The SAT Problem

Alexander Baumgartner

Research Institute for Symbolic Computation

Boolean Satisfiability Problem

- Boolean *variables* X.
- Binary/Unary boolean *functions* F (e.g.: $\wedge, \vee, \Longrightarrow, \oplus, \neg, Id, \dots$).
- Boolean *expressions* are built from X, F and parenthesis.
- *Truth assignment:* Assignment of boolean values to the variables.
  - We use 0 for false and 1 for true.
- *Satisfying truth assignment:* Expression evaluates to 1.

Boolean Satisfiability Problem

Definition (SAT Problem)

Given a boolean expression, does it have a satisfying truth assignment?

Boolean Satisfiability Problem

Definition (SAT Problem)

Given a boolean expression, does it have a satisfying truth assignment?

Example

The expression $\neg((x_1 \vee \neg(x_2 \wedge (x_3 \implies x_2))) \vee \neg x_3)$ is SAT.
There is a satisfying truth assignment $x_1 = 0, x_2 = 1, x_3 = 1$.
$x_1 \wedge \neg x_1$ is UNSAT. There is no satisfying truth assignment.

SAT is *NP*-Complete

- SAT was the first known *NP*-complete problem.
- Proved in Cook Levin Theorem.
  - SAT is in *NP*.
  - All problems in *NP* are polynomially reducible to SAT.

CNF-SAT

- *Literal:* Boolean variable or its negation (e.g. x or $\neg x$).
- *Clause:* Logical OR of one or more literals (e.g. $x_1 \vee x_2 \vee \neg x_3$).
- A boolean expression is in *CNF* if it's the logical AND of clauses (e.g. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_4)$).

CNF-SAT is *NP*-Complete

- CNF-SAT is in *NP*.
  - Trivial, it's a special case of SAT.
- CNF-SAT is *NP*-hard.
  - Can be shown by reducing SAT to CNF-SAT.

CNF-SAT is *NP*-Complete

- CNF-SAT is in *NP*.
  - Trivial, it's a special case of SAT.
- CNF-SAT is *NP*-hard.
  - Can be shown by reducing SAT to CNF-SAT.
    - Rewrite $\Longrightarrow, \Longleftrightarrow, \oplus, \ldots$ to combinations of $\wedge, \vee, \neg$. Consider 4 unary and 16 binary boolean functions. $O(n)$.

CNF-SAT is *NP*-Complete

- CNF-SAT is in *NP*.
  - Trivial, it's a special case of SAT.
- CNF-SAT is *NP*-hard.
  - Can be shown by reducing SAT to CNF-SAT.
    - Rewrite $\Longrightarrow, \Longleftrightarrow, \oplus, \ldots$ to combinations of $\wedge, \vee, \neg$.
      Consider 4 unary and 16 binary boolean functions. $O(n)$.
    - Convert expression so that negation is applied only to variables.

$$\begin{array}{rcl} \neg(f \vee g) & = & \neg f \wedge \neg g \\ \neg(f \wedge g) & = & \neg f \vee \neg g \\ \neg\neg f & = & f \end{array}$$

      Straightforward recursive implementation. $O(n)$.

CNF-SAT is *NP*-Complete

- CNF-SAT is in *NP*.
  - Trivial, it's a special case of SAT.
- CNF-SAT is *NP*-hard.
  - Can be shown by reducing SAT to CNF-SAT.
    - Rewrite $\Longrightarrow, \Longleftrightarrow, \oplus, \ldots$ to combinations of $\wedge, \vee, \neg$.
      Consider 4 unary and 16 binary boolean functions. $O(n)$.
    - Convert expression so that negation is applied only to variables.

$$
\begin{array}{rcl}
\neg(f \vee g) & = & \neg f \wedge \neg g \\
\neg(f \wedge g) & = & \neg f \vee \neg g \\
\neg\neg f & = & f
\end{array}
$$

      Straightforward recursive implementation. $O(n)$.
    - Construct from the result an expression in CNF.
      Consider base case where a formula ϕ is a literal.
      Consider two recursion cases $\phi = \phi_1 \wedge \phi_2$ and $\phi = \phi_1 \vee \phi_2$.

CNF-SAT is *NP*-Complete

- CNF-SAT is in *NP*.
  - Trivial, it's a special case of SAT.
- CNF-SAT is *NP*-hard.
  - Can be shown by reducing SAT to CNF-SAT.
    - Rewrite $\Longrightarrow, \Longleftrightarrow, \oplus, \ldots$ to combinations of $\wedge, \vee, \neg$.
      Consider 4 unary and 16 binary boolean functions. $O(n)$.
    - Convert expression so that negation is applied only to variables.

$$\begin{aligned} \neg(f \vee g) &= \neg f \wedge \neg g \\ \neg(f \wedge g) &= \neg f \vee \neg g \\ \neg\neg f &= f \end{aligned}$$

      Straightforward recursive implementation. $O(n)$.
    - Construct from the result an expression in CNF.
      Consider base case where a formula ϕ is a literal.
      Consider two recursion cases $\phi = \phi_1 \wedge \phi_2$ and $\phi = \phi_1 \vee \phi_2$.
- Proof yields an algorithm to rewrite SAT to CNF-SAT in polynomial time.

Example SAT to CNF-SAT

- $\neg((x_1 \vee \neg(x_2 \wedge (x_3 \implies x_2))) \vee \neg x_3)$
- $\neg((x_1 \vee \neg(x_2 \wedge (\neg x_3 \vee x_2))) \vee \neg x_3)$
- $((\neg x_1 \wedge (x_2 \wedge (\neg x_3 \vee x_2))) \wedge x_3)$
- $(\neg x_1) \wedge (x_2) \wedge (\neg x_3 \vee x_2) \wedge (x_3)$

Variants of CNF-SAT

- k-SAT $=$ k literals in each clause.
  - k-SAT is NP-complete.

Variants of CNF-SAT

- k-SAT = k literals in each clause.
  - k-SAT is NP-complete.
- 3-SAT = 3 literals in each clause.
  - e.g. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$
  - 3-SAT is NP-complete.
- 2-SAT = 2 literals in each clause.
  - 2-SAT is NL-complete.

Variants of CNF-SAT

- k-SAT $=$ *k* literals in each clause.
  - k-SAT is *NP*-complete.
- 3-SAT $=$ 3 literals in each clause.
  - e.g. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$
  - 3-SAT is *NP*-complete.
- 2-SAT $=$ 2 literals in each clause.
  - 2-SAT is *NL*-complete.
- Horn-SAT $=$ Each clause has at most one positive literal (head).
  - e.g. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1) \wedge (x_2 \vee \neg x_1)$
  - Horn-SAT is *P*-complete.

Variants of CNF-SAT

- k-SAT = k literals in each clause.
  - k-SAT is NP-complete.
- 3-SAT = 3 literals in each clause.
  - e.g. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$
  - 3-SAT is NP-complete.
- 2-SAT = 2 literals in each clause.
  - 2-SAT is NL-complete.
- Horn-SAT = Each clause has at most one positive literal (head).
  - e.g. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1) \wedge (x_2 \vee \neg x_1)$
  - Horn-SAT is P-complete.
- MUS-SAT = Minimal unsatisfiable subsets (MUSes).
  - MUS = Unsatisfiable subset of clauses such that any of its proper subsets is satisfiable.

SAT Solver

- Input: Boolean expression in CNF.
- Output: $\begin{cases} \text{SAT and satisfying truth assignment or} \\ \text{UNSAT and (maybe) certificate.} \end{cases}$

SAT Solver

- Input: Boolean expression in CNF.
- Output: $\begin{cases} \text{SAT and satisfying truth assignment or} \\ \text{UNSAT and (maybe) certificate.} \end{cases}$

- Trial and error (recursive, backtracking).
- Davis-Putnam-Logemann-Loveland (DPLL).
  - Systematic backtracking algorithm.
- Paturi-Pudlak-Saks-Zani (PPSZ).
  - Heuristic randomized algorithm.

DPLL Rules

- Partial truth assignments M, N; Set E of clauses C; Literals l.
- $M \models \neg C$, if C is false under M.
- $E \models C$, if C is true in all models of E.

DPLL Rules

- Partial truth assignments M, N; Set E of clauses C; Literals l.
- $M \models \neg C$, if C is false under M.
- $E \models C$, if C is true in all models of E.
  - **UnitPropagate**
    $M \parallel E \cup \{C \vee l\} \Longrightarrow M \cdot l \parallel E \cup \{C \vee l\}$,
    if $M \models \neg C$ and l is undefined in M.

DPLL Rules

- Partial truth assignments M, N; Set E of clauses C; Literals l.
- $M \models \neg C$, if C is false under M.
- $E \models C$, if C is true in all models of E.
  - **UnitPropagate**
    $M \parallel E \cup \{C \vee l\} \Longrightarrow M \cdot l \parallel E \cup \{C \vee l\}$,
    if $M \models \neg C$ and l is undefined in M.
  - **Decide**
    $M \parallel E \Longrightarrow M \cdot l^d \parallel E$,
    if l or $\neg l$ occurs in a clause of E and l is undefined in M.

DPLL Rules

- Partial truth assignments M, N; Set E of clauses C; Literals l.
- $M \models \neg C$, if C is false under M.
- $E \models C$, if C is true in all models of E.
  - **UnitPropagate**
    $$M \parallel E \cup \{C \vee l\} \Longrightarrow M \cdot l \parallel E \cup \{C \vee l\},$$
    if $M \models \neg C$ and l is undefined in M.
  - **Decide**
    $$M \parallel E \Longrightarrow M \cdot l^d \parallel E,$$
    if l or $\neg l$ occurs in a clause of E and l is undefined in M.
  - **Fail**
    $$M \parallel E \cup \{C\} \Longrightarrow \text{fail},$$
    if $M \models \neg C$ and M contains no decision literals.

DPLL Rules

- Partial truth assignments M, N; Set E of clauses C; Literals l.
- $M \models \neg C$, if C is false under M.
- $E \models C$, if C is true in all models of E.
  - **UnitPropagate**
    $$M \parallel E \cup \{C \vee l\} \Longrightarrow M \cdot l \parallel E \cup \{C \vee l\},$$
    if $M \models \neg C$ and l is undefined in M.
  - **Decide**
    $$M \parallel E \Longrightarrow M \cdot l^d \parallel E,$$
    if l or $\neg l$ occurs in a clause of E and l is undefined in M.
  - **Fail**
    $$M \parallel E \cup \{C\} \Longrightarrow \text{fail},$$
    if $M \models \neg C$ and M contains no decision literals.
  - **Backjump**
    $$M \cdot l^d \cdot N \parallel E \Longrightarrow M \cdot l' \parallel E,$$
    if there is some clause $C \vee l'$ such that $E \models C \vee l'$ and $M \models \neg C$ and l' is undefined in M and l or $\neg l$ occurs in a clause of E.

DPLL Efficiently

- $O(n)$ space complexity.
- $O(2^n)$ time complexity.
- Efficiency issues:
  - Efficient data structure for unit propagation.
  - Select literal in Decide rule.
  - Select literal in Backjump rule.
  - Reuse gained information after back jump - Reduce search space.

Some Implementation Strategies

- Variable (and value) selection heuristic.
- Clause learning.
- Conflict-directed backjumping.
- Assignment stack shrinking.
- Conflict clause minimization.
- The watched literals scheme.
- Fast backjumping.
- Randomized restarts.
- ...

Solver Types

- Single-engine solver.
- Portfolio approach.
- Interacting multi-engine approach.
- Parallel approach.

Events

- Biannual SAT-Race/Challenge (2012, June 17-20, Trento, Italy).
- Biannual SAT-Competition (2013, July 8-12, Helsinki, Finland).
- Clear input/output specification.
- Different problem types:
  - Application,
  - Hand crafted,
  - Random.

`http://www.satcompetition.org/`

SAT-Challenge 2012 / Application SAT+UNSAT

Rank	RiG	Solver	# solved	% solved	cum. run-time	median run-time
–	–	Virtual Best Solver (VBS)	568	94.7	56528	30.3
1	1	SATzilla2012 APP	531	88.5	85194	114.0
2	2	SATzilla2012 ALL	515	85.8	86638	122.2
3	1	Industrial SAT Solver	499	83.2	93705	160.2
–	–	lingeling (SAT Competition 2011 Bronze)	488	81.3	84715	135.3
4	2	interactSAT	480	80.0	87676	152.5
5	1	glucose	475	79.2	71501	114.4
6	2	SINN	472	78.7	86302	146.4
7	3	ZENN	468	78.0	74019	124.7
8	4	Lingeling	467	77.8	91973	185.5
9	5	linge_dyphase	458	76.3	90192	204.4
10	6	simpsat	453	75.5	95737	222.0

http://baldur.iti.kit.edu/SAT-Challenge-2012/

Input Format

```
c This is UnifRandomKSATGenerator
c uniform random 6-SAT generated instance with:
c clause length: 6
c number variables: 200
c number clauses: 8674
c clause to variable ratio: 43.37
c random number generator name: SHA1PRNG
c random number generator provider: SUN
c random number generator seed: 5915616858147256l8
p cnf 200 8674
-40 146 89 -186 107 -36 0
65 99 -6 73 -119 30 0
35 -41 -59 -180 -144 198 0
-91 -105 49 79 61 -18 0
-152 87 185 -130 -66 -119 0
...
```

Output Format

```
c predict which solver should be used ....
c solver ranking 2 4 8 3 1 0 5 6 7
c run and check best solver 26 ...
c child timeout set to 1200
c child exited successfully
c TIME USED: 0.000000
c clasp version 2.0.0-RC2
c Reading from test.cnf
c Solving...
c Answer: 1
v -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -20
...
v -199 200 0
s SATISFIABLE

c Models : 1+
c Time : 0.004s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
c CPU Time : 0.000s
```

Solving Problems with SAT

- State of the art SAT solvers are highly sophisticated.
- Encode problem as boolean expression.
- Use a SAT solver to find satisfying truth assignment.

Encode 3-Coloring Problem

- Given a graph (V, E) find an assignment of one of 3 colors to each vertex such that no two adjacent vertices share a color.

Encode 3-Coloring Problem

- Given a graph (V, E) find an assignment of one of 3 colors to each vertex such that no two adjacent vertices share a color.
- For each vertex $v \in V$:

$$(v(1) \vee v(2) \vee v(3)) \wedge (\neg v(1) \vee \neg v(2)) \wedge (\neg v(1) \vee \neg v(3)) \wedge (\neg v(2) \vee \neg v(3))$$

Encode 3-Coloring Problem

- Given a graph (V, E) find an assignment of one of 3 colors to each vertex such that no two adjacent vertices share a color.
- For each vertex $v \in V$:

$$(v(1) \vee v(2) \vee v(3)) \wedge (\neg v(1) \vee \neg v(2)) \wedge (\neg v(1) \vee \neg v(3)) \wedge (\neg v(2) \vee \neg v(3))$$

- For each edge $(v, u) \in E$:

$$(\neg v(1) \vee \neg u(1)) \wedge (\neg v(2) \vee \neg u(2)) \wedge (\neg v(3) \vee \neg u(3))$$

- Vertex v colored with color i iff $v(i)$ true in the model.