CSC2512
Advanced Propositional Reasoning

CSC2512: Modern CDCL Sat Solvers

CDCL = Conflict Driven Clause Learning

Input formula F in CNF. Determine whether or not F is satisfiable using DPLL with key modifications.

CSC2512: Modern Sat Solvers

DP—Variable Elimination. In each iteration we eliminate a variable **v** by replace the set of clauses C with C – X –Y + R—remove all clauses with the eliminated variable (X the clauses containing **v** and Y the clauses containing **–v**) and add R (all resolvants between clauses in X and Y).

If min(|X|,|Y|) = k, R can contain O(k^2) clauses, and the these clauses can be longer than any clause in X or Y.

So DP can take exponential **space** (and time).

DPLL on the other time requires only linear space (although exponential time). We only need to keep track of a single path in its depth-first search (n copies of the program stack, one for each recursive call, when we have n variables).

Generally speaking space is more constraining than time on modern machines.

CDCL solvers lie somewhere in the middle. They use more space than DPLL, but generally less than DP.

CSC2512: Modern Sat Solvers

Modern DPLL based **CDCL** SAT Solvers (Conflict driven clause learning)

1. **Clause Database**: Each clause is stored as an array/vector of literals.
   – Typically we encode the literals as numbers, e.g., x = 0, -x =1, y = 2, -y = 3. So a clause [x, -y] would be stored as the vector [0, 3]. Under such a scheme negated variables are odd, positive ones are even.
2. **Watch Literals:** We distinguish two literals of each clause as being the watch literals. Each of these literals is said to watch the clause. (Input unit clauses don't have two literals so they are placed directly on the trail). Typically the literals at index 0 and index 1 are used for watches.

3. **Literal Watch Lists:** For each literal we store a list of **watched** clauses—these are the clauses that the literal serves as a watch for.

CSC2512: Sat Solvers

Main Data structures:

4. **Trail:** an array/vector storing the current partial truth assignment being explored. We grow the trail as we descend the search tree, shrink it as we backtrack.
   - Each element on the trail is a pair **(literal, clause index/pointer)**.
   - Implemented as an array treated as a stack where there is a **top pointer (trail_top)** indicating the last entry in the stack. Removing items is done by decreasing **trail_top**. New items are added to the array at index **trail_top.**

5. **UP Stack.** The trail also doubles as a **UP Stack.** We need two stack pointers, **trail_top** that points to next empty slot on the trail, and **up_stack_top** that points to the next literal that needs to be Unit Propagated. We can tell if the **UP Stack** is empty by testing to see if **up_stack_top == trail_top**

Detecting Units the Old Way

For each literal keep a list of clauses it appears in.

Keep a count of the false literals in the clause.

If x is made false, increment the count for every clause it is in. If that count is equal to the clause length -1 the clause has become unit.

Examine the clause to find the literal it implies

Requires work for every clause x appears in

Requires work to restore the counts on backtrack.

Detecting Units the new way with watch literals

UP—processes a clause only when one of its watches become **false.** Then either:

- The other watch is true and we don't need to do anything (the clause is already satisfied)
- the false watch is replaced by a new unset literal.
- If no replacement can be found, we set the other watch to be true.
- The other watch is already false we know that all literals in the clause are false, and we have a conflict (a falsified clause)

CSC2512: Sat Solvers

Unit Propagation:

While up_stack_top != trail_top (more literals to UP)

1. **x** = **Trail[up_stack_top]**; **up_stack_top += 1 //nxt var to UP**
2. **For** each clause **C** watched by -**x //-x is now false**

   a. **y** = **C**'s other watch.

   b. **If y** is **TRUE continue**

   c. **If** there exists **z** = a **non-false** literal in **c** with z ≠ x and z ≠ y **then** move C from x's watched clause list to z's watched clause list.

   d. **Else** //all lits in C are false except possibly for y.

      1. **If y** is **FALSE** **return C** as a conflict clause
      2. **Else** set **y** to **TRUE** and put (**y, c**) on the trail

CSC2512: Sat Solvers

So to update with a newly false literal we need only check a fraction of the clauses the literal appears in (only those it watches).

No work needs to be done on backtrack—if the watches are valid, they will remain valid on backtrack.

CSC2512: Sat Solvers

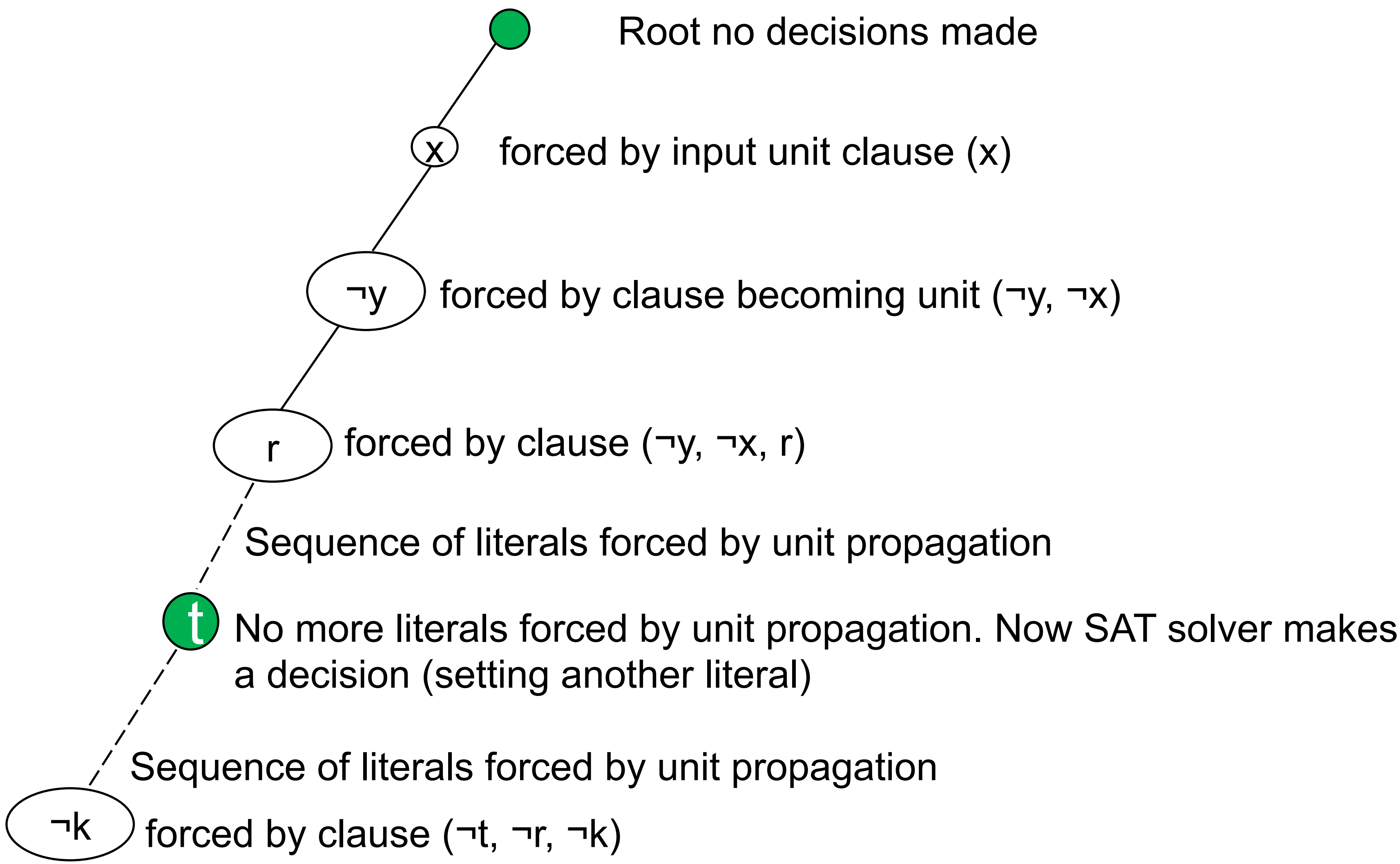
Decision Levels.

The solver operates by (a) making decisions—choosing which literal to set to true, then (b) running UP until the UP stack is empty or a conflict is detected.

The literal set by decision + all of the literals forced by UP after setting the decision literal constitute a section of the trail called **a decision level.**

When the solver **backtracks** it always unsets a full decision level—a decision literal and all of the literals UP'ed by it. It might unset multiple decision levels, but never a subset of a decision level.

CSC2512: Sat Solvers



CSC2512: Sat Solvers

Unit Propagation:

The solver maintains the **invariant** that after each decision level is added **or removed** from the trail every clause has

1. two unassigned watches
2. at least one true watch, or
3. or is a conflict (all literals, and both watches are false).

(One False one unassigned watch not possible).

The invariant is true as the start of the search: every clause has two unassigned watches.

Note that at level 0, no decisions have been made, but we might have unit clauses in F. The invariant holds before these units are propagated, and after UP is finished.

CSC2512: Sat Solvers

Add a decision level **D** to the trail, insert newly decided on literal, and run UP to completion). For each clause either

1. Both watches remain unassigned at level **D**
2. at least one of the watches was true before **D**
3. A watch is made false at level **D** so it is
   1. replaced by an unset watch
   2. the other watch is made true
   3. Both watches have become false and the clause is detected to be a conflict and we backtrack from level **D**
      (Note this means that the conflict could not occurred above level **D**, else we would have backtracked from that prior level and never gotten to level **D**)

Invariant still holds.

CSC2512: Sat Solvers

Backtrack from a decision level **D** to the trail. Either

1. The clause has two unassigned watches at level **D** so they remain unassigned.
2. The clause has two false watches at level **D**. Then both must have been made false at level **D** so on backtrack both will be unset.
3. The clause has a true watch set above level D, and it remains set on backtrack
4. The clause has a true watch set at level D. If the other watch is false it must have been set at level D and both will be unset on backtrack.

Invariant is preserved and more importantly, **no clause needs to be examined on backtrack**! Only need to unassign the literals removed from the trail by backtracking.

CSC2512: Sat Solvers

```
Sat(F)
Build Clause Database and literal watch lists, add units to trail
Dlevel = 0
while (TRUE)
    conflict = UP()
    if (conflict)
        if Dlevel == 0 return UNSAT
        newClause = LearnClause(conflict)
        addToClauseDataBase(newClause)
        backtrack(assertionLevel(newClause)) //undo decision levels
        assign(assertedLiteral(newClause), newClause) //put on trail
    else if  all literals assigned, return SAT (true lits are satisfying assignment)
    else
        x = PickNextLiteral()
        Dlevel = Dlevel+1
        assign(x, NIL)  //Literals made true by decision have no clause reason
```

CSC2512: Clause Learning (Trail)

- X
  - A
  - ¬B
  - C
- ¬Y
  - D
  - ¬E
  - F
- Z
  - H
  - I
  - ¬J
  - ¬K

(K,¬I,¬H, ¬F,E, ¬D,B)

- X,Y,Z: Decision Variables.
- A,¬B,C,D,¬E,F,H,I,¬J,¬K: forced by unit propagation
- (K,¬I,¬H, ¬F,E, ¬D,B): **Conflict Clause**

CSC2512: Clause Learning (Trail)

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- Each forced literal was forced by some clause becoming unit.

CSC2512: Clause Learning (Trail)

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

Each clause reason contains

1. One true literal on the path (the literal it forced)
2. Literals falsified higher up on the path.

CSC2512: Clause Learning (Trail)

- X
  - A ← …
  - ¬B ← (¬B, ¬A)
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- We can resolve away any sequence of forced literals in the conflict clause.
- Such resolutions always yield a new falsified clause.

1. (K,¬I,¬H,¬F,E, ¬D,B), (D,B,Y) → (K,¬I,¬H,¬F,E,B,Y), (¬B, ¬A) → (K,¬I,¬H,¬F,E,¬A,Y)
2. (K,¬I,¬H,¬F,E, ¬D,B), (¬K,¬I,¬H,E,B) → (¬I,¬H,¬F,E, ¬D,B)
3. (K,¬I,¬H,¬F,E, ¬D,B), (H,B,E,¬Z) → (K,¬I,¬F,E,¬D,B,¬Z)
4. …

CSC2512: Clause Learning (Trail)

- Any forced literal x in any conflict clause can be resolved with the reason clause for –x to generate a new conflict clause.
- If we continued this process until all forced literals are resolved away we would end up with a clause containing decision literals only (**All-decision clause**).
- But empirically the all-decision clause tends not be very effective.
  - Too specific to this particular part of the search to be useful later on.

CSC2512: 1-UIP clauses

- The standard clause learned is a 1-UIP clause
  - **LearnClause** learns a 1-UIP clause
- This continually involves resolves the trail deepest literal in the conflict clause until there is only one literal left in the clause that is at the deepest level.
  - Since every resolution step replaces a literal by literals falsified higher up the trail, we must eventually achieve this condition
  - The sole remaining literal at the deepest level is called the **asserted literal.**

CSC2512: 1-UIP clauses

- A **1-UIP** clause is sometimes called an **empowering clause**. It allows UP to force a literal that it wasn't able to before.

CSC2512: 1-UIP Clause (Trail)

- X
  - A ← …
  - ¬B ← (¬B, ¬A)
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

1. (K,¬I,¬H, ¬F,E, ¬D,B), (¬K,¬I,¬H,E,B)
   ➔ (¬I,¬H, ¬F,E, ¬D,B)
2. (¬I,¬H, ¬F,E, ¬D,B), (I,¬H,¬D,¬X)
   ➔ (¬H, ¬F,E, ¬D,B,¬X)

CSC2512: 1-UIP clauses

- The 1-UIP clause forces its asserted literal at a prior decision level (if we had the clause before we would have forced the asserted literal before).
- We backtrack so as to fix the trail to account for the new 1-UIP clause.
- The asserted literal is forced as soon as all other literals in the clause became false. The **assertionLevel** is the second deepest decision level in the clause (the asserted literal is at the deepest level)
- So we backtrack to that level (not undoing the decision or anything forced at that level), add the asserted literal to the trail, with the 1-UIP clause as its reason, then apply UP again.

CSC2512: 1-UIP Clause (Trail)

- X
  - A ← …
  - ¬B ← (¬B, ¬A)
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
  - ¬H ← (¬H,¬F,E, ¬D,B,¬X)

More unit propagation

(¬H, ¬F,E, ¬D,B,¬X)
1-UIP clause

CSC2512: 1-UIP clauses

- On backtrack the newly asserted literal can generate another conflict after UP, this will result in learning a new clause and backtrack further.
- Also note that we are jumping back across incompletely tested decisions.
  - We backtracked over **Z**, but we don't know if **¬Z** might not have lead to a solution.
  - All we know is that the trail is now patched to account for the newly learnt clause
  - Search is no longer "exhaustive" like DPLL
- Empirical evidence is not clear, but (a) it is cheap to backtrack, (b) going back far enough to fix the trail makes the implementation more efficient, (c) allows the search to explore a different area of the space.

CSC2512: 1-UIP clauses

- What happens if the 1-UIP clause is unit?
  - Where do we backtrack to?

CSC2512: Sat Solvers

```
Sat(F)
Build Clause Database and literal watch lists, add units to trail
Dlevel = 0
while (TRUE)
    conflict = UP()
    if (conflict)
        if Dlevel == 0 return UNSAT
        newClause = LearnClause(conflict)
        addToClauseDataBase(newClause)
        backtrack(assertionLevel(newClause)) //undo decision levels
        assign(assertedLiteral(newClause), newClause) //put on trail
    else if  all literals assigned, return SAT (true lits are satisfying assignment)
    else
        x = PickNextLiteral()
        Dlevel = Dlevel+1
        assign(x, NIL)  //Literals made true by decision have no clause reason
```

CSC2512: Clause Learning

LearnClause(conflict)

//Starting with a clause that is falsified by the trail learn a new clause
//(also falsified by the trail) by resolution steps.

1. newClause = conflict
2. **while(**number of lits at decision level Dlevel > 1)
3.   (l, cls) = pop(Trail)
4.   **if** $\neg l \in$ newClause //why can't l be in newClause?
5.   newClause = resolve(cls, newClause) //number of lits at Dlevel may change
6. **Return**(newClause)

assign(lit,cls_reason**)**

1. push(lit,cls) on **Trail** //UP-stack top not updated, so will be UP'ed
2. lit = **True**
3. var(lit).dlevel = Dlevel //record Dlevel of assignment with lit's variable

CSC2512: Clause Learning

assertionLevel(clause)

//Clause must be falsified by trail

1. return(second highest Dlevel of any variable in clause)

assertedLiteral(clause)

//Clause must have only one literal with maximum **Dlevel**

1. return(literal with maximum Dlevel in clause)

backtrack(newDlevel)

//Remove all lits from trail that are at decision levels greater than newDlevel

```
1.  while Dlevel > newDlevel
2.      (l, cls) = pop(Trail)
3.      l = UNASSIGNED
4.      if cls = NIL //decision lit
5.          Dlevel = Dlevel -1
```

CSC2512: VSIDS Heuristic

- Heuristic for selecting next decision literal (variable)
- Variable State Independent Decaying Sum
- Scientific analysis is scant and intuitions vary: but VSIDS is thought to encourage resolutions involving most recently learnt clauses.
  - A counter for each variable. Increment the counter of all variables in the original conflict clause (the clause that was found to be empty by Unit Prop), and the variables in each reason clause resolved with the conflict to generate the 1-UIP clause. (Each such variable has its counter incremented only once. Periodically divide all counts by 2.
  - Pick the unassigned variable with highest count at each decision
- Low overhead (counters updated only for variables in conflict). Lits kept on heap ordered by counter.

CSC2512: VSIDS Heuristic

- The variables appearing in recently used clauses (i.e., clauses used in resolution steps to generate new learnt clauses) will, as we divide by 2, get higher VSIDS scores.
- Variables that at this point in the search are not being used in resolution steps will get their VSIDS scores decayed.

- More recent work (Reading for next week)
  An Empirical Study of Branching Heuristics through the Lens of Global Learning Rate
  Jia Hui Liang, Hari Govind, Pascal Poupart, Krzysztof Czarnecki, and **Vijay Ganesh.**
  In the Proceedings of the 20th International Conference on Theory and Application of Satisfiability Testing **(SAT 2017)**, Aug 28 – Sep 1, 2017, Melbourne, Australia

CSC2512: Phase Saving/Restarts

Restarts

- Periodically restarting the solver (undoing all decisions) is useful.
    - Various strategies have been investigated for when to restart.
    - Note also that all newly learnt units act as a restart---search is backtracked to decision level 0.

Phase Savings

- We decide to branch on a variable: what literal to try first?
- Phase saving: use the literal that was the most recent setting of the variable on the trail.

Interaction: phase saving and restarts interact. The VSIDS scores are unchanged after a restart, so a similar set of decisions will typically be made after a restart. Similarly, phase savings tends to decide on the same value of the decision variables as was used before. So with **phase savings** restarts will tend to put is back into the same part of the search space. But perhaps the small changes are important. This runs counter to the original intuition behind restarts.

CSC2512: Phase Saving/Restarts

Papers:

1. **Randomization in Backtrack Search: Exploiting Heavy-Tailed Profiles for Solving Hard Scheduling Problems. Carla P. Gomes, Bart Selman, Ken McAloon, Carol Tretkoff: AIPS 1998: 208-213**

2. A Lightweight Component Caching Scheme for Satisfiability Solvers Knot Pipatsrisawat and Adnan Darwiche.

CSC2512: Resolution Power

- With these various features it can be show that CDCL solvers (Conflict Driven Clause Learning) are no longer limited to **tree-resolution** instead they can **p-simulate general resolution**
- Remains an open question whether or not CDCL without restarts is as powerful as general resolution.

CSC2512: Clause Minimization

First a few observations:

1. **A Conflict Clause** is a clause that is falsified by the literals made true on the trail.
2. **A Reason clause** is a clause associated with a unit implied literal on the trail. If **R** is the reason clause for the literal x. Then:
   1. x is on the trail (i.e. has been made **true**).
   2. The clause **R** contains x, and other literals $\neg l_1$, $\neg l_2$, ..., $\neg l_k$: **R =** (x, $\neg l_1$, $\neg l_2$, ..., $\neg l_k$) where each $\neg l_i$ has been made **false** on the trail (l_i has been made true).
   3. Each l_i is on the trail above x
3. **The decision level** of a variable x is the decision level at which either x ¬x it is on the trail. (Unset variables do not have decision levels). Remember that the decision levels start at zero and each decision level consists of a decided upon literal along with all the literals forced by unit propagation until the next decision.

CSC2512: Clause Minimization

4. **The decision levels of a Conflict Clause or a reason clause** are the set of different decision levels of its variables.
5. **A trail resolution** is a resolution of a conflict clause and a reason clause. For example a 1-UIP clause is produced by a sequence of **trail resolutions.**

CSC2512: Clause Minimization

Observation: Trail resolutions cannot reduce the number of decision levels in a conflict clause.

Each reason clause $(x, \neg l_1, \neg l_2, \ldots, \neg l_k)$ must contain at least one literal $\neg l_1$ that is at the same decision level as x.

- All the l_1 are above x on the trail, so their decision levels are less than or equal to x. If they all had a decision level less than x, the reason clause would have become unit at a previous decision level.

So if we resolve away $\neg x$ from a conflict clause, we must introduce at least one other literal in the clause at x's decision level.

CSC2512: Clause Minimization

Observation: The minimum size clause that we can produce by doing trail resolutions against a conflict clause has size equal to the number of decision levels in the clause.

CSC2512: Clause Minimization

Clause minimization. Given a conflict clause (typically the 1-UIP clause) $C = (\neg l_1, \neg l_2, \ldots, \neg l_k)$ where each $\neg li$ has been made **false** on the trail, we want to compute via a sequence of **trail resolutions** a new clause C' such that

$$C' \subset C$$

Optimally we want to compute the smallest such C'

CSC2512: Clause Minimization

- X
  - A ← …
  - ¬B ← (¬B, ¬X)
  - C ← (C,B)
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← (¬E, ¬D)
  - F ← (F,¬C, B,E)
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H,¬F,E,¬D,B)

1. (K,¬I,¬H,¬F,E, ¬D,B), (¬K,¬I,¬H,E,B)
   → (¬I,¬H,¬F,E, ¬D,B)
2. (¬I,¬H,¬F,E, ¬D,B), (I,¬H,¬D,¬X)
   → (¬H,¬F,E, ¬D,B,¬X) == 1-UIP clause

3. Further reduction steps
4. (¬H,¬F,E,¬D,B,¬X), (F,¬C,B,E) →
   (¬H,¬**C**, E,¬D,B,¬X)
5. (¬H,¬**C**, E,¬D,B,¬X), (C,B) →
   (¬H, E,¬D,B,¬X)
6. (¬H,E,¬D,B,¬X),(¬E,¬D) →
   (¬H,¬D,B,¬X)
7. (¬H,¬D,B,¬X),(¬B,¬X) →
   (¬H,¬D,¬X)

CSC2512: Clause Minimization

The example shows that clause minimization can have a tremendous effect on the size of the clause. How do we do this:

Clause reduction: simple non recursive method.

```
Proc Reduce(C)
    for literal x ∈ C {
        if (x.ReasonClause \ {x}) ⊂ C
            C = C \ {x}
    }
    return C
```

CSC2512: Clause Minimization

Clause reduction: more sophisticated method.

```
Proc Reduce(C)
   for literal x ∈ C {
      if (lit_is_removable(x, C))
         C = C \ {x}
   }
   return C

Proc lit_is_removable(x, C)
   if (x.ReasonClause = NULL) return FALSE
   if ((x.ReasonClause \ {¬x}) ⊂ C ) return TRUE
   if (for each lit ∈ (x.ReasonClause \ {¬x}) lit_is_removable(lit, C) return TRUE
   …
```

In general, there is a recursive definition. x is removable from C if every literal in x.ReasonClause (other than ¬x) is either in C or is removable from C. Clever techniques are used to remember literals whose "lit_is_removable" status has already been determined, and faster tests to determine "lit_is_removable" in special cases.