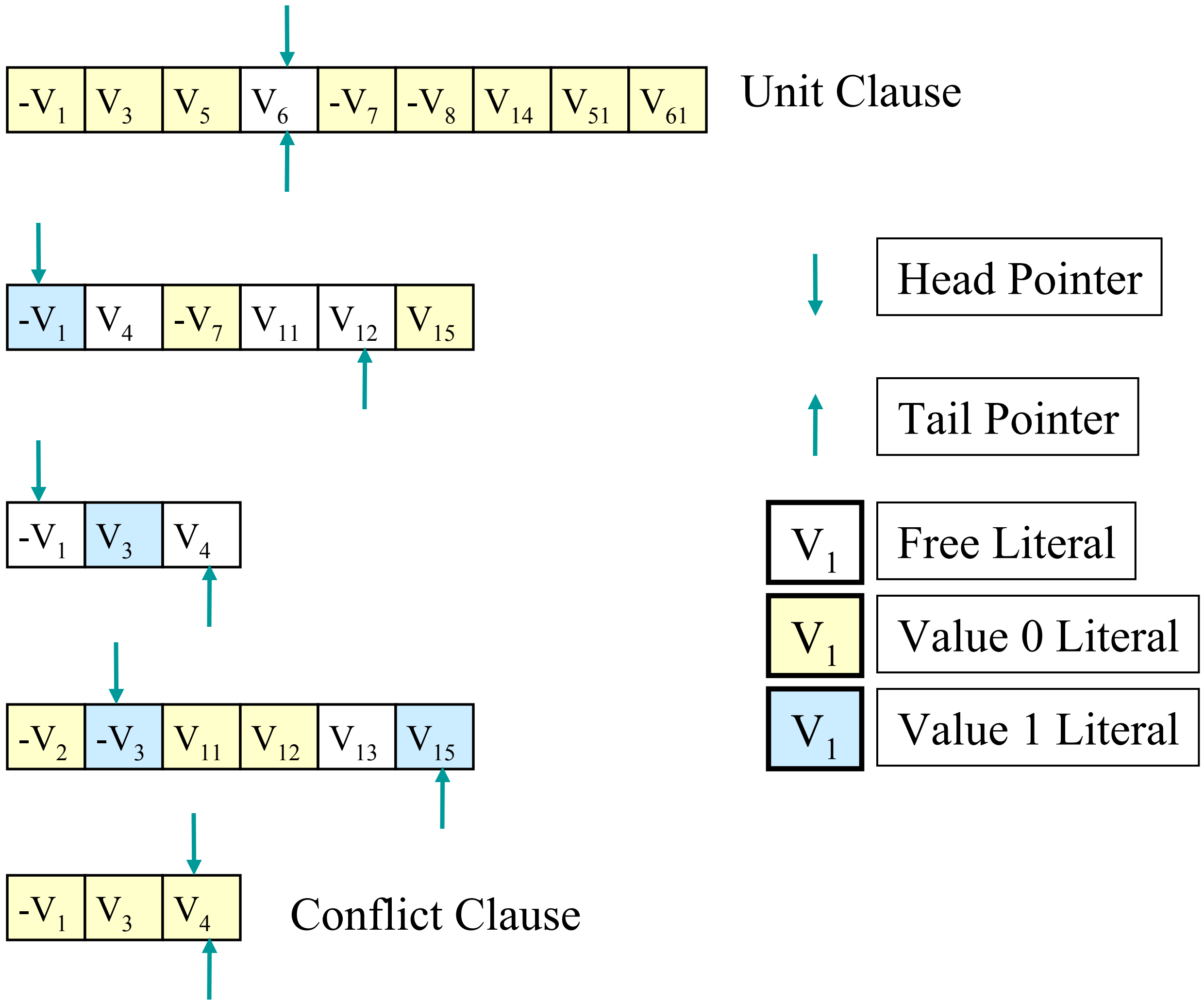
BCP in SATO

- There is no need to update the status of a clause whenever a literal of the clause is getting assigned a value.
  - not counter based!
- Literals of a clause are arranged in a linear array.
- Each clause has two pointers.
- *head* pointer points to the first literal from the beginning of the clause that is either free (unassigned) or has value 1.
  - If a free head literal was assigned 0, head pointer will be moved towards the end of the clause to find another literal that satisfy this criterion.
- *tail* pointer points to the last literal from the end of the clause that is either free (unassigned) or has value 1,
  - It will move towards the beginning when assigned value 0.

BCP in SATO

- Each variable will keep 4 lists:
  - positive heads
  - negative heads
  - positive tails
  - negative tails
- Whenever a variable gets assigned a value, the corresponding clauses with it as head or tail literal need to be visited and modified.
- The clause is a unit or a conflict clause when head meets tail
- A SAT instance with n variables, m clauses, each clause has l literals on the average:
  - A variable assignment/unassignment takes *m / n* operations on the average
  - Each operation may be more expensive than literal counting

BCP in SATO



-V1 V3 V5 V6 -V7 -V8 V14 V51 V61
Unit Clause
-V1 V4 -V7 V11 V12 V15
-V1 V3 V4
-V2 -V3 V11 V12 V13 V15
-V1 V3 V4
Conflict Clause
Head Pointer
Tail Pointer
V1
Free Literal
V1
Value 0 Literal
V1
Value 1 Literal

Chaff BCP Algorithm (1/8)

- What "causes" an implication? When can it occur?
  - All literals in a clause but one are assigned to F
    - (v1 + v2 + v3): implied cases: (0 + 0 + v3) or (0 + v2 + 0) or (v1 + 0 + 0)
  - For an N-literal clause, this can only occur after N-1 of the literals have been assigned to F
  - So, (theoretically) we could completely ignore the first N-2 assignments to this clause
  - In reality, we pick two literals in each clause to "watch" and thus can ignore any assignments to the other literals in the clause.
    - Example: (v1 + v2 + v3 + v4 + v5)
    - (**v1=X** + **v2=X** + v3=? {i.e. X or 0 or 1} + v4=? + v5=?)

BCP Algorithm (1.1/8)

- Big Invariants
  - Each clause has two watched literals.
  - If a clause can become newly implied via any sequence of assignments, then this sequence will include an assignment of one of the watched literals to F.
    - Example again: (v1 + v2 + v3 + v4 + v5)
    - (**v1=X** + **v2=X** + v3=? + v4=? + v5=?)
- BCP consists of identifying implied clauses (and the associated implications) while maintaining the "Big Invariants"

BCP Algorithm (2/8)

- Let's illustrate this with an example:

```
( 2  3  1  4  5)
( 1  2 -3)
( 1 -2)
(-1  4)
(-1)
```

BCP Algorithm (2.1/8)

- Let's illustrate this with an example:

watched literals →

```
( 2  3  1  4  5)
( 1  2 -3)
( 1 -2)
(-1  4)
(-1)
```

- Conceptually, we identify the first two literals in each clause as the watched ones

BCP Algorithm (2.2/8)

- Let's illustrate this with an example:

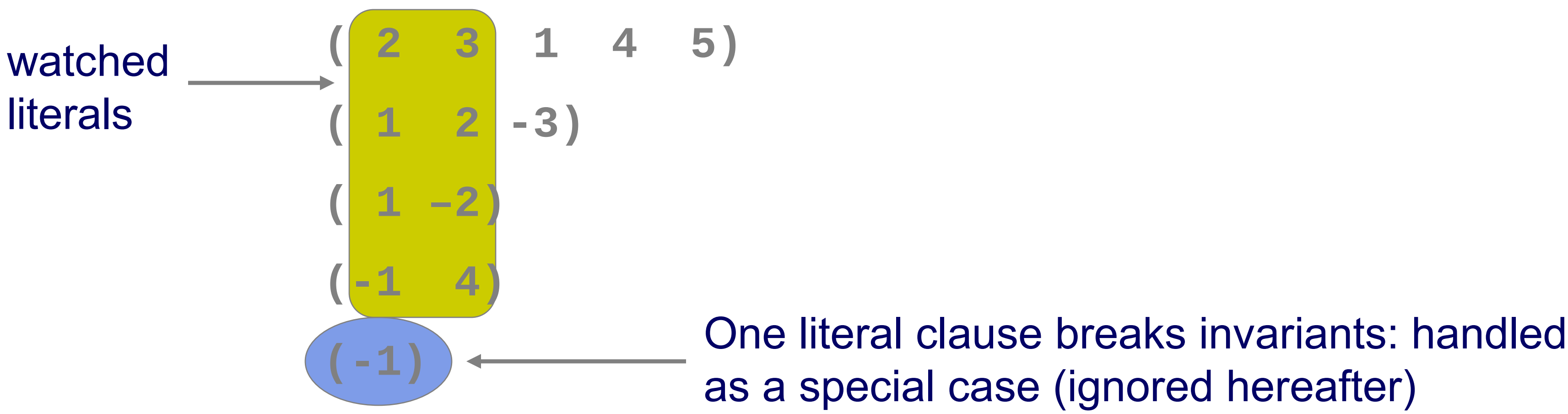
watched literals →

```
( 2  3  1  4  5)
( 1  2 -3)
( 1 -2)
(-1  4)
(-1)
```

- Conceptually, we identify the first two literals in each clause as the watched ones
- Changing which literals are watched is represented by reordering the literals in the clause (which comes into play later)

BCP Algorithm (2.3/8)

- Let's illustrate this with an example:



- Conceptually, we identify the first two literals in each clause as the watched ones
- Changing which literals are watched is represented by reordering the literals in the clause (which comes into play later)
- Clauses of size one are a special case

BCP Algorithm (3/8)

- We begin by processing the assignment v1 = F (which is implied by the size one clause)

```
State:(v1=F)
Pending:
```

```
( 2  3  1  4  5)
( 1  2 -3)
( 1 -2)
(-1  4)
```

BCP Algorithm (3.1/8)

- We begin by processing the assignment v1 = F (which is implied by the size one clause)

```
                         ( 2  3  1  4  5)
State:(v1=F)       =>    ( 1  2 -3)
Pending:           =>    ( 1 -2)
                         (-1  4)
```

- To maintain our invariants, we must examine each clause where the assignment being processed has set a watched literal to F.

BCP Algorithm (3.2/8)

- We begin by processing the assignment v1 = F (which is implied by the size one clause)

```
State:(v1=F)          ( 2  3  1  4  5)
Pending:              ( 1  2 -3)
                      ( 1 -2)
                   ⇨  (-1  4)
```

- To maintain our invariants, we must examine each clause where the assignment being processed has set a watched literal to F.
- We need not process clauses where a watched literal has been set to T, because the clause is now satisfied and so can not become implied.

BCP Algorithm (3.3/8)

- We begin by processing the assignment v1 = F (which is implied by the size one clause)

```
State:(v1=F)

Pending:
```

```
⇨ ( 2  3  1  4  5)
  ( 1  2 -3)
  ( 1 -2)
  (-1  4)
```

- To maintain our invariants, we must examine each clause where the assignment being processed has set a watched literal to F.
- We need not process clauses where a watched literal has been set to T, because the clause is now satisfied and so can not become implied.
- We *certainly* need not process any clauses where neither watched literal changes state (in this example, where v1 is not watched).

Lintao Zhang

BCP Algorithm (4/8)

- Now let's actually process the second and third clauses:

```
( 2  3  1  4  5)
( 1  2 -3)
( 1 -2)
(-1  4)
```

```
State:(v1=F)
Pending:
```

BCP Algorithm (4.1/8)

- Now let's actually process the second and third clauses:

```
( 2  3  1  4  5)        ( 2  3  1  4  5)
( 1  2 -3)         =>   (-3  2  1)
( 1 -2)                 ( 1 -2)
(-1  4)                 (-1  4)

State:(v1=F)            State:(v1=F)
Pending:                Pending:
```

- For the second clause, we replace v1 with ¬v3 as a new watched literal. Since ¬v3 is not assigned to F, this maintains our invariants.

BCP Algorithm (4.2/8)

- Now let's actually process the second and third clauses:

```
( 2  3  1  4  5)
( 1  2 -3)
( 1 -2)
(-1  4)
```

```
State:(v1=F)
Pending:
```

```
( 2  3  1  4  5)
(-3  2  1)
( 1 -2)
(-1  4)
```

```
State:(v1=F)
Pending:(v2=F)
```

- For the second clause, we replace v1 with ¬v3 as a new watched literal. Since ¬v3 is not assigned to F, this maintains our invariants.
- The third clause is implied. We record the new implication of ¬v2, and add it to the queue of assignments to process. Since the clause cannot again become newly implied, our invariants are maintained.

BCP Algorithm (5/8)

- Next, we process ¬v2. We only examine the first 2 clauses.

```
( 2  3  1  4  5)
(-3  2  1)
( 1 -2)
(-1  4)
```

```
State:(v1=F, v2=F)
Pending:
```

```
( 4  3  1  2  5)
(-3  2  1)
( 1 -2)
(-1  4)
```

```
State:(v1=F, v2=F)
Pending:(v3=F)
```

- For the first clause, we replace v2 with v4 as a new watched literal. Since v4 is not assigned to F, this maintains our invariants.
- The second clause is implied. We record the new implication of v3, and add it to the queue of assignments to process. Since the clause cannot again become newly implied, our invariants are maintained.

BCP Algorithm (6/8)

- Next, we process ¬v3. We only examine the first clause.

```
( 4  3  1  2  5)        ( 4  5  1  2  3)
(-3  2  1)              (-3  2  1)
( 1 -2)                 ( 1 -2)
(-1  4)                 (-1  4)
```

```
State:(v1=F, v2=F, v3=F)      State:(v1=F, v2=F, v3=F)
Pending:                      Pending:
```

- For the first clause, we replace v3 with v5 as a new watched literal. Since v5 is not assigned to F, this maintains our invariants.
- Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. Both v4 and v5 are unassigned. Let's say we decide to assign v4=T and proceed.

BCP Algorithm (7/8)

- Next, we process v4. We do nothing at all.

```
( 4  5  1  2  3)          ( 4  5  1  2  3)
(-3  2  1)                (-3  2  1)
( 1 -2)                   ( 1 -2)
(-1  4)                   (-1  4)
```

State:(v1=F, v2=F, v3=F, v4=T)	State:(v1=F, v2=F, v3=F, v4=T)

- Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. Only v5 is unassigned. Let's say we decide to assign v5=F and proceed.

BCP Algorithm (8/8)

- Next, we process v5=F. We examine the first clause.

```
( 4  5  1  2  3)
(-3  2  1)
( 1 -2)
(-1  4)
```

```
State:(v1=F, v2=F, v3=F,
v4=T, v5=F)
```

```
( 4  5  1  2  3)
(-3  2  1)
( 1 -2)
(-1  4)
```

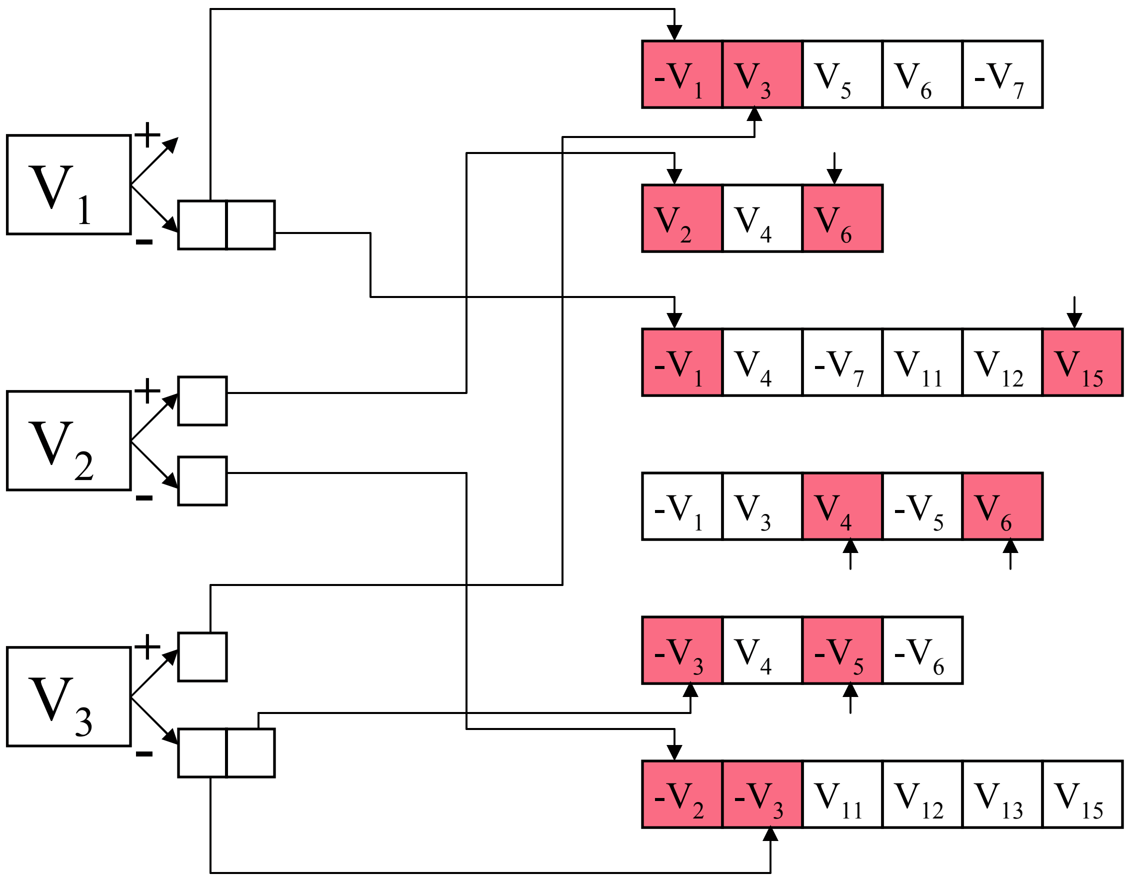
```
State:(v1=F, v2=F, v3=F,
v4=T, v5=F)
```

- The first clause is implied. However, the implication is v4=T, which is a duplicate (since v4=T already) so we ignore it.
- Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. No variables are unassigned, so the problem is sat, and we are done.

Chaff: BCP Algorithm Summary

- During forward progress: Decisions and Implications
  - Only need to examine clauses where watched literal is set to F
    - Can ignore any assignments of literals to T
    - Can ignore any assignments to non-watched literals
- During backtrack: Unwind Assignment Stack
  - Any sequence of chronological unassignments will maintain our invariants
    - So no action is required at all to unassign variables.
- Overall
  - Minimize clause access

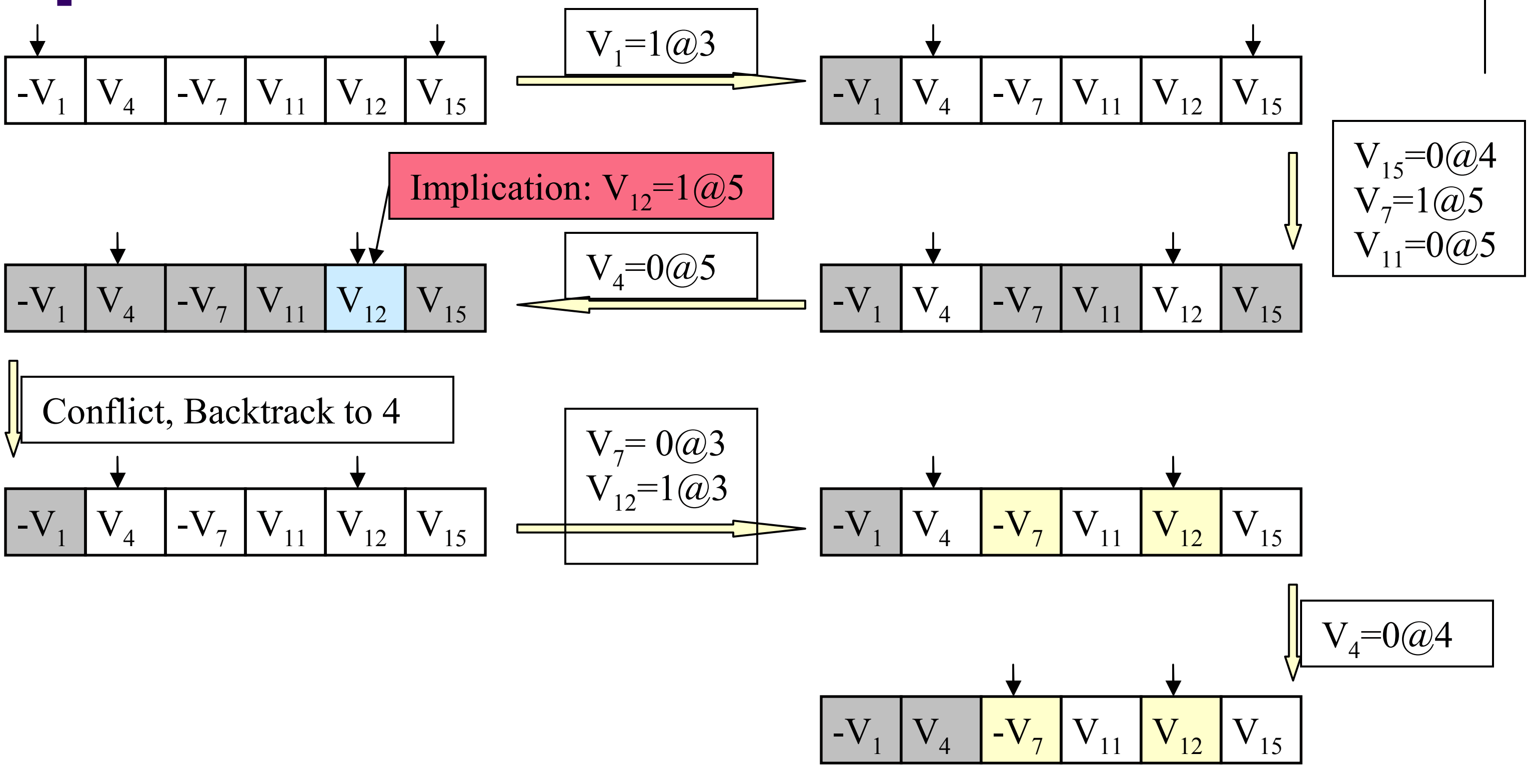
Implementation in ZChaff



Lintao Zhang

Microsoft Research

Implementation in ZChaff



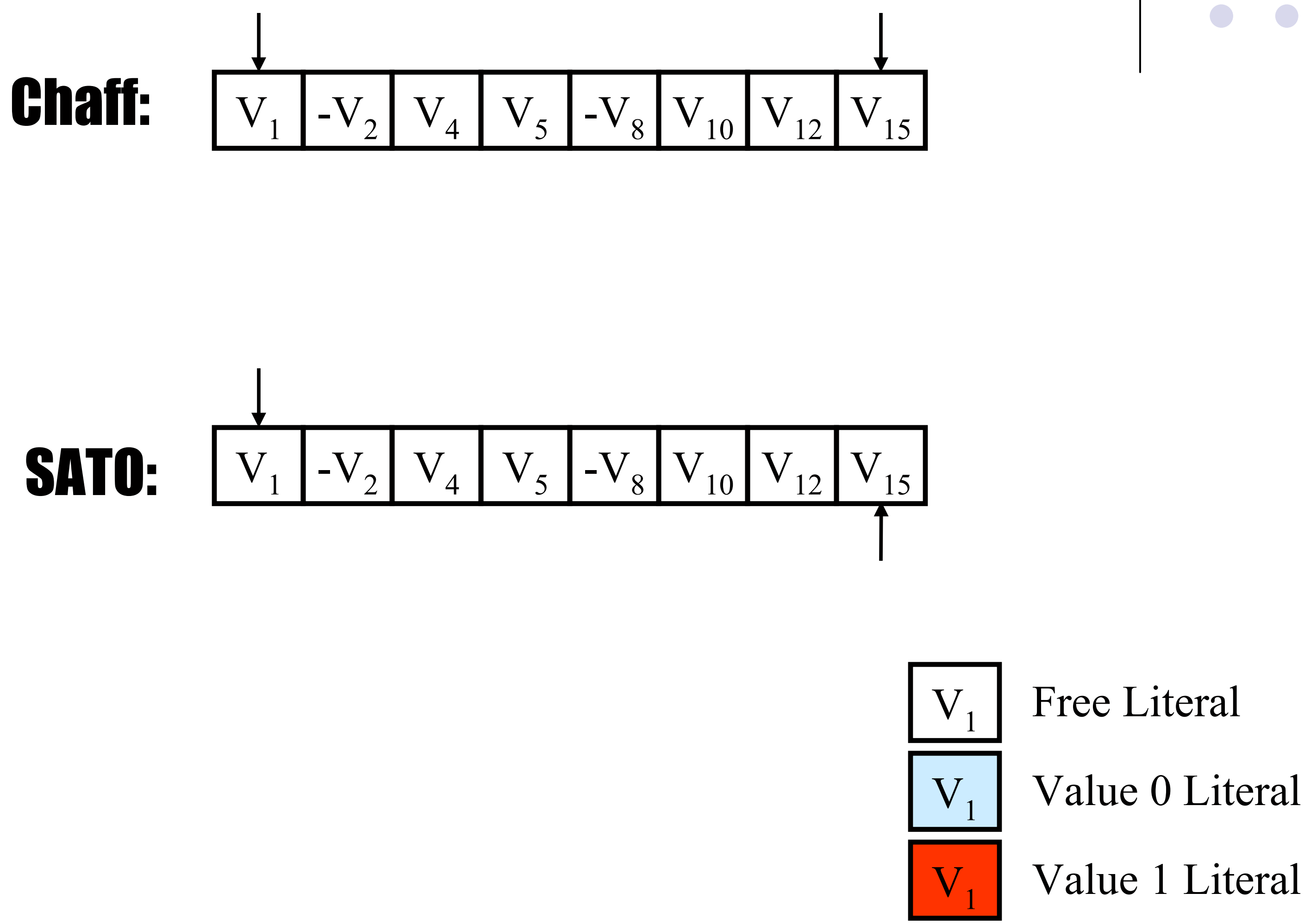
Always try to avoid watch literals with value 0

V_{12} Lit value unknown

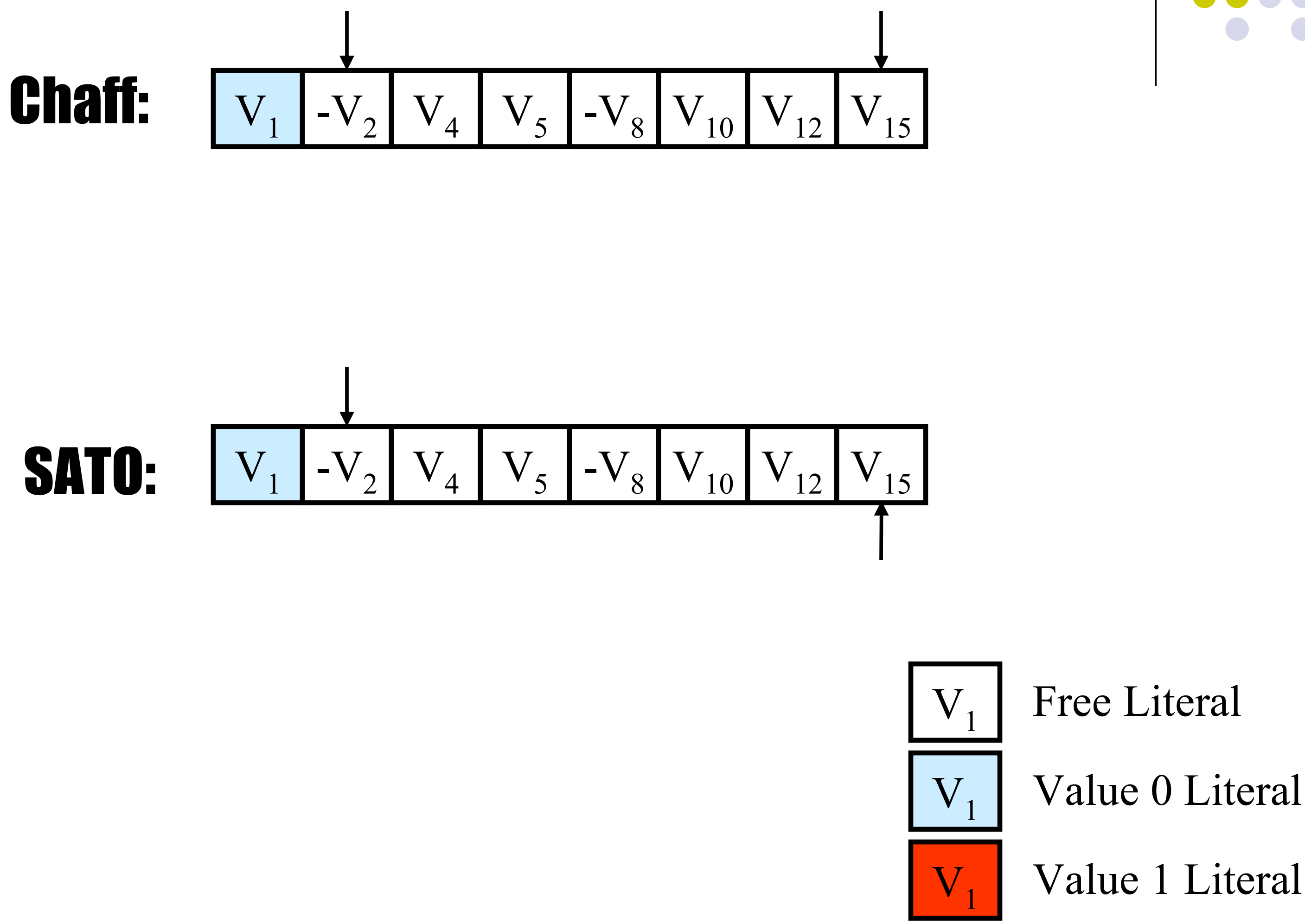
V_{12} Lit value 0

V_{12} Lit value 1

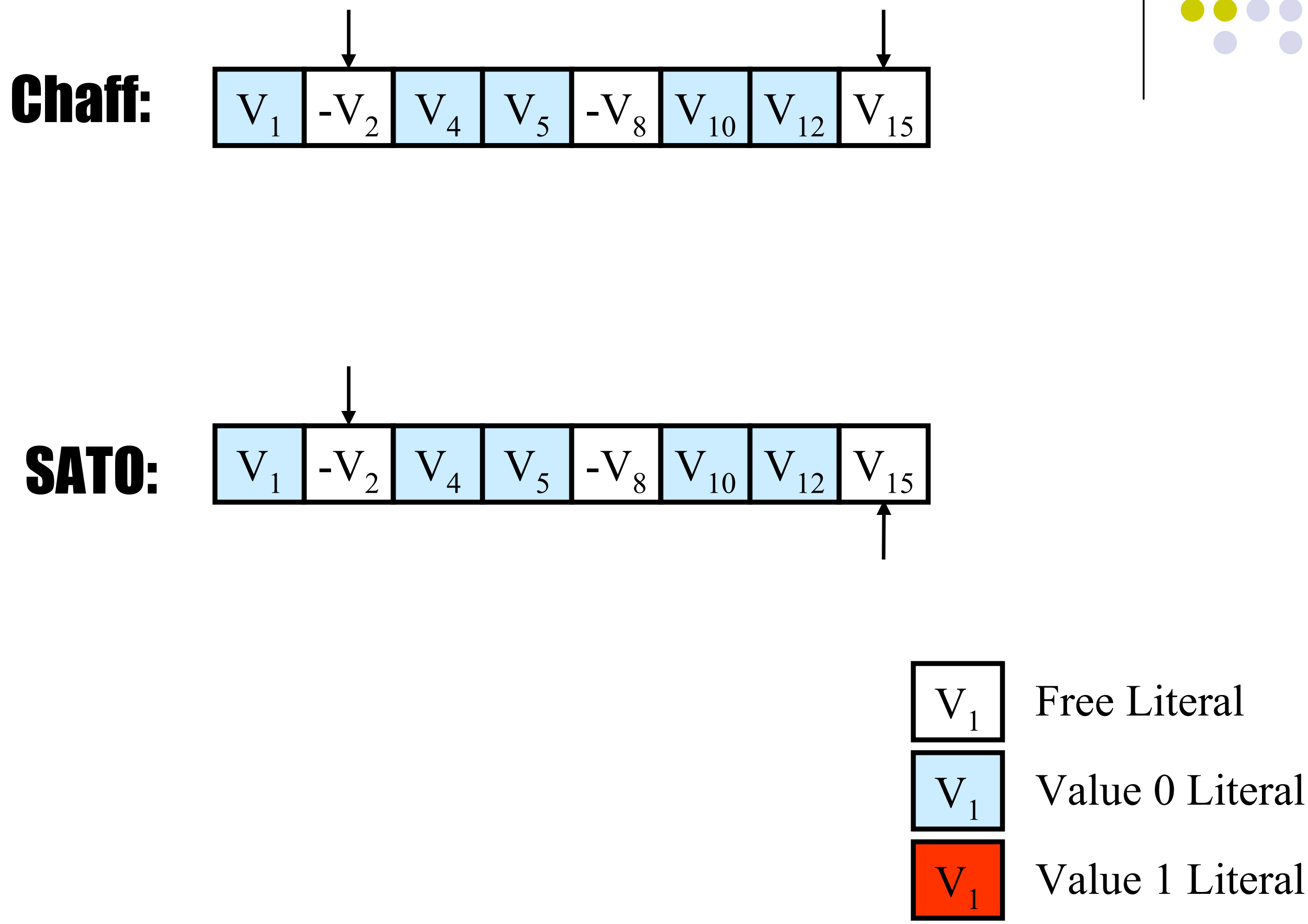
Difference of BCP in Chaff and SATO



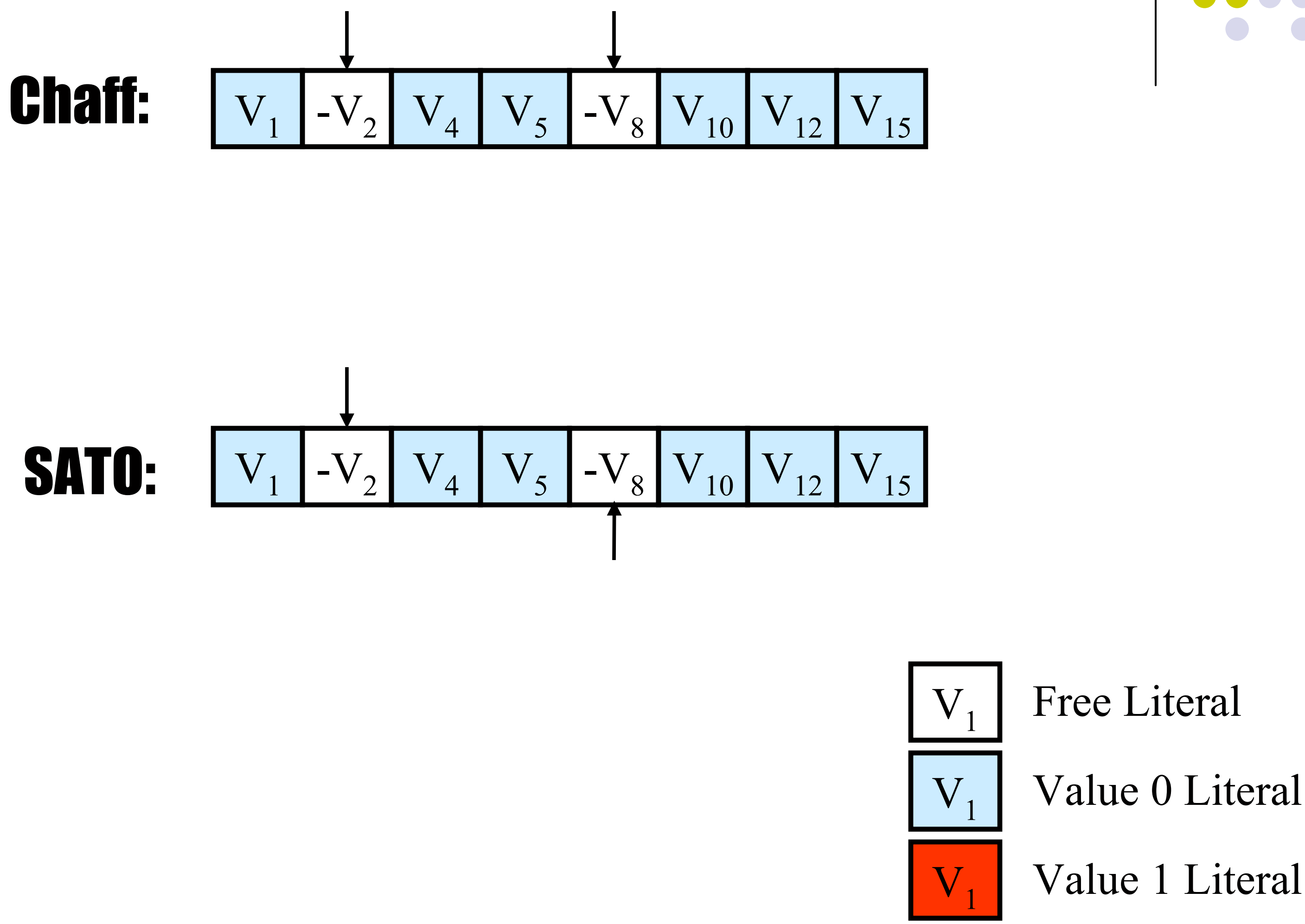
Difference of BCP in Chaff and SATO



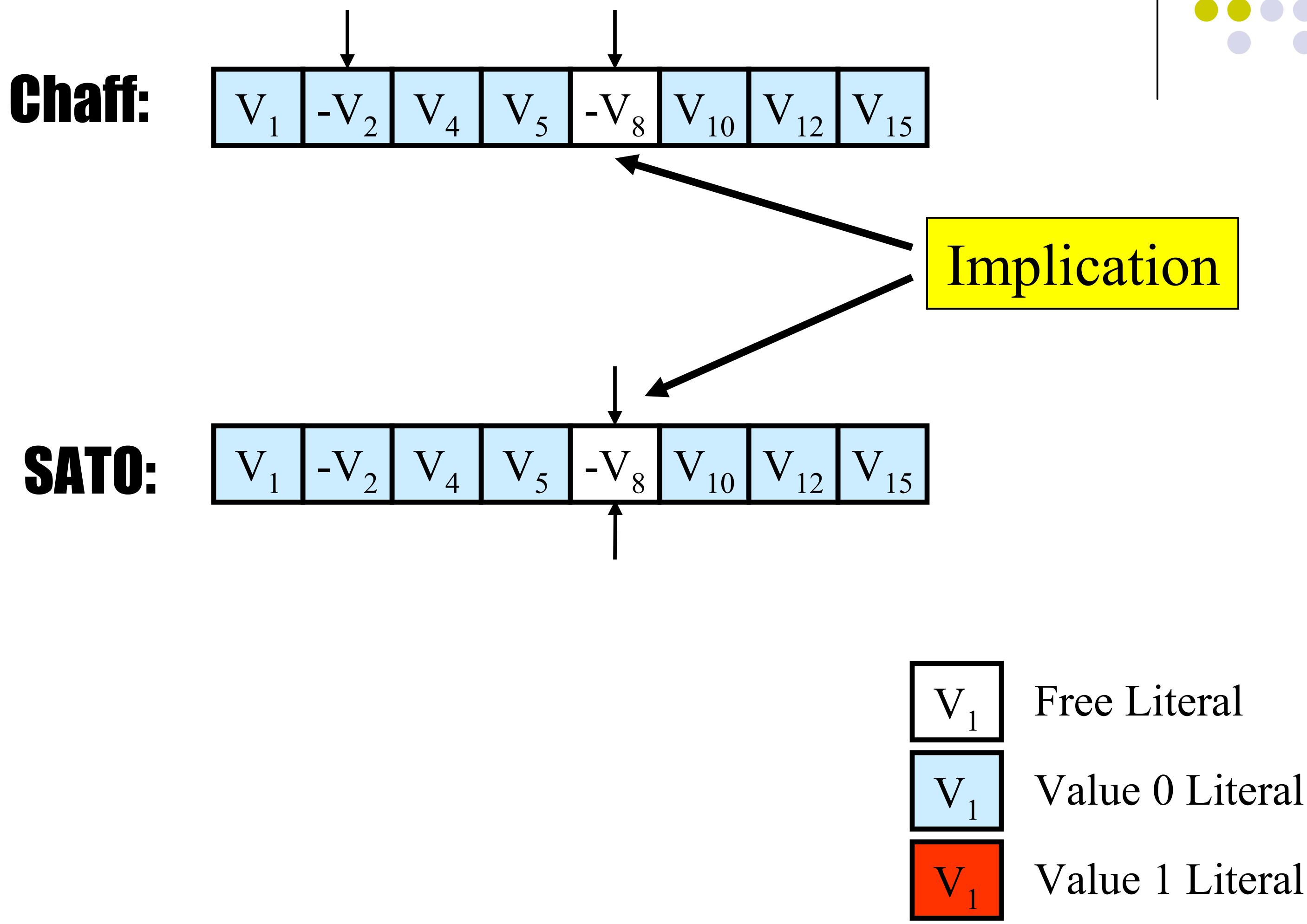
Difference of BCP in Chaff and SATO



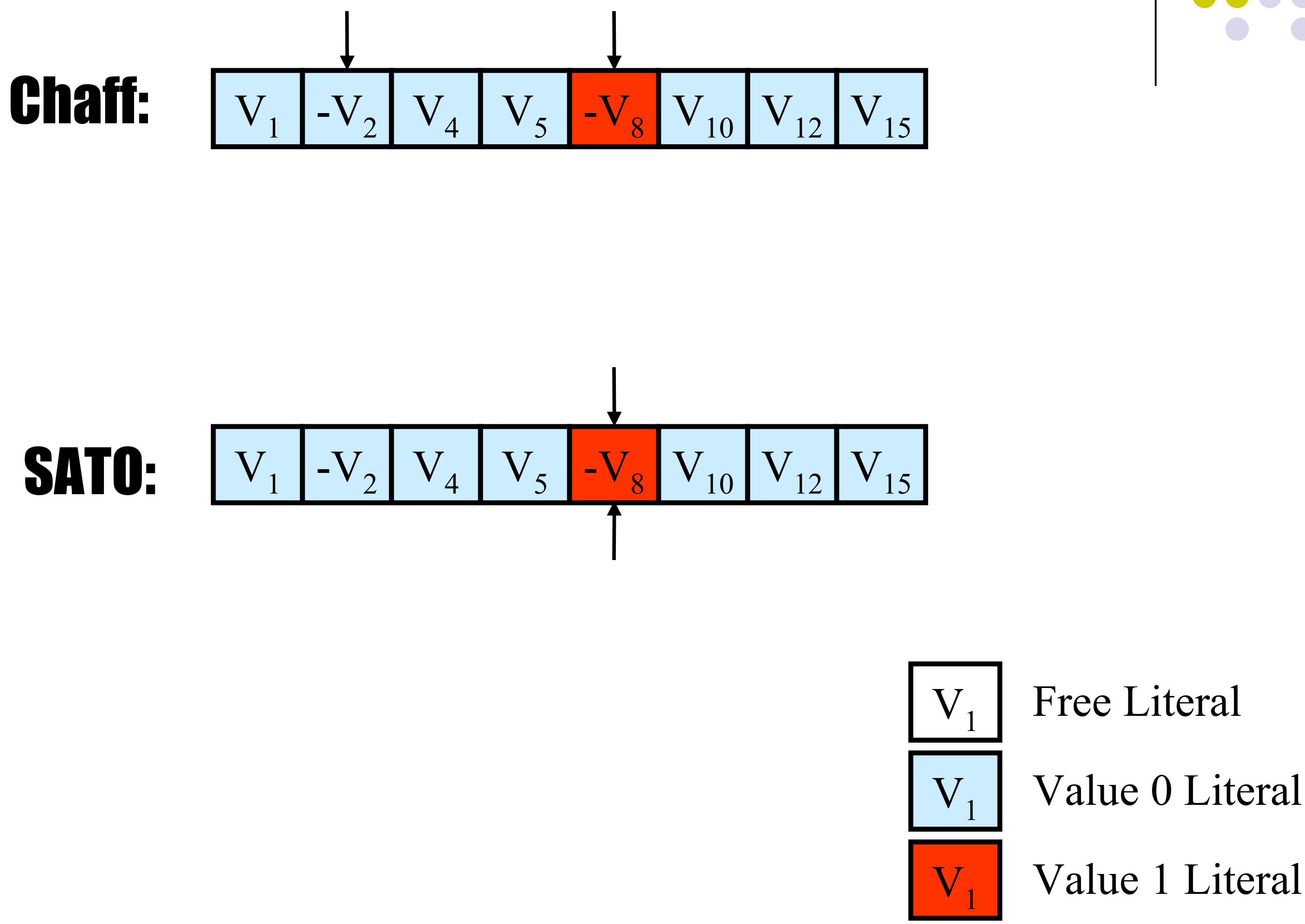
Difference of BCP in Chaff and SATO



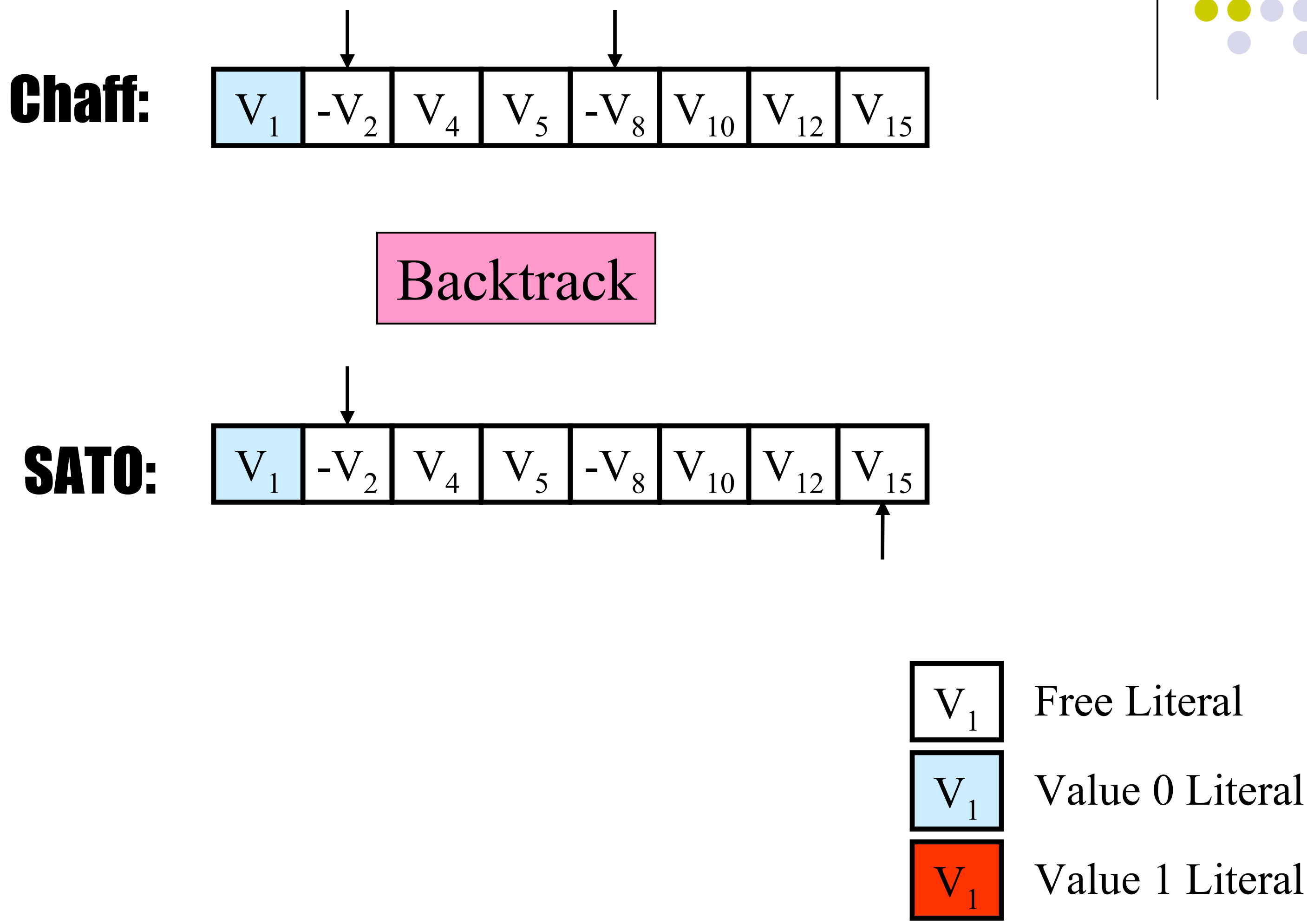
Difference of BCP in Chaff and SATO



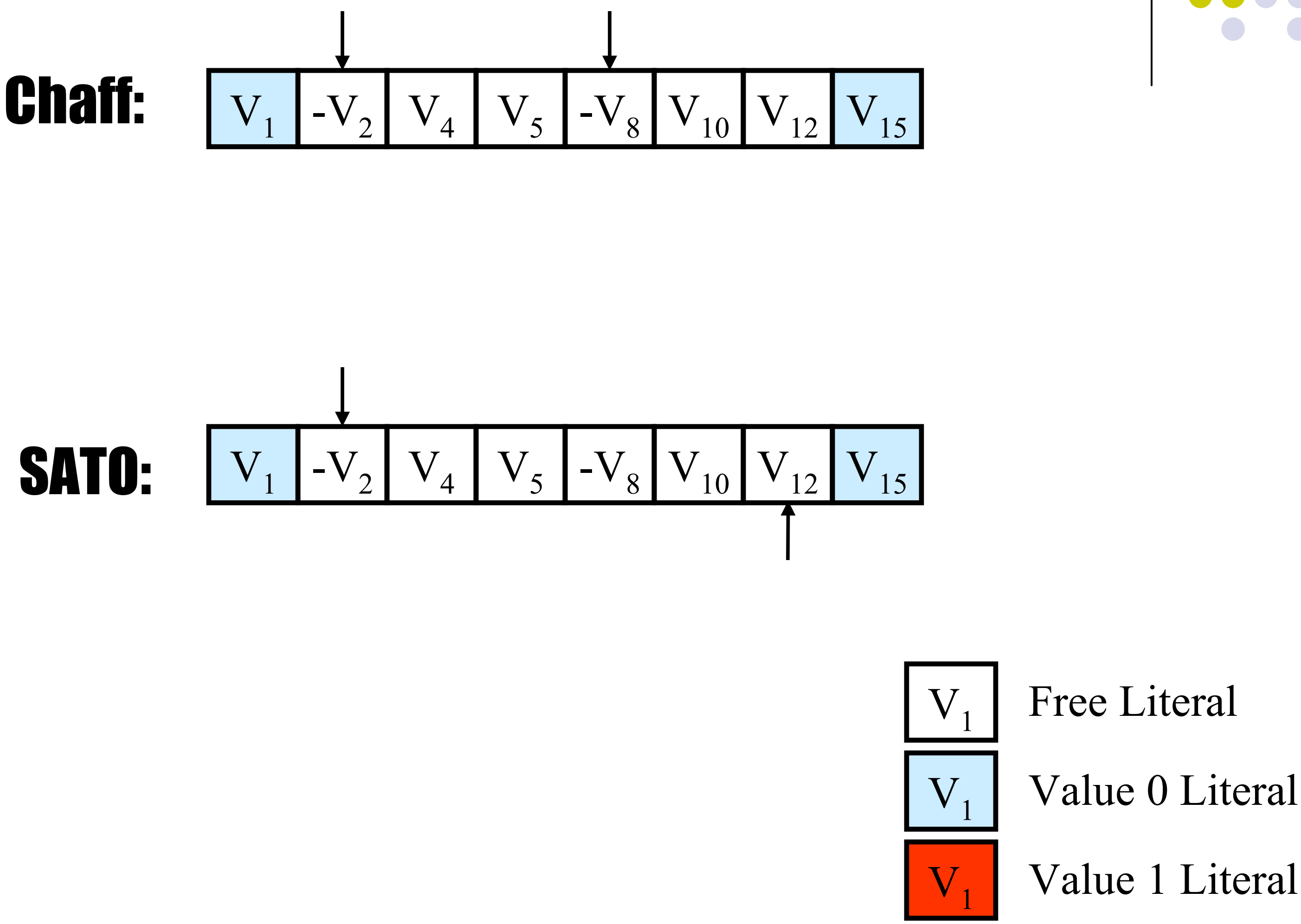
Difference of BCP in Chaff and SATO



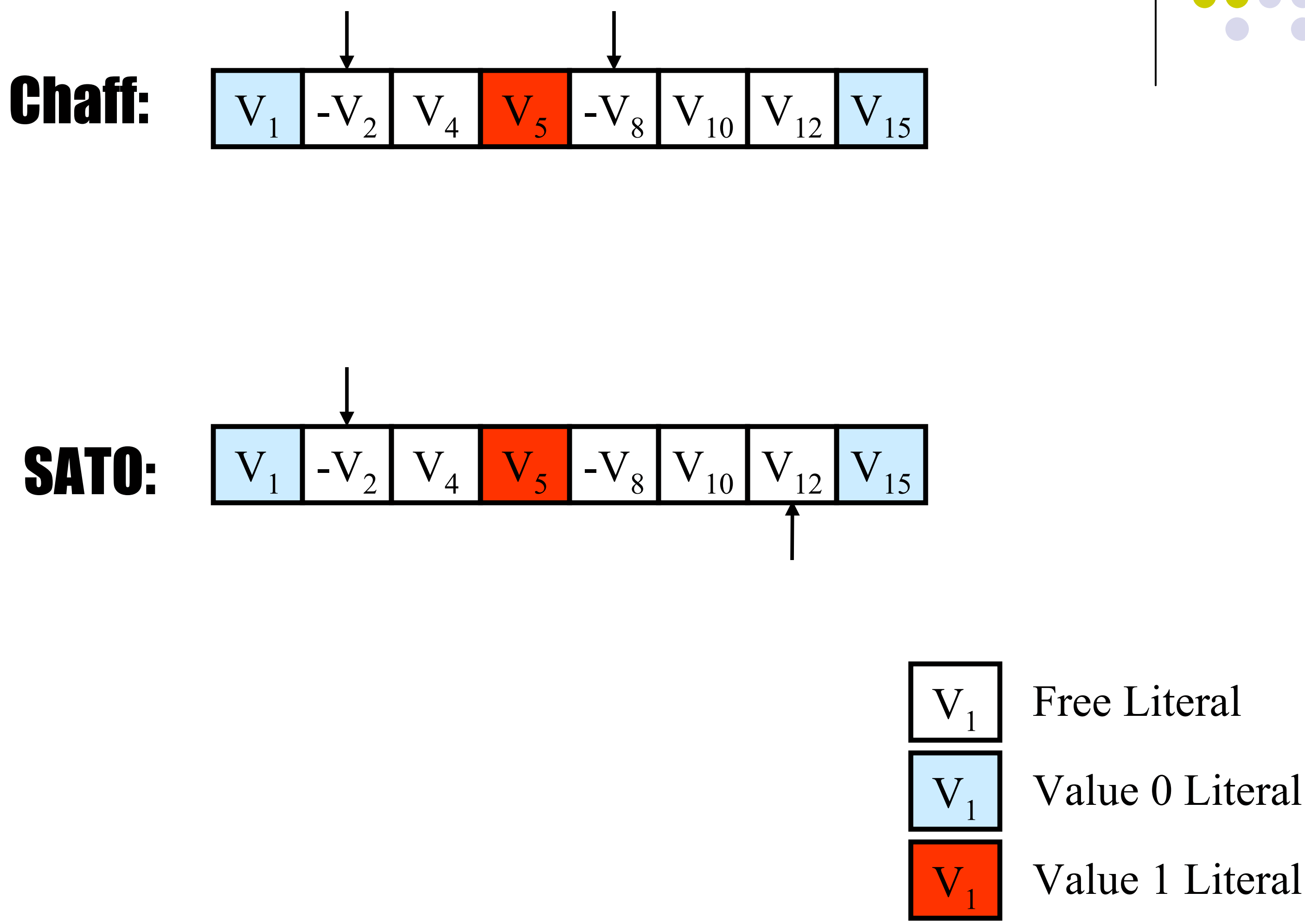
Difference of BCP in Chaff and SATO



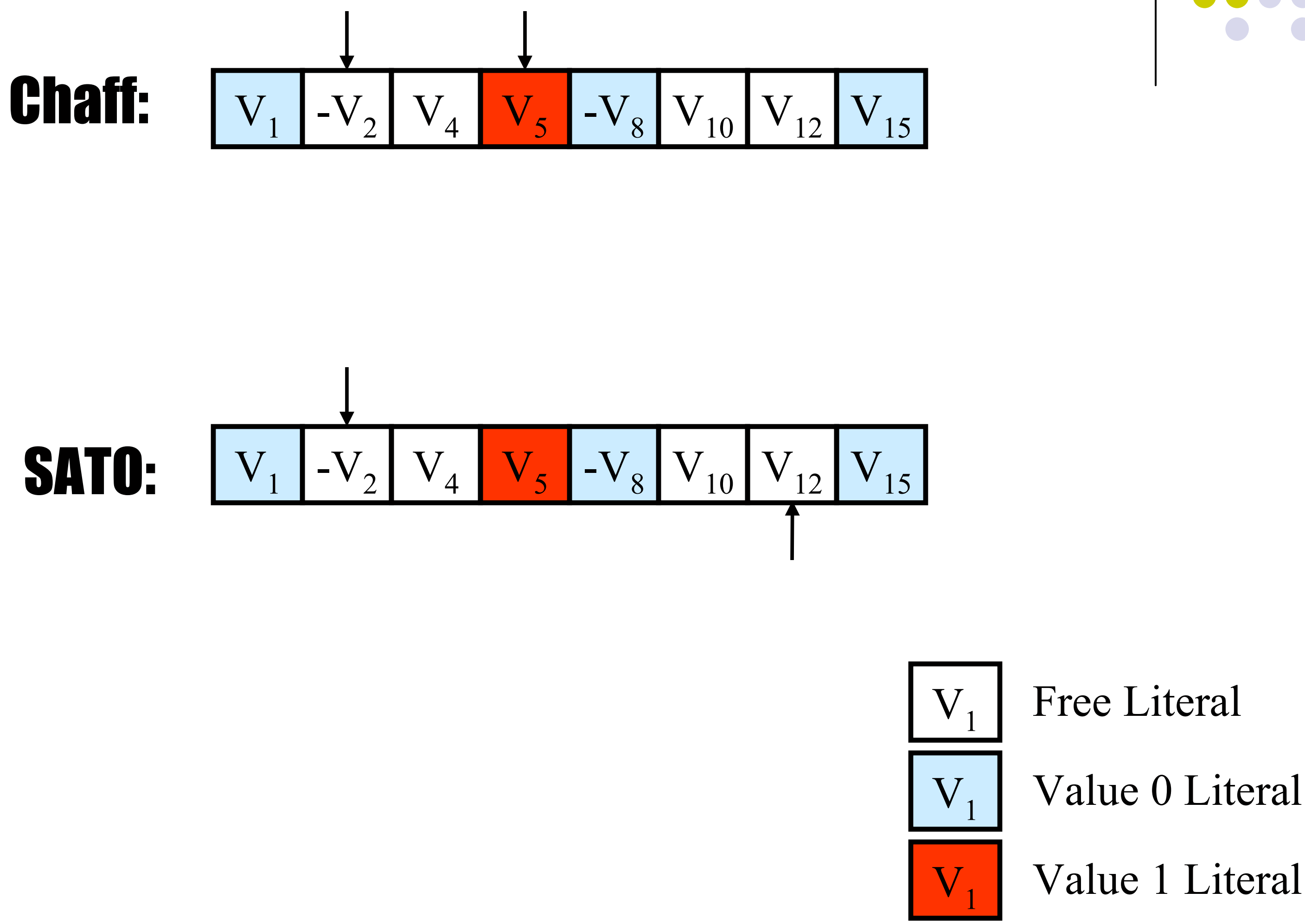
Difference of BCP in Chaff and SATO



Difference of BCP in Chaff and SATO



Difference of BCP in Chaff and SATO



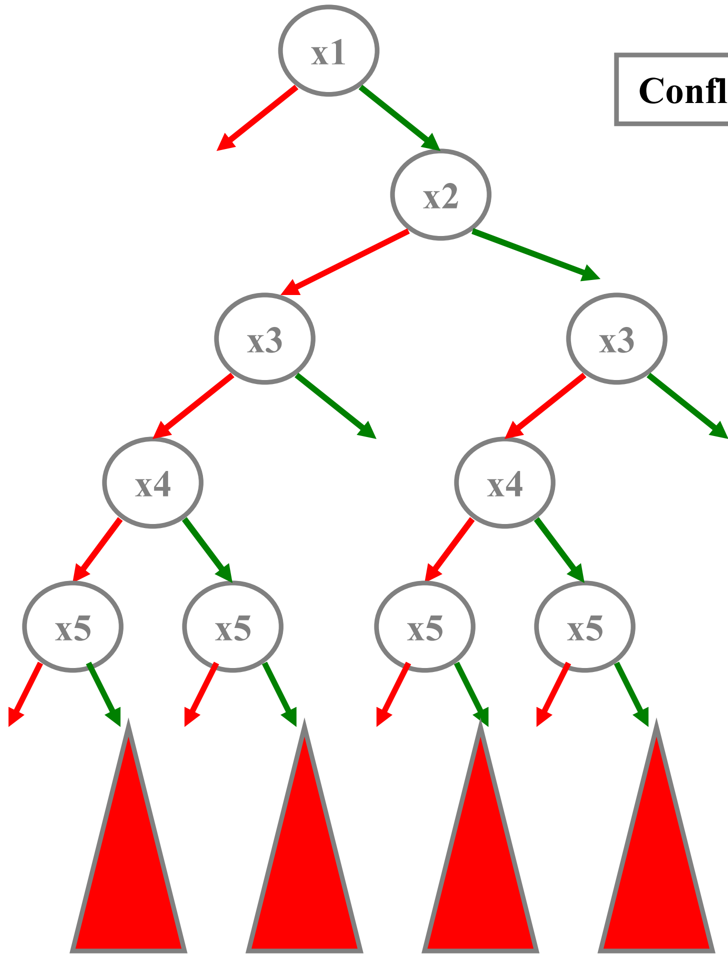
Efficient Implementation of SAT Solvers

```
while(1) {
   if (decide_next_branch()) { //Branching
       while(deduce()==conflict) { //Deducing
              blevel = analyze_conflicts(); //Learning
              if (blevel < 0)
                     return UNSAT;
              else back_track(blevel); //Backtracking
       }
   else //no branch means all variables got assigned.
       return SATISFIABLE;
   }
```

What is Learning?

- Adding information about the instance into the solution process without changing the satisfiability of the problem
  - In CNF representation, it is accomplished by the addition of clauses into the clause database.
- Knowledge of failure of search in a certain space may help search in other spaces
  - Conflict Driven Learning: record the reasons for failure of search as clauses, and add them to the database to help prune the space for future search
  - Non-Conflict Driven Learning:
    - Recursive learning
    - Probing
- Learning is very effective in pruning the search space for structured problems
  - It is of limited use for random instances
  - Why? It's still an open question.

What's the big deal?



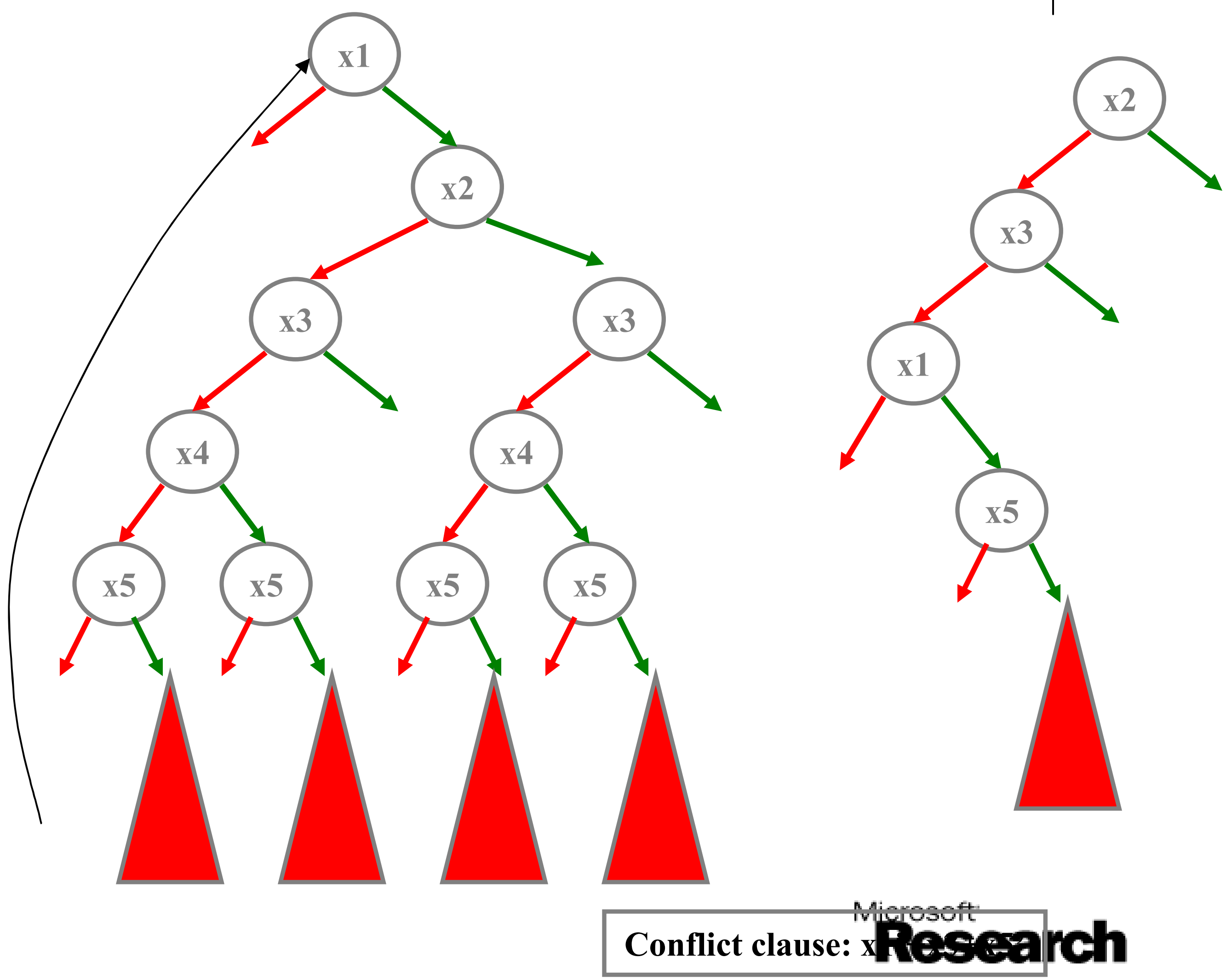
Conflict clause: x1'+x3+x5'

Significantly prune the search space – learned clause is useful forever!

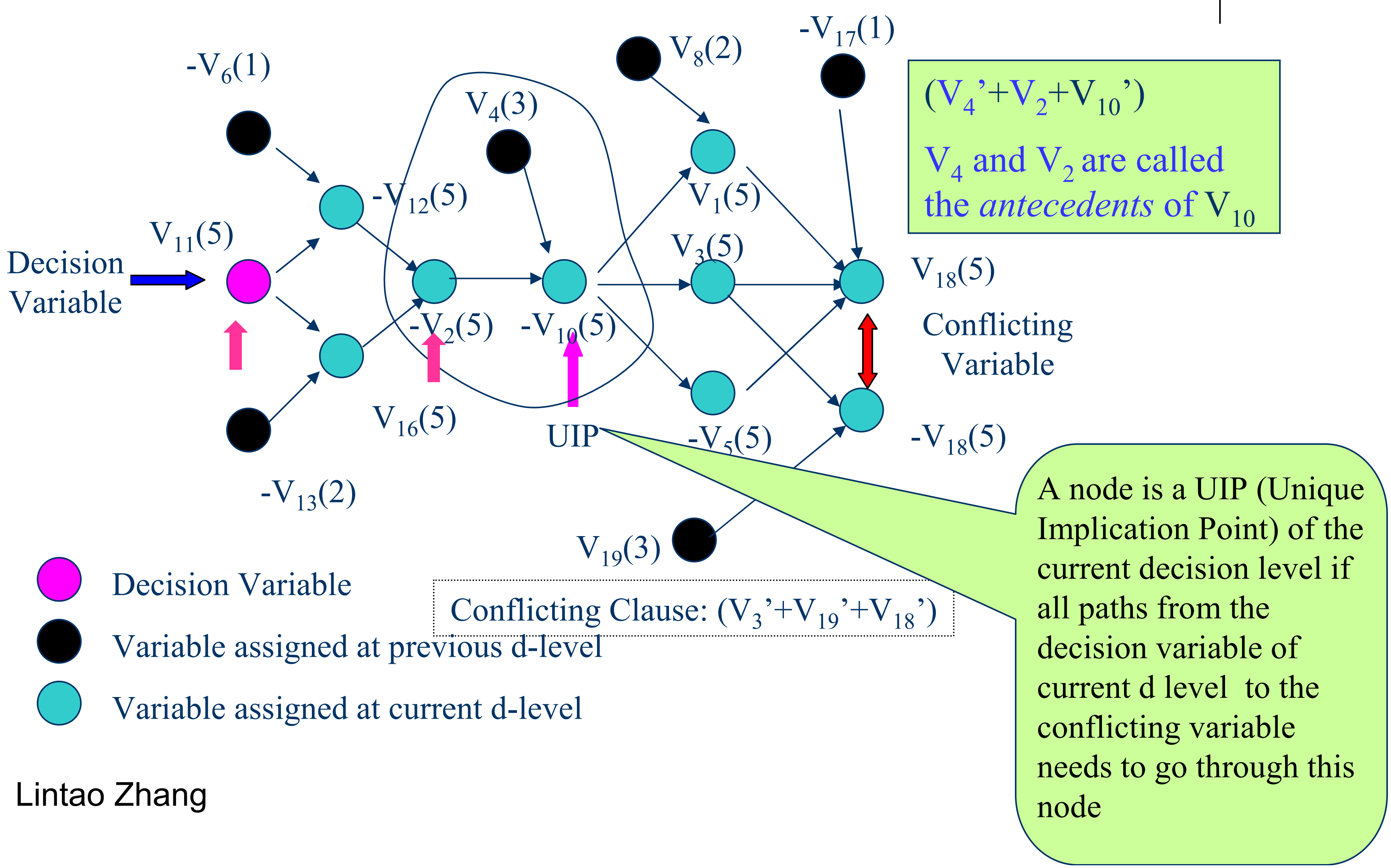
Useful in generating future conflict clauses.

Restart

- Abandon the current search tree and reconstruct a new one
- The clauses learned prior to the restart are *still there* after the restart and can help pruning the search space
- Adds to robustness in the solver

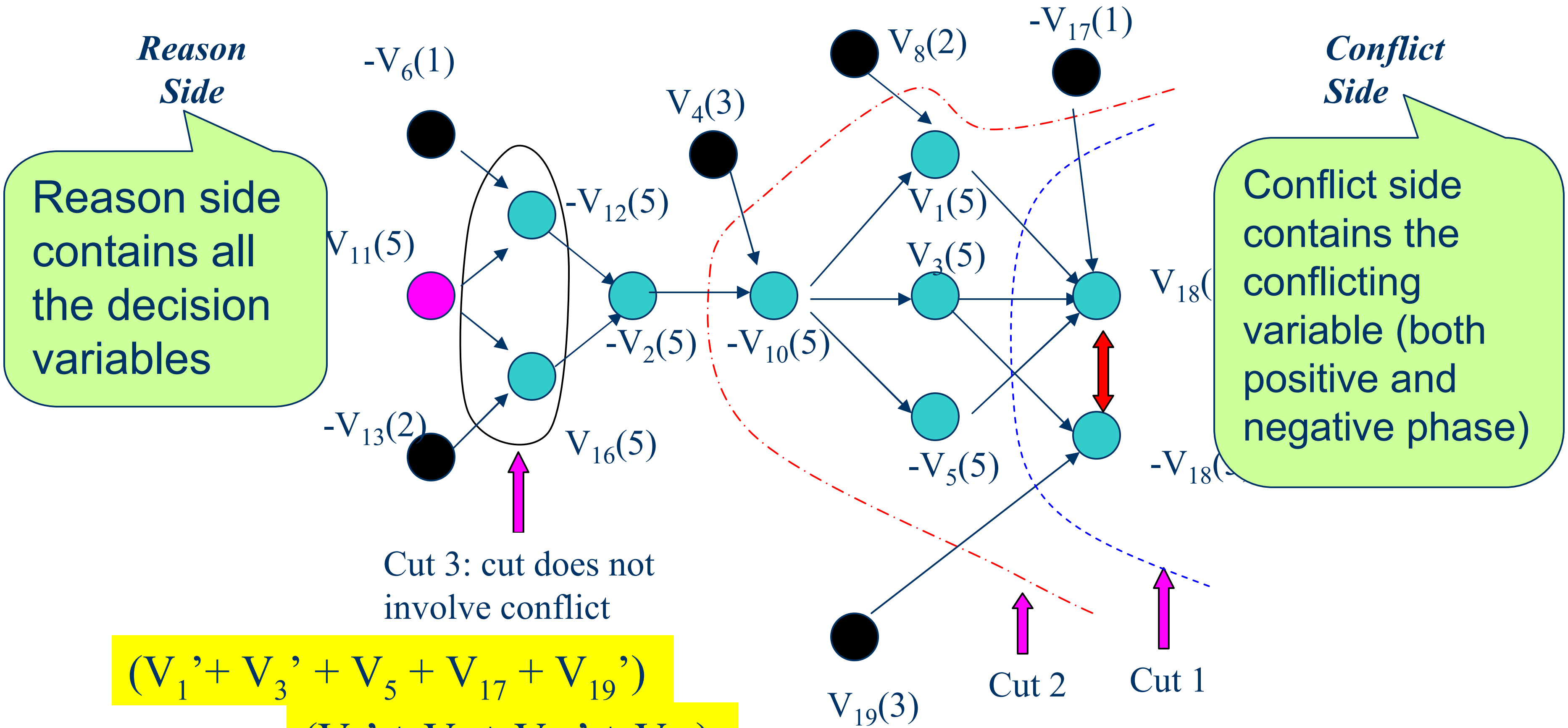


Implication Graph



Lintao Zhang

Bi-partition of the Implication Graph



$(V_1' + V_3' + V_5 + V_{17} + V_{19}')$

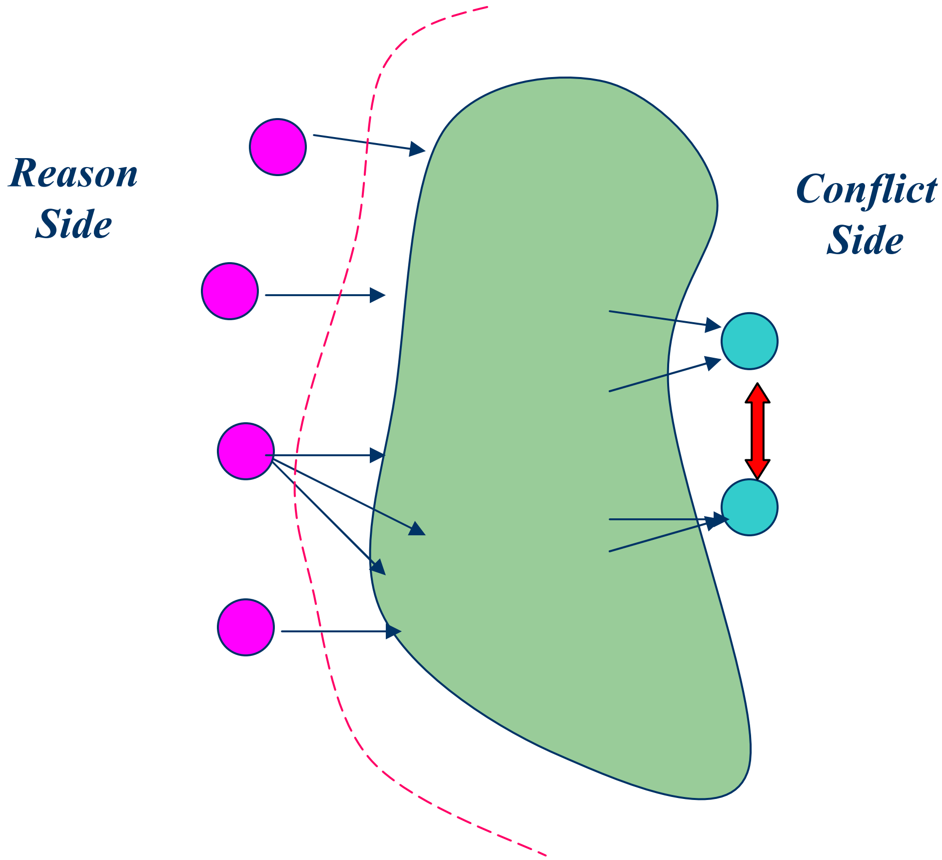
$(V_2' + V_6 + V_{11}' + V_{13})$

$(V_2 + V_4' + V_8' + V_{17} + V_{19}')$

Asserting Clauses and UIP

- Whenever a conflict occurs, conflict clauses can be generated and added to the database.
- If a conflict clause has only one literal at the highest decision level, after backtracking, the clause will become unit, and force the solver to explore a new search space.
  - Such a conflict clause is called an *asserting clause*.
  - It is desirable to make a conflict clause an asserting clause.
- To make a conflict clause an asserting clause, the partition needs to have
  - one UIP of the current decision level on the reason side
  - all vertices assigned after it on the conflict side.

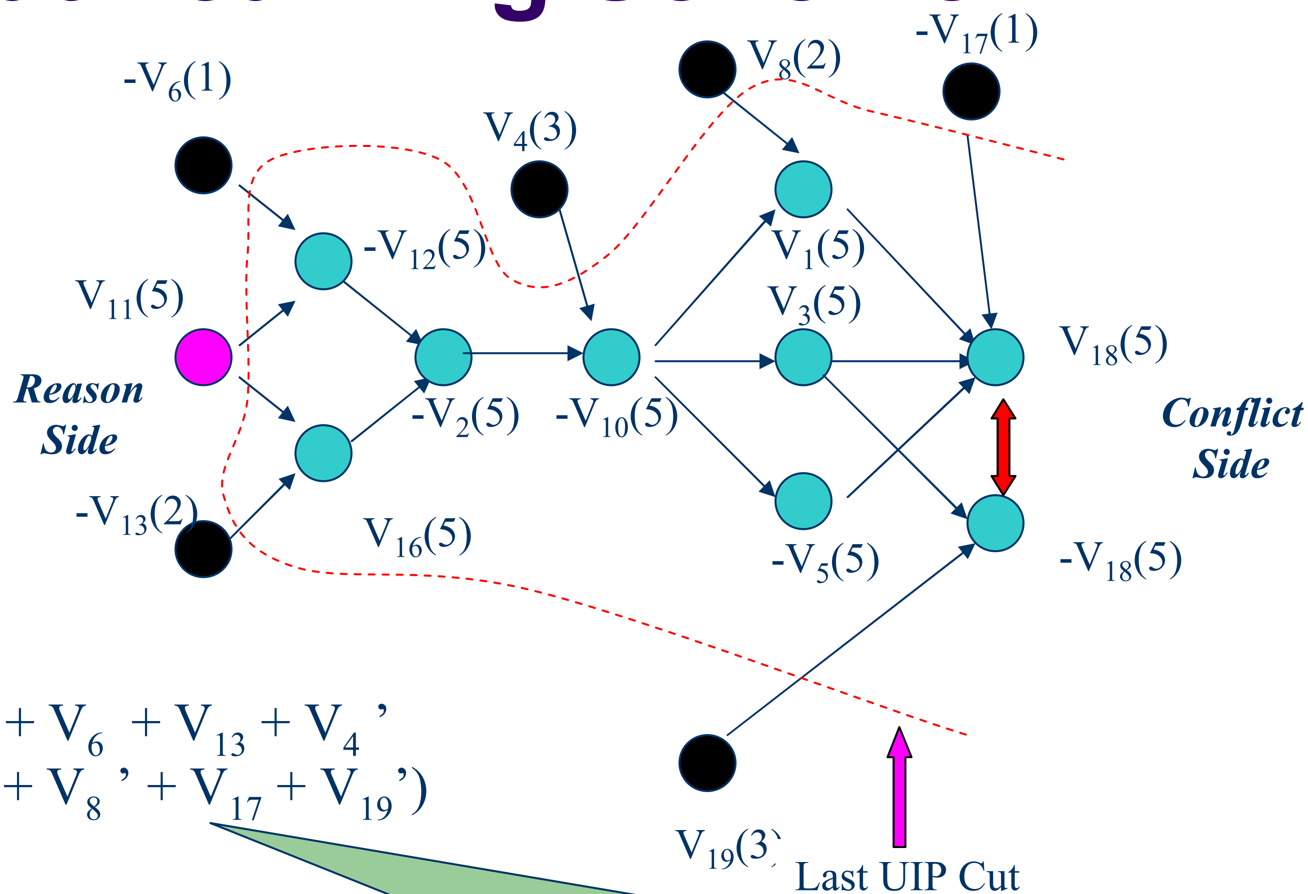
Decision Only Scheme



Conflict Clause consists of only decision variables.

Intuitively, this is not a good idea because the same decision sequence will not happen again. Therefore, this clause may be useful only if it does not contain all the decision variables.

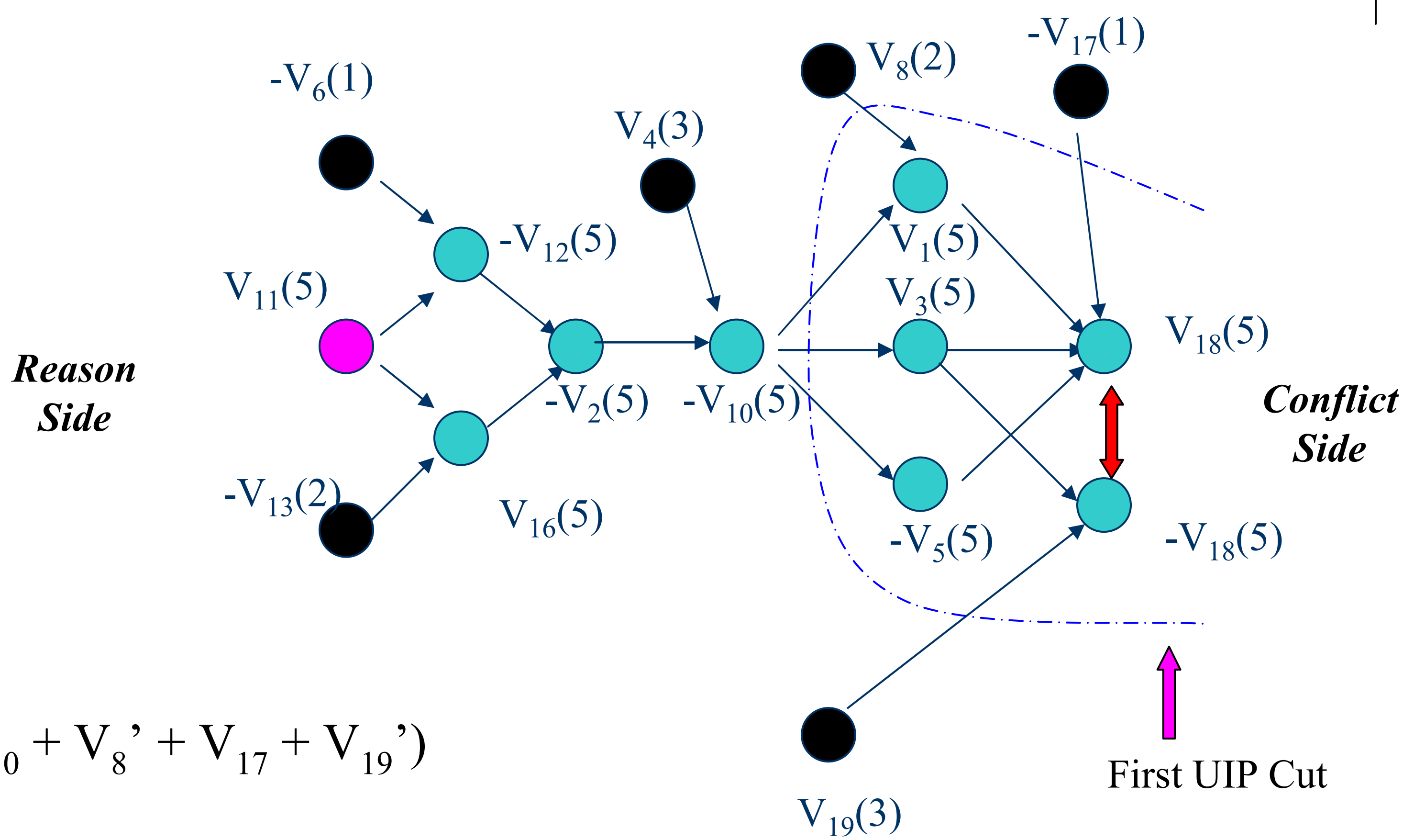
Relsat Learning Scheme



$$(V_{11}' + V_{6} + V_{13} + V_{4}' + V_{8}' + V_{17} + V_{19}')$$

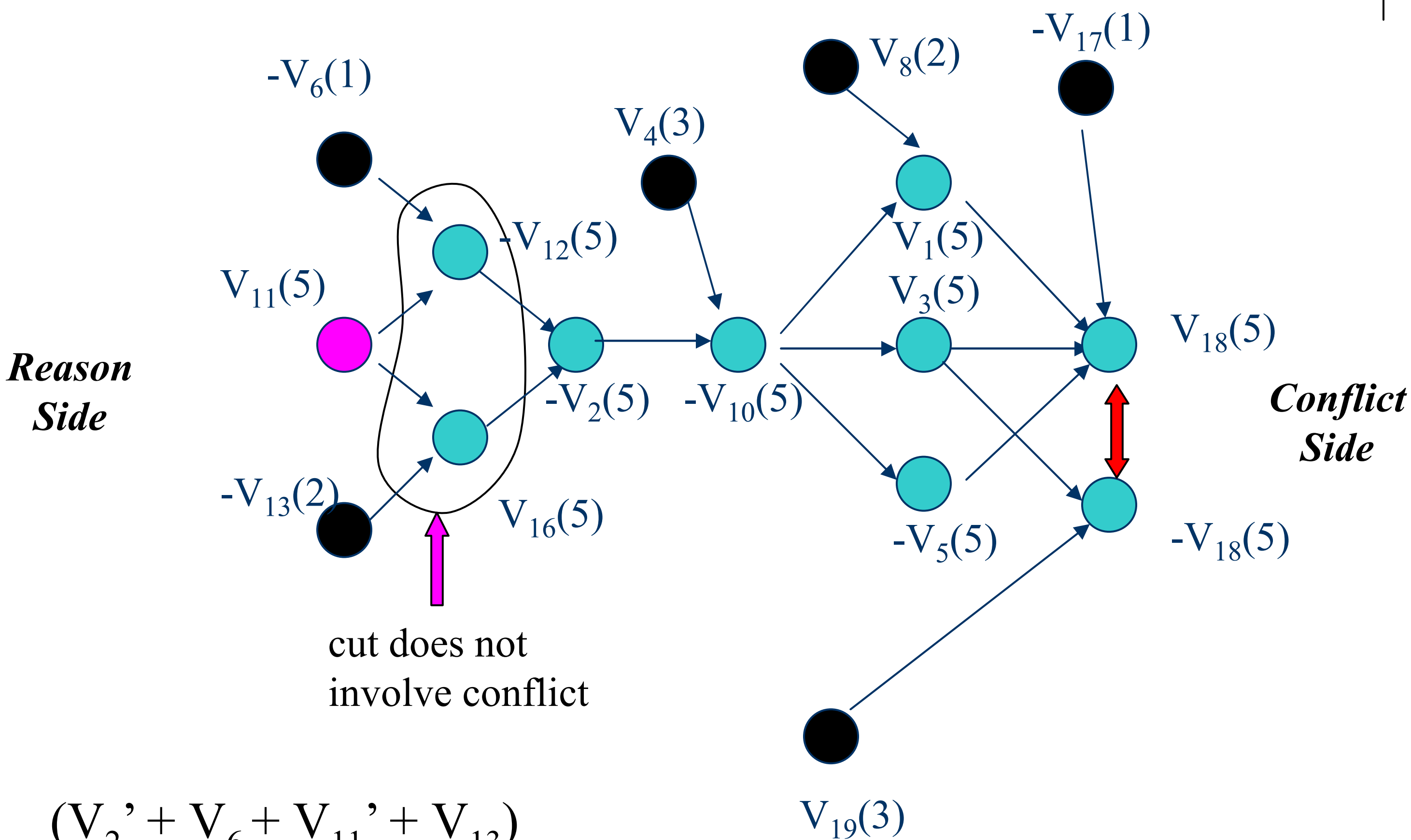
This clause only contains a single variable at the highest decision level. After backtracking and resolving the conflict, it will become a unit clause. The variable will be forced to flip, and it will assume the decision level of the highest of the rest literals

GRASP's Learning Scheme



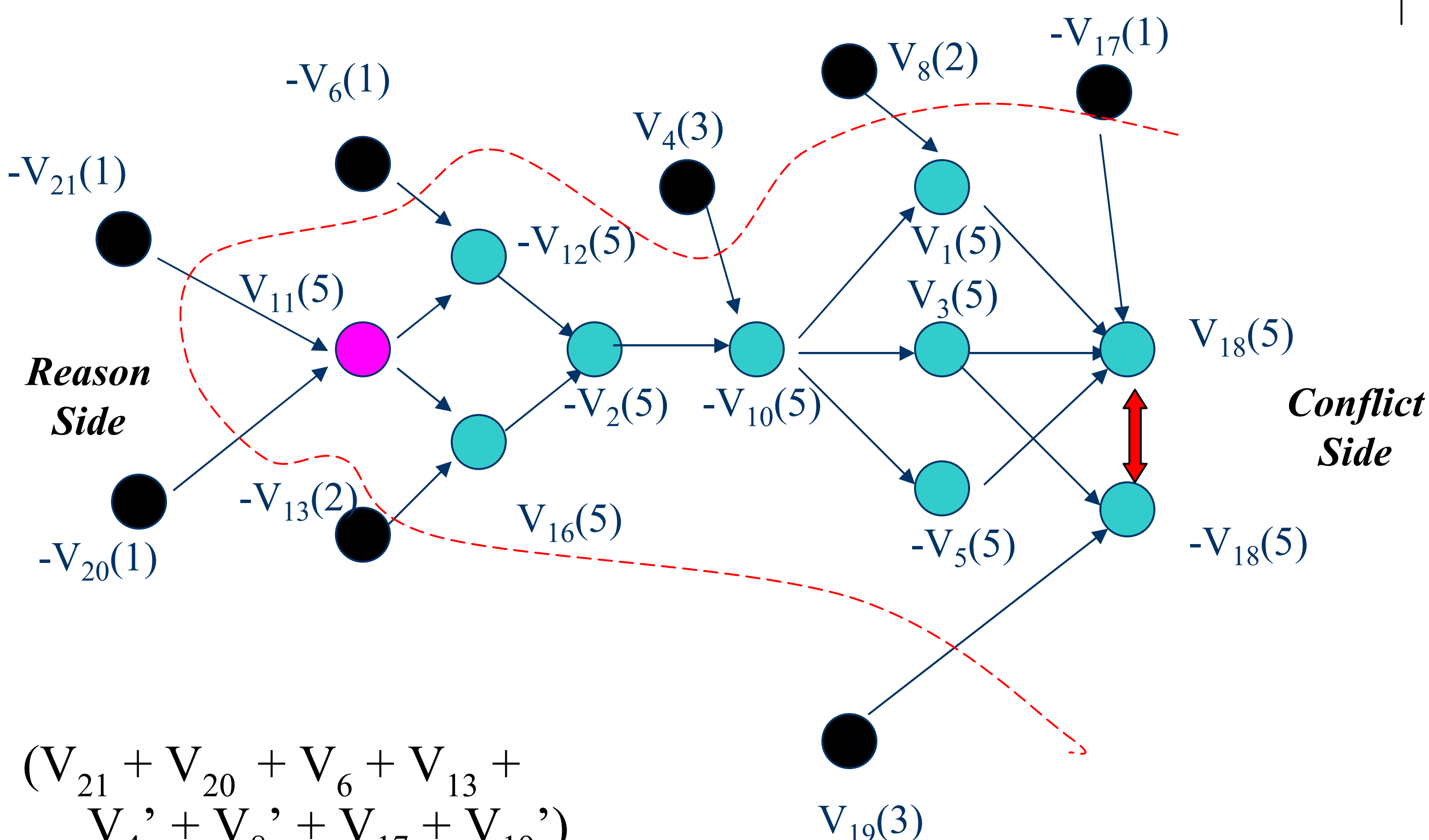
$(V_{10} + V_8' + V_{17} + V_{19}')$

GRASP's Learning Scheme



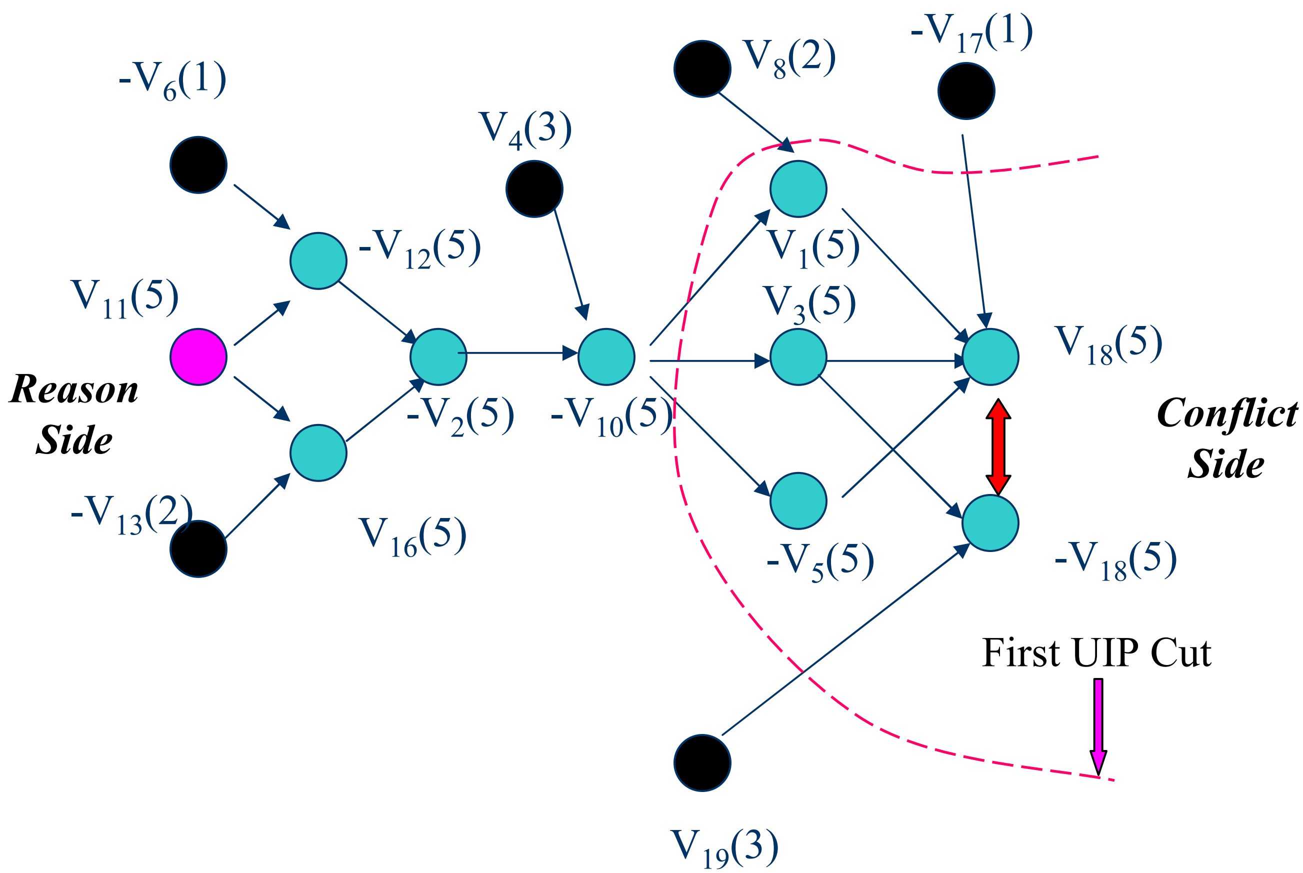
$(V_2' + V_6 + V_{11}' + V_{13})$

GRASP's Learning Scheme

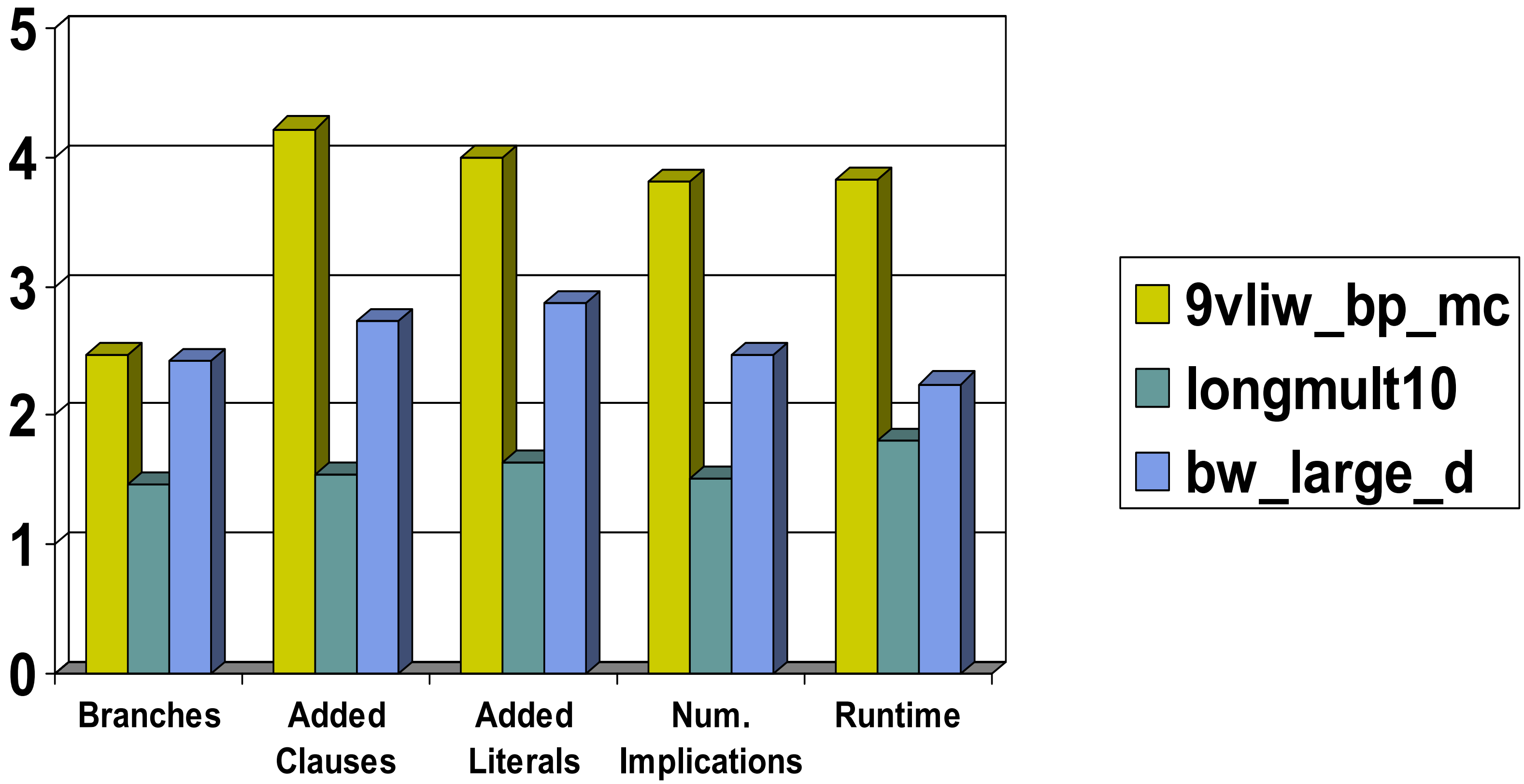


$$(V_{21} + V_{20} + V_{6} + V_{13} + V_{4}' + V_{8}' + V_{17} + V_{19}')$$

First UIP scheme

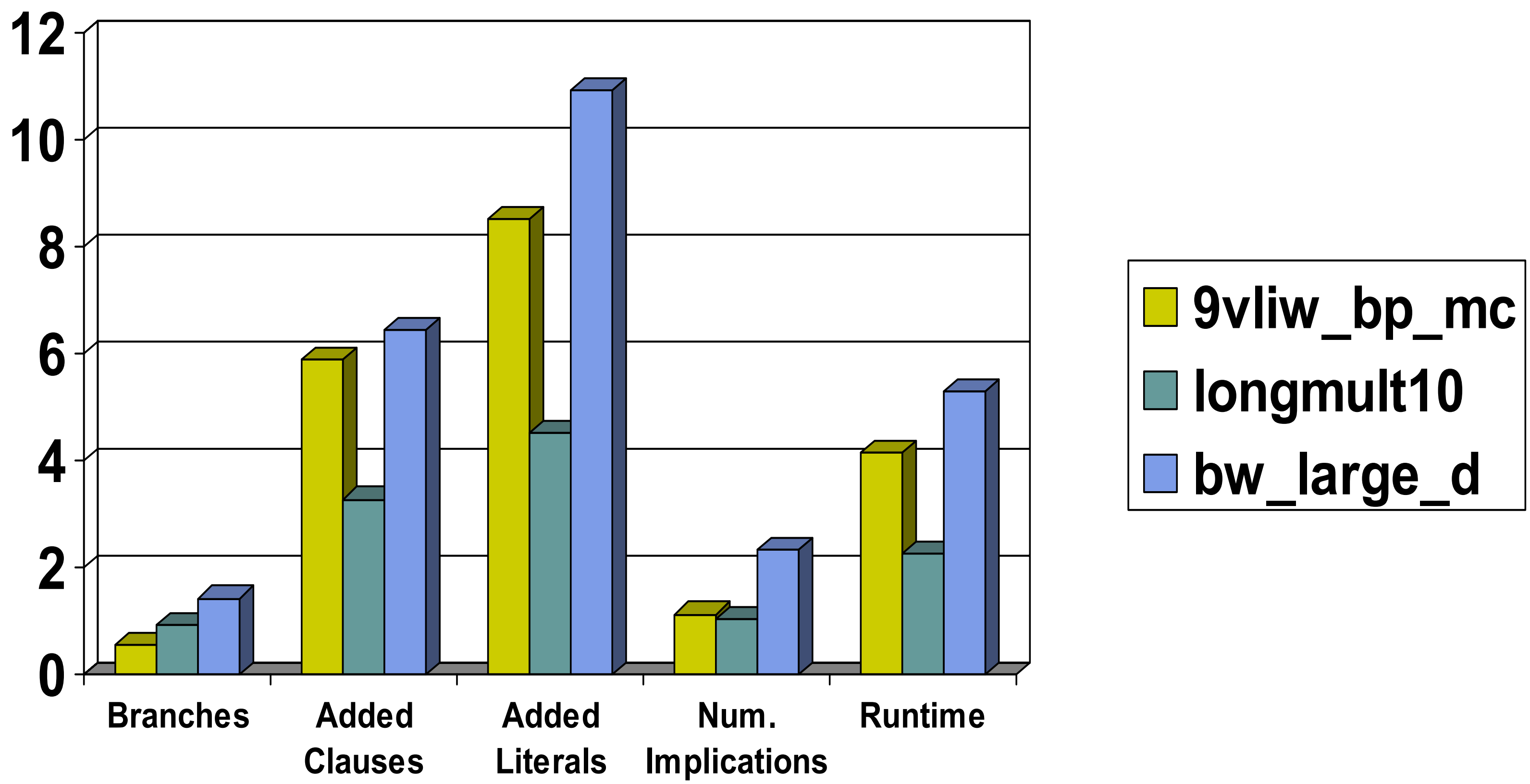


Relsat v.s. 1UIP



Ratio of statistics: relsat/1UIP

GRASP v.s. 1UIP



Ratio of statistics: GRASP/1UIP

Other Issues

- Random Restart
  - Periodically, throw away current search tree and start from the beginning
  - Very important for robustness
  - Employed by all modern SAT solvers
- Clause Deletion
  - Learned clauses slows down BCP, and eat up memory
  - Have to be deleted periodically
  - Various heuristics are proposed, based on
    - Clause age
    - Clause length
    - Clause relevance
    - Etc.

Engineering Issues

- Data structure tuning
  - Avoid linked list, always use array (vector)
- Memory management
  - Garbage collection
- Careful Coding
  - How to maintain the the decision priority queue?
- Cache performance

Cache Friendliness: Test Cases

		1dlx_c_mc_ex_bp_f	Hanoi4
Num Variables		776	718
Num Clauses		3725	4934
Num Literals		10045	12200
Z-Chaff	Run Time	0.22	3.94
	Branch	3166	8685
	Inst. Executed	86,610,942	1,299,030,113
SATO (-g100)	Run Time	4.41	10.91
	Branch	3771	4176
	Inst. Executed	620,374,889	1,762,565,056
Grasp	Run Time	11.78	26.50 (DNF)
	Branch	1795	1927
	Inst. Executed	1,415,933,580	2,914,228,045

Cache Friendliness (Data Only)

		1dlx_c_mc_ex_bp_f		Hanoi4	
		Num Access	Miss Rate	Num Access	Miss Rate
Z-Chaff	L1	24,029,356	4.75%	364,782,257	5.38%
	L2	1,659,877	4.63%	30,396,519	11.65%
SATO (-g100)	L1	188,352,975	36.76%	465,160,957	41.76%
	L2	79,422,894	9.74%	202,495,679	16.77%
Grasp	L1	415,572,501	32.89%	876,250,978	32.53%
	L2	153,490,555	50.25%	335,713,542	51.15%

The programs are compiled with –O3 using g++ 2.8.1(for GRASP and Chaff) or gcc 2.8.1 (for Sato3.2.1) on Sun OS 4.1.2 Trace was generated with QPT quick tracing and profiling tool. Trace was processed with dineroIV, the memory configuration is similar to a Pentium III processor:

L1: 16K Data, 16K Instruction, L2: 256k Unified. Both have 32 byte cache line, 4 way set associativity.