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.

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 are directly placed 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**

CSC2512: Sat Solvers

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.

CSC2512: Sat Solvers

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)

Unit Propagation:

While UP-stack is not empty

1. **x** = **Trail[up_stack_top]**; **up_stack_top += 1 //nxt var to UP**
2. **For** each clause **C** watched by -**x //-x is false**
   //(check if that clause has become unit).
   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

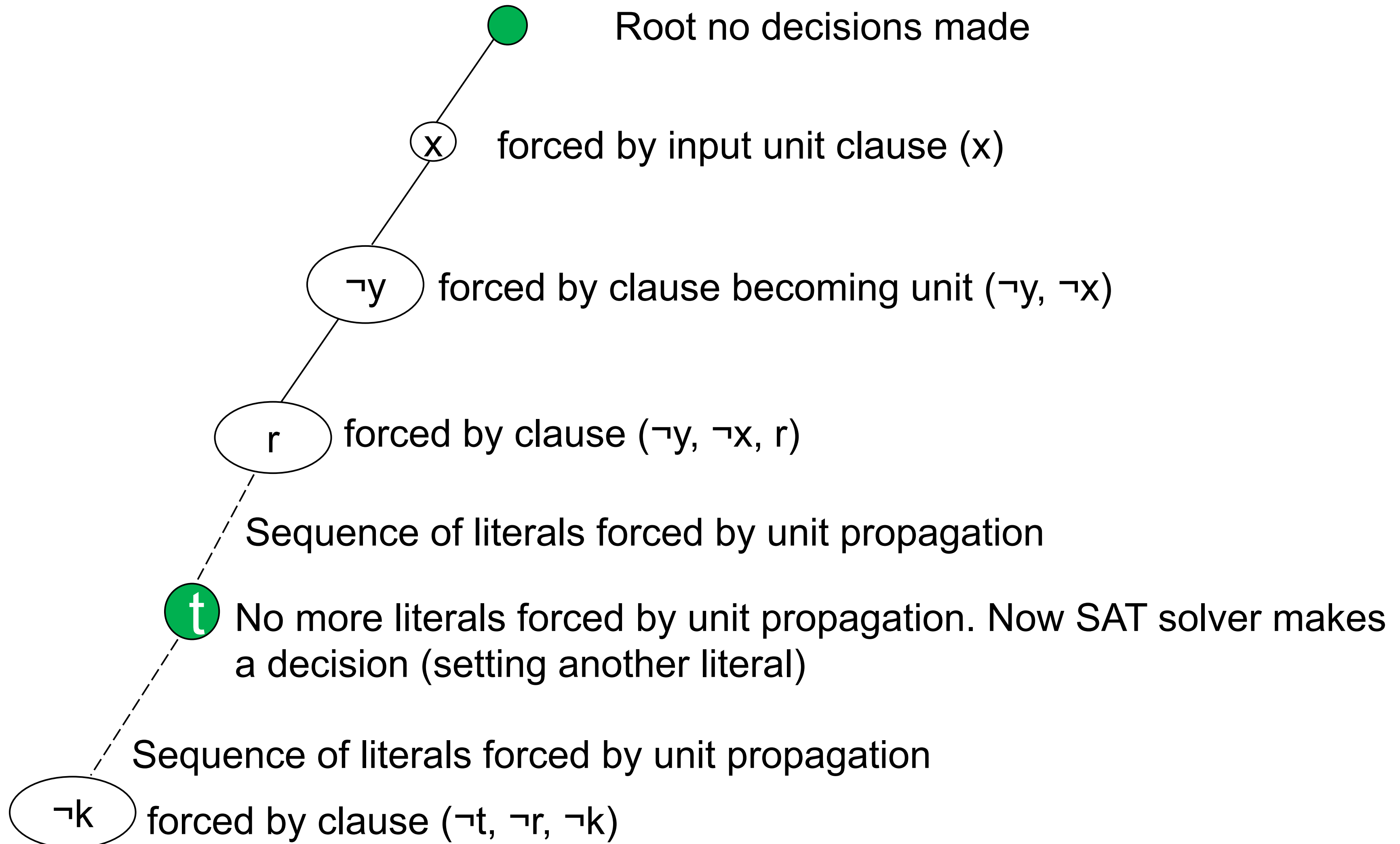
Decision Levels.

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 as a consequence of that decision 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.

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)

1. Build Clause Database and literal watch lists, add units to trail
2. Dlevel = 0
3. while (TRUE)
4. conflict = UP()
5. if (conflict)
6. if Dlevel == 0 return UNSAT
7. newClause = **LearnClause**(conflict)
8. addToClauseDataBase(newClause)
9. backtrack(assertionLevel(newClause)) //undo decision levels
10. assign(assertedLiteral(newClause), newClause) //put on trail
11. else if all literals assigned, return SAT (true lits are satisfying assignment)
12. else
13. x = PickNextLiteral()
14. Dlevel = Dlevel+1
15. assign(x, NIL) //Literals made true by decision have no clause reason

CSC2512: Sat Solvers

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: Sat Solvers

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: 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 **assserted 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

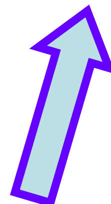
- The 1-UIP clause forces its asserted literal at a prior decision level (if we had the clause before we would have forced the asserted literal before).
- We backtrack so as to fix the trail to account for the new 1-UIP clause.
- The asserted literal is forced as soon as all other literals in the clause 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)

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: 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 $\neg l_1, \neg l_2, \ldots, \neg l_k$: **R =** $(x, \neg l_1, \neg l_2, \ldots, \neg l_k)$ 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 ¬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
   …
```

In general, there is a recursive definition. x is removable from C if every literal (other than ¬x) is either in C or is removable from C.

CSC2512: Other Work: Proof extraction

- For any clause c, if we unit propagate –c, in the formula F and obtain an empty clause (a conflict) then it must be the case that $F \models c$ by the soundness of UP.
- However, we can have $F \models c$ but UP(-c) does not generate a conflict. UP is not a complete rule of inference.
- Nevertheless, it is “complete” along a sequence of resolution steps.

CSC2512: Other Work: Proof extraction

- Given a resolution proof as a sequence of clauses where c_n is not an input clause.

$$c_1, c_2, \ldots, c_n$$

- we can observe that if we negate c_n and unit propagate the literals in the formula $c_1, c_2, \ldots, c_{n-1}$ we will obtain a conflict (one of these clauses will be falsified)

  If c_n = (A, B) as a result of resolving (A,x) and (B,-x) UP falsifies one of these clauses (depending on if it propagates x or –x first).

CSC2512: Other Work: Proof extraction

- This gives rise to the RUP (reverse unit propagation) technique for extracting proofs from a clause learning SAT solver.

- Output to a log all learnt clauses in the sequence they are learnt.

- Verify each learnt clause c_i in the order they it was learnt by negating c_i and unit propagating through the set of clauses U $\{c_1, \ldots, c_{i-1}\}$

- If we obtain a conflict we know that c_i is a logical consequence of the input formula and the previously verified learnt clauses.

CSC2512: Other Work: Proof extraction

- Eventually we can verify a unit clause (x) whose partner (-x) has previously been verified thus showing that the proof is sound.
- This procedure verifies the UNSAT result (just like the satisfying assignment can be used to verify the SAT result).
- Furthermore, we can instrument the UP checking process so that we only keep the learnt clauses and input clauses that are eventually needed to verify the final empty clause.
- Note that this works even when we have lost track of the resolution steps involved in computing a learnt clause.
- This set of clauses responsible for UNSAT can be output.

CSC2512: Other Work: Proof extraction

- Note also that the final sequence of learnt clauses are not a traditional resolution proof. This sequence is called a "clausal" proof, and it can be much shorter than a resolution proof.

- Unit Prop is needed to verify a clausal proof, whereas a much simple algorithm can verify a resolution proof.

- A clausal proof can be expanded into a resolution proof by tracking the clauses the unit prop steps need to derive a contradiction.

Paper: Trimming while Checking Clausal Proofs Marijn J.H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler

CSC2512: Other work: Assumptions

Assumptions. A useful technique is solving subject some set of literals called assumptions:

- $A = \{l_1, l_2, \ldots, l_k\}$
- We start the SAT solver and force it to pick a next unassigned literal in A as a **decision** until there are no more unassigned literals in A.
- If a literal of A is forced to TRUE we skip over it for the next decision.
- If a literal of A is forced to FALSE we stop: Say $l_1, l_2 \ldots, l_i$ are the decisions already made, and l_j is forced to FALSE: then we have the following clause

$$(\neg l_1, \neg l_2, \ldots, \neg l_i, \neg l_j)$$

CSC2512: Other work: Assumptions

- If we assign all literals in A we then continue the normal SAT solving process with freedom to pick the decision variables as we want.
- If this results in UNSAT, some clause (perhaps empty) falsified by the A decisions will be learnt.

- In any event, if the formula becomes UNSAT under A, we obtain a clause falsified by A (the clause specifies that some subset of A is impossible).

CSC2512: Other work: Assumptions

- Assumptions are very useful for **incremental** SAT solving where we want to SAT solver a sequence of related formulas F_1, F_2, …, F_n

- If $F_1 \subseteq F_2 \subseteq \cdots \subseteq F_n$ then we can use one instance of the SAT solver. Solve F_1 then add the additional clauses of F_2 and solve again, add the additional clauses of F_2 …
  - The advantage if this is that the SAT solver gets to reuse all of its learnt clauses.
- But if we must remove clauses between SAT solver invocations we have a problem: some of the learnt clauses might no longer be valid.

CSC2512: Other work: Assumptions

- For clause removals we can use assumptions: if we will later want to remove the clause $C = (x_1, x_2, \ldots, x_n)$ we can add a brand new variable (often called a selection variable) to the clause: $(x_1, x_2, \ldots, x_n, s)$
- Then if we want to include C we assume $\neg$**s.** Any learnt clauses were derived from resolving against C will now also contain **s** (**s** appears nowhere else in the formula so it can't be resolved away)
- When we want to exclude C from the formula we don't assume anything. The clause can always be satisfied by the SAT solver be making **s** true—and again since **s** appears nowhere else we will no longer learn any clauses from C. (Alternately we can assume **s**)

CSC2512: Papers for Next Time

- 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.
- 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
- A Lightweight Component Caching Scheme for Satisfiability Solvers Knot Pipatsrisawat and Adnan Darwiche.
- Trimming while Checking Clausal Proofs Marijn J.H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler

CSC2512: Clause Deletion

- A new clause is learned from every conflict.
- In practice the solver starts to slow down after it accumulates too many clauses.
- So deleting some of these learned clauses has proved to be effective.
- Earlier clauses were deleted whenever memory was about to be exhausted. Clauses were deleted by size (delete the largest ones first) or by activity (delete those clauses that had not recently been used in learning new clauses.

CSC2512: Clause Deletion

- In 2009 Audemard and Simon developed a new idea for selecting which clauses to delete called the LBD score.
- Once we learn a 1-UIP clause, minimize it, and use it to backtrack asserting a new literal, we can count the # of different decision levels in the clause. This is the LBD score.
- Audemard and Simon found that very aggressive clause deletion where every 10,000 learnt clauses ½ with highest LBD score are deleted, give a significant boost in performance.
- Now however theoretical completeness is sacrificed (although it can be regained by slowing increasing the clause deletion trigger from 10,000 clauses to 20,000, 30,000, etc. (any increasing sequence will suffice).

CSC2512: Clause Deletion

- Papers for next time:

1. Predicting Learnt Clauses Quality in Modern SAT Solvers, Gilles Audemard, Laurent Simon, IJCAI 2009.
2. Coverage-Based Clause Reduction Heuristics for CDCL Solvers, Hidetomo Nabeshima and Katsumi Inoue, SAT 2017

CSC2512: Preprocessing

- An additional essential part of modern SAT solvers is preprocessing and inprocessing.
- Preprocessing is the technique of converting the input CNF F to a new CNF F' such that if F' is UNSAT then so is F, and furthermore if π is a satisfying model of F' then π can in poly-time be converted into π' a satisfying model for F.

CSC2512: Preprocessing

- A number of useful techniques for preprocessing have been developed. The most important of these is Bounded Variable Elimination, Clause subsumption, and self-subsuming resolutions.

CSC2512: Preprocessing

- Bounded Variable Elimination. This is a like a single step of DP. We eliminate the variable x from the formula by taking all clauses A containing x and all clauses B containing ¬x and generate all resolvant pairs:
  $R = \{R[c_{1,}c_2] \mid c_1 \in A \;\; c_2 \in B\}$
- All tautologies are removed from R. Furthermore, clauses in R might be subsumed by other clauses. So we reduce R by removing these clauses.
- **Bounded**: we preform this step if
  $|R| < |A| + |B|$
  (i.e., we obtain fewer clauses)

CSC2512: Preprocessing

- Implementing this efficiently requires clever scheduling and data structure techniques.

CSC2512: Preprocessing

- Subsumption. Checking for subsumption can be speeded up using **Bloom Filters**. We map each literal in F to a number in the range [0,63]. Then for each clause **c** we construct a 64 bit map, by setting all bits mapped to by literals in **c.**
- Now if **c**’ $\subseteq$ **c** then **c**’ bit map must be a subset of **c** bit map. That is, the and of these two bit maps must equal **c’** bit map.
  - **E.g.**, **x = 0**, **-x = 1**, **y = 2, -y = 3**, **z = 4**, **-z = 5.**
    c = [x,y,z] = [1, 0, 1, 0, 1, 0]
    c’= [x,z] = [1, 0, 0, 0, 1, 0]

    [1, 0, 1, 0, 1, 0] AND [1, 0, 0, 0, 1, 0] = [1, 0, 0, 0, 1, 0] = c’ bit map
- This test is fast, and if it fails then we know that **c** is not subsumed by **c’.** If it succeeds then we actually have to test for subsumption as this is a one-way test.
  - **E.g. r = 4**
    c” = [x,r] = [1, 0, 0, 0, 1, 0] – same bit map as c’

CSC2512: Preprocessing

- (A,x) (B,¬x) where B $\subseteq$ A. Clearly(B,¬x) is not a subset of (A,x), so there is no clause subsumption. However, consider the resolvant: (A,B) == (A) (since B $\subseteq$ A). The resolvant subsumes (A,x). So in this case we can remove x from (A,x). This is called a **self-subsuming** resolution.
  - E.g.
    (a, b, ¬c, **x**) and (b, ¬**x**)
    ➔ (a,b,¬c) and (b, ¬**x**)

CSC2512: Paper for next time.

1. Effective Preproessing in SAT through Variable and Clause Elimination, Niklas Een and Armin Biere, SAT 2005.

CSC2512: MUSes and MCSes

- MUS computation.

- In many applications we want to know why something is unsatisfiable. We can extract a minimal unsatisfiable subset of the formula: a **MUS**.
- Note not a **minimum** unsatisfiable subset (which is a much harder problem)
- Since the **MUS** typically much smaller than the input formula **F.** It can provide much more specific information about a cause of unsatisfiablity in **F.**

CSC2512: MUSes and MCSes

- A MUS (Minimal Unsatisfiable Set) **M** is an UNSAT set of clauses **M** such that for any clause c in **M**:
  M \ {c} is SAT //M is set inclusion minimal
  - If **F** is SAT then it contains no MUSes. If it is UNSAT it contains at least one MUS and usually contains many different MUSes.

- A correction set **C** of a CNF **F** is a subset of **F** such that:
  F \ C is SAT
  A correction set **C** is a **minimal correction set (MCS)** if no proper subset of **C** is a correction set of **F**
  - If **F** is SAT only the empty set is a MCS. But if **F** is UNSAT, then any MCS cannot be empty and generally, there are many MCSes.

CSC2512: MUSes and MCSes

- MUS/MCS hitting set duality (Reiter AIJ 2087).

- Consider an UNSAT formula **F**, let **AllMuses(F)** be the collection of all MUSes in **F.** Each $M \in$ **AllMuses(F)** is a set of clauses, a subset of **F**. That is **AllMuses(F)** is a collection of sets.
- Similarly, let **AllMCSes(F)** be the collection of all MCSes of **F**

CSC2512: MUSes and MCSes

- Given a collection of sets **K**, **HS** is a **hitting set** of **K** iff for every set $S \in \mathbf{K}$ we have that $\mathbf{HS} \cap \mathbf{S} \neq \emptyset$
  - **HS** has a non-empty intersection with every set in the collection.
- A set **HS** is a **minimal hitting set** of **K** if it is a hitting set and no proper subset of **HS** is a hitting set of **K.**

CSC2512: MUSes and MCSes

- Reiter's result:

A set **C** ⊆ **F** is an MCS of **F iff** it is a minimal hitting set of **AllMuses(F)**. And a set **M** ⊆ **F** is an MUS of **F iff** it is a minimal hitting set of **AllMCSes(F).**

Also, a set **C** ⊆ **F** is an correction set (not necessarily minimal) of **F iff** it is a hitting set of all unsatisfiable subsets of **F** (not necessarily minimal). And a set **M** ⊆ **F** is unsatisfiable (not necessarily minimal) **iff** it is a hitting set of all correction sets (not necessarily minimal) of **F.**

CSC2512: MUS extraction

- Given an UNSAT formula **F**, we want to compute one of its **MUSes** (and we don't care which one).
- We can do this with a sequence of calls to a SAT solver.

CSC2512: MUS extraction

- A **critical** clause of an UNSAT formula **U**, is a clause whose removal makes **U** SAT.
  - A MUS **M** is an UNSAT formula all of whose clauses are critical.

- Divide **F** into two sets
  - **crits:** a set of clauses that we know must be in the **MUS** we are extracting (they are critical).
  - **unkn** a set of clauses that might be in the **MUS** but we don't know yet.

  - **crits U unkn** is the working formula—it is an unsat formula that is a subset of **F** and thus it contains one of **F's** MUSes

CSC2512: MUS extraction

1. crits ⟵ ∅ unkn ⟵ **F**
2. **while** unkn ≠ ∅
    1. c ⟵ **choose** c ∈ unkn
    2. sat? = SatSolve(crits U unkn \ {c})
    3. **if sat?**
        1. crits = crits U {c}
    4. unkn = unkn - \{c}

This simple algorithm iteratively tests the clauses of an initial UNSAT formula (**F**) removing clauses not needed to retain UNSAT, and keeping those clauses whose removal makes the formula SAT.

CSC2512: MUS extraction

This simple algorithm can be significantly improved.

1. When we find that removing **c** from unkn makes crits U unkn (so **c** is critical for the **MUS** contained in crits U unkn) we can use the satisfying truth assignment π to find other critical clauses.
2. When removing **c** from unkn keeps crits U unkn UNSAT (**c** need not be in the **MUS** we are extracting), then we can extract from the SAT solver a subset of the clauses in crits U unkn sufficient to cause UNSAT using **assumptions**

CSC2512: MUS extraction

Model Rotation. Given that **c** is found to be critical, find other critical clauses.

When is a clause **c** critical for the working formula crits U unkn.

There exists a truth assignment satisfying (crits U unkn) \ {**c**}—removing c makes the working formula SAT

We have found a truth assignment π satisfying (crits U unkn) \ {**c**}

So we try to change one of the truth assignments in π so that we satisfy **c** and every other clause in (crits U unkn), except for one other clause **c'**. So now we have a new truth assignment π' that satisfies (crits U unkn) \ {**c'**}

This shows that **c'** is also critical for the current working formula and we can move it into crits.

CSC2512: MUS extraction

Assumptions. Instead of giving the SAT solver the CNF (crits U unkn) we add a **selector** variable to each clause of **F.**

The selector variables are brand new variables (one new variable per clause). So every clause $c_i \in$ **F** is replaced with the clause $(c_i \vee b_i)$ where b_i is the new selector variable for clause c_i.

Then we call the SAT solver with all clauses in **F** and the assumptions $\{\neg b_1, \neg b_2, \ldots, \neg b_m\}$. These assumptions force the SAT solver to try to satisfy all of the clauses c_i

CSC2512: MUS extraction

Assumptions.

When we want to remove the clause ci from the working formula, we stop assuming ¬bi. Now the SAT solver is free to satisfy ($c_i \vee b_i$) by simply making b_i true.

If the working formula is UNSAT then the SAT solver will return a subset of the assumptions $\{\neg b_1, \neg b_2, \ldots, \neg b_m\}$ in a conflict clause $(b_{j1}, b_{j2}, \ldots, b_{jk})$. This clause says that the subset of clauses $\{c_{j1}, c_{j2}, \ldots, c_{jk}\}$ is UNSAT (at least one of them must be falsified by any truth assignment.

Now we can use this subset to further reduce the working formula.

CSC2512: MUS extraction

1. crits_A ⟵ ∅
2. unkn_A = $\{\neg b_i \mid c_i \in \mathbf{F}\}$ //Two sets of assumptions
3. **while** unkn_A ≠ ∅
    1. **choose** $\neg b_i \in$ unkn_A
    2. (sat?, π, conflict) =
       SatSolve(**F**, crits_A U unkn_A \ $\{\neg b_i\}$)
    3. **if sat?**
        1. new_crits = Model_Rotate(c_i, π)
        2. crits_A = crits_A U $\{\neg b_i \mid c_i \in$ new_crits}
        3. unkn_A = unkn_A \ $\{\neg b_i \mid c_i \in$ new_crits}
    4. **else**
        1. unkn_A = unkn_A ∩ $\{\neg b_i \mid b_i \in$ conflict}

CSC2512: MUS extraction

However we can get even better improvements by moving beyond the simple algorithm.

Paper for next time:

Using Minimal Correction Sets to more Efficiently Compute Minimal Unsatisfiable Sets, Fahiem Bacchus and George Katsirelos (CAV 2015).