SAT Solving with Conflict Driven Clause Learning

William Schultz

CS 7240 Final Project

April 27, 2022

Overview and Project Goals

- Satisfiability is the canonical NP-complete problem.
- Much work has been devoted to building efficient SAT solvers over last decades.
- **Project Goal:** Implement a basic SAT solver based on *conflict driven clause learning* (CDCL), the dominant core technique used in modern solvers.
  - Gain a deeper understanding of the DPLL and CDCL based algorithms for SAT solving
  - Use as a platform for potentially exploring new SAT solving techniques and understanding limitations of existing ones

Review: The SAT Problem

The SAT problem:

Given a boolean formula in conjunctive normal form, determine whether there exists an assignment to the variables of the formula that makes the overall formula true.

Review: The SAT Problem

The SAT problem:

Given a boolean formula in conjunctive normal form, determine whether there exists an assignment to the variables of the formula that makes the overall formula true.

e.g.

$$(x_1 \lor x_2) \land (\neg x_3 \lor \neg x_1)$$

Review: The SAT Problem

The SAT problem:

Given a boolean formula in conjunctive normal form, determine whether there exists an assignment to the variables of the formula that makes the overall formula true.

e.g.

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

$$\text{SAT, with } \{x_1 = 1, x_2 = 0, x_3 = 0\}.$$

Review: The SAT Problem

The SAT problem:

Given a boolean formula in conjunctive normal form, determine whether there exists an assignment to the variables of the formula that makes the overall formula true.

e.g.

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

$$\text{SAT, with } \{x_1 = 1, x_2 = 0, x_3 = 0\}.$$

CNF notation:

$$\{\{x_1, x_2\}, \{\neg x_3, \neg x_1\}\}$$

DPLL: SAT as Search

- A basic approach to solving SAT is to view it as a search problem over possible assignments.
- This is the basis of the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [DLL62]
- Basic idea of DPLL is to do a depth first, brute force search with backtracking along with some basic formula simplification as you go.
    - Also employs the *unit propagation rule*

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.
- A *unit clause* is a clause that contains exactly one literal.

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.
- A *unit clause* is a clause that contains exactly one literal.
- If a CNF formula contains a unit clause then we can apply unit propagation i.e. set that literal to the appropriate truth value to satisfy its clause e.g.

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.
- A *unit clause* is a clause that contains exactly one literal.
- If a CNF formula contains a unit clause then we can apply unit propagation i.e. set that literal to the appropriate truth value to satisfy its clause e.g.

$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.
- A *unit clause* is a clause that contains exactly one literal.
- If a CNF formula contains a unit clause then we can apply unit propagation i.e. set that literal to the appropriate truth value to satisfy its clause e.g.

$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$
$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.
- A *unit clause* is a clause that contains exactly one literal.
- If a CNF formula contains a unit clause then we can apply unit propagation i.e. set that literal to the appropriate truth value to satisfy its clause e.g.

$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$
$$\{\{\color{green}{b}\}, \{\color{red}{\neg b}, \neg c\}, \{c, \neg d\}\}$$
$$\{\{\neg c\}, \{c, \neg d\}\}$$

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.
- A *unit clause* is a clause that contains exactly one literal.
- If a CNF formula contains a unit clause then we can apply unit propagation i.e. set that literal to the appropriate truth value to satisfy its clause e.g.

$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$
$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$
$$\{\{\neg c\}, \{c, \neg d\}\}$$
$$\{\{\neg c\}, \{c, \neg d\}\}$$

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.
- A *unit clause* is a clause that contains exactly one literal.
- If a CNF formula contains a unit clause then we can apply unit propagation i.e. set that literal to the appropriate truth value to satisfy its clause e.g.

$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$
$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$
$$\{\{\neg c\}, \{c, \neg d\}\}$$
$$\{\{\neg c\}, \{c, \neg d\}\}$$
$$\{\{\neg d\}\}$$

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.
- A *unit clause* is a clause that contains exactly one literal.
- If a CNF formula contains a unit clause then we can apply unit propagation i.e. set that literal to the appropriate truth value to satisfy its clause e.g.

$$\{\{b\},\{\neg b,\neg c\},\{c,\neg d\}\}$$
$$\{\{b\},\{\neg b,\neg c\},\{c,\neg d\}\}$$
$$\{\{\neg c\},\{c,\neg d\}\}$$
$$\{\{\neg c\},\{c,\neg d\}\}$$
$$\{\{\neg d\}\}$$
$$\{\{\neg d\}\}$$

Unit Propagation

- Core simplification rule employed in DPLL, and also in CDCL as we will see later.
- A *unit clause* is a clause that contains exactly one literal.
- If a CNF formula contains a unit clause then we can apply unit propagation i.e. set that literal to the appropriate truth value to satisfy its clause e.g.

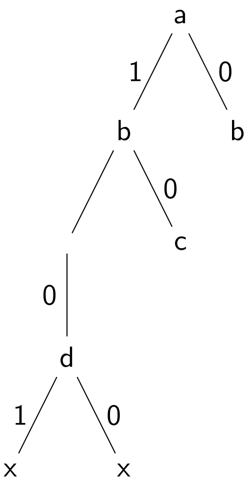
$$\{\{b\},\{\neg b,\neg c\},\{c,\neg d\}\}$$
$$\{\{b\},\{\neg b,\neg c\},\{c,\neg d\}\}$$
$$\{\{\neg c\},\{c,\neg d\}\}$$
$$\{\{\neg c\},\{c,\neg d\}\}$$
$$\{\{\neg d\}\}$$
$$\{\{\neg d\}\}$$
$$\{\} \quad \text{(SAT)}$$

DPLL: Example

$\{\neg a, b\}$

$\{\neg b, \neg c\}$

$\{\neg c, \neg d\}$

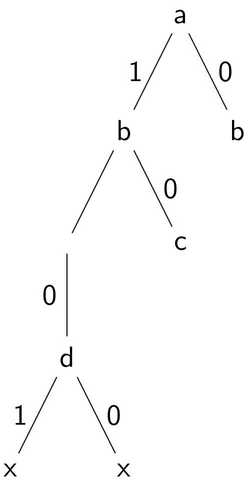


DPLL: Example

$\{\neg a, b\}$

$\{\neg b, \neg c\}$

$\{\neg c, \neg d\}$

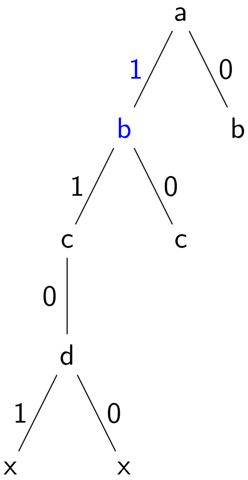


DPLL: Example

$\{\neg a, b\}$

$\{\neg b, \neg c\}$

$\{\neg c, \neg d\}$



Beyond DPLL: Learning from Conflicts

- DPLL is a rather naive algorithm

Beyond DPLL: Learning from Conflicts

- DPLL is a rather naive algorithm
- An extension to this basic framework is to *learn from conflicts*

Beyond DPLL: Learning from Conflicts

- DPLL is a rather naive algorithm
- An extension to this basic framework is to *learn from conflicts*
- When you encounter a conflict in the search tree, *learn* a clause that prevents you from making the similar mistakes again

Beyond DPLL: Learning from Conflicts

- DPLL is a rather naive algorithm
- An extension to this basic framework is to *learn from conflicts*
- When you encounter a conflict in the search tree, *learn* a clause that prevents you from making the similar mistakes again
- This fundamental approch is known as *conflict-driven clause learning* (CDCL) and started being employed in SAT solvers around the late 90s and early 2000s.

Beyond DPLL: Learning from Conflicts

- DPLL is a rather naive algorithm
- An extension to this basic framework is to *learn from conflicts*
- When you encounter a conflict in the search tree, *learn* a clause that prevents you from making the similar mistakes again
- This fundamental approch is known as *conflict-driven clause learning* (CDCL) and started being employed in SAT solvers around the late 90s and early 2000s.
- In addition, employ *non-chronological backtracking*

CDCL

- When using CDCL, if a conflict is encountered, we not only backtrack to the previous level, as in DPLL
- We try to learn a *conflict clause* along with a *backjump* level, which determines how far back in the search tree to unwind to.

CDCL: Example

$\{a, b\}$

$\{b, c\}$

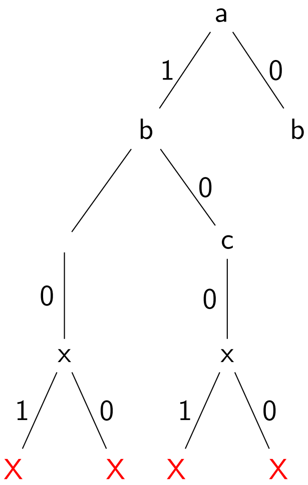
$\{\neg a, \neg x, y\}$

$\{\neg a, x, z\}$

$\{\neg a, \neg y, z\}$

$\{\neg a, x, \neg z\}$

$\{\neg a, \neg y, \neg z\}$



CDCL: Example

$\{a, b\}$

$\{b, c\}$

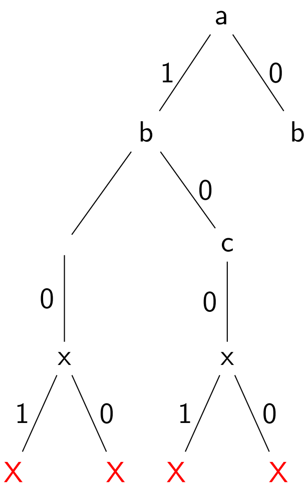
$\{\neg a, \neg x, y\}$

$\{\neg a, x, z\}$

$\{\neg a, \neg y, z\}$

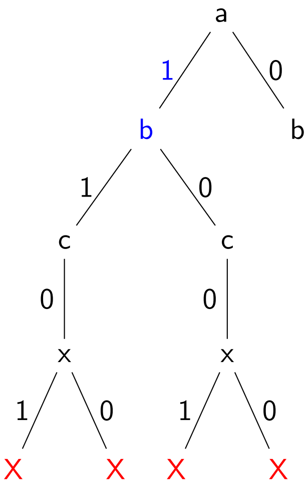
$\{\neg a, x, \neg z\}$

$\{\neg a, \neg y, \neg z\}$



CDCL: Example

$\{a, b\}$
$\{b, c\}$
$\{\neg a, \neg x, y\}$
$\{\neg a, x, z\}$
$\{\neg a, \neg y, z\}$
$\{\neg a, x, \neg z\}$
$\{\neg a, \neg y, \neg z\}$



SAT Solver Implementation

- Worked on implementing my own CDCL SAT solver as a framework for exploring future potential SAT enhancements
- Around 1500 lines of C++, tested on a variety of easy to medium SAT benchmark problems
- `https://github.com/will62794/mysat`

Evaluation

- Some peformance results of my SAT solver against a performant, modern solver.

Future Extensions

- Resolution proofs
- Variable ordering heuristics
- Learning heuristics
- Learning end to end SAT solver (neuroSAT)

Martin Davis, George Logemann, and Donald Loveland.
A machine program for theorem-proving.
Commun. ACM, 5(7):394–397, jul 1962.