SAT Solvers: Clause Learning

Barnaby Martin
barnaby.d.martin@durham.ac.uk

A search algorithm

A search algorithm

- Algorithm A(F)
- outputs *sat* or *unsat*
- Pseudocode:

if $\mathrm{var}(F) = \emptyset$ then

    if $F = \emptyset$ then exit(sat).

    else exit(unsat). /* $F = \{\Box\}$ where $\Box$ is empty clause */

else choose $x \in \mathrm{var}(F)$,

/* x is the branching variable */

if A($F[x = 0]$) $=$ sat then exit(sat).

else if A($F[x = 1]$) $=$ sat then exit(sat).

else exit(unsat).

Pure Literals

- A literal x of a clause-set F is a pure literal of F if some clauses of F contain x but no clause of F contains $\overline{x}$.
- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.

Pure Literals

- A literal x of a clause-set F is a pure literal of F if some clauses of F contain x but no clause of F contains $\overline{x}$.
- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.
- Let F' be the clause-set obtained from F by removing all clauses that contain pure literals.

  Then F and F' are equisatisfiable (why?).

  We say that F' is obtained from F by pure literal elimination.

Applying Pure Literal Elimination

- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.
- In the above example we obtain $F' = \{\{\overline{y}, \overline{z}\}\}$. Now $\overline{y}$ and $\overline{z}$ are pure literals of F', and we can apply pure literal elimination again, obtaining the empty clause-set.

Applying Pure Literal Elimination

- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.
- In the above example we obtain $F' = \{\{\overline{y}, \overline{z}\}\}$. Now $\overline{y}$ and $\overline{z}$ are pure literals of F', and we can apply pure literal elimination again, obtaining the empty clause-set.
- For a clause-set F we denote by $PL(F)$ the smallest clause-set that can be obtained from F by (possibly repeated) applications of pure literal elimination.
- In the above example we have $PL(F) = \emptyset$.

Applying Pure Literal Elimination

- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.
- In the above example we obtain $F' = \{\{\overline{y}, \overline{z}\}\}$. Now $\overline{y}$ and $\overline{z}$ are pure literals of F', and we can apply pure literal elimination again, obtaining the empty clause-set.
- For a clause-set F we denote by $PL(F)$ the smallest clause-set that can be obtained from F by (possibly repeated) applications of pure literal elimination.
- In the above example we have $PL(F) = \emptyset$.
- F and $PL(F)$ are always equisatisfiable.

Unit Propagation

- When a clause-set F contains a unit clause $\{\ell\}$, we can obtain by unit propagation the clause-set $F[\ell = 1]$ from F.
- In that case F and $F[\ell = 1]$ are equisatisfiable (why?)
- We write $UP(F)$ for the clause-set obtained from F by applying unit propagation as often as possible.

Unit Propagation

- When a clause-set F contains a unit clause $\{\ell\}$, we can obtain by unit propagation the clause-set $F[\ell = 1]$ from F.
- In that case F and $F[\ell = 1]$ are equisatisfiable (why?)
- We write $UP(F)$ for the clause-set obtained from F by applying unit propagation as often as possible.
- Example: Let $F = \{\{x, y\}, \{\overline{y}\}, \{z, \overline{x}, v\}\}$.
  - From F we obtain by unit propagation the clause set $F' = \{\{x\}, \{z, \overline{x}, v\}\}$.
  - With a second step of unit propagation we obtain from F' the clause-set $F'' = \{\{z, v\}\}$.

  Consequently $UP(F) = \{\{z, v\}\}$.
- F and $UP(F)$ are always equisatisfiable.

First UP and then PL, or vice versa?

Fact

For any clause-set F, we have $UP(PL(UP(F))) = PL(UP(F))$.

Proof: exercise (Question 3a.i in 2009 Advanced AI exam)

Fact

There is a clause-set F such that
$PL(UP(PL(F))) \neq UP(PL(F))$.

Proof: exercise (Question 3a.ii in 2009 Advanced AI exam)

So, first UP and then PL, or the other way around?

The DPLL algorithm

- Algorithm DPLL(F)
- outputs *sat* or *unsat*

```
F := UP(F).
F := PL(F).
if var(F) = ∅ then
    if F = ∅ then exit(sat).
    else exit(unsat).  /* F = {□} */
else choose x ∈ var(F),
/* x is the branching variable */
if A(F[x = 0]) = sat then exit(sat).
else if A(F[x = 1]) = sat then exit(sat).
else exit(unsat).
```

Some Well-Known DPLL-based SAT Solvers

- Important SAT solvers:
  - Grasp (Marques-Silva & Sakallah 1996)
  - Relsat (Bayardo Jr. & Schrag 1997)
  - chaff (Moskewicz et al 2001), zChaff (Zhang 2001)
  - Minisat (Een & Sörensson 2003)
  - RSat (Pipatsrisawat & Darwiche 2007)
  - Glucose (Audemard & Simon 2009)
  - CryptoMiniSat (Soos 2010)
  - Lingeling, PicoSAT (Biere 2010)
- Many of them are available for free and continually improved.

Some Well-Known DPLL-based SAT Solvers

- Important SAT solvers:
  - Grasp (Marques-Silva & Sakallah 1996)
  - Relsat (Bayardo Jr. & Schrag 1997)
  - chaff (Moskewicz et al 2001), zChaff (Zhang 2001)
  - Minisat (Een & Sörensson 2003)
  - RSat (Pipatsrisawat & Darwiche 2007)
  - Glucose (Audemard & Simon 2009)
  - CryptoMiniSat (Soos 2010)
  - Lingeling, PicoSAT (Biere 2010)
- Many of them are available for free and continually improved.
- Key improvements to DPLL that boosted the performance:
  - combination of clever branching heuristics with
  - clause learning,
  - non-chronological backtracking,
  - restart strategies,
  - implementation of propagation.
- Check *www.satlive.org* for more info on SAT solvers!

Conflict-Driven Clause Learning (CDCL): Basics

- CDCL is based on an *iterative version of DPLL*
- Clauses of conflict are cached and learnt

Conflict-Driven Clause Learning (CDCL): Basics

- CDCL is based on an *iterative version of DPLL*
- Clauses of conflict are cached and learnt
- Learnt clauses allow us to prune the search space in different parts of the search encountered later.

Iterative (CDCL) DPLL Algorithm

- initialise:
  - F is the given clause-set
  - τ is the empty assignment.

while(true) do

if $UP(F[\tau]) = \emptyset$ then exit(sat).

elseif $\square \in UP(F[\tau])$ and $\tau = \emptyset$ then exit(unsat)

elseif $\square \in UP(F[\tau])$ then

$C = \text{CONFLICT-CLAUSE}(F, \tau)$, set $F := F \cup \{C\}$

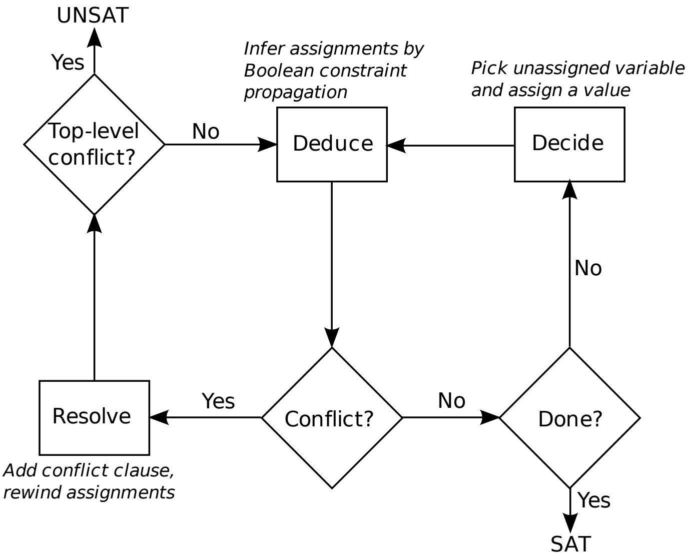
$\tau := \text{BACKTRACK}(\tau)$

else CHOOSE VARIABLE $x \in \text{var}(UP(F[\tau]))$, CHOOSE

VALUE $b \in \{0, 1\}$

Set $\tau := \tau \cup \{(x, b)\}$.

CDCL DPLL diagram



Remarks

- The subroutines CONFLICT-CLAUSE, BACKTRACK, and CHOOSE VARIABLE and VALUE are based on the heuristics and strategy of the particular solver.

Remarks

- The subroutines CONFLICT-CLAUSE, BACKTRACK, and CHOOSE VARIABLE and VALUE are based on the heuristics and strategy of the particular solver.
- The current assignment τ is associated with the set D of literals that are true under τ (e.g., if $\tau = \{(x, 1), (y, 0)\}$ then $D = \{x, \overline{y}\}$). The literals in D are called decision literals.

Remarks

- The subroutines CONFLICT-CLAUSE, BACKTRACK, and CHOOSE VARIABLE and VALUE are based on the heuristics and strategy of the particular solver.
- The current assignment τ is associated with the set D of literals that are true under τ (e.g., if $\tau = \{(x, 1), (y, 0)\}$ then $D = \{x, \overline{y}\}$). The literals in D are called decision literals.
- The algorithm also remembers the order in which decision literals are added to D
- In the step BACKTRACK, (one or more) most recent variable assignment is undone. This causes a backtracking.

Remarks

- The subroutines CONFLICT-CLAUSE, BACKTRACK, and CHOOSE VARIABLE and VALUE are based on the heuristics and strategy of the particular solver.
- The current assignment τ is associated with the set D of literals that are true under τ (e.g., if $\tau = \{(x, 1), (y, 0)\}$ then $D = \{x, \overline{y}\}$). The literals in D are called decision literals.
- The algorithm also remembers the order in which decision literals are added to D
- In the step BACKTRACK, (one or more) most recent variable assignment is undone. This causes a backtracking.
- Instead of *UP* the solver can use more sophisticated propagation techniques known as boolean constraint propagation (BCP).

Simple Example

- Init: $F = \{\{x, y\}, \{\overline{x}, y\}, \{x, \overline{y}\}, \{\overline{x}, \overline{y}\}\}$, $\tau = \emptyset$
- CHOOSE $x, 1$, set $\tau = \{(x, 1)\}$
- $\square \in UP(F[\tau])$
  $\{\overline{x}\} =$ CONFLICT-CLAUSE(F, τ) (explain later why)
  $F = \{\{\overline{x}\}, \{x, y\}, \{\overline{x}, y\}, \{x, \overline{y}\}, \{\overline{x}, \overline{y}\}\}$,
  $\tau = \emptyset =$ BACKTRACK$(\{(x, 1)\})$
- $\square \in UP(F[\tau])$
  Since $\tau = \emptyset$ exit(unsat).

The Implication Graph

The implication graph G is a directed graph defined w.r.t a particular state (F, τ) of the iterative DPLL algorithm. The nodes of G are literals. G is constructed as follows:

The Implication Graph

The implication graph G is a directed graph defined w.r.t a particular state (F, τ) of the iterative DPLL algorithm.
The nodes of G are literals. G is constructed as follows:

(1) Add all literals true under τ. These are called decision literals and form the sources of G (no incoming edges).

(2) If ℓ is not in G yet, but there is a clause $\{\ell_1, \ldots, \ell_k, \ell\} \in F$ such that $\overline{\ell_1}, \ldots, \overline{\ell_k}$ are already in G, then add node ℓ (called an implied literal) and add edges from $\overline{\ell_i}$ to ℓ for $1 \leq i \leq k$ (that aren't already there).
Repeat this step until stable.
   - If the graph contains two complementary literals x and $\overline{x}$, then these literals are called conflict literals.
   - Can stop here if a complementary pair of literals is found.

(3) Finally, add a special node "$\lightning$" and add edges from conflict literals (if there are any) to this node.

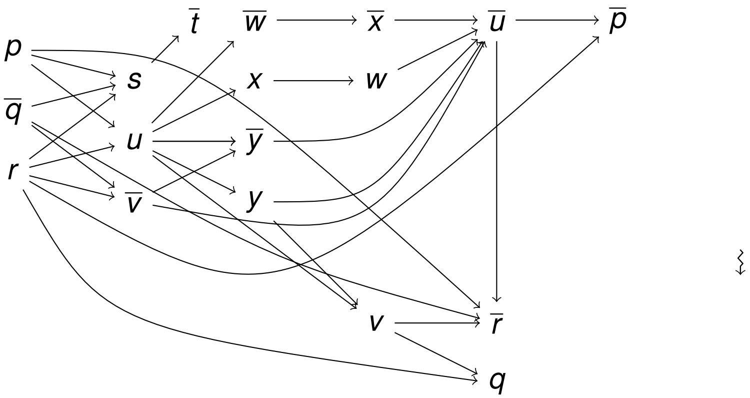
Example

Assume that

- current clause-set F is $\{\{\overline{p}, q, s\}, \{q, \overline{r}, s\}, \{\overline{p}, \overline{r}, u\}, \{q, \overline{r}, \overline{v}\}, \{\overline{s}, \overline{t}\}, \{\overline{u}, \overline{w}\}, \{\overline{u}, x\}, \{\overline{u}, y\}, \{\overline{u}, v, \overline{y}\}, \{w, \overline{x}\}\}$;
- current assignment $\tau = \{(p, 1), (q, 0), (r, 1)\}$.

Example of an implication graph

Note: The edges from conflict literals to $\lightning$ are not shown.



$F = \{\{\overline{p}, q, s\}, \{q, \overline{r}, s\}, \{\overline{p}, \overline{r}, u\}, \{q, \overline{r}, \overline{v}\}, \{\overline{s}, \overline{t}\}, \{\overline{u}, \overline{w}\},$
$\{\overline{u}, x\}, \{\overline{u}, y\}, \{\overline{u}, v, \overline{y}\}, \{w, \overline{x}\}\};$

Construction in detail

In the following "a" means "add the vertex a";
"$a \to b$" means "add an edge running from vertex a to vertex b".

p, $\overline{q}$, r (decision literals)
s, $p \to s$, $\overline{q} \to s$
$r \to s$
u, $p \to u$, $r \to u$
$\overline{v}$, $\overline{q} \to \overline{v}$, $r \to \overline{v}$
$\overline{t}$, $s \to \overline{t}$
$\overline{w}$, $u \to \overline{w}$
x, $u \to x$
y, $u \to y$
$\overline{y}$, $u \to \overline{y}$, $\overline{v} \to \overline{y}$.

Now we have the conflict literals $y, \overline{y}$; at this stage a SAT solver could already start to form a conflict graph; but we continue...

$\overline{u}, \overline{y} \rightarrow \overline{u}$

$v, u \rightarrow v, y \rightarrow v$

$\overline{v} \rightarrow \overline{u}, y \rightarrow \overline{u}$

$\overline{x}, \overline{w} \rightarrow \overline{x}$

$w, x \rightarrow w$

$\overline{r}, p \rightarrow \overline{r}, \overline{u} \rightarrow \overline{r}$

$\overline{p}, r \rightarrow \overline{p}\ \overline{u} \rightarrow \overline{p}$

$\overline{q} \rightarrow \overline{r}, v \rightarrow \overline{r}$

$q, r \rightarrow q, v \rightarrow q$

$w \rightarrow \overline{u}$

$\overline{x} \rightarrow \overline{u}$

Except for s all other added literals are conflict literals, therefore we connect all other literals to $\lightning$. (However, these edges are omitted in the drawing for sake of readability).

Conflict Graphs

- Assume that the implication graph G contains conflict literals. Then we can extract from G a conflict graph H with the following properties:
  - H contains exactly one pair of conflict variables.

Conflict Graphs

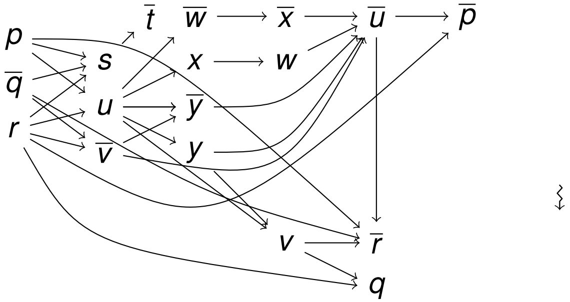
- Assume that the implication graph G contains conflict literals. Then we can extract from G a conflict graph H with the following properties:
  - H contains exactly one pair of conflict variables.
  - For every implied literal ℓ of H there is exactly one clause $\{\ell_1, \ldots, \ell_k, \ell\} \in F$ such that the literals $\overline{\ell_1}, \ldots, \overline{\ell_k}$ belong to H and there are edges from ℓ_i to ℓ.

Conflict Graphs

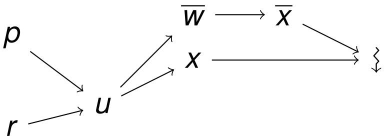
- Assume that the implication graph G contains conflict literals. Then we can extract from G a conflict graph H with the following properties:
  - H contains exactly one pair of conflict variables.
  - For every implied literal ℓ of H there is exactly one clause $\{\ell_1, \ldots, \ell_k, \ell\} \in F$ such that the literals $\overline{\ell_1}, \ldots, \overline{\ell_k}$ belong to H and there are edges from ℓ_i to ℓ.
  - From every node in H one can reach $\lightning$ via a directed path.

  In a certain sense H represents succinctly one cause of conflict (from the decision variables via unit propagation).

Example of a conflict graph



The implication graph from above gives rise to the following conflict graph.



Note that there can be different possibilities for obtaining a

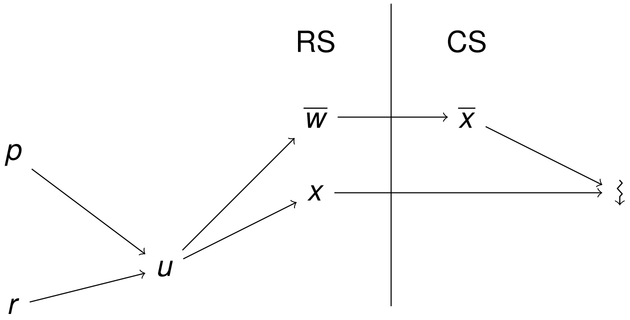
Remark

- Note that the implication graph is not necessarily acyclic, there can be directed cyles.
- However, a conflict graph is always acyclic.

Conflict Clauses

- Select a set of edges that divides the conflict graph into two parts, the reason side (RS) and the conflict side (CS) such that
  - RS contains all decision literals
  - CS contains $\lightning$ and at least one conflict literal.
- Form a clause C consisting of the complements of all literals in RS that have a neighbour in CS.
- This clause C is a conflict clause.
- The conflict clause is returned by the subroutine CONFLICT-CLAUSE and added to the given set of clauses.
- In general there are many different choices for the conflict clause, it is an important part of the solver's strategy which conflict clause is selected.

Example of a Conflict Clause

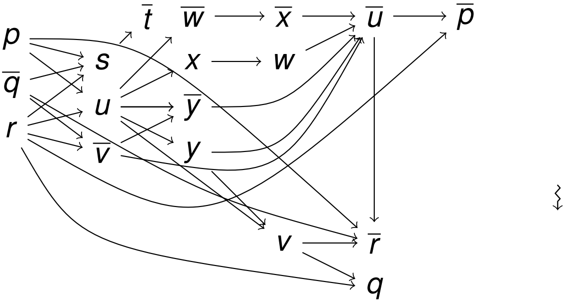


Conflict clause: $C = \{w, \overline{x}\}$.
Other possibilities: $C = \{\overline{p}, \overline{r}\}$, $C = \{\overline{u}\}$.

Backtracking

- After a clause is learnt we can backtrack via the subroutine BACKTRACK
- for example $\tau = \{(p, 1), (q, 0)\} = \text{BACKTRACK}(\{(p, 1), (q, 0), (r, 1)\})$
- or even $\tau = \emptyset = \text{BACKTRACK}(\{(p, 1), (q, 0), (r, 1)\})$.

Decision Levels

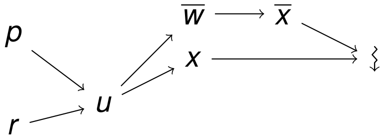


- A decision level consists of a decision literal ℓ and all implied literals that were added after ℓ has been added.
- Assume that in the example the latest decision literal is r.

  The decision level of r contains the implied literals $r, u, \overline{v}, \overline{w}, x, y, \overline{y}, \overline{x}$. The implied literals s and $\overline{t}$ belong to the previous decision level.

Unique Implication Points

- A unique implication point (UIP) is a literal ℓ of the current decision level such that every path from the current decision literal to any of the two conflict literals must run through ℓ.
- For example, the current decision literal is a UIP.
- Intuitively, a UIP is a *single reason* for a conflict.
- It is a common strategy to choose a cut that puts a UIP into the RS and its successors into CS.
- In the previous example, r and u are both UIPs. The corresponding conflict clauses are $\{\overline{p}, \overline{r}\}$ and $\{\overline{u}\}$.

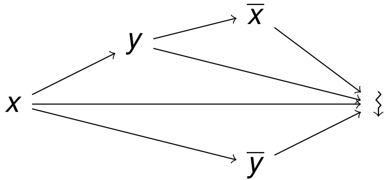


- For example the solver zChaff backtracks to decision level 0 if it learns a unit clause.

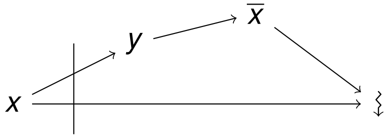
Simple Example (Continued)

Now we can see how clause $\{\overline{x}\}$ can be learnt in our example.

- $F = \{\{x, y\}, \{\overline{x}, y\}, \{x, \overline{y}\}, \{\overline{x}, \overline{y}\}\}$,
- $\tau = \{(x, 1)\}$
- Implication graph:



- Conflict graph:



- $\{\overline{x}\} = \text{CONFLICT-CLAUSE}(F, \tau)$

Example 2

Assume that

- current clause-set F is $\{\{\overline{x_1}, x_2\}, \{\overline{x_2}, x_3, x_4\}, \{\overline{x_2}, \overline{x_5}\}, \{\overline{x_4}, x_5, x_6\}, \{\overline{x_7}, x_8\}, \{\overline{x_8}, \overline{x_9}\}, \{x_9, \overline{x_{10}}\}, \{x_3, \overline{x_8}, x_{10}\}\}$;
- current assignment $\tau = \{(x_1, 1), (x_3, 0), (x_7, 1)\}$.

Clause Deletion

- The number of learnt clauses can sometimes be huge
- Efficient solvers have a “forgetting strategy”:
  - delete learnt causes that are too large, and/or
  - those that were not used in derivations recently

Backtracking

- After a clause is learnt we can backtrack via the subroutine BACKTRACK
- for example
  $\tau = \{(p, 1), (q, 0)\} = \text{BACKTRACK}(\{(p, 1), (q, 0), (r, 1)\})$
- or even $\tau = \emptyset = \text{BACKTRACK}(\{(p, 1), (q, 0), (r, 1)\})$.

Non-Chronological Backtracking

- Normal backtracking undoes the most recent assignment
- NCB (aka backjumping) can backtrack further back

Non-Chronological Backtracking

- Normal backtracking undoes the most recent assignment
- NCB (aka backjumping) can backtrack further back
- Assuming that we learnt a clause using a UIP
  - if the learnt clause is unit, NCB backtracks to $\tau = \emptyset$
  - otherwise, NCB backtracks to the <u>second</u> latest decision level of literals in learnt clause
  - The learnt clause will allow additional unit propagation
  - In the earlier example, if we learn $\{\overline{p}, \overline{r}\}$, then backtrack from $\tau = (\{(p, 1), (q, 0), (r, 1)\})$ to $\tau = \{(p, 1)\}$ and update the implication graph (e.g. add $\overline{r}$ to the decision level of p).
- NCB maintains the completeness, while improving efficiency in practice.

Exercise

Let F be the clause-set consisting of the following clauses:

- $C_1 = \{x_1, x_2\}$
- $C_2 = \{x_1, x_3, x_7\}$
- $C_3 = \{\overline{x_2}, \overline{x_3}, x_4\}$
- $C_4 = \{\overline{x_4}, x_5, x_8\}$
- $C_5 = \{\overline{x_4}, x_6, x_9\}$
- $C_6 = \{\overline{x_5}, \overline{x_6}\}$

Let $\tau = \{(x_7, 0), (x_8, 0), (x_9, 0), (x_1, 0)\}$, in this order.

- Draw the implication graph and a conflict graph
- Identify UIPs and clauses that can be learnt the conflict
- Identify the effect of non-chronological backtracking
- What changes if we swap $(x_7, 0)$ and $(x_9, 0)$ in τ?

Variable Choice Heuristics (aka Decision Heuristics)

- Strategies for choosing the next variable to assign
- Difficult to develop
  - Bad early decisions can be very costly
  - Hard to detect badness, especially with hard satisfiable clause-sets

Variable Choice Heuristics (aka Decision Heuristics)

- Strategies for choosing the next variable to assign
- Difficult to develop
  - Bad early decisions can be very costly
  - Hard to detect badness, especially with hard satisfiable clause-sets
- Clause Learning, Non-Chronological Backtracking, and Restarts compensate for the difficulty

Variable Choice Heuristics (aka Decision Heuristics)

- Strategies for choosing the next variable to assign
- Difficult to develop
  - Bad early decisions can be very costly
  - Hard to detect badness, especially with hard satisfiable clause-sets
- Clause Learning, Non-Chronological Backtracking, and Restarts compensate for the difficulty
- Common tendency: favour variables that appear in many short clauses
  - this facilitates unit propagation

Examples of Decision Heuristics

- MOMS: choose literal with Maximum number of occurrences in Minimum Size clauses

Examples of Decision Heuristics

- MOMS: choose literal with Maximum number of occurrences in Minimum Size clauses
- Jeroslow-Wang: Let $C(x)$ be the set of open clauses containing either polarity of a given variable x. Define the weight of x as

$$w(x) = \sum_{c \in C(x)} 2^{-|c|}$$

  Choose a variable with the maximum weight as the branching variable.

Decision Heuristic VSIDS

- Variable State Independent Decaying Sum
- For each literal, keep counter of how many learnt clauses it appears in
  - Periodically divide by constant to bias to recently learnt clauses
- At each decision, choose literal with highest counter

Decision Heuristic VSIDS

- Variable State Independent Decaying Sum
- For each literal, keep counter of how many learnt clauses it appears in
  - Periodically divide by constant to bias to recently learnt clauses
- At each decision, choose literal with highest counter
- Partial assignments more likely to lead to solution
  - Learnt clauses are resolvents of earlier clauses
  - Assignment satisfying resolvent extends to original clauses

Decision Heuristic VSIDS

- Variable State Independent Decaying Sum
- For each literal, keep counter of how many learnt clauses it appears in
  - Periodically divide by constant to bias to recently learnt clauses
- At each decision, choose literal with highest counter
- Partial assignments more likely to lead to solution
  - Learnt clauses are resolvents of earlier clauses
  - Assignment satisfying resolvent extends to original clauses
- Possibly leads to shorter learnt clauses
  - Learnt clauses share more literals, shortening resolvents

Optimising UP

- A major portion (80-90%) of solvers' run time is spent on UP.
- Hence, an efficient UP engine is key to any SAT solver.

Optimising UP

- A major portion (80-90%) of solvers' run time is spent on UP.
- Hence, an efficient UP engine is key to any SAT solver.
- Classic (naive) implementation:

Optimising UP

- A major portion (80-90%) of solvers' run time is spent on UP.
- Hence, an efficient UP engine is key to any SAT solver.
- Classic (naive) implementation:
  - For each clause, keep counts of satisfied, falsified and unresolved literals
  - When a literal is assigned or unassigned, visit all clauses with literal and update counters

Optimising UP

- A major portion (80-90%) of solvers' run time is spent on UP.
- Hence, an efficient UP engine is key to any SAT solver.
- Classic (naive) implementation:
  - For each clause, keep counts of satisfied, falsified and unresolved literals
  - When a literal is assigned or unassigned, visit all clauses with literal and update counters
- Drawback: assigning and unassigning expensive