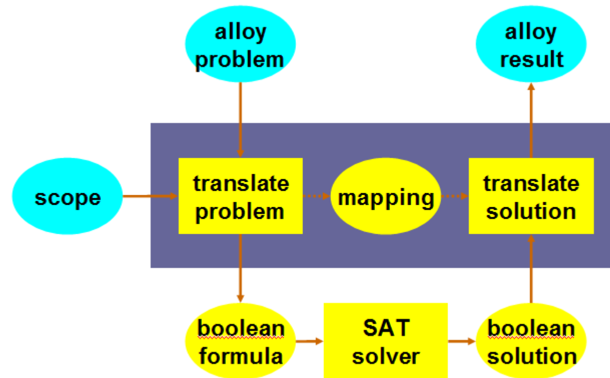
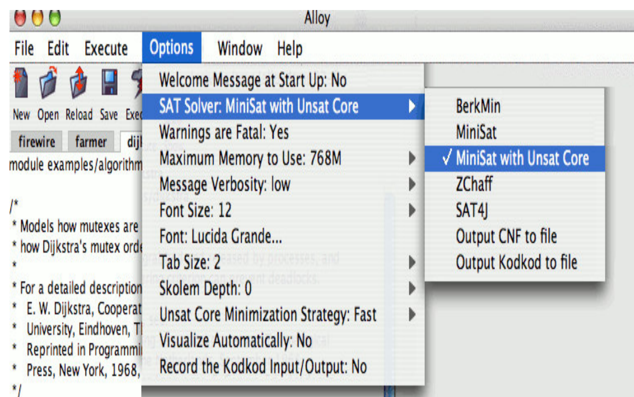
Alloy Analyzer Architecture

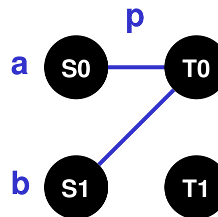


Analyzer Configuration



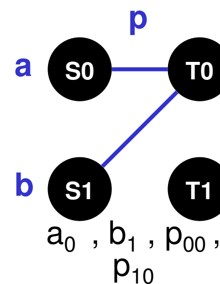
example

- problem
  a, b : S
  p : S -> T
  ! (a – b).p in (a.p – b.p)
- a model in a scope of 2
  S = {S0, S1}
  T = {T0, T1}
  p = {(S0, T0), (S1, T0)}
  a = {S0}
  b = {S1}



translation scheme

- represent
  - set as vector of bool var
    - a [a_0 a_1]
    - b [b_0 b_1]
  - relation as matrix
    p [p_{00} p_{01} , p_{10} p_{11}]
- translate
  - set expr to vector of bool formula
    - XT [a - b]$_i$ = XT [a]$_i$ $\wedge$ $\neg$XT [b]$_i$
    - XT [a . b]$_i$ = $\exists$j. XT [a]$_j$ $\wedge$ XT [b]$_{ji}$
  - relational expr to matrix of bool formula
  - formula to bool formulas



translation

a	$[a_0\ a_1]$
b	$[b_0\ b_1]$
p	$[p_{00}\ p_{01}\ ,\ p_{10}\ p_{11}]$
a – b	$[a_0 \wedge \neg b_0\ \ a_1 \wedge \neg b_1]$
(a – b).p	$[(a_0 \wedge \neg b_0 \wedge p_{00}) \vee (a_1 \wedge \neg b_1 \wedge p_{10})\ \ldots]$
a.p	$[(a_0 \wedge p_{00}) \vee (a_1 \wedge p_{10})\quad (a_0 \wedge p_{01}) \vee (a_1 \wedge p_{11})]$
b.p	$[(b_0 \wedge p_{00}) \vee (b_1 \wedge p_{10})\quad (b_0 \wedge p_{01}) \vee (b_1 \wedge p_{11})]$
a.p – b.p	$[((a_0 \wedge p_{00}) \vee (a_1 \wedge p_{10})) \wedge \neg ((b_0 \wedge p_{00}) \vee (b_1 \wedge p_{10}))\ \ldots]$

! (a – b).p in (a.p – b.p)

$$\neg (((a_0 \wedge \neg b_0 \wedge p_{00}) \vee (a_1 \wedge \neg b_1 \wedge p_{10}) \Rightarrow ((a_0 \wedge p_{00}) \vee (a_1 \wedge p_{10})) \wedge \neg ((b_0 \wedge p_{00}) \vee (b_1 \wedge p_{10})))) \wedge \ldots$$

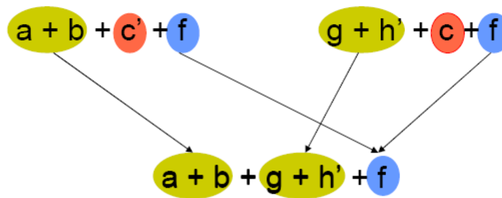
Once built, remember the mapping and pass the formula to a SAT Solver

Many SAT Solvers available

- Most of them have a standardized input interface.
- Therefore one could freely choose one and test the performance
  - ZCHAFF
  - MiniSat
  - Berkmin
  - SAT4J

Resolution

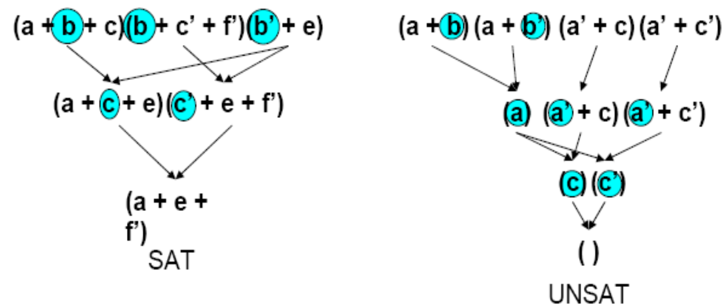
- Based on resolution
  - Resolution of a pair of clauses with exactly **ONE** incompatible variable
    - Two clauses are said to have distance 1
    - (a+b)(a'+c) = (a+b)(a'+c)(b+c)



David Putnam

M .Davis, H. Putnam, "A computing procedure for quantification theory", *J. of ACM*, Vol. 7, pp. 201-214, 1960

- Iteratively select a variable for resolution till no more variables are left.
- Can discard all original clauses after each iteration.



Potential memory explosion problem!

DLL Algorithm

- M. Davis, G. Logemann and D. Loveland, "A Machine Program for Theorem-Proving", *Communications of ACM*, Vol. 5, No. 7, pp. 394-397,1962
- Search based
- Basic framework for many modern SAT solvers
- Also known as DPLL for historical reasons

Deduction Rules for SAT

- Unit Literal Rule: If an unsatisfied clause has all but one of its literals evaluate to 0, then the *free* literal must be implied to be 1.

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

- Conflicting Rule: If all literals in a clause evaluate to 0, then the formula is unsatisfiable in this branch.

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

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)

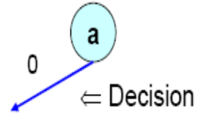
Basic DLL Procedure - DFS

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

a

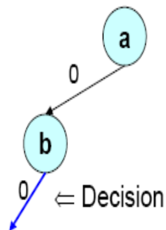
Basic DLL Procedure - DFS

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



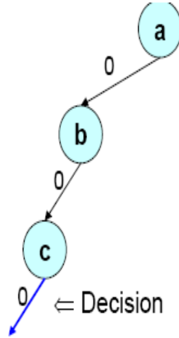
Basic DLL Procedure - DFS

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



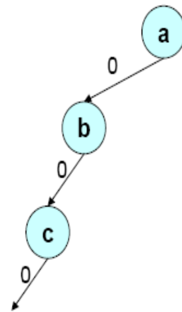
Basic DLL Procedure - DFS

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

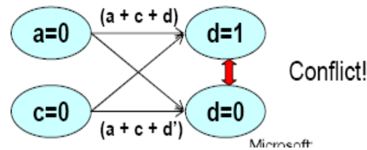


Basic DLL Procedure - DFS

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

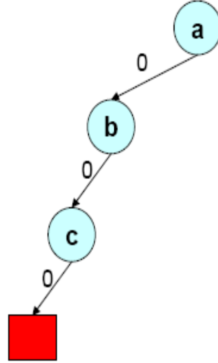


Implication Graph

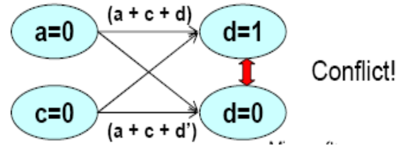


Basic DLL Procedure - DFS

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

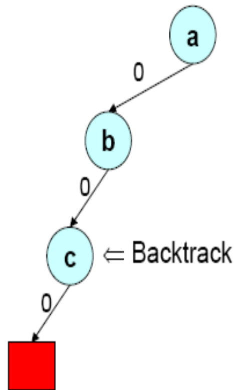


Implication Graph



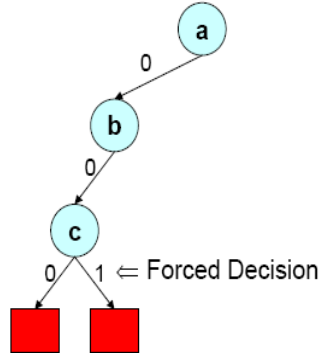
Basic DLL Procedure - DFS

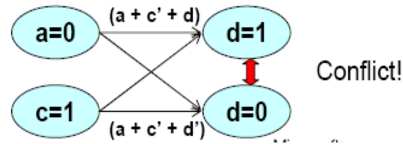
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

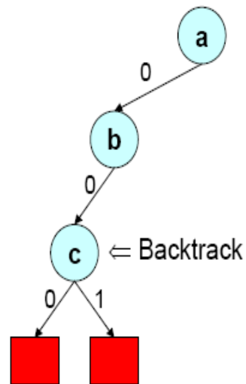
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)





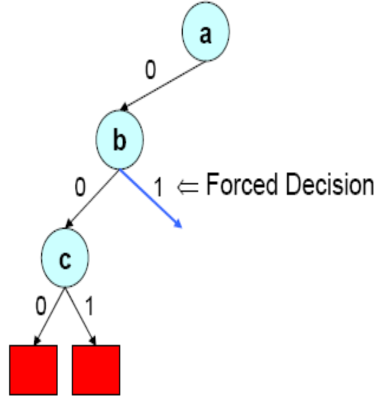
Basic DLL Procedure - DFS

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



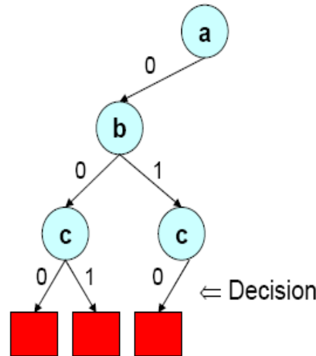
Basic DLL Procedure - DFS

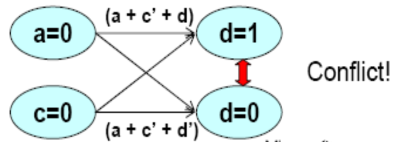
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

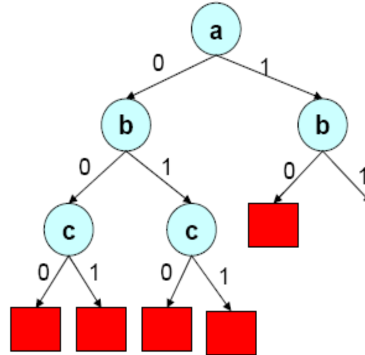
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

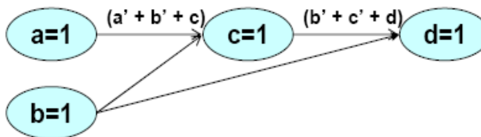




Basic DLL Procedure - DFS

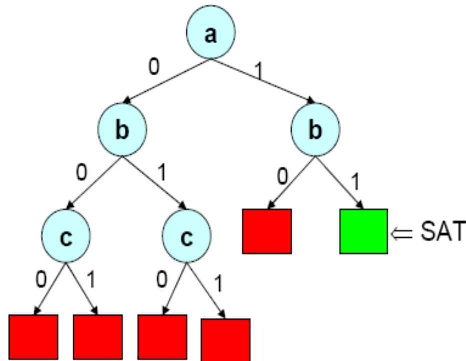
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)

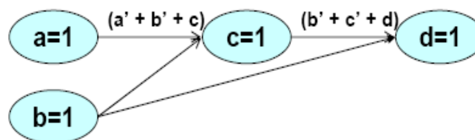




Basic DLL Procedure - DFS

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





Implications and BCP

- Implication
  - A variable is forced to be assigned to be True or False based on previous assignments.
- Unit clause rule (rule for elimination of one literal clauses)
  - An <u>unsatisfied</u> clause is a <u>unit</u> clause if it has exactly one unassigned literal.

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

$a = T$, $b = T$, c is unassigned

Satisfied Literal
Unsatisfied Literal
Unassigned Literal

  - The unassigned literal is implied because of the unit clause.
- Boolean Constraint Propagation (BCP)
  - Iteratively apply the unit clause rule until there is no unit clause available.
- Workhorse of DLL based algorithms.

Features of DLL

- Eliminates the exponential memory requirements of DP
- Exponential time is still a problem
- Limited practical applicability – largest use seen in automatic theorem proving
- The original DLL algorithm has seen a lot of success for solving random generated instances.

Chaff Philosophy

- Make the core operations fast
  - profiling driven, most time-consuming parts:
    - Boolean Constraint Propagation (BCP) and Decision
- Emphasis on coding efficiency
- Emphasis on optimizing data cache behavior
- Search space pruning: conflict resolution and learning

Based on DLL Algorithm

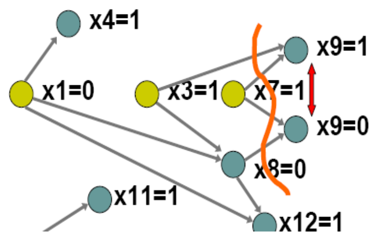
```
while(1) {
    if (decide_next_branch()) {             //Branching
        while(deduce()==conflict) {         //Deducing
            blevel = analyze_conflicts();   //Learning
            if (blevel < 0)
                return UNSAT;
            else back_track(blevel);        //Backtracking
  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

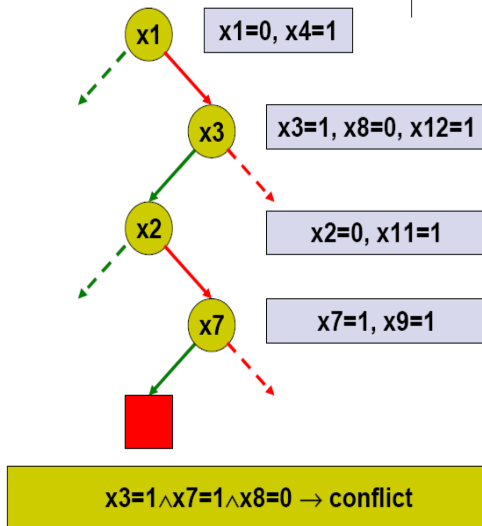
Analyze conflicts

- Backtracking to the highest decision level that has not been tried with both values
  - OK for randomly generated instances, bad for instances generated in practical applications
- Conflict Driven Learning and Non-Chronological Backtracking
  - Serveral approaches

Example(1)

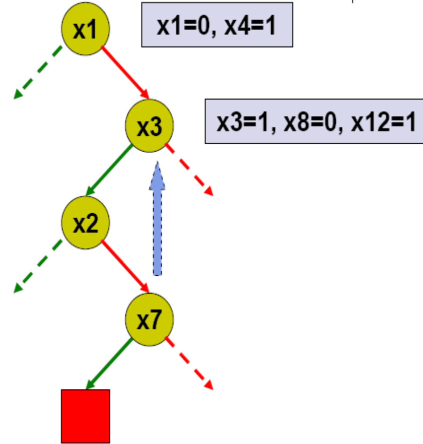
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

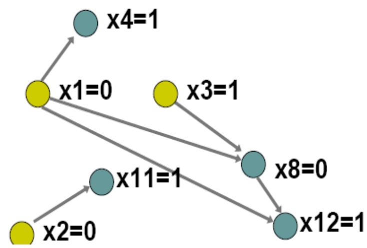




Example(2)

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'

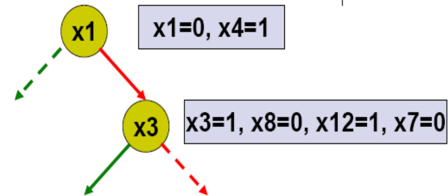


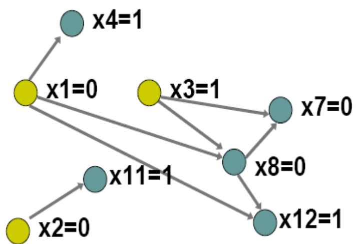


Backtrack to the decision level of x3=1:
x7 = 0

Example(3)

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'





Based on DLL Algorithm

```
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;
}
```


Decide next branch

- 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

Based on DLL Algorithm

```
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;
}
```


Chaff's Main Procedures

- Efficient BCP
  - Two watched literals
  - Fast backtracking
- Efficient decision heuristic
  - Localizes search space
- Restart
  - Increases robustness

Implication

- What "causes" an implication?
- When can it occur?
- All literals in a clause but one are assigned to False

Implication example

- The clause (v1 + v2 + v3) implies values only in the following cases
- In case (F + F + v3)
  - implies v3=T
- In case (F + v2 + F)
  - implies v2=T
- In case (v1 + F + F)
  - implies v1=T

Implication for N-literal clause

- Implication occurs after N-1 assignments to False to its literals
- Theoretically, we could ignore the first N-2 assignments to this clause
- The first N-2 assignments won't have any effect on the BCP

Watched Literals

- Each clause has two watched literals
- Ignore any assignments to the other literals in the clause.
- BCP Maintains the following invariant
  - By the end of BCP, one of the watched literal is true or both are undefined.
- Guaranteed to find all implications

BCP with watched Literals

- Identifying conflict clauses
- Identifying unit clauses
- Identifying associated implications
- Maintaining “BCP Invariant”

Example (1/13)

v2 + v3 + v1 + v4
v1 + v2 + v3’
v1 + v2’
v1’+ v4

Example (2/13)

Watched literals

v2 + v3 + v1 + v4
v1 + v2 + v3′
v1 + v2′
v1′+ v4

Example (3/13)

Stack:(**v1=F**)

v2 + v3 + v1 + v4
v1 + v2 + v3′
v1 + v2′
v1′+ v4

Assume we decide to set v1 the value F

Example (4/13)

Stack:(**v1=F**)

v2 + v3 + v1 + v4
v1 + v2 + v3′
v1 + v2′
→ v1′+ v4

- Ignore clauses with a watched literal whose value is T.

Example (5/13)

Stack:(**v1=F**)

→ v2 + v3 + v1 + v4
v1 + v2 + v3′
v1 + v2′
v1′+ v4

- Ignore clauses where neither watched literal value changes

Example (6/13)

Stack:(**v1=F**)

v2 + v3 + v1 + v4
→ v1 + v2 + v3'
→ v1 + v2'
v1'+ v4

- Examine clauses with a watched literal whose value is F

Example (7/13)

v2 + v3 + v1 + v4
v1 + v2 + v3'
v1 + v2'
v1'+ v4

Stack:(**v1=F**)

→

v2 + v3 + v1 + v4
v1 + v2 + v3'
v1 + v2'
v1'+ v4

Stack:(**v1=F**)

- In the second clause, replace the watched literal v1 with v3'

Example (8/13)

v2 + v3 + v1 + v4
v1 + v2 + v3′
v1 + v2′
v1′+ v4

Stack:(**v1=F**)

→

v2 + v3 + v1 + v4
v1 + v2 + v3′
v1 + v2′
v1′+ v4

Stack:(v1=F)
Pending: (v2=F)

- The third clause is a unit and implies v2=F
- We record the new implication, and add it to a queue of assignments to process.

Example (9/13)

v2 + v3 + v1 + v4
v1 + v2 + v3′
v1 + v2′
v1′+ v4

Stack:(v1=F, **v2=F**)

→ v2 + v3 + v1 + v4
→ v1 + v2 + v3′
v1 + v2′
v1′+ v4

Stack:(v1=F, v2=F)
Pending: (v3=F)

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

Example (10/13)

v2 + v3 + v1 + v4 → v2 + v3 + v1 + v4
v1 + v2 + v3' → v1 + v2 + v3'
v1 + v2'   v1 + v2'
v1' + v4   v1' + v4

Stack:(v1=F, **v2=F**)

Stack:(v1=F, v2=F)
Pending: (v3=F)

- In the first clause, we replace v2 with v4
- The second clause is a unit and implies v3=F
- We record the new implication, and add it to the queue

Example (11/13)

v2 + v3 + v1 + v4 → v2 + v3 + v1 + v4
v1 + v2 + v3'   v1 + v2 + v3'
v1 + v2'   v1 + v2'
v1' + v4   v1' + v4

Stack:(v1=F, v2=F, **v3=F**)

Stack:(v1=F, v2=F, v3=F)
Pending: ()

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

Example (12/13)

v2 + v3 + v1 + v4
v1 + v2 + v3'
v1 + v2'
v1'+ v4

Stack:(v1=F, v2=F, **v3=F**)

→

v2 + v3 + v1 + v4
v1 + v2 + v3'
v1 + v2'
v1'+ v4

Stack:(v1=F, v2=F, v3=F)
Pending: (v4=T)

- The first clause is a unit and implies v4=T.
- We record the new implication, and add it to the queue.

Example (13/13)

v2 + v3 + v1 + v4
v1 + v2 + v3'
v1 + v2'
v1'+ v4

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

- There are no pending assignments, and no conflict
- Therefore, BCP terminates and so does the SAT solver

Identify conflicts

v2 + v3 + v1
v1 + v2 + v3'
v1 + v2'
v1'+ v4

Stack:(v1=F, v2=F, **v3=F**)

- What if the first clause does not have v4?
- When processing v3', we examine the first clause.
- This time, there is no alternative literal to watch.
- BCP returns a conflict

Backtrack

v2 + v3 + v1
v1 + v2 + v3'
v1 + v2'
v1'+ v4

Stack:()

- We do not need to move any watched literal

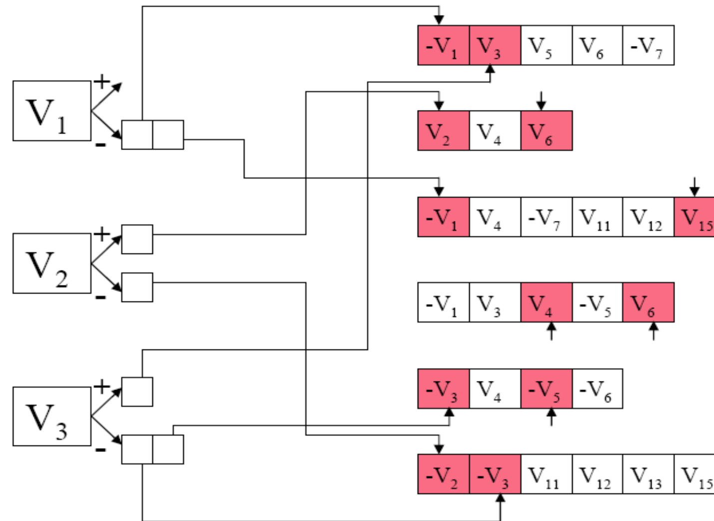
BCP Summary

- During forward progress (decisions, implications)
  - Examine clauses where watched literal is set to F
  - Ignore clauses with assignments of literals to T
  - Ignore clauses with assignments to non-watched literals

Backtrack Summary

- Unwind Assignment Stack
- No action is applied to the watched literals
- Overall
  - Minimize clause access

Implementation



Description of the Algorithm

- When var ***v*** is assigned 1, for each literal ***p*** pointed to by a pointer in the list of ***neg_watched(v)***, the solver searches for a literal ***l*** in the clause (containing ***p***) that ***is not set to 0.***

Four cases may occur:

- if the only such ***l*** is the other watched literal and it *evaluates to 1* - clause is *SAT*, do nothing.
- if exists such literal ***l*** and it is not the other watched literal, then e remove to p from ***neg_watched(v)***, and add pointer to ***l*** to the watched list of the variable corresponding to ***l*** - moving watched literal.
- if the only such ***l*** is the other watched literal and it is *free*, then - unit clause, and the other watched literal is the unit literal.
- if all literals in the clause are 0's, and no such ***l*** exists - a conflicting clause.

Conclusion

- Undoing a var assignment during backtrack takes constant time.
- The two watched literals are the last to be assigned to 0 => any backtracking will make sure that the literals being watched are either unassigned or assigned to 1.
- No updates on pointers for the literals being watched.
- The fastest algorithm today.

Conclusion

- Alloy (predicate logic with relational extension) can be translated to propositional logic for a defined scope
- SAT solver can be used to check Satisfiability
  - Fast algorithms were described
  - Also memory efficent
  - Practically limited to number of variables and clauses