Introduction to Satisfiability Solving with Practical Applications

Niklas Een

SAT solvers

Inner workings

The SAT problem

A **literal** p is a variable x or its negation $\neg x$.

A **clause** C is a disjunction of literals: $x_2 \vee \neg x_{41} \vee x_{15}$

A **CNF** is a conjunction of clauses:

$$(x_2 \vee \neg x_{41} \vee x_{15}) \wedge (x_6 \vee \neg x_2) \wedge (x_{31} \vee \neg x_{41} \vee \neg x_6 \vee x_{156})$$

The **SAT-problem** is:

- Find a boolean assignment
- such that each clause has a true literal

First problem shown to be NP-complete (1971)

What's a clause?

A clause of size n can be viewed as n propagation rules:

$$a \vee b \vee c$$

is equivalent to:

$$(\neg a \wedge \neg b) \rightarrow c$$

$$(\neg a \wedge \neg c) \rightarrow b$$

$$(\neg b \wedge \neg c) \rightarrow a$$

Example: Consider the constraint

$$t = \text{AND}(x, y)$$

x=0 $\rightarrow$ t=0
y=0 $\rightarrow$ t=0
x=1 and y=1 $\rightarrow$ t=1

$\neg x \rightarrow \neg t$
$\neg y \rightarrow \neg t$
$x \wedge y \rightarrow t$

$\{x, \neg t\}$
$\{y, \neg t\}$
$\{\neg x, \neg y, t\}$

$$\neg t \wedge y \rightarrow \neg x$$

Example

{3, 6, -7, 8}
{1, 4, 7}
{-8, 4}
{-1, -3, 8}
{-3, -4, -8}
{-1, -2, 3, 4, -6}

→

{**3**, 6, -7, 8}
{1, 4, 7}
{-8, 4}
{-1, **-3**, 8}
{**-3**, -4, -8}
{-1, -2, **3**, 4, -6}

→

{**3**, 6, -7, 8}
{**1**, 4, 7}
{-8, 4}
{**-1**, **-3**, 8}
{**-3**, -4, -8}
{**-1**, -2, **3**, 4, -6}

Unit clause (BCP)

→ . . .

. . . →

{**3**, 6, -7, **8**}
{**1**, 4, 7}
{**-8**, 4}
{**-1**, **-3**, **8**}
{**-3**, -4, **-8**}
{**-1**, -2, **3**, 4, -6}

Another unit clause (more BCP)

→

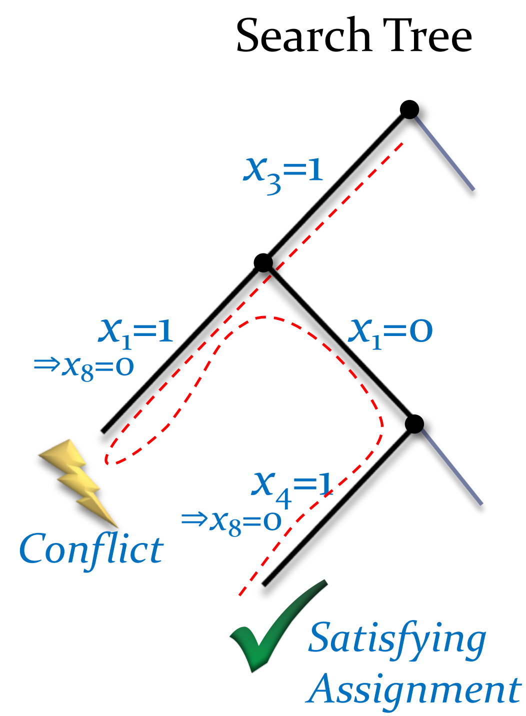
{**3**, 6, -7, **8**}
{**1**, **4**, 7}
{**-8**, **4**}
{**-1**, **-3**, **8**}
{**-3**, **-4**, **-8**}
{**-1**, -2, **3**, **4**, -6}

CONFLICT! (backtrack)

Search Components

- Decision heuristic
  - Static (x_1, x_2, x_3...)
  - State based
    - Shortest non-satisfied clause, most common literal etc.
  - History based
    - Pick variables that lead to conflicts in the past.
- Propagation
- Backtracking

Propagation
Backtracking



Search Components

Decision heuristic

Propagation

- Unit propagation ("BCP")
- Unate propagation
- Probing/Dilemma
- Equivalence classes

Backtracking

~~{3, 6, -7, 8}~~

{1, 4, 7}

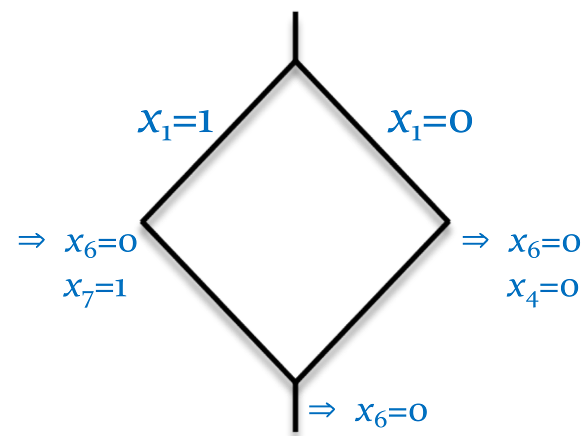
{-8, 4}

{-1, -3, 8}

{-3, -4, -8}

~~{-1, -2, 3, 4, -6}~~

Unate (pure literal)



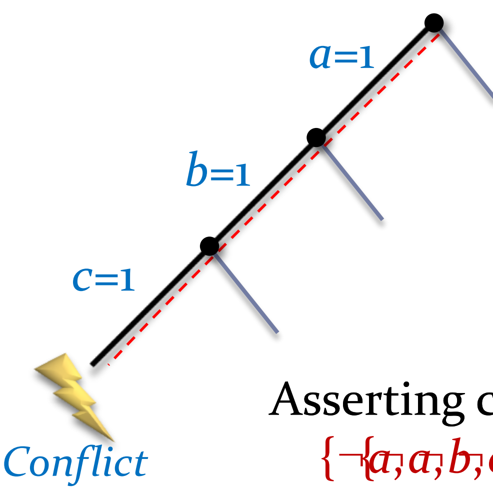
Search Components

- Decision heuristic
- Propagation
- Backtracking
  - Flip last decision (standard recursive backtracking)
  - Conflict analysis:
    - Learn an ***asserting clause***
    - [...]

May be expressed in any variables, not just decisions.

Must have only *one* variable from the last decision level.

```
dpll(assign){
  "do BCP";
  if "conflict": return FALSE;
  if "complete assign": return TRUE;
  "pick decision variable x";
  return dpll(assign[x=0])
         || dpll(assign[x=1]);
}
```



Asserting clause:
$\{\neg a, \neg b, \neg c\}$

What if b was irrelevant?

Search Components

Decision heuristic

Propagation

Backtracking

- Flip last decision (standard recursive backtracking)
- Conflict analysis:
  - Learn an asserting clause
  - Backjumping
  - No recursion
  - Can be viewed as a resolution strategy, guided by conflicts.
  - Together with *variable activity*, most important innovation.

```
dpll(assign){
  "do BCP";
  if "conflict": return FALSE;
  if "complete assign": return TRUE;
  "pick decision variable x";
  return dpll(assign[x=0])
         || dpll(assign[x=1]);
}
```

```
forever{                          – CDCL procedure
  "do BCP"
  if "no conflict":
     if "complete assign": return TRUE;
     "pick decision x=0 or x=1";
  else:
     if "at top-level": return FALSE;
     "analyze conflict"
     "undo assignments"
     "add conflict clause"
}
```

Conflict Analysis – Graph View

Conflicting clause:

{¬x10587, ¬x10592, ¬x10588}

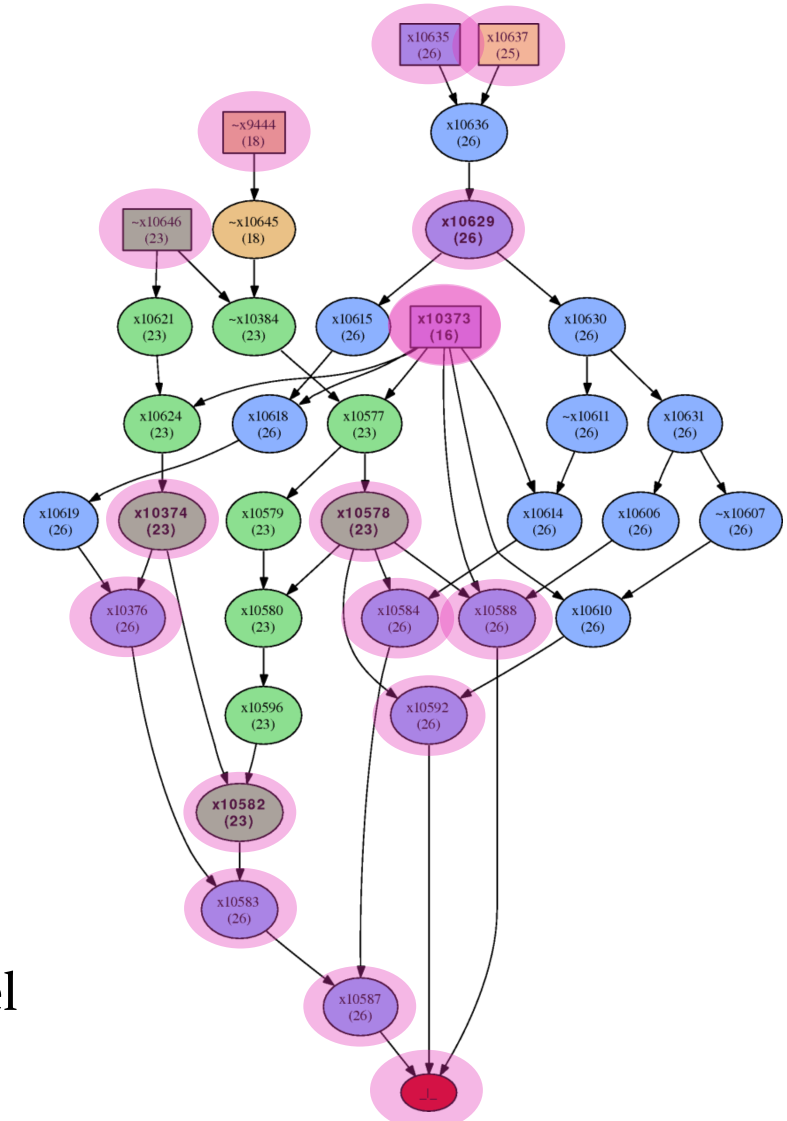
One option:

- Trace back to decision variables
- Would learn:

  {x10646, x9444, ¬x10373 , ¬x10635 ¬x10637}

Other option:

- Stop earlier
- Asserting if only one literal left at the highest decision level
- Keep expanding nodes from that level



Conflict Analysis – Resolution View

Decision	Implications
¬a	–
¬b	c
¬d	e, ¬f

Clause Database

[...]
{c, a, b} — *reason* for **c**
{¬f, ¬e, d, b} — *reason* for **¬f**
{e, d, ¬c, b, a} — *reason* for **e**
{f, ¬e, ¬c} — *conflicting clause*
[...]

start with the conflicting clause

resolve with reason of last assigned literal

{**f**, **¬e**, ¬c} {**¬f**, **¬e**, **d**, b}

resolve on **f**

keep resolving until only one literal of last decision level

{**¬e**, **d**, ¬c , b} {**e**, **d**, ¬c, b, a}

resolve on **e**

{**d**, ¬c, b, a}

Done!

or not?

Conflict Clause Minimization:

Continue to resolve if result is a strict subset

Resolution:

$\{x, A\}$ res. $\{\neg x, B\} = \{A, B\}$

blue = last decision level

Variable Activity

The VSIDS activity heuristic:

- Bump literals of the learned (conflict) clause
- Decay by halfing activity periodically

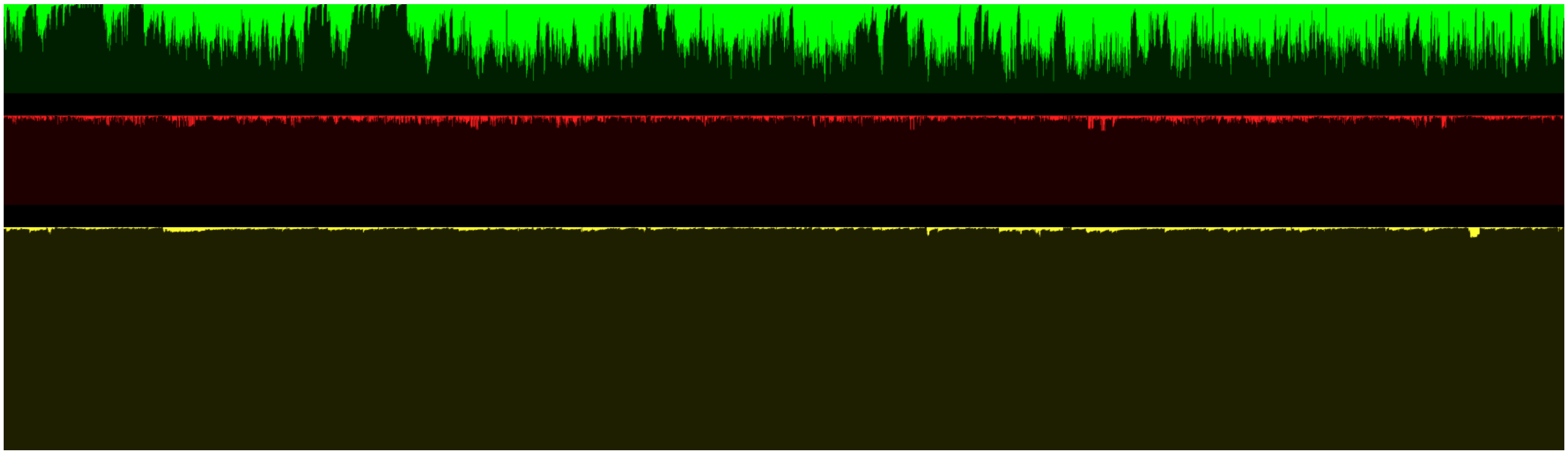
Modified activity heuristic:

- Bump *variables* of *all* clauses participating in analysis
- Decay after each conflict

Effect:

- Give preference to the very latest conflicts(Berkmin/VMTF)
- Longer memory (15000 decays before minimal float value)

Execution of CDCL Solver



Green – Activity of decision variable
Red – Length of learned clause
Yellow – Decision depth when conflict occurred

Other Techniques

Two watched literals

- not moved during backtrack;
- migrate to silent places
- improves with length of clauses
- most BCP in learned clauses (often 90%), which are long

Restarts with polarity memoization

- frequent restarts, except sometimes: 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8…
- not real restarts
- compresses assignment stack => more focus on active variables

Conflict-clause deletion

- remove clauses that don't participate in conflict analysis
- handles subsumed clauses better than original scheme (based on length)

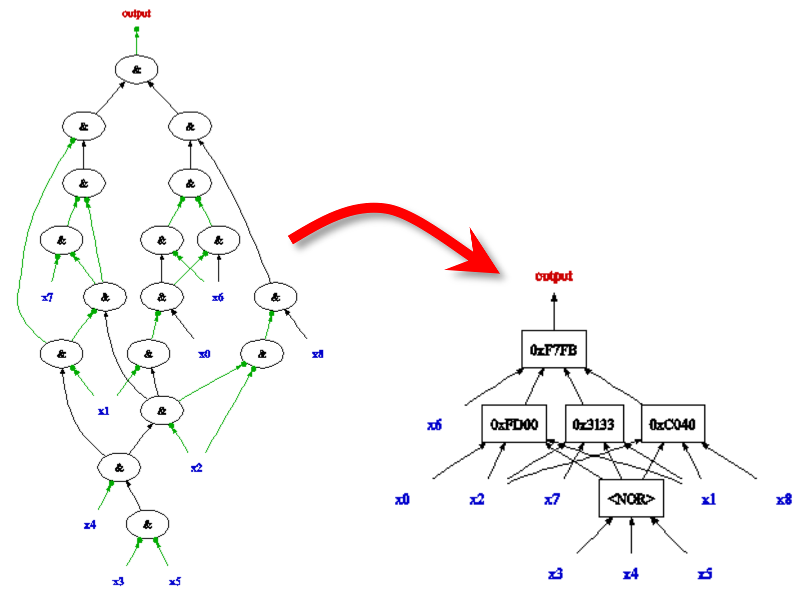
CNF preprocessing

- variable elimination
- subsumption, self-subsuming resolution

Other Techniques (cont.)

Better CNF generation

- If problem on circuit form:
  - Technology mapping for CNF
  - Fanout aware variable elimination
- Certain constraints (e.g. cardinality constraints) have known efficient encodings.



Improvements to incremental SAT

- Domain specific adjustments

Method	Approx. #conflicts (Charactersitics)
BMC	100
Interpolation	1,000 (clause deletion, proof logging)
PDR	10,000 (local problems, limited proof logging)
SAT-sweeping	100,000 (local problems)

SAT Research

Practical SAT is an experimental science.

There are three types of papers:

- The conclusion is wrong.
- The conclusion is correct,
  but not for the stated reasons.
- The conclusion is correct, the stated reasons are valid, but the experimental data does not support it.

Benchmark	1Cl	2Cl	4Cl	8
1dlx_c_bp_f	8.26	4.38	2.25	1.
1dlx_c_ex_bp_u_f	21.86	11.5	6.29	3.2
4pipe	3.12	1.7	0.91	0.49
5pipe	13.3	7.12	3.89	2.04
9vliw_bp_mc	30.64	16.36	8.27	4.62
engine_4_nd	3.85	2.03	1.13	0.68
engine_5_nd_case1	45.61	24.84	13.94	7.62
hanoi5	0.15	0.08	0.04	0.02

It is *hard* to improve the CDCL algorithm.

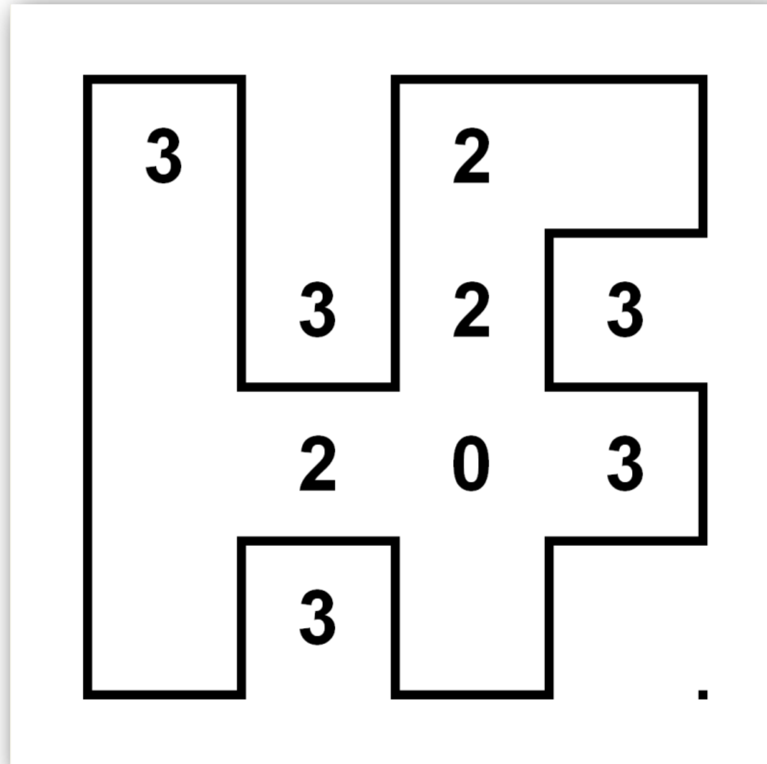
Applying SAT solvers

Solving puzzles

Slither Link

Rules

1. Each number must be surrounded by that many edges.
2. All edges must form a single closed loop.



Slither Link

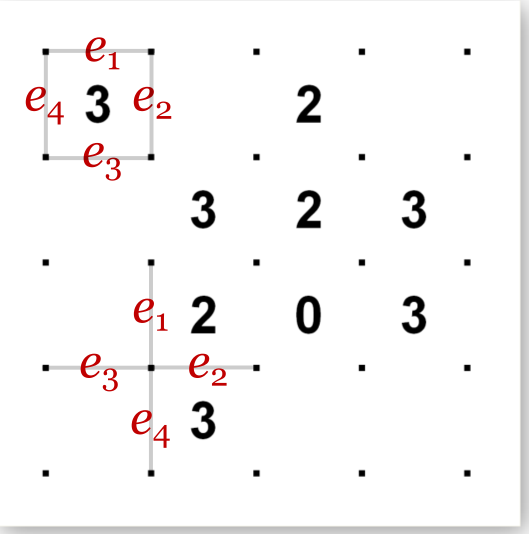
Rules

1. Each number must be surrounded by that many edges.
2. All edges must form a single closed loop.

Constraints

A. Rule 1 is easily expressed:
   - Let e_1, e_2, e_3, e_4 be the edges around a number k.
   - Encode in CNF: $\mathbf{card}(e_1, e_2, e_3, e_4) = k$

B. An approximation of rule 2 can be enforced locally:
   - Every crossing should have either zero or two edges.
   - Encode as: $\mathbf{card}(e_1, e_2, e_3, e_4) = 0 \text{ or } 2$



Example. $k = 1$:

$\{e_1, e_2, e_3, e_4\}$,
$\{\neg e_1, \neg e_2\}, \{\neg e_1, \neg e_3\}, \{\neg e_1, \neg e_4\}$,
$\{\neg e_2, \neg e_3\}, \{\neg e_2, \neg e_4\}, \{\neg e_3, \neg e_4\}$

Local loop constraint.

$\{\neg e_1, \neg e_2, \neg e_3\}, \{\neg e_1, \neg e_2, \neg e_4\}$,
$\{\neg e_1, \neg e_3, \neg e_4\}, \{\neg e_2, \neg e_3, \neg e_4\}$,
$\{e_1, e_2, e_3, \neg e_4\}, \{e_1, e_2, \neg e_3, e_4\}$,
$\{e_1, \neg e_2, e_3, e_4\}, \{\neg e_1, e_2, e_3, e_4\}$

Slither Link (cont.)

Lets run it…

…close, but no cigar.

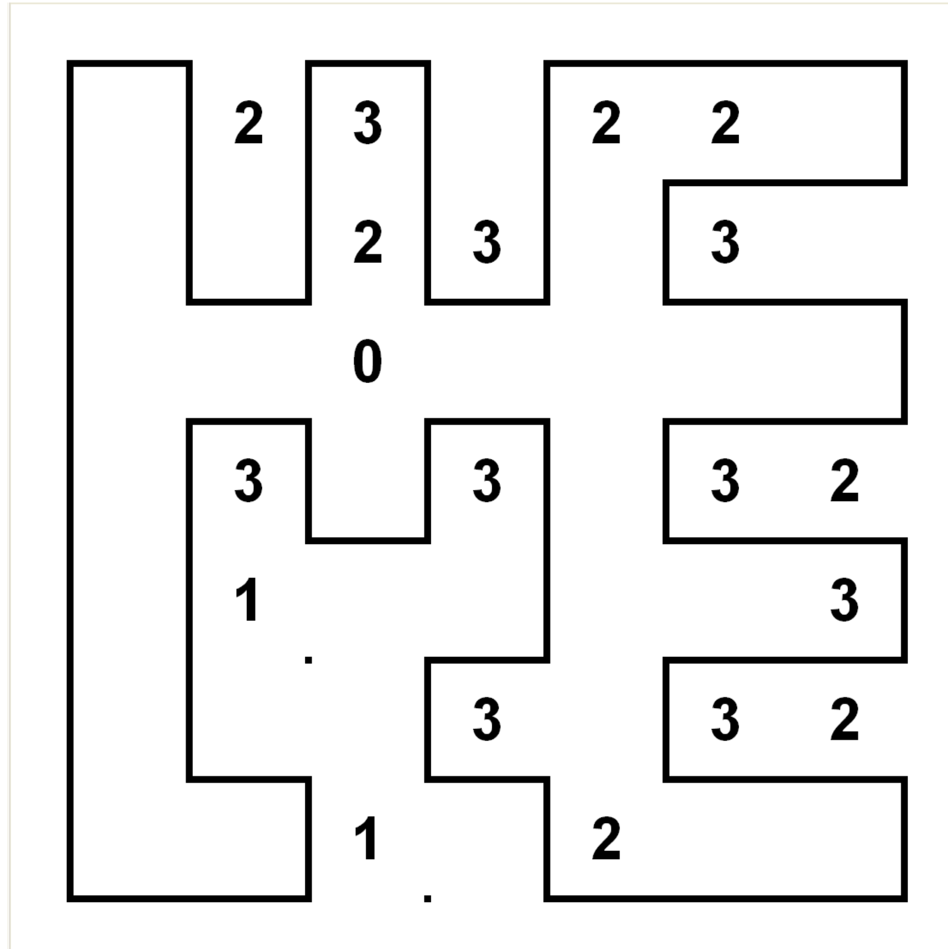
But with a *CEGAR*!

Refine by prohibiting these particular cycles.

Repeat

Repeat

Done!



Slither Link (cont.)

Incremental solution works well for larger sizes too.

Exercise: Formulate a SAT encoding that will solve *Slither Link* non-incrementally (one SAT call only).

Other nice puzzles

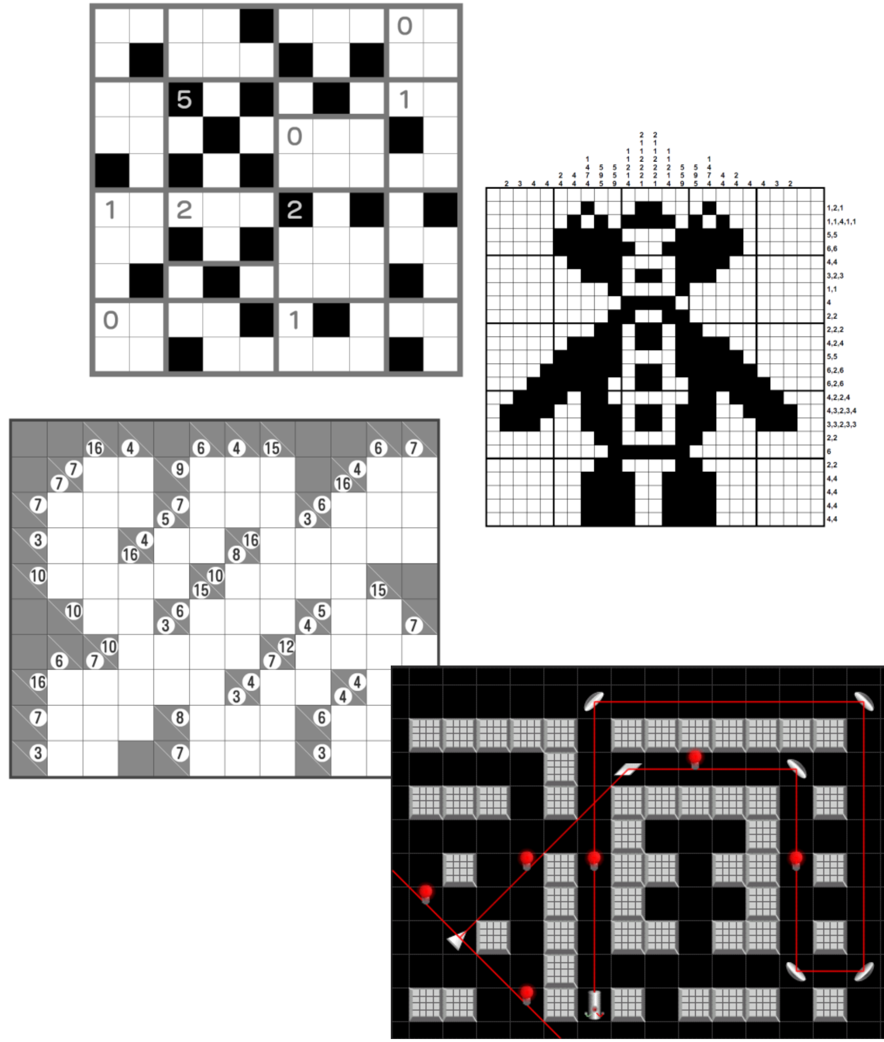
Heyawake

Hanjie

Kakuro

Reflections

…try one with SAT



http://games.erdener.org/laser/

Applying SAT solvers

Verification

Incremental SAT

MiniSat API

- void *addClause*(Vec<Lit> clause)
- bool *solve*(Vec<Lit> assumps)
- bool *readModel*(Var x) – *for SAT results*
- bool *assumpUsed*(Lit p) – *for UNSAT results*

The method *solve()* treats the literals in assumps as unit clauses to be temporary assumed during the SAT-solving.

More clauses can be added after *solve()* returns, then incrementally another SAT-solving executed.

Allows for…

Refinement loop

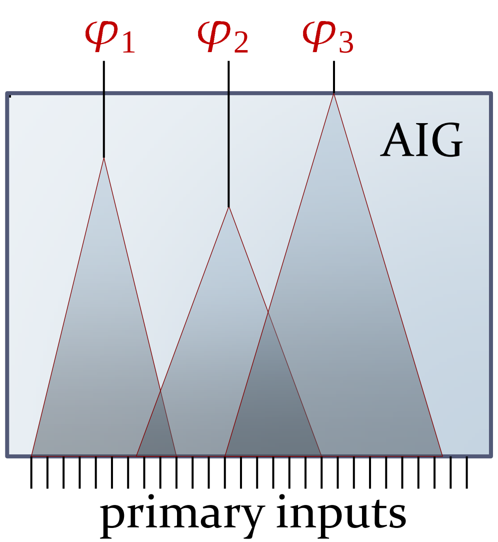
- More clauses can be added with *addClause()*

Restricted clause deletion

- Clauses can be tagged by an **activation literal** "a":
  $\{\neg a, p_0, p_1, \ldots, p_n\}, \{\neg a, q_0, q_1, \ldots, q_m\}, \ \ldots$
- Activated by passing a as part of assumps to *solve()*
- Deleted by *addClause*($\{\neg a\}$)

Poor-mans proof logging

- If we have several sets of clauses $A_1, A_2, \ldots$ with different activation literals $a_1, a_2, \ldots$, *assumpUsed()* tells us which sets were used for proving UNSAT
- Also works for output of cones of logic in a circuit

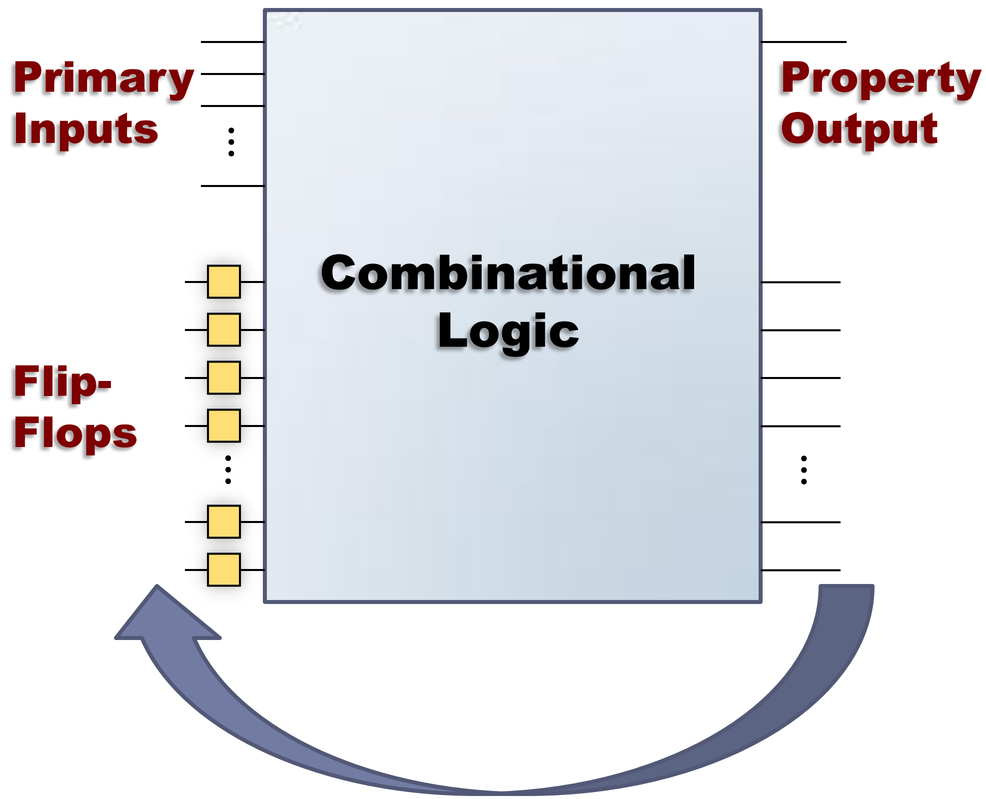


Bit-level Verification

Design is given as a netlist of:

- AND gates
- PIs
- Flops

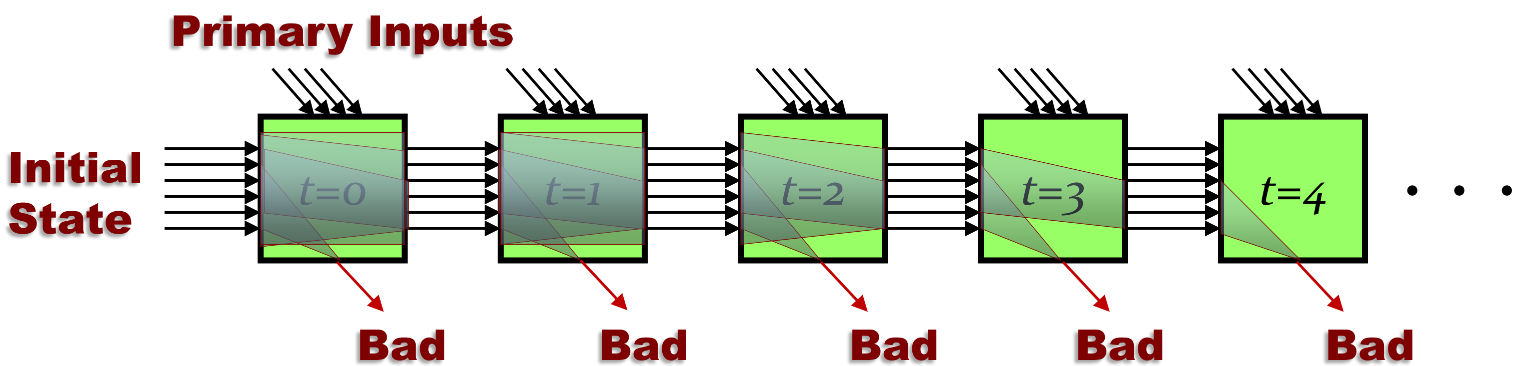
Wires can be comp-lemented. A special output is marked as the ***property***.



Bounded Model Checking

Unroll the design for 1, 2, 3, etc. time-frames.

Check if the property can fail in the last frame.



for k **in** 1..∞:
  p_{bad} = CNF(logic cone of **Bad**$_k$)
  if (*solve*({p_{bad}}))
    return CounterExample
  addClause({$\neg p_{bad}$})

Questions

- Why grow trace ”forward”?
- Increase by more than one frame at a time?
- How about SAT preprocessing?
- Better just skip incremental SAT?

Conclusions

- SAT-solvers are implication engines.
- Clauses are the ”assembly language” of propositional reasoning.
- Two important techniques of CDCL solvers are:
  - Conflict analysis
  - Variable activity
- Most applications use *incremental* SAT and encode an *abstraction* of the real problem.