Boolean Satisfiability

ICS 275

Winter 2016

Outline

Conflict Analysis: Implication Grpahs

- The combination of these techniques makes sure that unit resolution is empowered every time a conflict arises and that the solver will not repeat any mistake. The identification of conflict–driven clauses is done through a process known as conflict analysis, which analyzes a trace of unit resolution known as the implication graph.

Our csp conflict did not take arc-consistency into account
In SAT, conflict-dirven analysis does.

Implication graphs

=

1. {A,B}
2. {B,C}
3. { ¬A, ¬X, Y }
4. { ¬A,X,Z}
5. { ¬A, ¬Y,Z}
6. { ¬A,X, ¬Z}
7. { ¬A, ¬Y, ¬Z}

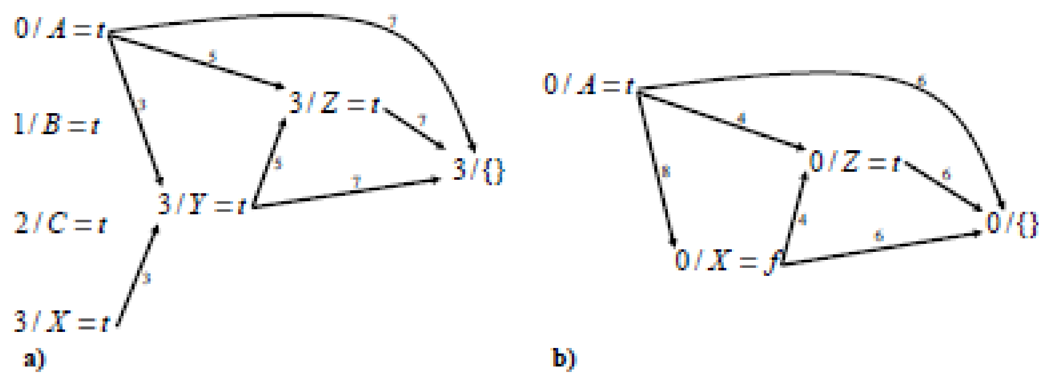


Figure 3.7. Two implication graphs.

Deriving conflict clause

- Every cut in the implication graph defines a conflict set as long as that cut seperates the decision variables (root nodes) from the contradiction (a leaf node).
- Any node (variable assignment) with an outgoing edge that cross the cut will be in the conflict set.

leading to conflict sets:{A=true,X=true}, {A=true, Y=true} and {A=true, Y=true,Z=true}.

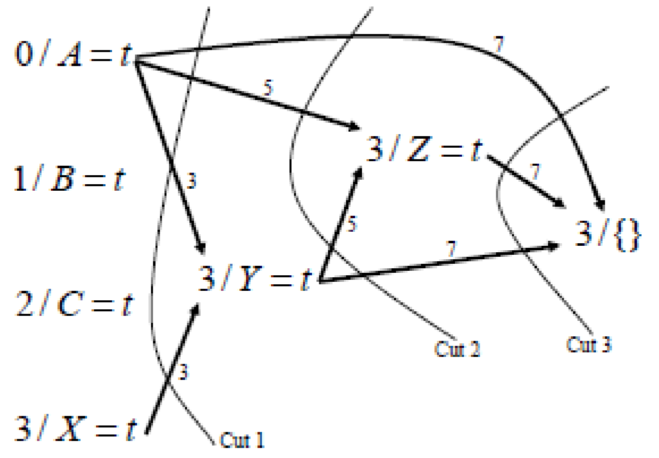


Figure 3.8. Three cuts in an implication graph, leading to three conflict sets.

Earliest minimal conflict?

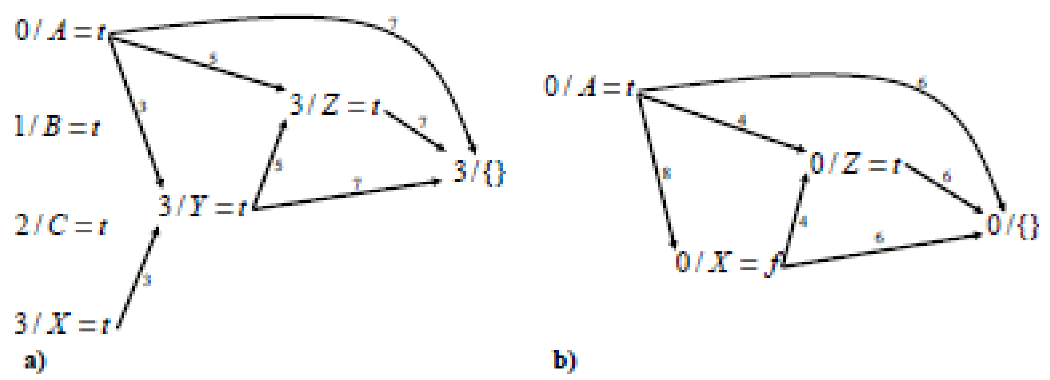


Figure 3.7. Two implication graphs.

For the graph in Figure 3.7(b), {A = true} is a conflict cut.

Conflict–driven clauses generated from cuts that contain exactly one variable assigned at the level of conflict are said to be asserting [ZMMM01]. Modern SAT solvers insist on learning only asserting clauses.

Where do we do with the conflict clauses?

The process of adding a conflict–driven clause to the CNF is
known as clause learning [MSS99, BS97, ZMMM01, BKS04]. A key question in
this regard is when exactly to add this clause. Consider the termination tree in
Figure 3.6 and the left most leaf node corresponding to the CNF |A,B,C,X.
Unit resolution discovers a contradiction in this CNF and by analyzing the implication
graph in Figure 3.7(a), we can identify the conflict–driven clause $\neg A \vee \neg X$.

The question now is: What to do next?
Since the contradiction was discovered after setting variable X to true, we
know that we have to at least undo that decision. Modern SAT solvers, however,
will undo all decisions made after the assertion level, which is the second highest
level in a conflict–driven clause. For example, in the clause $\neg A \vee \neg X$, A was set
at Level 0 and X was set at level 3. Hence, the assertion level is 0 in this case. If
the clause contains only literals from one level, its assertion level is then -1 by
definition. The assertion level is special in the sense that it is the deepest level
at which adding the conflict–driven clause would allow unit resolution to derive
a new implication using that clause. This is the reason why modern SAT solvers
would actually backtrack all the way to the assertion level, add the conflict–driven
clause to the CNF, apply unit resolution, and then continue the search process.
This particular method of performing non–chronological backtracking is referred
to as far-backtracking [SBK05].

DPLL and clause learning

Algorithm 5 DPLL+(CNF Δ): returns UNSATISFIABLE or SATISFIABLE.

```
$D \leftarrow ()$ {empty decision sequence}
$\Gamma \leftarrow \{\}$ {empty set of learned clauses}
while true do
    if unit resolution detects a contradiction in $(\Delta, \Gamma, D)$ then
        if $D = ()$ then {contradiction without any decisions}
            return UNSATISFIABLE
        else {backtrack to assertion level}
            $\alpha \leftarrow$ asserting clause
            $m \leftarrow$ assertion level of clause $\alpha$
            $D \leftarrow$ first $m$ decisions in $D$ {erase decisions $\ell_{m+1}, \ldots$}
            add clause $\alpha$ to $\Gamma$
    else {unit resolution does not detect a contradiction}
        if $\ell$ is a literal where neither $\ell$ nor $\neg\ell$ are implied by unit resolution from $(\Delta, \Gamma, D)$
        then
            $D \leftarrow D; \ell$ {add new decision to sequence $D$}
        else
            return SATISFIABLE
```

UIP: Unique Implication points

A UIP of a decision level in an implication
graph is a variable setting at that decision level which lies on every path
from the decision variable of that level to the contradiction. Intuitively, a UIP of
a level is an assignment at the level that, by itself, is sufficient for implying the
contradiction. In Figure 3.9, the variable setting 3/Y=true and 3/X=true would
be UIPs as they lie on every path from the decision 3/X=true to the contradiction
3/{}.

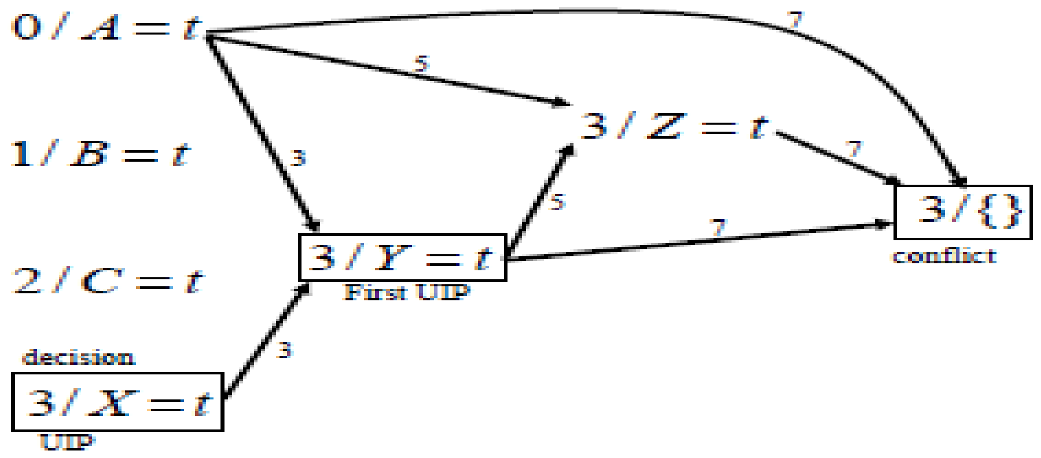
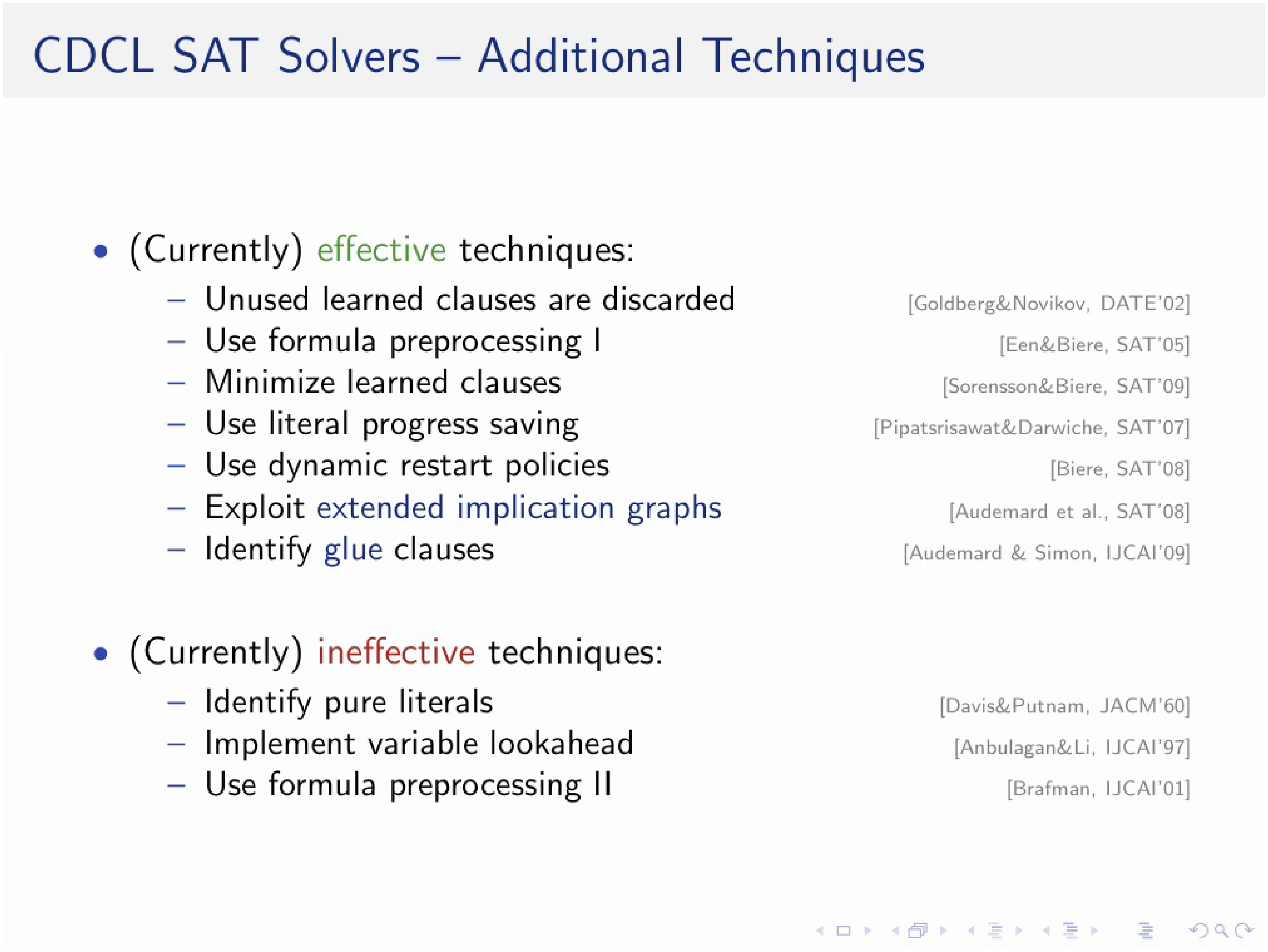


Figure 3.9. An example of a unique implication point (UIP).

CDCL SAT Solvers – Basic Techniques

- Based on DPLL [Davis et al., JACM'60, CACM'62]
  - Must be able to prove unsatisfiability
- New clauses are learned from conflicts [Marques-Silva&Sakallah, ICCAD'96]
  - Backtracking can be non-chronological
- Structure of conflicts is exploited (UIPs) [Marques-Silva&Sakallah, ICCAD'96]
- Backtrack search is periodically restarted [Gomes et al., AAAI'98]
- Lazy data structures are used [Moskewicz et al, DAC'01]
  - Compact with low maintenance overhead
- Branching is guided by conflicts [Moskewicz et al, DAC'01]
  - E.g. VSIDS, etc.



CDCL SAT Solvers – Additional Techniques

- (Currently) effective techniques:
  - Unused learned clauses are discarded [Goldberg&Novikov, DATE'02]
  - Use formula preprocessing I [Een&Biere, SAT'05]
  - Minimize learned clauses [Sorensson&Biere, SAT'09]
  - Use literal progress saving [Pipatsrisawat&Darwiche, SAT'07]
  - Use dynamic restart policies [Biere, SAT'08]
  - Exploit extended implication graphs [Audemard et al., SAT'08]
  - Identify glue clauses [Audemard & Simon, IJCAI'09]
- (Currently) ineffective techniques:
  - Identify pure literals [Davis&Putnam, JACM'60]
  - Implement variable lookahead [Anbulagan&Li, IJCAI'97]
  - Use formula preprocessing II [Brafman, IJCAI'01]

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots$$

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = 0$ and $f = 0$

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = 0$ and $f = 0$
  - Assign $a = 0$ and imply assignments

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = 0$ and $f = 0$
  - Assign $a = 0$ and imply assignments

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = 0$ and $f = 0$
  - Assign $a = 0$ and imply assignments
  - A conflict is reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots$$

  - Assume decisions $c = 0$ and $f = 0$
  - Assign $a = 0$ and imply assignments
  - A conflict is reached: $(\neg d \lor \neg e \lor f)$ is unsatisfied
  - $(a = 0) \land (c = 0) \land (f = 0) \Rightarrow (\varphi = 0)$

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = 0$ and $f = 0$
  - Assign $a = 0$ and imply assignments
  - A conflict is reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $(a = 0) \wedge (c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$
  - $(\varphi = 1) \Rightarrow (a = 1) \vee (c = 1) \vee (f = 1)$

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = 0$ and $f = 0$
  - Assign $a = 0$ and imply assignments
  - A conflict is reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $(a = 0) \wedge (c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$
  - $(\varphi = 1) \Rightarrow (a = 1) \vee (c = 1) \vee (f = 1)$
  - Learn new clause $(a \vee c \vee f)$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi \quad = \quad & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi \quad = \quad & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

  - Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi \;=\; & (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land \\ & (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k)\end{aligned}$$

  - Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
  - Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \lor c \lor f)$ implies $a = 1$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi \;=\; & (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land \\ & (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k)\end{aligned}$$

  - Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
  - Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \lor c \lor f)$ implies $a = 1$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi \;=\; & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

  - Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
  - Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi \ = \ & (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \land \\ & (a \lor c \lor f) \land (\neg a \lor g) \land (\neg g \lor b) \land (\neg h \lor j) \land (\neg i \lor k)\end{aligned}$$

  - Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
  - Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \lor c \lor f)$ implies $a = 1$
  - A conflict is again reached: $(\neg d \lor \neg e \lor f)$ is unsatisfied

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\varphi \;=\; (a \vee b) \wedge (\neg b \vee \boxed{c} \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee \boxed{f}) \wedge \\ (a \vee \boxed{c} \vee \boxed{f}) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

  - Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
  - Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$
  - A conflict is again reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $(c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

  - Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
  - Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$
  - A conflict is again reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $(c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$
  - $(\varphi = 1) \Rightarrow (c = 1) \vee (f = 1)$

Non-Chronological Backtracking

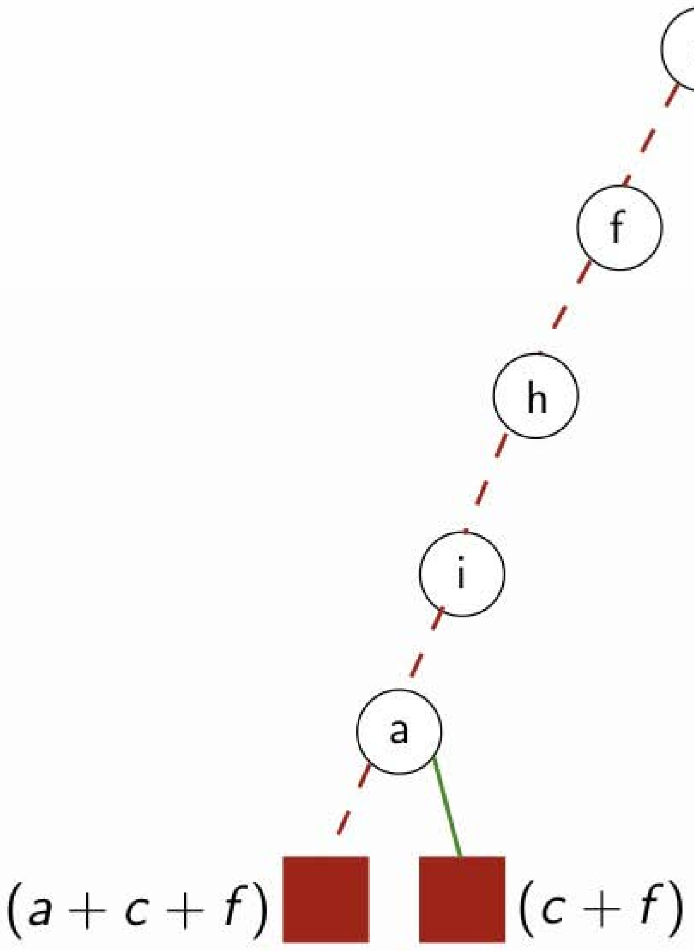
- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi = {} & (a \vee b) \wedge (\neg b \vee \boxed{c} \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee \boxed{f}) \wedge \\ & (a \vee \boxed{c} \vee \boxed{f}) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

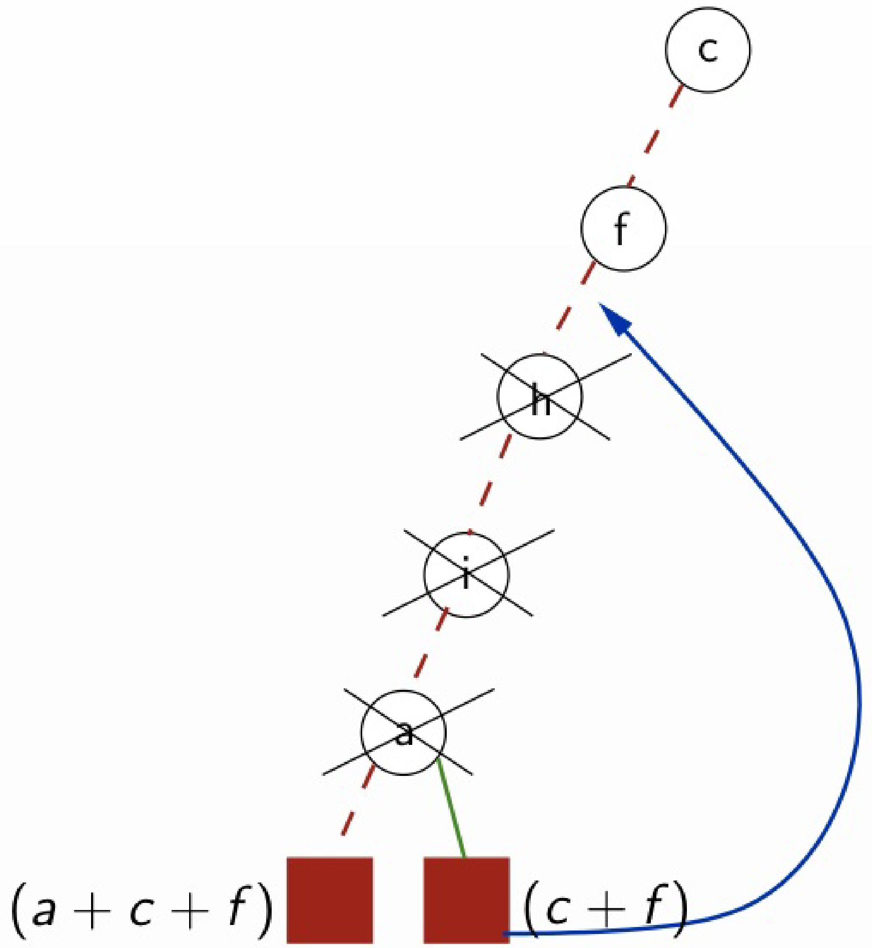
  - Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
  - Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$
  - A conflict is again reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $(c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$
  - $(\varphi = 1) \Rightarrow (c = 1) \vee (f = 1)$

  - Learn new clause $(c \vee f)$

Non-Chronological Backtracking



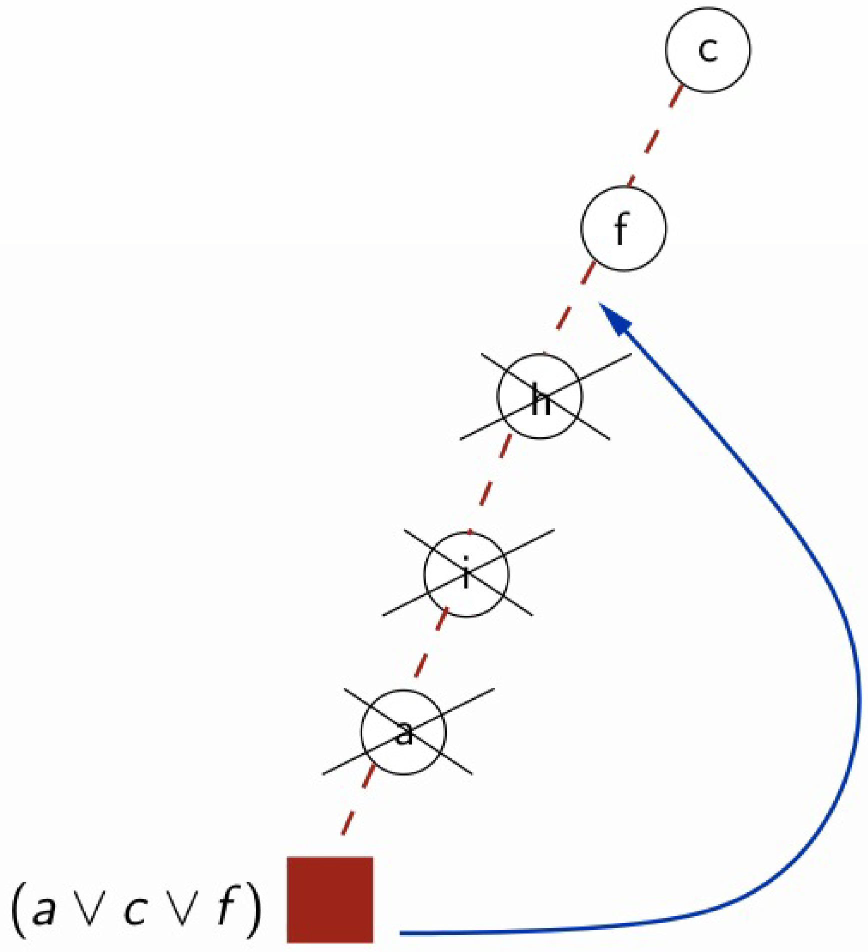
Non-Chronological Backtracking



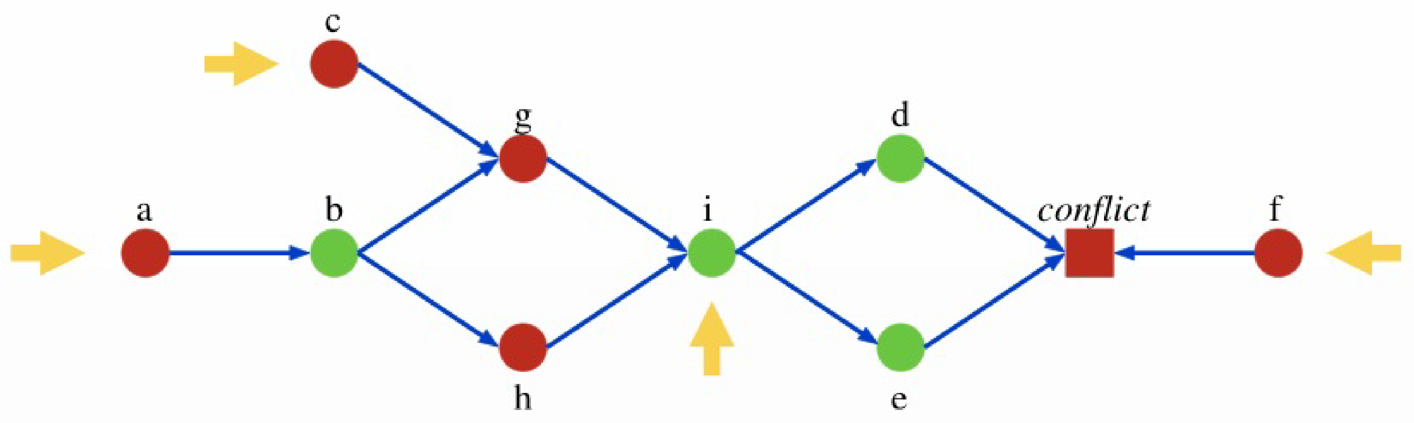
- Learnt clause: $(c \lor f)$
- Need to backtrack, given new clause
- Backtrack to most recent decision: $f = 0$

- Clause learning and non-chronological backtracking are hallmarks of modern SAT solvers

Most Recent Backtracking Scheme



Unique Implication Points (UIPs)



- Exploit structure from the implication graph
  - To have a more aggressive backtracking policy
- Identify additional clauses to learn [Marques-Silva&Sakallah'96]
  - Create clauses $(a \vee c \vee f)$ and $(\neg i \vee f)$
  - Imply not only $a = 1$ but also $i = 0$
- 1st UIP scheme is the most effective [Zhang et al.'01]
  - Create only one clause $(\neg i \vee f)$
  - Avoid creating similar clauses involving the same literals

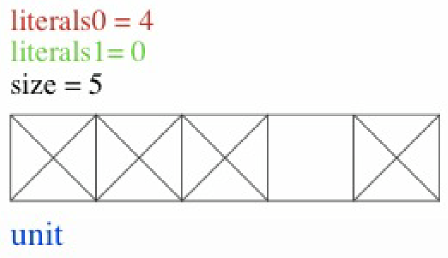
Clause deletion policies

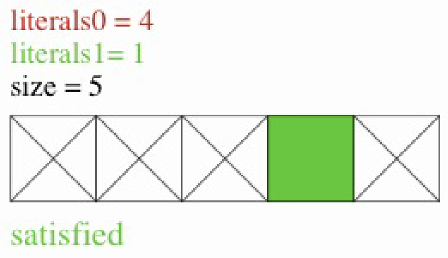
- Keep only the small clauses [Marques-Silva&Sakallah'96]
  - For each conflict record one clause
  - Keep clauses of size $\leq K$
  - Large clauses get deleted when become unresolved
- Keep only the relevant clauses [Bayardo&Schrag'97]
  - Delete unresolved clauses with $\leq M$ free literals
- Keep only the clauses that are used [Goldberg&Novikov'02]
  - Keep track of clauses activity

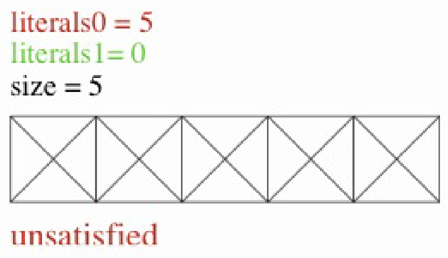
Data Structures

- Key point: only unit and unsatisfied clauses *must* be detected during search
  - Formula is unsatisfied when at least one clause is unsatisfied
  - Formula is satisfied when all the variables are assigned and there are no unsatisfied clauses
- In practice: unit and unsatisfied clauses may be identified using only two references
- Standard data structures (adjacency lists):
  - Each variable x keeps a reference to all clauses containing a literal in x
- Lazy data structures (watched literals):
  - For each clause, only two variables keep a reference to the clause, i.e. only 2 literals are watched

Standard Data Structures (adjacency lists)

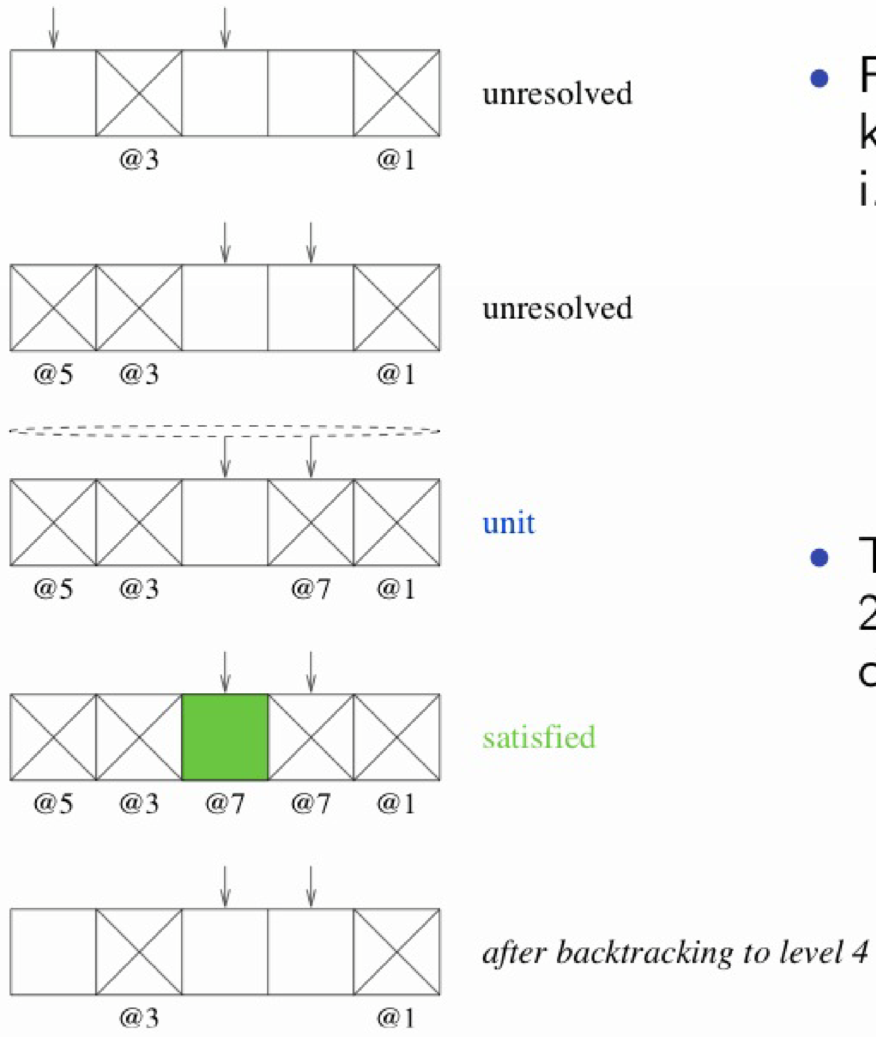






- Each variable x keeps a reference to all clauses containing a literal in x
  - If variable x is assigned, then all clauses containing a literal in x are evaluated
  - If search backtracks, then all clauses of all newly unassigned variables are updated
- Total number of references is L, where L is the number of literals

Lazy Data Structures (watched literals)



- For each clause, only two variables keep a reference to the clause, i.e. only 2 literals are watched
  - If variable x is assigned, only the clauses where literals in x are watched need to be evaluated
  - If search backtracks, then nothing needs to be done
- Total number of references is $2 \times C$, where C is the number of clauses
  - In general $L \gg 2 \times C$, in particular if clauses are learnt

BCP Algorithm (1/8)

- What "causes" an implication? When can it occur?
  - All literals in a clause but one are assigned to False
    - (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 False
  - 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 unit 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 unit (and conflict) clauses (and the associated implications) while maintaining the “Big Invariants”

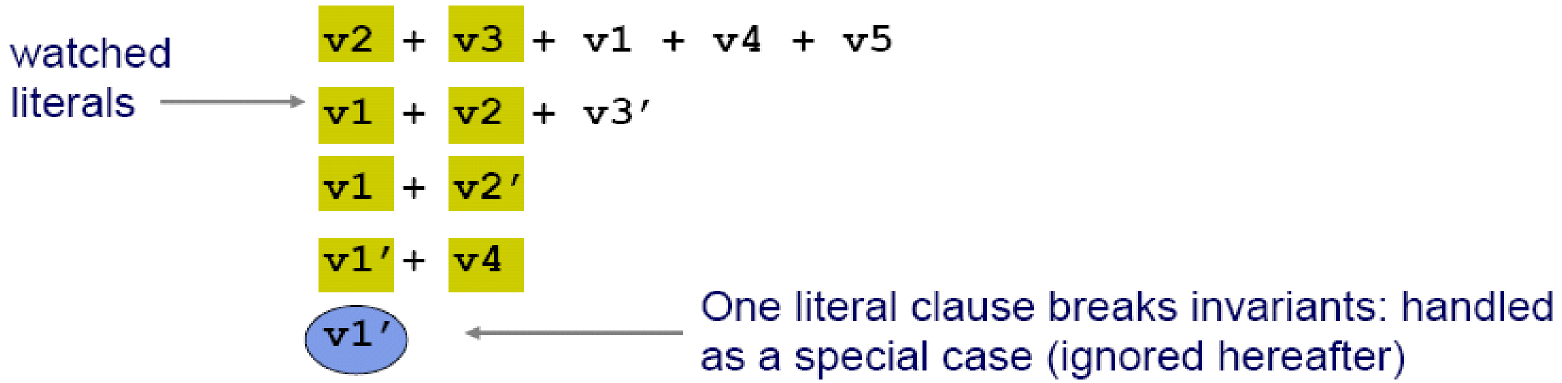
BCP Algorithm (2/8)

- Let’s illustrate this with an example:

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3′
v1 + v2′
v1′+ v4
v1′
```

BCP Algorithm (2.1/8)

- Let's illustrate this with an example:


watched literals
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'
v1 + v2'
v1' + v4
v1'
One literal clause breaks invariants: handled as a special case (ignored hereafter)

  - Initially, we identify any two literals in each clause as the watched ones
  - 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:
```

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'
v1 + v2'
v1'+ v4
```

BCP Algorithm (3.1/8)

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

```
                            v2 + v3 + v1 + v4 + v5
 State:(v1=F)         ⇨     v1 + v2 + v3'
 Pending:             ⇨     v1 + v2'
                            v1'+ v4
```

- 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)

Pending:
```

```
   v2 + v3 + v1 + v4 + v5
   v1 + v2 + v3'
   v1 + v2'
⇨  v1'+ v4
```

- 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 unit.

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:
```

```
⇨ v2 + v3 + v1 + v4 + v5
  v1 + v2 + v3'
  v1 + v2'
  v1'+ v4
```

- 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 unit.
- We *certainly* need not process any clauses where neither watched literal changes state (in this example, where v1 is not watched).

BCP Algorithm (4/8)

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

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'
v1 + v2'
v1'+ v4
```

```
State:(v1=F)

Pending:
```

BCP Algorithm (4.1/8)

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

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'
v1 + v2'
v1'+ v4
```

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

$\Rightarrow$

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'
v1 + v2'
v1'+ v4
```

```
State: (v1=F)
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:

```
v2 + v3 + v1 + v4 + v5          v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'                   v1 + v2 + v3'
v1 + v2'                  ⇨     v1 + v2'
v1'+ v4                         v1'+ v4

  State: (v1=F)                 State: (v1=F)
  Pending:                      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 unit. We record the new implication of v2', and add it to the queue of assignments to process. Since the clause cannot again become unit, our invariants are maintained.

BCP Algorithm (5/8)

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

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3’
v1 + v2’
v1’+ v4
```

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

Pending:
```

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3’
v1 + v2’
v1’+ v4
```

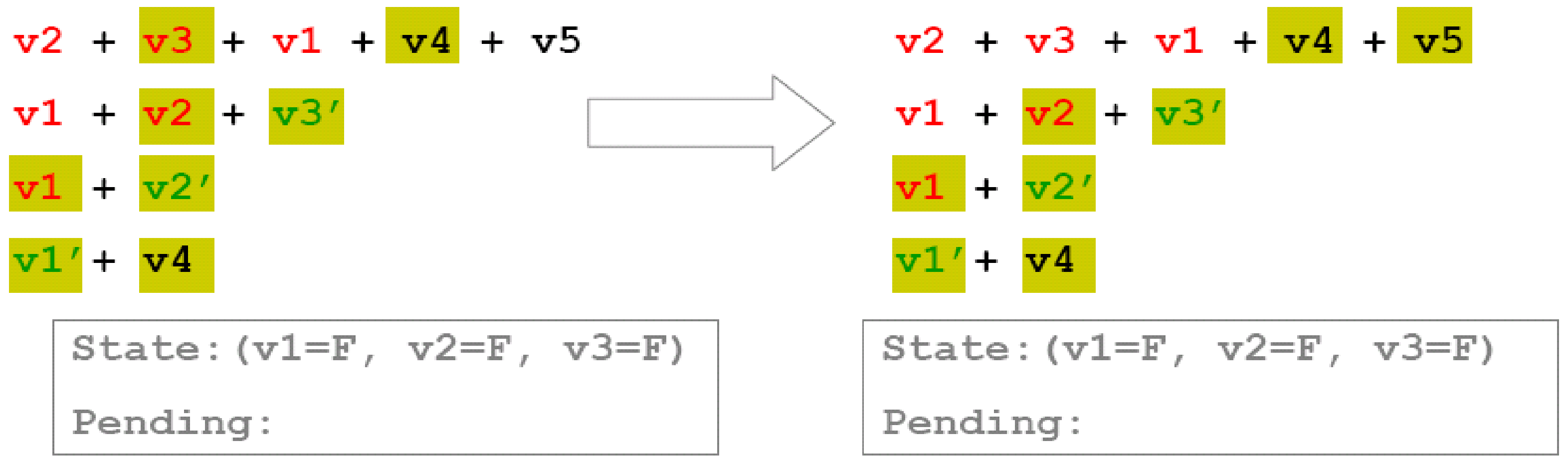
```
State:(v1=F, v2=F)

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 unit. We record the new implication of v3’, and add it to the queue of assignments to process. Since the clause cannot again become unit, our invariants are maintained.

BCP Algorithm (6/8)

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



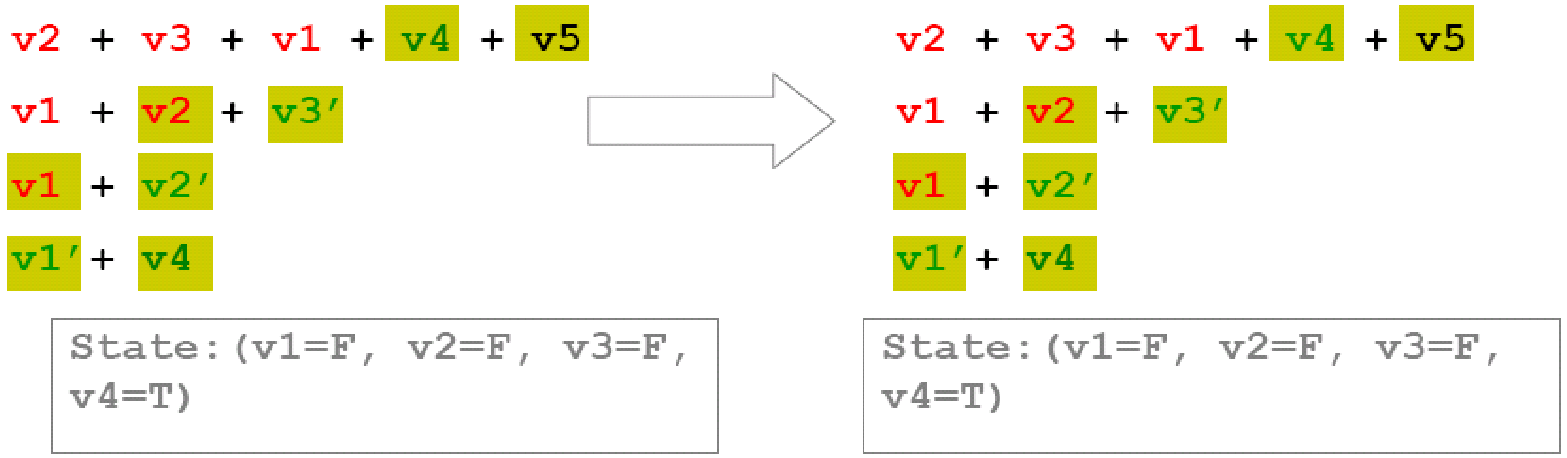
```
v2 + v3 + v1 + v4 + v5          v2 + v3 + v1 + v4 + v5
v1 + v2 + v3’                   v1 + v2 + v3’
v1 + v2’                        v1 + v2’
v1’+ v4                         v1’+ v4

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.



```
v2 + v3 + v1 + v4 + v5          v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'             ⇒     v1 + v2 + v3'
v1 + v2'                        v1 + v2'
v1'+ v4                         v1'+ v4

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

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3'

v1 + v2'

v1' + v4

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

$\Rightarrow$

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3'

v1 + v2'

v1' + v4

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

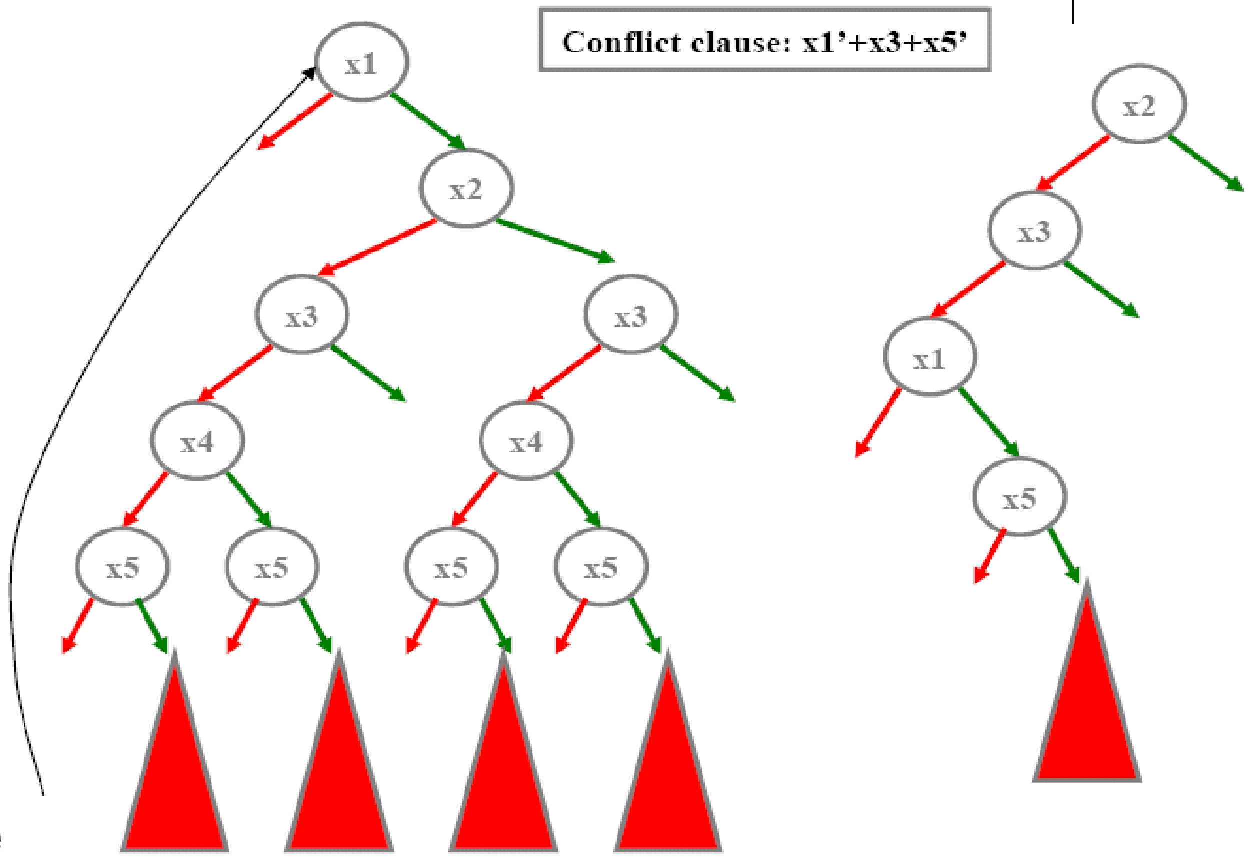
- The first clause is already satisfied by v4 so we ignore it.
- Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. No variables are unassigned, so the instance is SAT, and we are done.

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

Restart

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



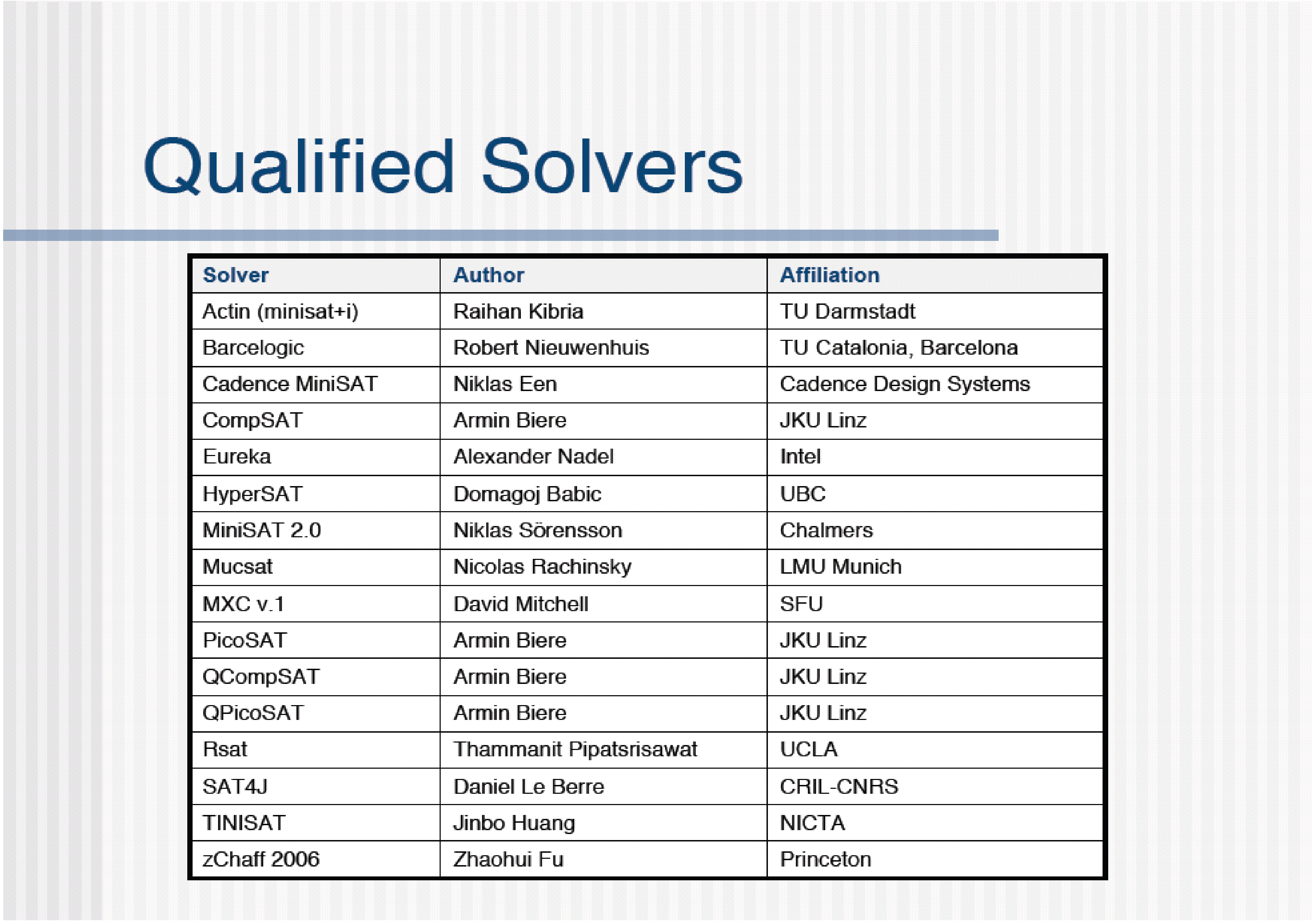
Evolution of SAT Solvers

Instance	Posit'94	Grasp'96	Chaff'03	Minisat'03	Picosat'08
ssa2670-136	13.57	0.22	0.02	0.00	0.01
bf1355-638	310.93	0.02	0.02	0.00	0.03
design_3	> 1800	3.93	0.18	0.17	0.93
design_1	> 1800	34.55	0.35	0.11	0.68
4pipe_4_ooo	> 1800	> 1800	17.47	110.97	44.95
fifo8_300	> 1800	> 1800	348.50	53.66	39.31
w08_15	> 1800	> 1800	> 1800	99.10	71.89
9pipe_9_ooo	> 1800	> 1800	> 1800	> 1800	> 1800
c6288	> 1800	> 1800	> 1800	> 1800	> 1800

- Modern SAT algorithms can solve instances with hundreds of thousands of variables and tens of millions of clauses

Benchmarks

- Random
- Crafted
- Industrial



Qualified Solvers

Solver	Author	Affiliation
Actin (minisat+i)	Raihan Kibria	TU Darmstadt
Barcelogic	Robert Nieuwenhuis	TU Catalonia, Barcelona
Cadence MiniSAT	Niklas Een	Cadence Design Systems
CompSAT	Armin Biere	JKU Linz
Eureka	Alexander Nadel	Intel
HyperSAT	Domagoj Babic	UBC
MiniSAT 2.0	Niklas Sörensson	Chalmers
Mucsat	Nicolas Rachinsky	LMU Munich
MXC v.1	David Mitchell	SFU
PicoSAT	Armin Biere	JKU Linz
QCompSAT	Armin Biere	JKU Linz
QPicoSAT	Armin Biere	JKU Linz
Rsat	Thammanit Pipatsrisawat	UCLA
SAT4J	Daniel Le Berre	CRIL-CNRS
TINISAT	Jinbo Huang	NICTA
zChaff 2006	Zhaohui Fu	Princeton

SAT race 2006